

IMSL[®]

Fortran Numerical Library

User's Guide MATH LIBRARY

VERSION 7.0



CORPORATE HEADQUARTERS

Rogue Wave Software 5500 Flatiron
Parkway Suite 200 Boulder, CO 80301
USA
T: 303.545.3220 F: 303.473.9137

IMSL Libraries Contact Information

USA Toll Free: 800.222.4675
T: 713.784.3131 F: 713.781.9260
Email: info@vni.com Web site: www.vni.com

Worldwide Offices

USA • UK • France • Germany • Japan For
contact information, please visit
www.vni.com/contact/worldwideoffices.php

© 1970-2010 Rogue Wave Software, Visual Numerics, IMSL and PV-WAVE are registered trademarks of Rogue Wave Software, Inc. in the U.S. and other countries. JMSL, JWAVE, TS-WAVE, PyIMSL and Knowledge in Motion are trademarks of Rogue Wave Software, Inc. All other company, product or brand names are the property of their respective owners.

IMPORTANT NOTICE: Information contained in this documentation is subject to change without notice. Use of this document is subject to the terms and conditions of a Rogue Wave Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability. If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. This documentation may not be copied or distributed in any form without the express written consent of Rogue Wave Software.



Embeddable mathematical and statistical algorithms available for C, C#.NET, Java™, Fortran and Python applications

Table of Contents

Introduction	i
The IMSL Fortran Numerical Library	i
User Background	i
Getting Started	ii
Finding the Right Routine	iii
Organization of the Documentation	iii
Naming Conventions.....	iv
Using Library Subprograms	v
Programming Conventions	vi
Module Usage	vi
Using MPI Routines.....	vii
Programming Tips.....	viii
Optional Subprogram Arguments	ix
Optional Data	ix
Overloaded =, /=, etc., for Derived Types.....	x
Error Handling	xi
Printing Results	xii
Fortran 90 Constructs.....	xii
Shared-Memory Multiprocessors and Thread Safety.....	xii
Using Operators and Generic Functions	xiii
Using ScaLAPACK, LAPACK, LINPACK, and EISPACK	xv
Using ScaLAPACK Enhanced Routines.....	xix
Chapter 1: Linear Systems	1
Routines	1
Usage Notes	5
Matrix Types	5
Solution of Linear Systems	6
Multiple Right Sides.....	7
Determinants	7
Iterative Refinement	7
Singularity	8
Special Linear Systems.....	8
Iterative Solution of Linear Systems	8
QR Decomposition	9
LIN_SOL_GEN	10
LIN_SOL_SELF	18
LIN_SOL_LSQ.....	27

LIN_SOL_SVD.....	36
LIN_SOL_TRI.....	45
LIN_SVD.....	57
Parallel Constrained Least-Squares Solvers.....	66
Solving Constrained Least-Squares Systems.....	66
PARALLEL_NONNEGATIVE_LSQ.....	67
PARALLEL_BOUNDED_LSQ.....	75
LSARG.....	83
LSLRG.....	87
LFCRG.....	93
LFTRG.....	99
LFSRG.....	103
LFIRG.....	108
LFDRG.....	113
LINRG.....	115
LSACG.....	119
LSLCG.....	123
LFCCG.....	128
LFTCG.....	134
LFSCG.....	138
LFICG.....	143
LFDCG.....	148
LINCG.....	150
LSLRT.....	154
LFCRT.....	158
LFDRT.....	162
LINRT.....	163
LSLCT.....	165
LFCCT.....	169
LFDCT.....	173
LINCT.....	175
LSADS.....	177
LSLDS.....	181
LFCDS.....	186
LFTDS.....	191
LFSDS.....	195
LFIDS.....	199
LFDDS.....	204
LINDS.....	206
LSASF.....	210
LSLSF.....	213
LFCSF.....	215
LFTSF.....	218
LFSSF.....	221
LFISF.....	223
LFDSF.....	226
LSADH.....	227
LSLDH.....	232
LFCDH.....	237

LFTDH.....	243
LFSDH.....	248
LFIDH.....	252
LFDDH.....	258
LSAHF.....	259
LSLHF.....	262
LFCHF.....	265
LFTHF.....	268
LFSHF.....	271
LFIHF.....	273
LFDHF.....	276
LSLTR.....	278
LSLCR.....	279
LSARB.....	282
LSLRB.....	285
LFCRB.....	290
LFTRB.....	293
LFSRB.....	296
LFIRB.....	298
LFDRB.....	301
LSAQS.....	303
LSLQS.....	305
LSLPB.....	308
LFCQS.....	311
LFTQS.....	314
LFSQS.....	316
LFIQS.....	318
LFDQS.....	320
LSLTQ.....	322
LSLCQ.....	324
LSACB.....	327
LSLCB.....	330
LFCCB.....	333
LFTCB.....	336
LFSCB.....	339
LFICB.....	341
LFDCB.....	344
LSAQH.....	346
LSLQH.....	349
LSLQB.....	352
LFCQH.....	355
LFTQH.....	358
LFSQH.....	360
LFIQH.....	362
LFDQH.....	365
LSLXG.....	366
LFTXG.....	372
LFSXG.....	377
LSLZG.....	380

LFTZG	385
LFSZG	391
LSLXD	394
LSCXD	399
LNFXD	403
LFSXD	408
LSLZD	412
LNFXD	416
LFSZD	421
LSLTO	424
LSLTC	426
LSLCC	428
PCGRC	431
JCGRC	437
GMRES	440
ARPACK_SVD	451
LSQRR	451
LQRRV	457
LSBRR	463
LCLSQ	467
LQRRR	471
LQERR	478
LQRSL	483
LUPQR	489
LCHRG	494
LUPCH	496
LDNCH	499
LSVRR	503
LSVCR	510
LSGRR	514

Chapter 2: Eigensystem Analysis 521

Routines	521
Usage Notes	522
Reformulating Generalized Eigenvalue Problems	525
Using ARPACK for Ordinary and Generalized Eigenvalue Problems	526
LIN_EIG_SELF	526
LIN_EIG_GEN	533
LIN_GEIG_GEN	542
EVLRG	549
EVCRC	552
EPIRG	555
EVLGC	557
EVCCG	559
EPICG	562
EVLSF	564
EVCSF	566
EVASF	568

EVESF	570
EVBSF	573
EVFSF	575
EPISF	578
EVLBS	580
EVCBS	582
EVASB	585
EVESB	588
EVBSB	591
EVFSB	593
EPISB	596
EVLHF	598
EVCHF	601
EVAHF	604
EVEHF	606
EVBHF	609
EVFHF	612
EPIHF	615
EVLRH	617
EVCRH	619
EVLCH	621
EVCCH	623
GVLRG	626
GVCRG	629
GPIRG	632
GVLCG	634
GVCCG	637
GPICG	640
GVLSP	642
GVCSP	645
GPISP	648
Eigenvalues and Eigenvectors Computed with ARPACK	651
The Abstract Interfaces for User-Written Array Functions	652
The Base Class ARPACKBASE	653
ARPACK_SYMMETRIC	654
ARPACK_SVD	668
ARPACK_NONSYMMETRIC	676
ARPACK_COMPLEX	685

Chapter 3: Interpolation and Approximation 693

Routines	693
Usage Notes	695
Piecewise Polynomials	695
Splines and B-splines	695
Cubic Splines	697
Tensor Product Splines	698
Quadratic Interpolation	699
Multi-dimensional Interpolation	699

Least Squares	699
Smoothing by Cubic Splines	699
Rational Chebyshev Approximation.....	699
Using the Univariate Spline Routines.....	699
Choosing an Interpolation Routine	701
SPLINE_CONSTRAINTS	702
SPLINE_VALUES	703
SPLINE_FITTING.....	704
SURFACE_CONSTRAINTS	714
SURFACE_VALUES	715
SURFACE_FITTING	716
CSIEZ	727
CSINT	729
CSDEC	732
CSHER.....	737
CSAKM	740
CSCON	742
CSPER	746
CSVAL.....	749
CSDER.....	750
CS1GD	753
CSITG.....	756
SPLEZ.....	758
BSINT	761
BSNAK	765
BSOPK.....	768
BS2IN	771
BS3IN	776
BSVAL.....	782
BSDER.....	783
BS1GD.....	786
BSITG.....	789
BS2VL	792
BS2DR	794
BS2GD.....	797
BS2IG	801
BS3VL	805
BS3DR	807
BS3GD.....	811
BS3IG	817
BSCPP.....	821
PPVAL.....	823
PPDER	825
PP1GD	828
PPITG	831
QDVAL.....	833
QDDER.....	835
QD2VL.....	838
QD2DR	840

QD3VL	843
QD3DR	847
SURF	851
SURFND	855
RLINE	858
RCURV	861
FNLSQ	865
BSLSQ	870
BSVLS	874
CONFT	879
BSLS2	889
BSLS3	894
CSSD	900
CSSMH	904
CSSCV	907
RATCH	910

Chapter 4: Integration and Differentiation **915**

Routines	915
Usage Notes	916
Univariate Quadrature	916
Multivariate Quadrature	917
Gauss Rules and Three-term Recurrences	917
Numerical Differentiation	918
QDAGS	918
QDAG	922
QDAGP	925
QDAG1D	929
QDAGI	935
QDAWO	938
QDAWF	942
QDAWS	946
QDAWC	949
QDNG	953
TWODQ	955
QDAG2D	960
QDAG3D	966
QAND	973
QMC	976
QQRUL	979
QQRCF	983
RECCF	986
RECQR	988
FQRUL	991
DERIV	995

Chapter 5: Differential Equations **999**

Routines	999
----------------	-----

Usage Notes	1000
Ordinary Differential Equations	1000
Differential-algebraic Equations	1001
Partial Differential Equations	1001
Summary	1002
IVPRK	1003
IVMRK	1011
IVPAG	1021
BVPFD	1037
BVPMS	1050
DAESL	1057
DASPG	1072
IVOAM	1072
Introduction to Subroutine PDE_1D_MG	1080
PDE_1D_MG	1081
Description	1089
Remarks on the Examples	1090
Example 1 - Electrodynamics Model	1092
Example 2 - Inviscid Flow on a Plate	1095
Example 3 - Population Dynamics	1098
Example 4 - A Model in Cylindrical Coordinates	1101
Example 5 - A Flame Propagation Model	1102
Example 6 - A 'Hot Spot' Model	1105
Example 7 - Traveling Waves	1107
Example 8 - Black-Scholes	1109
Example 9 - Electrodynamics, Parameters Studied with MPI	1111
MMOLCH	1115
MOLCH	1128
FEYNMAN_KAC	1128
HQSVAL	1185
FPS2H	1188
FPS3H	1194
SLEIG	1201
SLCNT	1213

Chapter 6: Transforms 1217

Routines	1217
Usage Notes	1218
Fast Fourier Transforms	1218
Continuous versus Discrete Fourier Transform	1219
Inverse Laplace Transform	1220
FAST_DFT	1220
FAST_2DFT	1227
FAST_3DFT	1233
FFTRF	1236
FFTRB	1240
FFTRI	1243
FFTCF	1245

FFTCB	1248
FFTCI.....	1251
FSINT	1253
FSINI	1255
FCOST	1257
FCOSI.....	1259
QSINF.....	1261
QSINB	1263
QSINI.....	1266
QCOSF.....	1268
QCOSB	1270
QCOSI	1272
FFT2D.....	1274
FFT2B	1277
FFT3F	1281
FFT3B	1285
RCONV	1289
CCONV	1294
RCORL	1299
CCORL	1304
INLAP.....	1309
SINLP	1311

Chapter 7: Nonlinear Equations **1319**

Routines	1319
Usage Notes	1319
Zeros of a Polynomial	1319
Zero(s) of a Function	1320
Root of System of Equations	1320
ZPLRC	1320
ZPORC.....	1322
ZPOCC.....	1324
ZANLY	1325
ZUNI.....	1328
ZBREN	1331
ZREAL.....	1334
NEQNF	1337
NEQNJ.....	1340
NEQBF	1344
NEQBJ	1350

Chapter 8: Optimization **1357**

Routines	1357
Usage Notes	1358
Unconstrained Minimization	1358
Minimization with Simple Bounds	1359
Linearly Constrained Minimization.....	1359
Nonlinearly Constrained Minimization	1359

Selection of Routines.....	1360
UVMIF.....	1362
UVMID.....	1365
UVMGS.....	1369
UMINF.....	1372
UMING.....	1377
UMIDH.....	1384
UMIAH.....	1389
UMCGF.....	1395
UMCGG.....	1399
UMPOL.....	1403
UNLSF.....	1407
UNLSJ.....	1413
BCONF.....	1420
BCONG.....	1427
BCODH.....	1434
BCOAH.....	1441
BCPOL.....	1448
BCLSF.....	1452
BCLSJ.....	1459
BCNLS.....	1466
READ_MPS.....	1475
MPS File Format.....	1480
NAME Section.....	1481
ROWS Section.....	1481
COLUMNS Section.....	1481
RHS Section.....	1481
RANGES Section.....	1482
BOUNDS Section.....	1483
QUADRATIC Section.....	1484
ENDATA Section.....	1484
MPS_FREE.....	1485
DENSE_LP.....	1488
DLPRS.....	1494
SLPRS.....	1497
TRAN.....	1504
QPROG.....	1506
LCONF.....	1510
LCONG.....	1516
NNLPF.....	1522
NNLPG.....	1528
CDGRD.....	1536
FDGRD.....	1538
FDHES.....	1541
GDHES.....	1543
DDJAC.....	1546
FDJAC.....	1555
CHGRD.....	1558
CHHES.....	1561

CHJAC.....	1565
GGUES.....	1569

Chapter 9: Basic Matrix/Vector Operations 1573

Routines.....	1573
Basic Linear Algebra Subprograms.....	1576
Programming Notes for Level 1 BLAS.....	1576
Descriptions of the Level 1 BLAS Subprograms.....	1577
Programming Notes for Level 2 and Level 3 BLAS.....	1588
Descriptions of the Level 2 and Level 3 BLAS.....	1589
Programming Notes for BLAS Using NVIDIA.....	1601
CUBLAS_GET.....	1607
CUBLAS_SET.....	1609
CHECK_BUFFER_ALLOCATION.....	1611
CUDA_ERROR_PRINT.....	1612
Other Matrix/Vector Operations.....	1614
CRGRG.....	1615
CCGCG.....	1616
CRBRB.....	1617
CCBCB.....	1619
CRGRB.....	1621
CRBRG.....	1622
CCGCB.....	1624
CCBCG.....	1626
CRGCG.....	1627
CRRCR.....	1629
CRBCB.....	1631
CSFRG.....	1632
CHFCG.....	1634
CSBRB.....	1635
CHBCB.....	1637
TRNRR.....	1639
MXTXF.....	1641
MXTYF.....	1643
MXYTF.....	1645
MRRRR.....	1647
MCRCR.....	1649
HRRRR.....	1651
BLINF.....	1653
POLRG.....	1655
MURRV.....	1657
MURBV.....	1659
MUCRV.....	1661
MUCBV.....	1663
ARBRB.....	1665
ACBCB.....	1667
NRIRR.....	1670
NRIRR.....	1671

NR2RR.....	1673
NR1RB.....	1674
NR1CB.....	1676
DISL2.....	1677
DISL1.....	1679
DISLI.....	1681
VCONR.....	1683
VCONC.....	1685
Extended Precision Arithmetic.....	1687

Chapter 10: Linear Algebra Operators and Generic Functions 1691

Routines	1691
Usage Notes	1692
Matrix Optional Data Changes.....	1692
Dense Matrix Computations.....	1694
Dense Matrix Functions	1696
Dense Matrix Parallelism Using MPI	1697
General Remarks	1697
Getting Started with Modules <code>MPI_setup_int</code> and <code>MPI_node_int</code>	1697
Using Processors.....	1699
Sparse Matrix Computations	1700
Introduction	1700
Derived Type Definitions	1702
Overloaded Assignments	1704
.x.....	1707
.tx.....	1711
.xt.....	1714
.hx.....	1717
.xh.....	1720
.t.....	1723
.h.....	1726
.i.....	1728
.ix.....	1730
.xi.....	1740
CHOL.....	1743
COND	1746
DET.....	1750
DIAG.....	1753
DIAGONALS	1754
EIG.....	1755
EYE.....	1759
FFT.....	1761
FFT_BOX	1763
IFFT	1765
IFFT_BOX.....	1767
isNaN	1769
NaN.....	1770
NORM.....	1771

ORTH.....	1774
RAND.....	1777
RANK.....	1779
SVD.....	1780
UNIT.....	1783

Chapter 11: Utilities 1787

Routines.....	1787
Usage Notes for ScaLAPACK Utilities.....	1789
ScaLAPACK Supporting Modules.....	1792
ScaLAPACK_SETUP.....	1792
ScaLAPACK_GETDIM.....	1794
ScaLAPACK_READ.....	1795
ScaLAPACK_WRITE.....	1797
ScaLAPACK_MAP.....	1805
ScaLAPACK_UNMAP.....	1807
ScaLAPACK_EXIT.....	1809
ERROR_POST.....	1810
SHOW.....	1813
WRRRN.....	1817
WRRRL.....	1819
WRIRN.....	1822
WRIRL.....	1825
WRCRN.....	1827
WRCRL.....	1830
WROPT.....	1833
PGOPT.....	1840
PERMU.....	1842
PERMA.....	1844
SORT_REAL.....	1846
SVRGN.....	1849
SVRGP.....	1850
SVIGN.....	1852
SVIGP.....	1853
SVRBN.....	1855
SVRBP.....	1856
SVIBN.....	1857
SVIBP.....	1859
SRCH.....	1860
ISRCH.....	1862
SSRCH.....	1864
ACHAR.....	1867
IACHAR.....	1868
ICASE.....	1869
IICSR.....	1870
IIDEX.....	1872
CVTSI.....	1873
CPSEC.....	1874

TIMDY.....	1875
TDATE.....	1876
NDAYS.....	1877
NDYIN.....	1878
IDYWK.....	1880
VERML.....	1881
RAND_GEN.....	1882
RNGET.....	1891
RNSET.....	1892
RNOPT.....	1893
RNIN32.....	1894
RNGE32.....	1895
RNSE32.....	1897
RNIN64.....	1897
RNGE64.....	1898
RNSE64.....	1900
RNUNF.....	1900
RNUN.....	1902
FAURE_INIT.....	1904
FAURE_FREE.....	1905
FAURE_NEXT.....	1905
IUMAG.....	1908
UMAG.....	1911
DUMAG.....	1914
PLOTP.....	1914
PRIME.....	1917
CONST.....	1919
CUNIT.....	1921
HYPOT.....	1925
MP_SETUP.....	1926

Reference Material

1931

Contents.....	1931
User Errors.....	1931
What Determines Error Severity.....	1931
Kinds of Errors and Default Actions.....	1932
Errors in Lower-Level Routines.....	1933
Routines for Error Handling.....	1933
ERSET.....	1933
IERCD and NIRTY.....	1934
Examples.....	1934
Machine-Dependent Constants.....	1937
IMACH.....	1937
AMACH.....	1939
DMACH.....	1940
IFNAN(X).....	1940
UMACH.....	1942
Matrix Storage Modes.....	1943

Reserved Names.....	1954
Deprecated Features and Renamed Routines	1955
Automatic Workspace Allocation	1955
Changing the Amount of Space Allocated	1955
Character Workspace.....	1956
Appendix A: GAMS Index	i
Description.....	i
IMSL MATH LIBRARY	ii
Appendix B: Alphabetical Summary of Routines	i
Routines	i
Appendix C: References	i
Appendix D: Benchmarking or Timing Programs	i
Scalar Program Descriptions.....	i
Parallel Program Descriptions.....	v
Product Support	11
Contacting IMSL Support.....	11
Index	i

Introduction

The IMSL Fortran Numerical Library

The IMSL Fortran Numerical Library consists of two separate but coordinated Libraries that allow easy user access. These Libraries are organized as follows:

- MATH/LIBRARY general applied mathematics and special functions

The User's Guide for IMSL MATH/LIBRARY has two parts:

1. MATH/LIBRARY
2. MATH/LIBRARY Special Functions

- STAT/LIBRARY statistics

Most of the routines are available in both single and double precision versions. Many routines for linear solvers and eigensystems are also available for complex and double -complex precision arithmetic. The same user interface is found on the many hardware versions that span the range from personal computer to supercomputer.

This library is the result of a merging of the products: IMSL Fortran Numerical Libraries and IMSL Fortran 90 Library.

User Background

To use this product you should be familiar with the Fortran 90 language as well as the withdrawn Fortran 77 language, which is, in practice, a subset of Fortran 90. A summary of the ISO and ANSI standard language is found in Metcalf and Reid (1990). A more comprehensive illustration is given in Adams et al. (1992).

Those routines implemented in the IMSL Fortran Numerical Library provide a simpler, more reliable user interface than was possible with Fortran 77. Features of the IMSL Fortran Numerical Library include the use of descriptive names, short required argument lists, packaged user-interface blocks, a suite of testing and benchmark software, and a collection of examples. Source code is provided for the benchmark software and examples.

Some of the routines in the IMSL Fortran Numerical Library can take advantage of a standard (MPI) Message Passing Interface environment but do not require an MPI environment if the user chooses to not take advantage of MPI.

The MPI logo shown below cues the reader when this is the case:



Routines documented with the MPI Capable logo can be called in a scalar or one computer environment.

Other routines in the IMSL Library take advantage of MPI and require that an MPI environment be present in order to use them. The MPI Required logo shown below clues the reader when this is the case:



NOTE: It is recommended that users considering using the MPI capabilities of the product read the following sections of the MATH Library documentation:

Introduction: [Using MPI Routines](#),

Introduction: [Using ScaLAPACK Enhanced Routines](#),

Chapter 10, Linear Algebra Operators and Generic Functions – see [Dense Matrix Parallelism Using MPI](#).

Vendor Supplied Libraries Usage

The IMSL Fortran Numerical Library contains functions which may take advantage of functions in vendor supplied libraries such as Intel's[®] Math Kernel Library (MKL) or Sun's[™] High Performance Library. Functions in the vendor supplied libraries are finely tuned for performance to take full advantage of the environment for which they are supplied. For these functions, the user of the IMSL Fortran Numerical Library has the option of linking to code which is based on either the IMSL legacy functions or the functions in the vendor supplied library. The following icon in the function documentation alerts the reader when this is the case:



Details on linking to the appropriate IMSL Library and alternate vendor supplied libraries are explained in the online README file of the product distribution.

Getting Started

The IMSL MATH/LIBRARY is a collection of Fortran routines and functions useful in mathematical analysis research and application development. Each routine is designed and documented for use in research activities as well as by technical specialists.

To use any of these routines, you must write a program in Fortran 90 (or possibly some other language) to call the MATH/LIBRARY routine. Each routine conforms to established conventions in programming and documentation. We give first priority in development to efficient algorithms, clear documentation, and accurate results. The uniform design of the routines makes it easy to use more than one routine in a given application. Also, you will find that the design consistency enables you to apply your experience with one MATH/LIBRARY routine to other IMSL routines that you use.

Finding the Right Routine

The MATH/LIBRARY is organized into chapters; each chapter contains routines with similar computational or analytical capabilities. To locate the right routine for a given problem, you may use either the table of contents located in each chapter introduction, or the alphabetical list of routines. The GAMS index uses GAMS classification (Boisvert, R.F., S.E. Howe, D.K. Kahaner, and J. L. Springmann 1990, *Guide to Available Mathematical Software*, National Institute of Standards and Technology NISTIR 90-4237). Use the GAMS index to locate which MATH/LIBRARY routines pertain to a particular topic or problem.

Often the quickest way to use the MATH/LIBRARY is to find an example similar to your problem and then to mimic the example. Each routine document has at least one example demonstrating its application. The example for a routine may be created simply for illustration, it may be from a textbook (with reference to the source), or it may be from the mathematical literature.

Organization of the Documentation

This manual contains a concise description of each routine, with at least one demonstrated example of each routine, including sample input and results. You will find all information pertaining to the MATH/LIBRARY in this manual. Moreover, all information pertaining to a particular routine is in one place within a chapter.

Each chapter begins with an introduction followed by a table of contents that lists the routines included in the chapter. Documentation of the routines consists of the following information:

- IMSL Routine's Generic Name
- Purpose: a statement of the purpose of the routine. If the routine is a function rather than a subroutine the purpose statement will reflect this fact.
- Function Return Value: a description of the return value (for functions only).
- Required Arguments: a description of the required arguments in the order of their occurrence. Input arguments usually occur first, followed by input/output arguments, with output arguments described last. Furthermore, the following terms apply to arguments:

Input Argument must be initialized; it is not changed by the routine.

Input/Output Argument must be initialized; the routine returns output through this argument; cannot be a constant or an expression.

Input[/Output] Argument must be initialized; the routine may return output through this argument based on other optional data the user may choose to pass to this routine; cannot be a constant or an expression.

Input or Output Select appropriate option to define the argument as either input or output. See individual routines for further instructions.

Output No initialization is necessary; cannot be a constant or an expression. The routine returns output through this argument.

- Optional Arguments: a description of the optional arguments in the order of their occurrence.
- Fortran 90 Interface: a section that describes the generic and specific interfaces to the routine.
- Fortran 77 Style Interface: an optional section, which describes Fortran 77 style interfaces, is supplied for backwards compatibility with previous versions of the Library.
- ScaLAPACK Interface: an optional section, which describes an interface to a ScaLAPACK based version of this routine.
- Description: a description of the algorithm and references to detailed information. In many cases, other IMSL routines with similar or complementary functions are noted.
- Comments: details pertaining to code usage.
- Programming notes: an optional section that contains programming details not covered elsewhere.
- Example: at least one application of this routine showing input and required dimension and type statements.
- Output: results from the example(s). **Note** that unique solutions may differ from platform to platform.
- Additional Examples: an optional section with additional applications of this routine showing input and required dimension and type statements.

Naming Conventions

The names of the routines are mnemonic and unique. Most routines are available in both a single precision and a double precision version, with names of the two versions sharing a common root. The root name is also the generic interface name. The name of the double precision specific version begins with a “D_” and the single precision specific version begins with an “S_”. For example, the following pairs are precision specific names of routines in the two different precisions: S_GQRUL/D_GQRUL (the root is “GQRUL,” for “Gauss quadrature rule”) and S_RECCF/D_RECCF (the root is “RECCF,” for “recurrence coefficient”). The precision specific names of the IMSL routines that return or accept the type complex data begin with the letter “C_” or “Z_” for complex or double complex, respectively. Of course, the generic name can be used as an entry point for all precisions supported.

When this convention is not followed the generic and specific interfaces are noted in the documentation. For example, in the case of the BLAS and trigonometric intrinsic functions where standard names are already established, the standard names are used as the precision specific names. There may also be other interfaces supplied to the routine to provide for backwards compatibility with previous versions of the IMSL Fortran Numerical Library. These alternate interfaces are noted in the documentation when they are available.

Except when expressly stated otherwise, the names of the variables in the argument lists follow the Fortran default type for integer and floating point. In other words, a variable whose name begins with one of the letters “I” through “N” is of type `INTEGER`, and otherwise is of type `REAL` or `DOUBLE PRECISION`, depending on the precision of the routine.

An assumed-size array with more than one dimension that is used as a Fortran argument can have an assumed-size declarator for the last dimension only. In the MATH/LIBRARY routines, the information about the first dimension is passed by a variable with the prefix “LD” and with the array name as the root. For example, the argument `LDA` contains the leading dimension of array `A`. In most cases, information about the dimensions of arrays is obtained from the array through the use of Fortran 90’s *size* function. Therefore, arguments carrying this type of information are usually defined as optional arguments.

Where appropriate, the same variable name is used consistently throughout a chapter in the MATH/LIBRARY. For example, in the routines for random number generation, `NR` denotes the number of random numbers to be generated, and `R` or `IR` denotes the array that stores the numbers.

When writing programs accessing the MATH/LIBRARY, the user should choose Fortran names that do not conflict with names of IMSL subroutines, functions, or named common blocks. The careful user can avoid any conflicts with IMSL names if, in choosing names, the following rules are observed:

- Do not choose a name that appears in the Alphabetical Summary of Routines, at the end of the *User’s Manual*, nor one of these names preceded by a `D`, `S_`, `D_`, `C_`, or `Z_`.
- Do not choose a name consisting of more than three characters with a numeral in the second or third position.

For further details, see the section on “[Reserved Names](#)” in the Reference Material.

Using Library Subprograms

The documentation for the routines uses the generic name and omits the prefix, and hence the entire suite of routines for that subject is documented under the generic name.

Examples that appear in the documentation also use the generic name. To further illustrate this principle, note the `LIN_SOL_GEN` documentation (see [Chapter 1, Linear Systems](#)), for solving general systems of linear algebraic equations. A description is provided for just one data type. There are four documented routines in this subject area: `s_lin_sol_gen`, `d_lin_sol_gen`, `c_lin_sol_gen`, and `z_lin_sol_gen`.

These routines constitute single-precision, double-precision, complex, and double-complex precision versions of the code.

The Fortran 90 compiler identifies the appropriate routine. Use of a module is required with the routines. The naming convention for modules joins the suffix “_int” to the generic routine name. Thus, the line “`use lin_sol_gen_int`” is inserted near the top of any routine that calls the subprogram “`lin_sol_gen`”. More inclusive modules are also available, such as `imsl_libraries` and `numerical_libraries`. To avoid name conflicts, Fortran 90 permits re-labeling names defined in modules so they do not conflict with names of routines or variables in the user’s program. The user can also restrict access to names defined in IMSL Library modules by use of the “`: ONLY, <list of names>`” qualifier.

When dealing with a complex matrix, all references to the *transpose* of a matrix, A^T , are replaced by the *adjoint* matrix

$$\bar{A}^T \equiv A^* = A^H$$

where the overstrike denotes complex conjugation. IMSL Fortran Numerical Library linear algebra software uses this convention to conserve the utility of generic documentation for that code subject. All references to *orthogonal* matrices are to be replaced by their complex counterparts, *unitary* matrices. Thus, an $n \times n$ orthogonal matrix Q satisfies the condition $Q^T Q = I_n$. An $n \times n$ unitary matrix V satisfies the analogous condition for complex matrices, $V^* V = I_n$.

Programming Conventions

In general, the IMSL MATH/LIBRARY codes are written so that computations are not affected by underflow, provided the system (hardware or software) places a zero value in the register. In this case, system error messages indicating underflow should be ignored.

IMSL codes are also written to avoid overflow. A program that produces system error messages indicating overflow should be examined for programming errors such as incorrect input data, mismatch of argument types, or improper dimensioning.

In many cases, the documentation for a routine points out common pitfalls that can lead to failure of the algorithm.

Library routines detect error conditions, classify them as to severity, and treat them accordingly. This error-handling capability provides automatic protection for the user without requiring the user to make any specific provisions for the treatment of error conditions. See the section on “[User Errors](#)” in the Reference Material for further details.

Module Usage

Users are required to incorporate a “use” statement near the top of their program for the IMSL routine being called when writing new code that uses this library. However, legacy code which calls routines in the previous version of the library without the use of a “use” statement will continue to work as before. Also, code that employed the “use numerical_libraries” statement from the previous version of the library will continue to work properly with this version of the library.

Users wishing to update existing programs so as to call other routines from this library should incorporate a use statement for the specific new routine being called. (Here, the term “new routine” implies any routine in the library, only “new” to the user’s program.) Use of the more encompassing “imsl_libraries” module in this case could result in argument mismatches for the “old” routine(s) being called. (The compiler would catch this.)

Users wishing to update existing programs to call the new generic versions of the routines must change their calls to the existing routines to match the new calling sequences and use either the routine specific interface modules or the all-encompassing “imsl_libraries” module.

Using MPI Routines



Users of the IMSL Fortran Numerical Library benefit by having a standard (MPI) Message Passing Interface environment. This is needed to accomplish parallel computing within parts of the Library. *Either of the icons above clues the reader when this is the case.* If parallel computing is not required, then the IMSL Library suite of dummy MPI routines can be substituted for standard MPI routines. All requested MPI routines called by the IMSL Library are in this dummy suite. Warning messages will appear if a code or example requires more than one process to execute. Typically users need not be aware of the parallel codes.

NOTE: that a standard MPI environment is not part of the IMSL Fortran Numerical Library. The standard includes a library of MPI Fortran and C routines, MPI “include” files, usage documentation, and other run-time utilities.

NOTE: Details on linking to the appropriate libraries are explained in the online README file of the product distribution.

There are three situations of MPI usage in the IMSL Fortran Numerical Library:

1. There are some computations that are performed with the ‘box’ data type that benefit from the use of parallel processing. For computations involving a single array or a single problem, there is no IMSL use of parallel processing or MPI codes. The box type data type implies that several problems of the same size and type are to be computed and solved. Each rack of the box is an independent problem. This means that each problem could potentially be solved in parallel. The default for computing a box data type calculation is that a single processor will do all of the problems, one after the other. If this is acceptable there should be no further concern about which version of the libraries is used for linking. If the problems are to be solved in parallel, then the user must link with a working version of an MPI Library and the appropriate IMSL Library. Examples demonstrating the use of box type data may be found in Chapter 10, “[Linear Algebra Operators and Generic Functions](#)”.

NOTE: Box data type routines are marked with the MPI Capable icon.

2. Various routines in Chapter 1, “[Linear Systems](#)” allow the user to interface with the ScaLAPACK Library routines. If the user chooses to run on only one processor then these routines will utilize either IMSL Library code or LAPACK Library code based on the libraries the user chooses to use during linking. If the user chooses to run on multiple processors then working versions of MPI, ScaLAPACK, PBLAS, and Blacs will need to be present. These routines are marked with the MPI Capable icon.
3. There are some routines or operators in the Library that require that a working MPI Library be present in order for them to run. Examples are the large-scale parallel solvers and the ScaLAPACK utilities. Routines of this type are marked with the MPI Required icon. For these routines, the user must link with a working version of an MPI Library and the appropriate IMSL Library.

In all cases described above it is the user’s responsibility to supply working versions of the aforementioned third party libraries when those libraries are required.

Table A below lists the chapters and IMSL routines calling MPI routines or the replacement non-parallel package.

Chapter Name and Number	Routine with MPI Utilized
Linear Systems, 1	PARALLEL_NONNEGATIVE_LSQ
Linear Systems, 1	PARALLEL_BOUNDED_LSQ
Linear Systems, 1	Those routines which utilize ScaLAPACK listed in Table D below.
Linear Algebra and Generic Functions, 10	See entire following Table B.1 – Defined Operators and Generic Functions
Utilities, 11	ScaLAPACK_SETUP
Utilities, 11	ScaLAPACK_GETDIM
Utilities, 11	ScaLAPACK_READ
Utilities, 11	ScaLAPACK_WRITE
Utilities, 11	ScaLAPACK_MAP
Utilities, 11	ScaLAPACK_UNMAP
Utilities, 11	ScaLAPACK_EXIT
Reference Material	Entire Error Processor Package for IMSL Library, if MPI is utilized

Table A - IMSL Routines Calling MPI Routines or Replacement Non-Parallel Package

Programming Tips

Each subject routine called or otherwise referenced requires the “use” statement for an interface block designed for that subject routine. The contents of this interface block are the interfaces to the separate routines available for that subject. Packaged descriptive names for option numbers that modify documented optional data or internal parameters might also be provided in the interface block. Although this seems like an additional complication, many errors are avoided at an early stage in development through the use of these interface blocks. The “use” statement is required for each routine called in the user’s program. As illustrated in Examples 3 and 4 in routine `lin_geig_gen`, the “use” statement is required for defining the secondary option flags.

The function subprogram for `s_NaN()` or `d_NaN()` does not require an interface block because it has only a single “required” dummy argument. Also, if one is only using the Fortran 77 interfaces supplied for backwards compatibility then the “use” statements are not required.

Optional Subprogram Arguments

IMSL Fortran Numerical Library routines have *required* arguments and may have *optional* arguments. All arguments are documented for each routine. For example, consider the routine `lin_sol_gen` that solves the linear algebraic matrix equation $Ax = b$. The required arguments are three rank-2 Fortran 90 arrays: A , b , and x . The input data for the problem are the A and b arrays; the solution output is the x array. Often there are other arguments for this linear solver that are closely connected with the computation but are not as compelling as the primary problem. The inverse matrix A^{-1} may be needed as part of a larger application. To output this parameter, use the optional argument given by the “ainv=” keyword. The rank-2 output array argument used on the right-hand side of the equal sign contains the inverse matrix. See Example 2 in [Chapter 1, “Linear Systems”](#) of `LIN_SOL_GEN` for an example of computing the inverse matrix.

For compatibility with previous versions of the IMSL Libraries, the `NUMERICAL_LIBRARIES` interface module includes backwards-compatible positional argument interfaces to all routines that existed in the Fortran 77 version of the Library. Note that it is not necessary to include “use” statements when calling these routines by themselves. Existing programs that called these routines will continue to work in the same manner as before.

Some of the primary routines have arguments “epack=” and “iopt=”. As noted the “epack=” argument is of derived type `s_error` or `d_error`. The prefix “s_” or “d_” is chosen depending on the precision of the data type for that routine. These optional arguments are part of the interface to certain routines, and are used to modify internal algorithm choices or other parameters.

Optional Data

This additional optional argument (available for some routines) is further distinguished—a derived type array that contains a number of parameters to modify the internal algorithm of a routine. This derived type has the name `?_options`, where “?” is either “s_” or “d_”. The choice depends on the precision of the data type. The declaration of this derived type is packaged within the modules for these codes.

The definition of the derived types is:

```
type ?_options
integer idummy; real(kind(?)) rdummy
end type
```

where the “?” is either “s_” or “d_”, and the `kind` value matches the desired data type indicated by the choice of “s_” or “d_”.

Example 3 in [Chapter 1, “Linear Systems”](#) of `LIN_SOL_GEN` illustrates the use of iterative refinement to compute a double-precision solution based on a single-precision factorization of the matrix. This is communicated to the routine using an optional argument with optional data. For efficiency of iterative refinement, perform the factorization step once, and then save the factored matrix in the array A and the pivoting information in the rank-1 integer array, `ipivots`. By default, the factorization is normally discarded. To enable the routine to be re-entered with a previously computed factorization of the matrix, optional data are used as array entries in the “iopt=” optional argument. The packaging of `LIN_SOL_GEN` includes the definitions of the self-documenting integer parameters `lin_sol_gen_save_LU` and `lin_sol_gen_solve_A`. These

parameters have the values 2 and 3, but the programmer usually does not need to be aware of it. The following rules apply to the “`iopt=iopt`” optional argument:

1. Define a relative index, for example `IO`, for placing option numbers and data into the array argument `iopt`. Initially, set `IO = 1`. Before a call to the IMSL Library routine, follow Steps 2 through 4.
2. The data structure for the optional data array has the following form:

```
iopt (IO) = ?_options (Option_number, Optional_data)
[iopt (IO + 1) = ?_options (Option_number, Optional_data)]
```

The length of the data set is specified by the documentation for an individual routine. (The *Optional_data* is output in some cases and may not be used in other cases.) The square braces [. .] denote optional items.

Illustration: Example 3 in [Chapter 2, “Singular Value and Eigenvalue Decomposition”](#) of `LIN_EIG_SELF`, a new definition for a small diagonal term is passed to `LIN_SOL_SELF`. There is one line of code required for the change and the new tolerance:

```
iopt (1) = d_options(d_lin_sol_self_set_small,
epsilon(one) *abs (d(i)))
```

3. The internal processing of option numbers stops when `Option_number == 0` or when `IO > SIZE(iopt)`. This signals each routine having this optional argument that all desired changes to default values of internal parameters have been made. This implies that the last option number is the value zero or the value of `SIZE (iopt)` matches the last optional value changed.
4. To add more options, replace `IO` with `IO + n`, where `n` is the number of items required for the previous option. Go to Step 2.

Option numbers can be written in any order, and any selected set of options can be changed from the defaults. They may be repeated. Example 3 in [Chapter 1, “Linear Solvers”](#) of `LIN_SOL_SELF` uses three and then four option numbers for purposes of computing an eigenvector associated with a known eigenvalue.

Overloaded =, /=, etc., for Derived Types

To assist users in writing compact and readable code, the IMSL Fortran Numerical Library provides overloaded assignment and logical operations for the derived types `s_options`, `d_options`, `s_error`, and `d_error`. Each of these derived types has an individual record consisting of an integer and a floating-point number. The components of the derived types, in all cases, are named `idummy` followed by `rdummy`. In many cases, the item referenced is the component `idummy`. This integer value can be used exactly as any integer by use of the component selector character (%). Thus, a program could assign a value and test after calling a routine:

```

s_epack(1)%idummy = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1)%idummy > 0) call error_post(s_epack)

```

Using the overloaded assignment and logical operations, this code fragment can be written in the equivalent and more readable form:

```

s_epack(1) = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1) > 0) call error_post(s_epack)

```

Generally the assignments and logical operations refer only to component `idummy`. The assignment “`s_epack(1)=0`” is equivalent to “`s_epack(1)=s_error(0,0E0)`”. Thus, the floating-point component `rdummy` is assigned the value `0E0`. The assignment statement “`I=s_epack(1)`”, for `I` an integer type, is equivalent to “`I=s_epack(1)%idummy`”. The value of component `rdummy` is ignored in this assignment. For the logical operators, a single element of any of the IMSL Fortran Numerical Library derived types can be in either the first or second operand.

Derived Type	Overloaded Assignments and Tests						
<code>s_options</code>	<code>I=s_options(1);s_options(1)=I</code>	<code>= =</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
<code>s_options</code>	<code>I=d_options(1);d_options(1)=I</code>	<code>= =</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
<code>d_epack</code>	<code>I=s_epack(1);s_epack(1)=I</code>	<code>= =</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
<code>d_epack</code>	<code>I=d_epack(1);d_epack(1)=I</code>	<code>= =</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>

In the examples, `operator_ex01`, ..., `_ex37`, the overloaded assignments and tests have been used whenever they improve the readability of the code.

Error Handling



The routines in the IMSL MATH/LIBRARY attempt to detect and report errors and invalid input. Errors are classified and are assigned a code number. By default, errors of moderate or worse severity result in messages being automatically printed by the routine. Moreover, errors of worse severity cause program execution to stop. The severity level and the general nature of the error are designated by an “error type” ranging from 0 to 5. An error type 0 is no error; types 1 through 5 are progressively more severe. In most cases, you need not be concerned with our method of handling errors. For those interested, a complete description of the error-handling system is given in the [Reference Material](#), which also describes how you can change the default actions and access the error code numbers.

A separate error handler is provided to allow users to handle errors of differing types being reported from several nodes without danger of “jumbling” or mixing error messages. The design of this error handler is described more fully in Hanson (1992). The primary feature of the design is the use of a separate array for each parallel call to a routine. This allows the user to summarize errors using the routine `error_post` in a non-parallel part of an application. For a more detailed discussion of the use of this error handler in applications which use MPI for distributed computing, see the [Reference Material](#).

Printing Results

Most of the routines in the IMSL MATH/LIBRARY (except the line printer routines and special utility routines) do not print any of the results. The output is returned in Fortran variables, and you can print these yourself. See [Chapter 11, “Utilities,”](#) for detailed descriptions of these routines.

A commonly used routine in the examples is the IMSL routine `UMACH` (see the [Reference Material](#)), which retrieves the Fortran device unit number for printing the results. Because this routine obtains device unit numbers, it can be used to redirect the input or output. The section on “[Machine-Dependent Constants](#)” in the [Reference Material](#) contains a description of the routine `UMACH`.

Fortran 90 Constructs



The IMSL Fortran Numerical Library contains routines which take advantage of Fortran 90 language constructs, including Fortran 90 array data types. One feature of the design is that the default use may be as simple as the problem statement. Complicated, professional-quality mathematical software is hidden from the casual or beginning user.

In addition, high-level operators and functions are provided in the Library. They are described in Chapter 10, “[Linear Algebra Operators and Generic Functions](#)”.

Shared-Memory Multiprocessors and Thread Safety



The IMSL Fortran Numerical Library allows users to leverage the high-performance technology of shared memory parallelism (SMP) when their environment supports it. Support for SMP systems within the IMSL Library is delivered through various means, depending upon the availability of technologies such as OpenMP, high performance LAPACK and BLAS, and hardware-specific IMSL algorithms. Use of the IMSL Fortran Numerical Library on SMP systems can be achieved by using the appropriate link environment variable when building your application. Details on the available link environment variables for your installation of the IMSL Fortran Numerical Library can be found in the online README file of the product distribution.

The IMSL Fortran Numerical Library is thread-safe in those environments that support OpenMP 2.0. This was achieved by using OpenMP directives that define global variables located in the code so they are private to the individual threads. Thread safety allows users to create instances of routines running on multiple threads and to include any routine in the IMSL Fortran Numerical Library in these threads.

Using Operators and Generic Functions

For users who are primarily interested in easy-to-use software for numerical linear algebra, see [Chapter 10, “Linear Algebra Operators and Generic Functions”](#). This compact notation for writing Fortran 90 programs, when it applies, results in code that is easier to read and maintain than traditional subprogram usage.

Users may begin their code development using operators and generic functions. If a more efficient executable code is required, a user may need to switch to equivalent subroutine calls using IMSL Fortran Numerical Library routines.

[Table B](#) contain lists of the defined operators and some of their generic functions.

Defined Array Operation	Matrix Operation
A .x. B	AB
.i. A	A^{-1}
.t. A, .h. A	A^T, A^*
A .ix. B	$A^{-1}B$
B .xi. A	BA^{-1}
A .tx. B, or (.t. A) .x. B A .hx. B, or (.h. A) .x. B	$A^T B, A^* B$
B .xt. A, or B .x. (.t. A) B .xh. A, or B .x. (.h. A)	BA^T, BA^*
S=SVD(A [,U=U, V=V])	$A = USV^T$
E=EIG(A [[,B=B, D=D], V=V, W=W])	$(AV = VE), AVD = BVE, (AW = WE), AWD = BWE$
R=CHOL(A)	$A = R^T R$
Q=ORTH(A [,R=R])	$(A = QR), Q^T Q = I$
U=UNIT(A)	$u_1, \dots = a_1 / \ a_1\ , \dots$
F=DET(A)	$det(A) = \text{determinant}$
K=RANK(A)	$rank(A) = \text{rank}$

Defined Array Operation	Matrix Operation
P=NORM(A[, [type=] i])	$p = \ A\ _1 = \max_j \left(\sum_{i=1}^m a_{ij} \right)$ $p = \ A\ _2 = s_1 = \text{largest singular value}$ $p = \ A\ _{\infty \leftrightarrow \text{huge}(1)} = \max_i \left(\sum_{j=1}^n a_{ij} \right)$
C=COND(A)	$\ A^{-1}\ \cdot \ A\ $
Z=EYE(N)	$Z = I_N$
A=DIAG(X)	$A = \text{diag}(x_1, \dots)$
X=DIAGONALS(A)	$x = (x_{11}, \dots)$
W=FFT(Z) ; Z=IFFT(W)	Discrete Fourier Transform, Inverse
A=RAND(A)	random numbers, $0 < A < 1$
L=isNaN(A)	test for NaN, if (l) then...

Table B.1 – Defined Operators and Generic Functions for Dense Arrays

Defined Operation	Matrix Operation
Data Management	<i>Define entries of sparse matrices</i>
A .x. B	AB
.t. A, .h. A	A^T, A^*
A .ix. B	$A^{-1}B$
B .xi. A	BA^{-1}
A .tx. B, or (.t. A) .x. B A .hx. B, or (.h. A) .x. B	$A^T B, A^* B$
B .xt. A, or B .x. (.t. A) B .xh. A, or B .x. (.h. A)	BA^T, BA^*
A+B	<i>Sum of two sparse matrices</i>

Defined Operation	Matrix Operation
C=COND (A)	$\ A^{-1}\ \cdot \ A\ $

Table B.2 – Defined Operators and Generic Functions for Harwell-Boeing Sparse Matrices

Using ScaLAPACK, LAPACK, LINPACK, and EISPACK

Many of the codes in the IMSL Library are based on LINPACK, Dongarra et al. (1979), and EISPACK, Smith et al. (1976), collections of subroutines designed in the 1970s and early 1980s. LAPACK, Anderson et al. (1999), was designed to make the linear solvers and eigensystem routines run more efficiently on high performance computers. For a number of IMSL routines, the user of the IMSL Fortran Numerical Library has the option of linking to code which is based on either the legacy routines or the more efficient LAPACK routines.

Table C below lists the IMSL routines that make use of LAPACK codes. The intent is to obtain improved performance for IMSL codes by using LAPACK codes that have good performance by virtue of using BLAS with good performance. To obtain improved performance we recommend linking with High Performance versions of LAPACK and BLAS, if available. The LAPACK codes are listed where they are used. Details on linking to the appropriate IMSL Library and alternate libraries for LAPACK and BLAS are explained in the online README file of the product distribution.

Generic Name of IMSL Routine	LAPACK Routines used when Linking with High Performance Libraries
LSARG	?GERFS, ?GETRF, ?GECON, ?=S/D
LSLRG	?GETRF, ?GETRS, ?=S/D
LFCRG	?GETRF, ?GECON, ?=S/D
LFTRG	?GETRF, ?=S/D
LFSRG	?GETRS, ?=S/D
LFIRG	?GETRS, ?=S/D
LINRG	?GETRF, ?GETRI, ?=S/D
LSACG	?GETRF, ?GETRS, ?GECON, ?=C/Z
LSLCG	?GETRF, ?GETRS, ?=C/Z
LFCCG	?GETRF, ?GECON, ?=C/Z
LFTCG	?GETRF, ?C/Z
LFSCG	?GETRS, ?C/Z
LFICG	?GERFS, ?GETRS, ?=C/Z

Generic Name of IMSL Routine	LAPACK Routines used when Linking with High Performance Libraries
LINCG	?GETRF, ?GETRI, ?=C/Z
LSLRT	?TRTRS, ?=S/D
LFCRT	?TRCON, ?=S/D
LSLCT	?TRTRS, ?=C/Z
LFCCT	?TRCON, ?=C/Z
LSADS	?PORFS, ?POTRS, ?=S/D
LSLDS	?POTRF, ?POTRS, ?=S/D
LFCDS	?POTRF, ?POCON, ?=S/D
LFTDS	?POTRF, ?=S/D
LFSDS	?POTRS, ?=S/D
LFIDS	?PORFS, ?POTRS, ?=S/D
LINDS	?POTRF, ?=S/D
LSASF	?SYRFS, ?SYTRF, ?SYTRS, ?=S/D
LSLSF	?SYTRF, ?SYTRS, ?=S/D
LFCSF	?SYTRF, ?SYCON, ?=S/D
LFTSF	?SYTRF, ?=S/D
LFSSF	?SYTRF, ?=S/D
LFISF	?SYRFS, ?=S/D
LSADH	?POCON, ?POTRF, ?POTRS, ?=C/Z
LSLDH	?TRTRS, ?POTRF, ?=C/Z
LFCDH	?POTRF, ?POCON, ?=C/Z
LFTDH	?POTRF, ?=C/Z
LFSDH	?TRTRS, ?=C/Z
LFIDH	?PORFS, ?POTRS, ?=C/Z
LSAHF	?HECON, ?HERFS, ?HETRF, ?HETRS, ?=C/Z
LSLHF	?HECON, ?HETRF, ?HETRS, ?=C/Z
LFCHF	?HETRF, ?HECON, ?=C/Z
LFTHF	?HETRF, ?=C/Z
LFSHF	?HETRS, ?=C/Z
LFIHF	?HERFS, ?HETRS, ?=C/Z
LSARB	?GBTRF, ?GBTRS, ?GBRFS ?=S/D
LSLRB	?GBTRF, ?GBTRS, ?=S/D
LFCRB	?GBTRF, ?GBCON, ?=S/D

Generic Name of IMSL Routine	LAPACK Routines used when Linking with High Performance Libraries
LFTRB	?GBTRF, ?=S/D
LFSRB	?GBTRS, ?=S/D
LFIRB	?GBTRS, ?GBRFS, ?=S/D
LSQRR	?GEQP3, ?GEQRF, ?ORMQR, ?TRTRS. ?=S/D
LQRRV	?GEQP3, ?GEQRF, ?ORMQR, ?=S/D
LSBRR	?GEQRF, ?=S/D
LQRRR	?GEQRF, ?=S/D
LSVRR	?GESVD, ?-S/D
LSVCR	?GESVD, ?=C/Z
LSGRR	?GESVD, ?=S/D
LQRSL	?TRTRS, ?ORMQR, ?=S/D
LQERR	?ORGQR, ?=S/D
EVLRG	?GEBAL, ?GEHRD, ?HSEQR ?=S/D
EVCRG	?GEEVX, ?=S/D
EVLCG	?HSEQR, ?GEBAL, ?GEHRD, ?=C/Z
EVCCG	?GEEV, ?=C/Z
EVLSE	?SYEV, ?=S/D
EVCSF	?SYEV, ?=S/D
EVLHF	?HEEV, ?=C/Z
EVCHF	?HEEV, ?=C/Z
GVLRG	?GEQRF, ?ORMQR, ?GGHRD, ?HGEQZ, ?=S/D
GVCRG	?GEQRF, ?ORMQR, ?GGHRD, ?HGEQZ, ?TGEVC, ?=S/D
GVLCG	?GEQRF, ?UMMQR, ?GGHRD, ?HGEQZ, ?=C/Z
GVCCG	?GEQRF, ?UMMQR, ?GGHRD, ?HGEQZ, ?TGEVC, ?=C/Z
GVLSP	?SYGV, ?=S/D
GVCSF	?SYGV, ?=S/D

Table C – IMSL Routines and LAPACK Routines Utilized Within

ScaLAPACK, Blackford et al. (1997), includes a subset of LAPACK codes redesigned for use on distributed memory MIMD parallel computers. A number of IMSL Library routines make use of a subset of the ScaLAPACK library.

Table D below lists the IMSL routines that make use of ScaLAPACK codes. The intent is to provide access to the ScaLAPACK codes through the familiar IMSL routine interface. The IMSL routines that utilize ScaLAPACK codes have a ScaLAPACK Interface documented in addition to the FORTRAN 90 Interface. Like the LAPACK codes, access to the ScaLAPACK codes is made

by linking to the appropriate library. Details on linking to the appropriate IMSL Library and alternate libraries for ScaLAPACK and BLAS are explained in the online README file of the product distribution.

Generic Name of IMSL Routine	ScaLAPACK Routines used when Linking with High Performance Libraries
LSARG	P?GERFS, P?GETRF, P?GETRS, ?=S/D
LSLRG	P?GETRF, P?GETRS, ?=S/D
LFCRG	P?GETRF, P?GECON, ?=S/D
LFTRG	P?GETRF, ?=S/D
LFSRG	P?GETRS, ?=S/D
LFIRG	P?GETRS, P?GERFS, ?=S/D
LINRG	P?GETRF, P?GETRI, ?=S/D
LSACG	P?GETRF, P?GETRS, P?GERFS, ?=C/Z
LSLCG	P?GETRF, P?GETRS, ?=C/Z
LFCCG	P?GETRF, P?GECON, ?=C/Z
LFTCG	P?GETRF, ?C/Z
LFSCG	P?GETRS, ?C/Z
LFICG	P?GERFS, P?GETRS, ?=C/Z
LINGC	P?GETRF, P?GETRI, ?=C/Z
LSLRT	P?TRTRS, ?=S/D
LCFRT	P?TRCON, ?=S/D
LSLCT	P?TRTRS, ?=C/Z
LCFCT	P?TRCON, ?=C/Z
LSADS	P?PORFS, P?POTRF, P?POTRS, ?=S/D
LSLDS	P?POTRF, P?POTRS, ?=S/D
LCFDS	P?POTRF, P?POCON, ?=S/D
LFTDS	P?POTRF, ?-S/D
LFSDS	P?POTRS, ?-S/D
LFIDS	P?PORFS, P?POTRS, ?=S/D
LINDS	P?GETRF, P?GETRI, ?=S/D
LSADH	P?POTRF, P?PORFS, P?POTRS, ?=C/Z
LSLDH	P?POTRS, P?POTRF, ?=C/Z
LCFDH	P?POTRF, P?POCON, ?=C/Z
LFTDH	P?POTRF, ?=C/Z
LFSDH	P?POTRS, ?=C/Z

Generic Name of IMSL Routine	ScaLAPACK Routines used when Linking with High Performance Libraries
LFIDH	P?PORFS, P?POTRS, ?=C/Z
LSLRB	P?GBTRF, P?GBTRS, ?=S/D
LSQRR	P?GEQPF, P?GEQRF, P?ORMQR, P?TRTRS, ?=S/D
LQRRV	P?TRTRS, P?GEQRF, P?ORMQR, ?=S/D
LQRRR	P?GEQRF, P?GEQPF, P?ORMQR, ?=S/D
LSVRR	P?GESVD, ?=S/D
LSGRR	P?GESVD, ?=S/D
LQRSL	P?TRTRS, P?ORMQR, ?=S/D
LQERR	P?ORGQR, ?=S/D

Table D – IMSL Routines and ScaLAPACK Routines Utilized Within

Using ScaLAPACK Enhanced Routines



General Remarks

Use of the ScaLAPACK enhanced routines allows a user to solve large linear systems of algebraic equations at a performance level that might not be achievable on one computer by performing the work in parallel across multiple computers. One might also use these routines on linear systems that prove to be too large for the address space of the target computer. Visual Numerics has tried to facilitate the use of parallel computing in these situations by providing interfaces to ScaLAPACK routines which accomplish the task. The IMSL Library solver interface has the same look and feel whether one is using the routine on a single computer or across multiple computers.

The basic steps required to utilize the IMSL routines which interface with ScaLAPACK routines are:

1. Initialize MPI
2. Initialize the processor grid
3. Define any necessary array descriptors
4. Allocate space for the local arrays
5. Set up local matrices across the processor grid
6. Call the IMSL routine which interfaces with ScaLAPACK
7. Gather the results from across the processor grid
8. Release the processor grid
9. Exit MPI

Utilities are provided in the IMSL Library that facilitate these steps for the user. Each of these utilities is documented in [Chapter 11, “Utilities”](#). We visit the steps briefly here:

1. Initialize MPI

The user should call `MP_SETUP()` at this step. This function is described in detail in [“Getting Started with Modules `MPI_setup_int` and `MPI_node_int`”](#) in Chapter 10, Linear Algebra Operators and Generic Functions of this manual. For ScaLAPACK usage, suffice it to say that following a call to the function `MP_SETUP()`, the module `MPI_node_int` will contain information about the number of processors, the rank of a processor, and the communicator for the application. A call to this function will return the number of processors available to the program. Since the module `MPI_node_int` is used by `MPI_setup_int`, it is not necessary to explicitly use the module `MPI_node_int`. If `MP_SETUP()` is not called, then the program will compute entirely on one node. No routine from MPI will be called.

2. Initialize the processor grid

`SCALAPACK_SETUP` (see [Chapter 11, “Utilities”](#)) is called at this step. This call will set up the processor grid for the user, define the context ID variable, `MP_ICTXT`, for the processor grid, and place `MP_ICTXT` into the module `GRIDINFO_INT`. Use of `SCALAPACK_SUPPORT` will make the information in `MPI_NODE_INT` and `GRIDINFO_INT` available to the user’s program.

3. Define any necessary array descriptors

Consider the generic matrix *A* which is to be carved up and distributed across the processors in the processor grid. In ScaLAPACK parlance, we refer to *A* as being the “*global*” array *A* which is to be distributed across the processor grid in 2D block cyclic fashion ([Chapter 11, “Utilities”](#)). Each processor in the grid will then have access to a subset of the global array *A*. We refer to the subset array to which the individual processor has access as the “*local*” array *A₀*. Just as it is sometimes necessary for a program to be aware of the leading dimension of the global array *A*, it is also necessary for the program to be aware of other critical information about the local array *A₀*. This information can be obtained by calling the IMSL utility `SCALAPACK_GETDIM` ([Chapter 11, “Utilities”](#)). ScaLAPACK Library utility `DESCINIT` (see the Usage Notes section of [Chapter 11, “Utilities”](#)) is then used to store this information in a vector.

4. Allocate space for the local arrays

The array dimensions, obtained in the previous step, are used at this point to allocate space for any local arrays that will be used in the call to the IMSL routine.

5. Set up local matrices across the processor grid

If the matrices to be used by the solvers have not been distributed across the processor grid, IMSL provides utility routines `SCALAPACK_READ` and `SCALAPACK_MAP` to help in the distribution of global arrays across processors. `SCALAPACK_READ` will read data from a file while `SCALAPACK_MAP` will map a global array to the processor grid. Users may choose to distribute the arrays themselves as long as they distribute the arrays in 2D block cyclic fashion consistent with the array descriptors that have been defined.

6. Call the IMSL routine which interfaces with ScaLAPACK

The IMSL routines which interface with ScaLAPACK are listed in [Table D](#).

7. Gather the results from across the processor grid

IMSL provides utility routines `SCALAPACK_WRITE` and `SCALAPACK_UNMAP` to help in the gathering of results from across processors to a global array or file. `SCALAPACK_WRITE` will write data to a file while `SCALAPACK_UNMAP` will map local arrays from the processor grid to a global array.

8. Release the processor grid

This is accomplished by a call to `SCALAPACK_EXIT`.

9. Exit MPI

A call to `MP_SETUP` with the argument 'FINAL' will shut down MPI and set the value of `MP_NPROCS = 0`. This flags that MPI has been initialized and terminated. It cannot be initialized again in the same program unit execution. No MPI routine is defined when `MP_NPROCS` has this value.

Chapter 1: Linear Systems

Routines

1.1	Linear Solvers		
1.1.1	Solves a general system of linear equations $Ax = b$	LIN_SOL_GEN	10
1.1.2	Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix	LIN_SOL_SELF	18
1.1.3	Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense.....	LIN_SOL_LSQ	27
1.1.4	Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition.....	LIN_SOL_SVD	36
1.1.5	Solves multiple systems of linear equations.....	LIN_SOL_TRI	45
1.1.6	Computes the singular value decomposition (SVD) of a rectangular matrix, A	LIN_SVD	57
1.2.	Large-Scale Parallel Solvers		
1.2.1	Parallel Constrained Least-Squares Solvers		66
1.2.2	Solves a linear, non-negative constrained least-squares system.....	PARALLEL_NONNEGATIVE_LSQ	67
1.2.3	Solves a linear least-squares system with bounds on the unknowns.....	PARALLEL_BOUNDED_LSQ	75
1.3.	Solution of Linear Systems, Matrix Inversion, and Q Determinant Evaluation		
1.3.1	Real General Matrices		
	High accuracy linear system solution	LSARG	83
	Solves a linear system	LSLRG	87
	Factors and computes condition number	LFCRG	93
	Factors	LFTRG	99
	Solves after factoring	LFSRG	103
	High accuracy linear system solution after factoring	LFIRG	108
	Computes determinant after factoring	LFDRG	113

	Inverts.....	LINRG	115
1.3.2	Complex General Matrices		
	High accuracy linear system solution.....	LSACG	119
	Solves a linear system.....	LSLCG	123
	Factors and computes condition number.....	LFCCG	128
	Factors.....	LFTCG	134
	Solves a linear system after factoring.....	LFSCG	138
	High accuracy linear system solution after factoring.....	LFICG	143
	Computes determinant after factoring.....	LFDCG	148
	Inverts.....	LINCG	150
1.3.3	Real Triangular Matrices		
	Solves a linear system.....	LSLRT	154
	Computes condition number.....	LCFRT	158
	Computes determinant after factoring.....	LCFRT	162
	Inverts.....	LINRT	163
1.3.4	Complex Triangular Matrices		
	Solves a linear system.....	LSLCT	165
	Computes condition number.....	LCFCT	169
	Computes determinant after factoring.....	LCFCT	173
	Inverts.....	LINCT	175
1.3.5	Real Positive Definite Matrices		
	High accuracy linear system solution.....	LSADS	177
	Solves a linear system.....	LSLDS	181
	Factors and computes condition number.....	LCFDS	186
	Factors.....	LFTDS	191
	Solve a linear system after factoring.....	LFSDS	195
	High accuracy linear system solution after factoring.....	LFIDS	199
	Computes determinant after factoring.....	LFDDS	204
	Inverts.....	LINDS	206
1.3.6	Real Symmetric Matrices		
	High accuracy linear system solution.....	LSASF	210
	Solves a linear system.....	LSLSF	213
	Factors and computes condition number.....	LCFSF	215
	Factors.....	LFTSF	218
	Solves a linear system after factoring.....	LFSSF	221
	High accuracy linear system solution after factoring.....	LFISF	223
	Computes determinant after factoring.....	LFDSF	226
1.3.7	Complex Hermitian Positive Definite Matrices		
	High accuracy linear system solution.....	LSADH	227
	Solves a linear system.....	LSLDH	232
	Factors and computes condition number.....	LCFDH	237
	Factors.....	LFTDH	243
	Solves a linear system after factoring.....	LFSDH	248
	High accuracy linear system solution after factoring.....	LFIDH	252
	Computes determinant after factoring.....	LFDDH	258
1.3.8	Complex Hermitian Matrices		
	High accuracy linear system solution.....	LSAHF	259

	Solves a linear system	LSLHF	262
	Factors and computes condition number	LFCHF	265
	Factors	LFTHF	268
	Solves a linear system after factoring	LFSHF	271
	High accuracy linear system solution after factoring	LFIHF	273
	Computes determinant after factoring	LFDFH	276
1.3.9	Real Band Matrices in Band Storage		
	Solves a tridiagonal system	LSLTR	278
	Solves a tridiagonal system: Cyclic Reduction	LSLCR	279
	High accuracy linear system solution	LSARB	282
	Solves a linear system	LSLRB	285
	Factors and compute condition number	LFCRB	290
	Factors	LFTRB	293
	Solves a linear system after factoring	LFSRB	296
	High accuracy linear system solution after factoring	LFIRB	298
	Computes determinant after factoring	LFDRB	301
1.3.10	Real Band Symmetric Positive Definite Matrices in Band Storage		
	High accuracy linear system solution	LSAQS	303
	Solves a linear system	LSLQS	305
	Solves a linear system	LSLPB	308
	Factors and computes condition number	LFCQS	311
	Factors	LFTQS	314
	Solves a linear system after factoring	LFSQS	316
	High accuracy linear system solution after factoring	LFIQS	318
	Computes determinant after factoring	LFDQS	320
1.3.11	Complex Band Matrices in Band Storage		
	Solves a tridiagonal system	LSLTQ	322
	Solves a tridiagonal system: Cyclic Reduction	LSLCQ	324
	High accuracy linear system solution	LSACB	327
	Solves a linear system	LSLCB	330
	Factors and computes condition number	LFCCB	333
	Factors	LFTCB	336
	Solves a linear system after factoring	LFSCB	339
	High accuracy linear system solution after factoring	LFICB	341
	Computes determinant after factoring	LFDCB	344
1.3.12	Complex Band Positive Definite Matrices in Band Storage		
	High accuracy linear system solution	LSAQH	346
	Solves a linear system	LSLQH	349
	Solves a linear system	LSLQB	352
	Factors and compute condition number	LFCQH	355
	Factors	LFTQH	358
	Solves a linear system after factoring	LFSQH	360
	High accuracy linear system solution after factoring	LFIQH	362
	Computes determinant after factoring	LFDQH	365
1.3.13	Real Sparse Linear Equation Solvers		
	Solves a sparse linear system	LSLXG	366
	Factors	LFTXG	372
	Solves a linear system after factoring	LFSXG	377
1.3.14	Complex Sparse Linear Equation Solvers		

	Solves a sparse linear system	LSLZG	380
	Factors.....	LFTZG	385
	Solves a linear system after factoring	LFSZG	391
1.3.15	Real Sparse Symmetric Positive Definite Linear Equation Solvers		
	Solves a sparse linear system	LSLXD	394
	Symbolic Factor.....	LSCXD	399
	Computes Factor.....	LNF XD	403
	Solves a linear system after factoring	LFSXD	408
1.3.16	Complex Sparse Hermitian Positive Definite Linear Equation Solvers		
	Solves a sparse linear system	LSLZD	412
	Computes Factor.....	LNFZD	416
	Solves a linear system after factoring	LFSZD	421
1.3.17	Real Toeplitz Matrices in Toeplitz Storage		
	Solves a linear system	LSLTO	424
1.3.18	Complex Toeplitz Matrices in Toeplitz Storage		
	Solves a linear system	LSLTC	426
1.3.19	Complex Circulant Matrices in Circulant Storage		
	Solves a linear system	LSLCC	428
1.3.20	Iterative Methods		
	Preconditioned conjugate gradient.....	PCGRC	431
	Jacobi conjugate gradient	JCGRC	437
	Generalized minimum residual.....	GMRES	440
	Partial Singular Value Decomposition.....	ARPACK_SVD	451
1.4.	Linear Least Squares and Matrix Factorization		
1.4.1	Least Squares, QR Decomposition and Generalized Inverse		
	Solves a Least-squares system	LSQRR	451
	Solves a Least-squares system	LQRRV	457
	High accuracy Least squares.....	LSBRR	463
	Linearly constrained Least squares	LCLSQ	467
	QR decomposition.....	LQRRR	471
	Accumulation of QR decomposition	LQERR	478
	QR decomposition Utilities	LQRSL	483
	QR factor update	LUPQR	489
1.4.2	Cholesky Factorization		
	Cholesky factoring for rank deficient matrices	LCHRG	494
	Cholesky factor update.....	LUPCH	496
	Cholesky factor down-date.....	LDNCH	499
1.4.3	Singular Value Decomposition (SVD)		
	Real singular value decomposition	LSVRR	503
	Complex singular value decomposition.....	LSVCR	510
	Generalized inverse	LSGRR	514

Usage Notes

Section 1.1 describes routines for solving systems of linear algebraic equations by direct matrix factorization methods, for computing only the matrix factorizations, and for computing linear least-squares solutions.

Section 1.2 describes routines for solving systems of parallel constrained least-squares.

Many of the routines described in sections 1.3 and 1.4 are for matrices with special properties or structure. Computer time and storage requirements for solving systems with coefficient matrices of these types can often be drastically reduced, using the appropriate routine, compared with using a routine for solving a general complex system.

The appropriate matrix property and corresponding routine can be located in the “Routines” section. Many of the linear equation solver routines in this chapter are derived from subroutines from LINPACK, Dongarra et al. (1979). Other routines have been developed by Visual Numerics, derived from draft versions of LAPACK subprograms, Bischof et al. (1988), or were obtained from alternate sources.

A system of linear equations is represented by $Ax = b$ where A is the $n \times n$ coefficient data matrix, b is the known right-hand-side n -vector, and x is the unknown or solution n -vector. Figure 1-1 summarizes the relationships among the subroutines. Routine names are in boxes and input/output data are in ovals. The suffix $**$ in the subroutine names depend on the matrix type. For example, to compute the determinant of A use $LFC**$ or $LFT**$ followed by $LFD**$.

The paths using $LSA**$ or $LFI**$ use iterative refinement for a more accurate solution. The path using $LSA**$ is the same as using $LFC**$ followed by $LFI**$. The path using $LSL**$ is the same as the path using $LFC**$ followed by $LFS**$. The matrix inversion routines $LIN**$ are available only for certain matrix types.

Matrix Types

The two letter codes for the form of coefficient matrix, indicated by $**$ in Figure 1-1, are as follows:

RG	Real general (square) matrix.
CG	Complex general (square) matrix.
TR or CR	Real tridiagonal matrix.
RB	Real band matrix.
TQ or CQ	Complex tridiagonal matrix.
CB	Complex band matrix.
SF	Real symmetric matrix stored in the upper half of a square matrix.
DS	Real symmetric positive definite matrix stored in the upper half of a square matrix.
DH	Complex Hermitian positive definite matrix stored in the upper half of a complex square matrix.
HF	Complex Hermitian matrix stored in the upper half of a complex square matrix.

QS or PB	Real symmetric positive definite band matrix.
QH or QB	Complex Hermitian positive definite band matrix.
XG	Real general sparse matrix.
ZG	Complex general sparse matrix.
XD	Real symmetric positive definite sparse matrix.
ZD	Complex Hermitian positive definite sparse matrix.

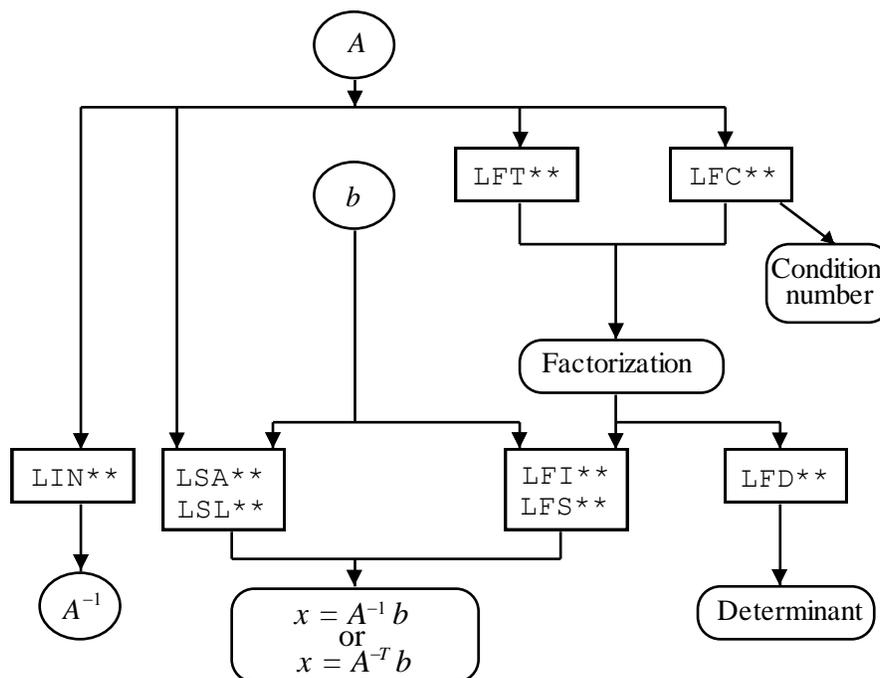


Figure 1-1 Solution and Factorization of Linear Systems

Solution of Linear Systems

The simplest routines to use for solving linear equations are `LSL**` and `LSA**`. For example, the mnemonic for matrices of real general form is `RG`. So, the routines `LSARG` and `LSLRG` are appropriate to use for solving linear systems when the coefficient matrix is of real general form. The routine `LSARG` uses iterative refinement, and more time than `LSLRG`, to determine a high accuracy solution.

The high accuracy solvers provide maximum protection against extraneous computational errors. They do not protect the results from instability in the mathematical approximation. For a more complete discussion of this and other important topics about solving linear equations, see Rice (1983), Stewart (1973), or Golub and van Loan (1989).

Multiple Right Sides

There are situations where the `LSL**` and `LSA**` routines are not appropriate. For example, if the linear system has more than one right-hand-side vector, it is most economical to solve the system by first calling a factoring routine and then calling a solver routine that uses the factors. After the coefficient matrix has been factored, the routine `LFS**` or `LFI**` can be used to solve for one right-hand side at a time. Routines `LFI**` uses iterative refinement to determine a high accuracy solution but requires more computer time and storage than routines `LFS**`.

Determinants

The routines for evaluating determinants are named `LFD**`. As indicated in Figure 1-1, these routines require the factors of the matrix as input. The values of determinants are often badly scaled. Additional complications in structures for evaluating them result from this fact. See Rice (1983) for comments on determinant evaluation.

Iterative Refinement

Iterative refinement can often improve the accuracy of a well-posed numerical solution. The iterative refinement algorithm used is as follows:

$$x_0 = A^{-1} b$$

For $i = 1, 50$

$$r_i = Ax_{i-1} - b \text{ computed in higher precision}$$

$$p_i = A^{-1} r_i$$

$$x_i = x_{i-1} - p_i$$

if $(\|p_i\|_\infty \leq \epsilon \|x_i\|_\infty)$ Exit

End for

Error — Matrix is too ill-conditioned

If the matrix A is in single precision, then the residual $r_i = Ax_{i-1} - b$ is computed in double precision. If A is in double precision, then quadruple-precision arithmetic routines are used.

The use of the value 50 is arbitrary. In fact a single correction is usually sufficient. It is also helpful even when r_i is computed in the same precision as the data.

Matrix Inversion

An inverse of the coefficient matrix can be computed directly by one of the routines named `LIN**`. These routines are provided for general matrix forms and some special matrix forms. When they do not exist, or when it is desirable to compute a high accuracy inverse, the two-step technique of calling the factoring routine followed by the solver routine can be used. The inverse

is the solution of the matrix system $AX = I$ where I denotes the $n \times n$ identity matrix, and the solution is $X = A^{-1}$

Singularity

The numerical and mathematical notions of singularity are not the same. A matrix is considered numerically singular if it is sufficiently close to a mathematically singular matrix. If error messages are issued regarding an exact singularity then specific error message level reset actions must be taken to handle the error condition. By default, the routines in this chapter stop. The solvers require that the coefficient matrix be numerically nonsingular. There are some tests to determine if this condition is met. When the matrix is factored, using routines `LF**`, the condition number is computed. If the condition number is large compared to the working precision, a warning message is issued and the computations are continued. In this case, the user needs to verify the usability of the output. If the matrix is determined to be mathematically singular, or ill-conditioned, a least-squares routine or the singular value decomposition routine may be used for further analysis.

Special Linear Systems

Toeplitz matrices have entries which are constant along each diagonal, for example:

$$A = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_{-1} & p_0 & p_1 & p_2 \\ p_{-2} & p_{-1} & p_0 & p_1 \\ p_{-3} & p_{-2} & p_{-1} & p_0 \end{bmatrix}$$

Real Toeplitz systems can be solved using `LSLTO`. Complex Toeplitz systems can be solved using `LSLTC`.

Circulant matrices have the property that each row is obtained by shifting the row above it one place to the right. Entries that are shifted off at the right reenter at the left. For example:

$$A = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_4 & p_1 & p_2 & p_3 \\ p_3 & p_4 & p_1 & p_2 \\ p_2 & p_3 & p_4 & p_1 \end{bmatrix}$$

Complex circulant systems can be solved using `LSLCC`.

Iterative Solution of Linear Systems

The preconditioned conjugate gradient routines `PCGRC` and `JCGRC` can be used to solve symmetric positive definite systems. The routines are particularly useful if the system is large and sparse. These routines use reverse communication, so A can be in any storage scheme. For general linear systems, use `GMRES`.

QR Decomposition

The QR decomposition of a matrix A consists of finding an orthogonal matrix Q , a permutation matrix P , and an upper trapezoidal matrix R with diagonal elements of nonincreasing magnitude, such that $AP = QR$. This decomposition is determined by the routines `LQRRR` or `LQRRV`. It returns R and the information needed to compute Q . To actually compute Q use `LQERR`. Figure 1-2 summarizes the relationships among the subroutines.

The QR decomposition can be used to solve the linear system $Ax = b$. This is equivalent to $Rx = Q^T P b$. The routine `LQRSL`, can be used to find $Q^T P b$ from the information computed by `LQRRR`. Then x can be computed by solving a triangular system using `LSLRT`. If the system $Ax = b$ is overdetermined, then this procedure solves the least-squares problem, i.e., it finds an x for which

$$\|Ax - b\|_2^2$$

is a minimum.

If the matrix A is changed by a rank-1 update, $A \rightarrow A + \alpha xy^T$, the QR decomposition of A can be updated/down-dated using the routine `LUPQR`. In some applications a series of linear systems which differ by rank-1 updates must be solved. Computing the QR decomposition once and then updating or down-dating it usually faster than newly solving each system.

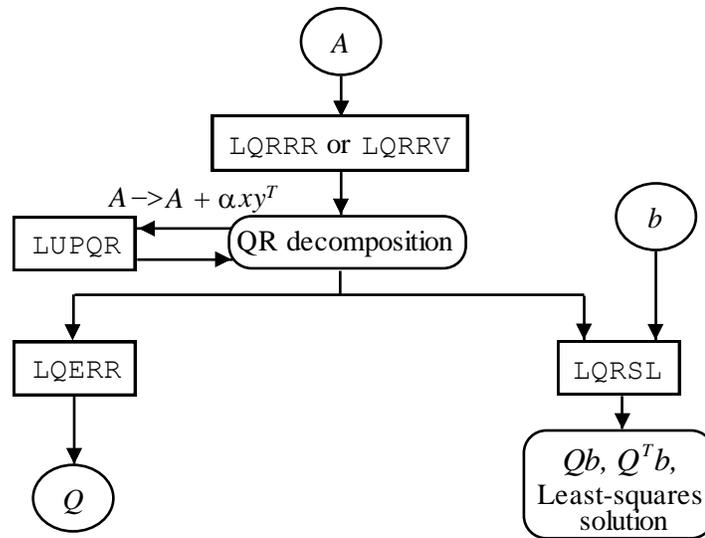


Figure 1-2 Least-Squares Routine

LIN_SOL_GEN



Solves a general system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the LU factorization of A using partial pivoting, representing the determinant of A , computing the inverse matrix A^{-1} , and solving $A^T x = b$ or $Ax = b$ given the LU factorization of A .

Required Arguments

- A** — Array of size $n \times n$ containing the matrix. (Input [/Output])
If the packaged option `lin_sol_gen_save_LU` is used then the LU factorization of A is saved in A . For solving efficiency, the diagonal reciprocals of the matrix U are saved in the diagonal entries of A .
- B** — Array of size $n \times nb$ containing the right-hand side matrix. (Input [/Output])
If the packaged option `lin_sol_gen_save_LU` is used then input B is used as work storage and is not saved.
- X** — Array of size $n \times nb$ containing the solution matrix.(Output)

Optional Arguments

- `NROWS = n` (Input)
Uses array `A(1:n, 1:n)` for the input matrix.
Default: `n = size(A, 1)`
- `NRHS = nb` (Input)
Uses array `b(1:n, 1:nb)` for the input right-hand side matrix.
Default: `nb = size(b, 2)`
Note that `b` must be a rank-2 array.
- `pivots = pivots(:)` (Output [/Input])
Integer array of size `n` that contains the individual row interchanges. To construct the permuted order so that no pivoting is required, define an integer array `ip(n)`. Initialize `ip(i) = i, i = 1, n` and then execute the loop, after calling `lin_sol_gen`,
- ```
k=pivots(i)
interchange ip(i) and ip(k), i=1,n
```
- The matrix defined by the array assignment that permutes the rows,  
`A(1:n, 1:n) = A(ip(1:n), 1:n)`, requires no pivoting for maintaining numerical

stability. Now, the optional argument “`iopt=`” and the packaged option number `?_lin_sol_gen_no_pivoting` can be safely used for increased efficiency during the *LU* factorization of *A*.

`det = det(1:2)` (Output)

Array of size 2 of the same type and kind as *A* for representing the determinant of the input matrix. The determinant is represented by two numbers. The first is the base with the sign or complex angle of the result. The second is the exponent. When `det(2)` is within exponent range, the value of this expression is given by `abs(det(1))**det(2) * (det(1))/abs(det(1))`. If the matrix is not singular, `abs(det(1)) = radix(det)`; otherwise, `det(1) = 0.`, and `det(2) = - huge(abs(det(1)))`.

`ainv = ainv(:, :)` (Output)

Array of the same type and kind as `A(1:n, 1:n)`. It contains the inverse matrix,  $A^{-1}$ , when the input matrix is nonsingular.

`iopt = iopt(:)` (Input)

Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>lin_sol_gen</code> |                                       |              |
|-----------------------------------------------|---------------------------------------|--------------|
| Option Prefix = ?                             | Option Name                           | Option Value |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_set_small</code>    | 1            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_save_LU</code>      | 2            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_solve_A</code>      | 3            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_solve_ADJ</code>    | 4            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_no_pivoting</code>  | 5            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_scan_for_NaN</code> | 6            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_no_sing_mess</code> | 7            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_gen_A_is_sparse</code>  | 8            |

`iopt(IO) = ?_options(?_lin_sol_gen_set_small, Small)`

Replaces a diagonal term of the matrix *U* if it is smaller in magnitude than the value *Small* using the same sign or complex direction as the diagonal. The system is declared singular. A solution is approximated based on this replacement if no overflow results. Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_sol_gen_save_LU, ?_dummy)`

Saves the *LU* factorization of *A*. Requires the optional argument “`pivots=`” if the routine will be used later for solving systems with the same matrix. This is the only case where the input arrays *A* and *b* are not saved. For solving efficiency, the diagonal reciprocals of the matrix *U* are saved in the diagonal entries of *A*.

`iopt(IO) = ?_options(?_lin_sol_gen_solve_A, ?_dummy)`  
 Uses the  $LU$  factorization of  $A$  computed and saved to solve  $Ax = b$ .

`iopt(IO) = ?_options(?_lin_sol_gen_solve_ADJ, ?_dummy)`  
 Uses the  $LU$  factorization of  $A$  computed and saved to solve  $A^T x = b$ .

`iopt(IO) = ?_options(?_lin_sol_gen_no_pivoting, ?_dummy)`  
 Does no row pivoting. The array `pivots (:)`, if present, are output as `pivots(i) = i`, for  $i = 1, \dots, n$ .

`iopt(IO) = ?_options(?_lin_sol_gen_scan_for_NaN, ?_dummy)`  
 Examines each input array entry to find the first value such that
   
`isNaN(a(i,j)) .or. isNaN(b(i,j)) ==.true.`
  
 See the `isNaN()` function, [Chapter 10](#).
   
 Default: Does not scan for NaNs.

`iopt(IO) = ?_options(?_lin_sol_gen_no_sing_mess, ?_dummy)`  
 Do not print an error message when the matrix  $A$  is singular.

`iopt(IO) = ?_options(?_lin_sol_gen_A_is_sparse, ?_dummy)`  
 Uses an indirect updating loop for the  $LU$  factorization that is efficient for sparse matrices where all matrix entries are stored.

## FORTRAN 90 Interface

Generic: `CALL LIN_SOL_GEN (A, B, X [, ...])`

Specific: The specific interface names are `S_LIN_SOL_GEN`, `D_LIN_SOL_GEN`, `C_LIN_SOL_GEN`, and `Z_LIN_SOL_GEN`.

## Description

Routine `LIN_SOL_GEN` solves a system of linear algebraic equations with a nonsingular coefficient matrix  $A$ . It first computes the  $LU$  factorization of  $A$  with partial pivoting such that  $LU = A$ . The matrix  $U$  is upper triangular, while the following is true:

$$L^{-1}A \equiv L_n P_n L_{n-1} P_{n-1} \cdots L_1 P_1 A \equiv U$$

The factors  $P_i$  and  $L_i$  are defined by the partial pivoting. Each  $P_i$  is an interchange of row  $i$  with row  $j \geq i$ . Thus,  $P_i$  is defined by that value of  $j$ . Every

$$L_i = I + m_i e_i^T$$

is an elementary elimination matrix. The vector  $m_i$  is zero in entries  $1, \dots, i$ . This vector is stored as column  $i$  in the strictly lower-triangular part of the working array containing the decomposition information. The reciprocals of the diagonals of the matrix  $U$  are saved in the diagonal of the working array. The solution of the linear system  $Ax = b$  is found by solving two simpler systems,

$$y = L^{-1}b \text{ and } x = U^{-1}y$$

More mathematical details are found in Golub and Van Loan (1989, Chapter 3).

## Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `LIN_SOL_GEN`. The messages are numbered 161–175; 181–195; 201–215; 221–235.

## Example 1: Solving a Linear System of Equations

This example solves a linear system of equations. This is the simplest use of `lin_sol_gen`. The equations are generated using a matrix of random numbers, and a solution is obtained corresponding to a random right-hand side matrix. Also, see `operator_ex01`, supplied with the product examples, for this example using the operator notation.

```

use lin_sol_gen_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 1 for LIN_SOL_GEN.

integer, parameter :: n=32
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) err
real(kind(1e0)) A(n,n), b(n,n), x(n,n), res(n,n), y(n**2)

! Generate a random matrix.
call rand_gen(y)
A = reshape(y, (/n,n/))

! Generate random right-hand sides.
call rand_gen(y)
b = reshape(y, (/n,n/))

! Compute the solution matrix of Ax=b.
call lin_sol_gen(A, b, x)

! Check the results for small residuals.
res = b - matmul(A,x)
err = maxval(abs(res))/sum(abs(A)+abs(b))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SOL_GEN is correct.'
end if

end

```

## Output

```
Example 1 for LIN_SOL_GEN is correct.
```

## Additional Examples

### Example 2: Matrix Inversion and Determinant

This example computes the inverse and determinant of  $A$ , a random matrix. Tests are made on the conditions

$$AA^{-1} = I$$

and

$$\det(A^{-1}) = \det(A)^{-1}$$

Also, see [operator\\_ex02](#).

```
use lin_sol_gen_int
use rand_gen_int

implicit none

! This is Example 2 for LIN_SOL_GEN.

integer i
integer, parameter :: n=32
real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
real(kind(1e0)) err
real(kind(1e0)) A(n,n), b(n,0), inv(n,n), x(n,0), res(n,n), &
 y(n**2), determinant(2), inv_determinant(2)

! Generate a random matrix.

call rand_gen(y)
A = reshape(y, (/n,n/))

! Compute the matrix inverse and its determinant.

call lin_sol_gen(A, b, x, nrhs=0, &
 ainv=inv, det=determinant)

! Compute the determinant for the inverse matrix.

call lin_sol_gen(inv, b, x, nrhs=0, &
 det=inv_determinant)

! Check residuals, A times inverse = Identity.

res = matmul(A,inv)
do i=1, n
 res(i,i) = res(i,i) - one
end do

err = sum(abs(res)) / sum(abs(a))
if (err <= sqrt(epsilon(one))) then
```

```

 if (determinant(1) == inv_determinant(1) .and. &
 (abs(determinant(2)+inv_determinant(2)) &
 <= abs(determinant(2))*sqrt(epsilon(one)))) then
 write (*,*) 'Example 2 for LIN_SOL_GEN is correct.'
 end if
 end if

end

```

## Output

Example 2 for LIN\_SOL\_GEN is correct.

### Example 3: Solving a System with Iterative Refinement

This example computes a factorization of a random matrix using single-precision arithmetic. The double-precision solution is corrected using iterative refinement. The corrections are added to the developing solution until they are no longer decreasing in size. The initialization of the derived type array `iopti(1:2) = s_option(0,0.0e0)` leaves the integer part of the second element of `iopti(:)` at the value zero. This stops the internal processing of options inside `lin_sol_gen`. It results in the *LU* factorization being saved after exit. The next time the routine is entered the integer entry of the second element of `iopt(:)` results in a solve step only. Since the *LU* factorization is saved in arrays `A(:, :)` and `ipivots(:)`, at the final step, solve only steps can occur in subsequent entries to `lin_sol_gen`. Also, see `operator_ex03`, [Chapter 10](#).

```

 use lin_sol_gen_int
 use rand_gen_int

 implicit none

! This is Example 3 for LIN_SOL_GEN.

 integer, parameter :: n=32
 real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
 real(kind(1d0)), parameter :: d_zero=0.0d0
 integer ipivots(n)
 real(kind(1e0)) a(n,n), b(n,1), x(n,1), w(n**2)
 real(kind(1e0)) change_new, change_old
 real(kind(1d0)) c(n,1), d(n,n), y(n,1)
 type(s_options) :: iopti(2)=s_options(0,zero)

! Generate a random matrix.

 call rand_gen(w)
 a = reshape(w, (/n,n/))

! Generate a random right hand side.

 call rand_gen(b(1:n,1))

! Save double precision copies of the matrix and right hand side.

 d = a
 c = b

```

```

! Start solution at zero.

 y = d_zero
 change_old = huge(one)

! Use packaged option to save the factorization.

 iopti(1) = s_options(s_lin_sol_gen_save_LU,zero)

 iterative_refinement: do
 b = c - matmul(d,y)
 call lin_sol_gen(a, b, x, &
 pivots=ipivots, iopt=iopti)
 y = x + y
 change_new = sum(abs(x))

! Exit when changes are no longer decreasing.

 if (change_new >= change_old) &
 exit iterative_refinement
 change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
 iopti(2) = s_options(s_lin_sol_gen_solve_A,zero)
 end do iterative_refinement
 write (*,*) 'Example 3 for LIN_SOL_GEN is correct.'
end

```

## Output

Example 3 for LIN\_SOL\_GEN is correct.

### Example 4: Evaluating the Matrix Exponential

This example computes the solution of the ordinary differential equation problem

$$\frac{dy}{dt} = Ay$$

with initial values  $y(0) = y_0$ . For this example, the matrix  $A$  is real and constant with respect to  $t$ . The unique solution is given by the matrix exponential:

$$y(t) = e^{At} y_0$$

This method of solution uses an eigenvalue-eigenvector decomposition of the matrix

$$A = XDX^{-1}$$

to evaluate the solution with the equivalent formula

$$y(t) = Xe^{Dt}z_0$$

where

$$z_0 = X^{-1}y_0$$

is computed using the complex arithmetic version of `lin_sol_gen`. The results for  $y(t)$  are real quantities, but the evaluation uses intermediate complex-valued calculations. Note that the computation of the complex matrix  $X$  and the diagonal matrix  $D$  is performed using the IMSL MATH/LIBRARY FORTRAN 77 interface to routine `EVCRG`. This is an illustration of intermixing interfaces of FORTRAN 77 and Fortran 90 code. The information is made available to the Fortran 90 compiler by using the FORTRAN 77 interface for `EVCRG`. Also, see `operator_ex04`, supplied with the product examples, where the Fortran 90 function `EIG()` has replaced the call to `EVCRG`.

```

use lin_sol_gen_int
use rand_gen_int
use Numerical_Libraries

implicit none

! This is Example 4 for LIN_SOL_GEN.

integer, parameter :: n=32, k=128
real(kind(1e0)), parameter :: one=1.0e0, t_max=1, delta_t=t_max/(k-1)
real(kind(1e0)) err, A(n,n), atemp(n,n), ytemp(n**2)
real(kind(1e0)) t(k), y(n,k), y_prime(n,k)
complex(kind(1e0)) EVAL(n), EVEC(n,n)
complex(kind(1e0)) x(n,n), z_0(n,1), y_0(n,1), d(n)
integer i

! Generate a random matrix in an F90 array.

call rand_gen(ytemp)
atemp = reshape(ytemp, (/n,n/))

! Assign data to an F77 array.
A = atemp

! Use IMSL Numerical Libraries F77 subroutine for the
! eigenvalue-eigenvector calculation.
CALL EVCRG(N, A, N, EVAL, EVEC, N)

! Generate a random initial value for the ODE system.
call rand_gen(ytemp(1:n))
y_0(1:n,1) = ytemp(1:n)

! Assign the eigenvalue-eigenvector data to F90 arrays.
d = EVAL; x = EVEC

! Solve complex data system that transforms the initial values, Xz_0=y_0.
call lin_sol_gen(x, y_0, z_0)
t = ((i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
y = matmul(x, exp(spread(d,2,k)*spread(t,1,n))* &
 spread(z_0(1:n,1),2,k))
y_prime = matmul(x, spread(d,2,k)* &
 exp(spread(d,2,k)*spread(t,1,n))* &
 spread(z_0(1:n,1),2,k))

```

```

! Check results. Is $y' - Ay = 0$?
 err = sum(abs(y_prime-matmul(atemp,y))) / &
 (sum(abs(atemp))*sum(abs(y)))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_SOL_GEN is correct.'
 end if

end

```

## Output

Example 4 for LIN\_SOL\_GEN is correct.

---

# LIN\_SOL\_SELF



Solves a system of linear equations  $Ax = b$ , where  $A$  is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using symmetric pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , or computing the solution of  $Ax = b$  given the factorization of  $A$ . An optional argument is provided indicating that  $A$  is positive definite so that the Cholesky decomposition can be used.

## Required Arguments

- A** — Array of size  $n \times n$  containing the self-adjoint matrix. (Input [/Output])  
If the packaged option `lin_sol_self_save_factors` is used then the factorization of  $A$  is saved in  $A$ . For solving efficiency, the diagonal reciprocals of the matrix  $R$  are saved in the diagonal entries of  $A$  when the Cholesky method is used.
- B** — Array of size  $n \times nb$  containing the right-hand side matrix. (Input [/Output])  
If the packaged option `lin_sol_self_save_factors` is used then input  $B$  is used as work storage and is not saved.
- X** — Array of size  $n \times nb$  containing the solution matrix. (Output)

## Optional Arguments

- NROWS** =  $n$  (Input)  
Uses array  $A(1:n, 1:n)$  for the input matrix.  
Default:  $n = \text{size}(A, 1)$

NRHS = nb (Input)

Uses the array  $b(1:n, 1:nb)$  for the input right-hand side matrix.

Default:  $nb = \text{size}(b, 2)$

Note that  $b$  must be a rank-2 array.

pivots = pivots(:) (Output [/Input])

Integer array of size  $n + 1$  that contains the individual row interchanges in the first  $n$  locations. Applied in order, these yield the permutation matrix  $P$ . Location  $n + 1$  contains the number of the first diagonal term no larger than *Small*, which is defined on the next page of this chapter.

det = det(1:2) (Output)

Array of size 2 of the same type and kind as  $A$  for representing the determinant of the input matrix. The determinant is represented by two numbers. The first is the base with the sign or complex angle of the result. The second is the exponent. When  $\text{det}(2)$  is within exponent range, the value of the determinant is given by the expression  $\text{abs}(\text{det}(1))^{*\text{det}(2)} * (\text{det}(1)/\text{abs}(\text{det}(1)))$ . If the matrix is not singular,  $\text{abs}(\text{det}(1)) = \text{radix}(\text{det})$ ; otherwise,  $\text{det}(1) = 0.$ , and  $\text{det}(2) = -\text{huge}(\text{abs}(\text{det}(1)))$ .

ainv = ainv(:, :) (Output)

Array of the same type and kind as  $A(1:n, 1:n)$ . It contains the inverse matrix,  $A^{-1}$  when the input matrix is nonsingular.

iopt = iopt(:) (Input)

Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>lin_sol_self</code> |                                        |              |
|------------------------------------------------|----------------------------------------|--------------|
| Option Prefix = ?                              | Option Name                            | Option Value |
| s_, d_, c_, z_                                 | <code>lin_sol_self_set_small</code>    | 1            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_save_factors</code> | 2            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_no_pivoting</code>  | 3            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_use_Cholesky</code> | 4            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_solve_A</code>      | 5            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_scan_for_NaN</code> | 6            |
| s_, d_, c_, z_                                 | <code>lin_sol_self_no_sing_mess</code> | 7            |

`iopt(IO) = ?_options(?_lin_sol_self_set_small, Small)`

When Aasen's method is used, the tridiagonal system  $Tu = v$  is solved using *LU* factorization with partial pivoting. If a diagonal term of the matrix  $U$  is smaller in magnitude than the value *Small*, it is replaced by *Small*. The system is declared singular. When the Cholesky method is used, the upper-triangular matrix  $R$ , (see "[Description](#)"), is obtained. If a diagonal term of the matrix  $R$  is smaller in magnitude than the value *Small*, it is replaced by *Small*. A solution is approximated based on this

replacement in either case.

Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_sol_self_save_factors, ?_dummy)`  
Saves the factorization of  $A$ . Requires the optional argument “pivots=” if the routine will be used for solving further systems with the same matrix. This is the only case where the input arrays  $A$  and  $b$  are not saved. For solving efficiency, the diagonal reciprocals of the matrix  $R$  are saved in the diagonal entries of  $A$  when the Cholesky method is used.

`iopt(IO) = ?_options(?_lin_sol_self_no_pivoting, ?_dummy)`  
Does no row pivoting. The array `pivots(:)`, if present, satisfies `pivots(i) = i + 1` for  $i = 1, \dots, n - 1$  when using Aasen’s method. When using the Cholesky method, `pivots(i) = i` for  $i = 1, \dots, n$ .

`iopt(IO) = ?_options(?_lin_sol_self_use_cholesky, ?_dummy)`  
The Cholesky decomposition  $PAP^T = R^T R$  is used instead of the Aasen method.

`iopt(IO) = ?_options(?_lin_sol_self_solve_A, ?_dummy)`  
Uses the factorization of  $A$  computed and saved to solve  $Ax = b$ .

`iopt(IO) = ?_options(?_lin_sol_self_scan_for_NaN, ?_dummy)`  
Examines each input array entry to find the first value such that

`isNaN(a(i,j)) .or. isNaN(b(i,j)) == .true.`

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs

`iopt(IO) = ?_options(?_lin_sol_self_no_sing_mess, ?_dummy)`  
Do not print an error message when the matrix  $A$  is singular.

## FORTRAN 90 Interface

Generic: `CALL LIN_SOL_SELF (A, B, X [, ...])`

Specific: The specific interface names are `S_LIN_SOL_SELF`, `D_LIN_SOL_SELF`, `C_LIN_SOL_SELF`, and `Z_LIN_SOL_SELF`.

## Description

Routine `LIN_SOL_SELF` routine solves a system of linear algebraic equations with a nonsingular coefficient matrix  $A$ . By default, the routine computes the factorization of  $A$  using Aasen’s method. This decomposition has the form

$$PAP^T = LTL^T$$

where  $P$  is a permutation matrix,  $L$  is a unit lower-triangular matrix, and  $T$  is a tridiagonal self-adjoint matrix. The solution of the linear system  $Ax = b$  is found by solving simpler systems,

$$u = L^{-1}Pb$$

$$Tv = u$$

and

$$x = P^T L^{-T} v$$

More mathematical details for real matrices are found in Golub and Van Loan (1989, Chapter 4).

When the optional Cholesky algorithm is used with a positive definite, self-adjoint matrix, the factorization has the alternate form

$$PAP^T = R^T R$$

where  $P$  is a permutation matrix and  $R$  is an upper-triangular matrix. The solution of the linear system  $Ax = b$  is computed by solving the systems

$$u = R^{-T} Pb$$

and

$$x = P^T R^{-1} u$$

The permutation is chosen so that the diagonal term is maximized at each step of the decomposition. The individual interchanges are optionally available in the argument “pivots”.

### Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `LIN_SOL_SELF`. These error messages are numbered 321–336; 341–356; 361–376; 381–396.

### Example 1: Solving a Linear Least-squares System

This example solves a linear least-squares system  $Cx \cong d$ , where  $C_{m \times n}$  is a real matrix with  $m \geq n$ . The least-squares solution is computed using the self-adjoint matrix

$$A = C^T C$$

and the right-hand side

$$b = A^T d$$

The  $n \times n$  self-adjoint system  $Ax = b$  is solved for  $x$ . This solution method is not as satisfactory, in terms of numerical accuracy, as solving the system  $Cx \cong d$  directly by using the routine `lin_sol_lsq`. Also, see `operator_ex05`, [Chapter 10](#).

```

use lin_sol_self_int
use rand_gen_int

implicit none

! This is Example 1 for LIN_SOL_SELF.

integer, parameter :: m=64, n=32
```

```

real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) err
real(kind(1e0)), dimension(n,n) :: A, b, x, res, y(m*n), &
 C(m,n), d(m,n)

! Generate two rectangular random matrices.
call rand_gen(y)
C = reshape(y, (/m,n/))

call rand_gen(y)
d = reshape(y, (/m,n/))

! Form the normal equations for the rectangular system.
A = matmul(transpose(C),C)
b = matmul(transpose(C),d)

! Compute the solution for Ax = b.
call lin_sol_self(A, b, x)

! Check the results for small residuals.
res = b - matmul(A,x)
err = maxval(abs(res))/sum(abs(A)+abs(b))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SOL_SELF is correct.'
end if

end

```

## Output

Example 1 for LIN\_SOL\_SELF is correct.

## Additional Examples

### Example 2: System Solving with Cholesky Method

This example solves the same form of the system as Example 1. The optional argument “`iopt=`” is used to note that the Cholesky algorithm is used since the matrix  $A$  is positive definite and self-adjoint. In addition, the sample covariance matrix

$$\Gamma = \sigma^2 A^{-1}$$

is computed, where

$$\sigma^2 = \frac{\|d - Cx\|^2}{m - n}$$

the inverse matrix is returned as the “`ainv=`” optional argument. The scale factor  $\sigma^2$  and  $\Gamma$  are computed after returning from the routine. Also, see `operator_ex06`, [Chapter 10](#).

```

use lin_sol_self_int
use rand_gen_int
use error_option_packet

```

```

implicit none

! This is Example 2 for LIN_SOL_SELF.

integer, parameter :: m=64, n=32
real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
real(kind(1e0)) err
real(kind(1e0)) a(n,n), b(n,1), c(m,n), d(m,1), cov(n,n), x(n,1), &
 res(n,1), y(m*n)
type(s_options) :: iopti(1)=s_options(0,zero)

! Generate a random rectangular matrix and a random right hand side.

call rand_gen(y)
c = reshape(y, (/m,n/))

call rand_gen(d(1:n,1))

! Form the normal equations for the rectangular system.

a = matmul(transpose(c),c)
b = matmul(transpose(c),d)

! Use packaged option to use Cholesky decomposition.

iopti(1) = s_options(s_lin_sol_self_Use_Cholesky,zero)

! Compute the solution of Ax=b with optional inverse obtained.

call lin_sol_self(a, b, x, ainv=cov, &
 iopt=iopti)

! Compute residuals, x - (inverse)*b, for consistency check.

res = x - matmul(cov,b)

! Scale the inverse to obtain the covariance matrix.

cov = (sum((d-matmul(c,x))**2)/(m-n)) * cov

! Check the results.

err = sum(abs(res))/sum(abs(cov))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_SOL_SELF is correct.'
end if

end

```

## Output

Example 2 for LIN\_SOL\_SELF is correct.

### Example 3: Using Inverse Iteration for an Eigenvector

This example illustrates the use of the optional argument “`i_opt=`” to reset the value of a *Small* diagonal term encountered during the factorization. Eigenvalues of the self-adjoint matrix

$$A = C^T C$$

are computed using the routine `lin_eig_self`. An eigenvector, corresponding to one of these eigenvalues,  $\lambda$ , is computed using inverse iteration. This solves the near singular system  $(A - \lambda I)x = b$  for an eigenvector,  $x$ . Following the computation of a normalized eigenvector

$$y = \frac{x}{\|x\|}$$

the consistency condition

$$\lambda = y^T A y$$

is checked. Since a singular system is expected, suppress the fatal error message that normally prints when the error post-processor routine `error_post` is called within the routine `lin_sol_self`. Also, see `operator_ex07`, [Chapter 10](#).

```
use lin_sol_self_int
use lin_eig_self_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 3 for LIN_SOL_SELF.

integer i, tries
integer, parameter :: m=8, n=4, k=2
integer ipivots(n+1)
real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
real(kind(1d0)) err
real(kind(1d0)) a(n,n), b(n,1), c(m,n), x(n,1), y(m*n), &
 e(n), atemp(n,n)
type(d_options) :: iopti(4)

! Generate a random rectangular matrix.

call rand_gen(y)
c = reshape(y, (/m,n/))

! Generate a random right hand side for use in the inverse
! iteration.

call rand_gen(y(1:n))
b = reshape(y, (/n,1/))

! Compute the positive definite matrix.
```

```

a = matmul(transpose(c),c)

! Obtain just the eigenvalues.

call lin_eig_self(a, e)

! Use packaged option to reset the value of a small diagonal.
iopti = d_options(0,zero)
iopti(1) = d_options(d_lin_sol_self_set_small,&
epsilon(one) * abs(e(1)))
! Use packaged option to save the factorization.
iopti(2) = d_options(d_lin_sol_self_save_factors,zero)
! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
iopti(3) = d_options(d_lin_sol_self_no_sing_mess,zero)
atemp = a
do i=1, n
a(i,i) = a(i,i) - e(k)
end do

! Compute A-eigenvalue*I as the coefficient matrix.
do tries=1, 2
call lin_sol_self(a, b, x, &
pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
iopti(4) = d_options(d_lin_sol_self_solve_A,zero)
! Reset right-hand side nearly in the direction of the eigenvector.
b = x/sqrt(sum(x**2))
end do

! Normalize the eigenvector.
x = x/sqrt(sum(x**2))

! Check the results.
err = dot_product(x(1:n,1),matmul(atemp(1:n,1:n),x(1:n,1))) - &
e(k)

! If any result is not accurate, quit with no summary printing.
if (abs(err) <= sqrt(epsilon(one))*e(1)) then
write (*,*) 'Example 3 for LIN_SOL_SELF is correct.'
end if

end

```

## Output

Example 3 for LIN\_SOL\_SELF is correct.

### Example 4: Accurate Least-squares Solution with Iterative Refinement

This example illustrates the accurate solution of the self-adjoint linear system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

computed using iterative refinement. This solution method is appropriate for least-squares problems when an accurate solution is required. The solution and residuals are accumulated in double precision, while the decomposition is computed in single precision. Also, see operator\_ex08, supplied with the product examples.

```

 use lin_sol_self_int
 use rand_gen_int

 implicit none

! This is Example 4 for LIN_SOL_SELF.

 integer i
 integer, parameter :: m=8, n=4
 real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
 real(kind(1d0)), parameter :: d_zero=0.0d0
 integer ipivots((n+m)+1)
 real(kind(1e0)) a(m,n), b(m,1), w(m*n), f(n+m,n+m), &
 g(n+m,1), h(n+m,1)
 real(kind(1e0)) change_new, change_old
 real(kind(1d0)) c(m,1), d(m,n), y(n+m,1)
 type(s_options) :: iopti(2)=s_options(0,zero)

! Generate a random matrix.

 call rand_gen(w)

 a = reshape(w, (/m,n/))

! Generate a random right hand side.

 call rand_gen(b(1:m,1))

! Save double precision copies of the matrix and right hand side.

 d = a
 c = b

! Fill in augmented system for accurately solving the least-squares
! problem.

 f = zero
 do i=1, m
 f(i,i) = one
 end do
 f(1:m,m+1:) = a
 f(m+1:,1:m) = transpose(a)

! Start solution at zero.

 y = d_zero
 change_old = huge(one)

! Use packaged option to save the factorization.

```

```

iopti(1) = s_options(s_lin_sol_self_save_factors,zero)

iterative_refinement: do
 g(1:m,1) = c(1:m,1) - y(1:m,1) - matmul(d,y(m+1:m+n,1))
 g(m+1:m+n,1) = - matmul(transpose(d),y(1:m,1))
 call lin_sol_self(f, g, h, &
 pivots=ipivots, iopt=iopti)
 y = h + y
 change_new = sum(abs(h))

! Exit when changes are no longer decreasing.

 if (change_new >= change_old) &
 exit iterative_refinement
 change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
 iopti(2) = s_options(s_lin_sol_self_solve_A,zero)
end do iterative_refinement
write (*,*) 'Example 4 for LIN_SOL_SELF is correct.'
end

```

## Output

Example 4 for LIN\_SOL\_SELF is correct.

---

## LIN\_SOL\_LSQ

Solves a rectangular system of linear equations  $Ax \cong b$ , in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using column and row pivoting, representing the determinant of  $A$ , computing the generalized inverse matrix  $A^\dagger$ , or computing the least-squares solution of

$$Ax \cong b$$

or

$$A^T y \cong b,$$

given the factorization of  $A$ . An optional argument is provided for computing the following unscaled covariance matrix

$$C = (A^T A)^{-1}$$

Least-squares solutions, where the unknowns are non-negative or have simple bounds, can be computed with [PARALLEL\\_NONNEGATIVE\\_LSQ](#) and [PARALLEL\\_BOUNDED\\_LSQ](#). These codes can be restricted to execute without MPI.

### Required Arguments

$A$  — Array of size  $m \times n$  containing the matrix. (Input [/Output])

If the packaged option `lin_sol_lsq_save_QR` is used then the factorization of  $A$  is

saved in `A`. For efficiency, the diagonal reciprocals of the matrix  $R$  are saved in the diagonal entries of `A`.

**B** — Array of size  $m \times nb$  containing the right-hand side matrix. When using the option to solve adjoint systems  $A^T x \cong b$ , the size of  $b$  is  $n \times nb$ . (Input [/Output])  
If the packaged option `lin_sol_lsq_save_QR` is used then input  $B$  is used as work storage and is not saved.

**X** — Array of size  $m \times nb$  containing the right-hand side matrix. When using the option to solve adjoint systems  $A^T x \cong b$ , the size of  $x$  is  $m \times nb$ . (Output)

## Optional Arguments

`MROWS` = `m` (Input)

Uses array `A(1:m, 1:n)` for the input matrix.

Default: `m = size(A, 1)`

`NCOLS` = `n` (Input)

Uses array `A(1:m, 1:n)` for the input matrix.

Default: `n = size(A, 2)`

`NRHS` = `nb` (Input)

Uses the array `b(1:, 1:nb)` for the input right-hand side matrix.

Default: `nb = size(b, 2)`

Note that `b` must be a rank-2 array.

`pivots` = `pivots(:)` (Output [/Input])

Integer array of size  $2 * \min(m, n) + 1$  that contains the individual row followed by the column interchanges. The last array entry contains the approximate rank of `A`.

`trans` = `trans(:)` (Output [/Input])

Array of size  $2 * \min(m, n)$  that contains data for the construction of the orthogonal decomposition.

`det` = `det(1:2)` (Output)

Array of size 2 of the same type and kind as `A` for representing the products of the determinants of the matrices  $Q$ ,  $P$ , and  $R$ . The determinant is represented by two numbers. The first is the base with the sign or complex angle of the result. The second is the exponent. When `det(2)` is within exponent range, the value of this expression is given by `abs(det(1))**det(2) * (det(1))/abs(det(1))`. If the matrix is not singular, `abs(det(1)) = radix(det)`; otherwise, `det(1) = 0.`, and `det(2) = -huge(abs(det(1)))`.

`ainv` = `ainv(:, :)` (Output)

Array with size  $n \times m$  of the same type and kind as `A(1:m, 1:n)`. It contains the generalized inverse matrix,  $A^\dagger$ .

`cov = cov(:, :)` (Output)

Array with size  $n \times n$  of the same type and kind as  $A(1:m, 1:n)$ . It contains the unscaled covariance matrix,  $C = (A^T A)^{-1}$ .

`iopt = iopt(:)` (Input)

Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>lin_sol_lsq</code> |                                          |              |
|-----------------------------------------------|------------------------------------------|--------------|
| Option Prefix = ?                             | Option Name                              | Option Value |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_set_small</code>       | 1            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_save_QR</code>         | 2            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_solve_A</code>         | 3            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_solve_ADJ</code>       | 4            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_no_row_pivoting</code> | 5            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_no_col_pivoting</code> | 6            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_scan_for_NaN</code>    | 7            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_lsq_no_sing_mess</code>    | 8            |

`iopt(IO) = ?_options(?_lin_sol_lsq_set_small, Small)`

Replaces with *Small* if a diagonal term of the matrix  $R$  is smaller in magnitude than the value *Small*. A solution is approximated based on this replacement in either case.

Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_sol_lsq_save_QR, ?_dummy)`

Saves the factorization of  $A$ . Requires the optional arguments “pivots=” and “trans=” if the routine is used for solving further systems with the same matrix. This is the only case where the input arrays  $A$  and  $b$  are not saved. For efficiency, the diagonal reciprocals of the matrix  $R$  are saved in the diagonal entries of  $A$ .

`iopt(IO) = ?_options(?_lin_sol_lsq_solve_A, ?_dummy)`

Uses the factorization of  $A$  computed and saved to solve  $Ax = b$ .

`iopt(IO) = ?_options(?_lin_sol_lsq_solve_ADJ, ?_dummy)`

Uses the factorization of  $A$  computed and saved to solve  $A^T x = b$ .

`iopt(IO) = ?_options(?_lin_sol_lsq_no_row_pivoting, ?_dummy)`

Does no row pivoting. The array `pivots(:)`, if present, satisfies `pivots(i) = i` for  $i = 1, \dots, \min(m, n)$ .

`iopt(IO) = ?_options(?_lin_sol_lsq_no_col_pivoting, ?_dummy)`

Does no column pivoting. The array `pivots(:)`, if present, satisfies `pivots(i + min(m, n)) = i` for  $i = 1, \dots, \min(m, n)$ .

`iopt(IO) = ?_options(?_lin_sol_lsq_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

```
isNaN(a(i,j)) .or. isNaN(b(i,j)) ==.true.
```

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs

```
iopt(IO) = ?_options(?_lin_sol_lsq_no_sing_mess,?_dummy)
```

Do not print an error message when  $A$  is singular or  $k < \min(m, n)$ .

## FORTRAN 90 Interface

Generic: `CALL LIN_SOL_LSQ (A, B, X [,...])`

Specific: The specific interface names are `S_LIN_SOL_LSQ`, `D_LIN_SOL_LSQ`, `C_LIN_SOL_LSQ`, and `Z_LIN_SOL_LSQ`.

## Description

Routine `LIN_SOL_LSQ` solves a rectangular system of linear algebraic equations in a least-squares sense. It computes the decomposition of  $A$  using an orthogonal factorization. This decomposition has the form

$$QAP = \begin{bmatrix} R_{k \times k} & 0 \\ 0 & 0 \end{bmatrix}$$

where the matrices  $Q$  and  $P$  are products of elementary orthogonal and permutation matrices. The matrix  $R$  is  $k \times k$ , where  $k$  is the approximate rank of  $A$ . This value is determined by the value of the parameter *Small*. See Golub and Van Loan (1989, Chapter 5.4) for further details. Note that the use of both row and column pivoting is nonstandard, but the routine defaults to this choice for enhanced reliability.

## Fatal and Terminal Error Messages

See the `messages.gls` file for error messages for `LIN_SOL_LSQ`. These error messages are numbered 241–256; 261–276; 281–296; 301–316.

## Example 1: Solving a Linear Least-squares System

This example solves a linear least-squares system  $Cx \cong d$ , where

$$C_{m \times n}$$

is a real matrix with  $m > n$ . The least-squares problem is derived from polynomial data fitting to the function

$$y(x) = e^x + \cos\left(\pi \frac{x}{2}\right)$$

using a discrete set of values in the interval  $-1 \leq x \leq 1$ . The polynomial is represented as the series

$$u(x) = \sum_{i=0}^N c_i T_i(x)$$

where the  $T_i(x)$  are Chebyshev polynomials. It is natural for the problem matrix and solution to have a column or entry corresponding to the subscript zero, which is used in this code. Also, see `operator_ex09`, supplied with the product examples.

```

use lin_sol_lsq_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 1 for LIN_SOL_LSQ.

integer i
integer, parameter :: m=128, n=8
real(kind(1d0)), parameter :: one=1d0, zero=0d0
real(kind(1d0)) A(m,0:n), c(0:n,1), pi_over_2, x(m), y(m,1), &
 u(m), v(m), w(m), delta_x

! Generate a random grid of points.
call rand_gen(x)

! Transform points to the interval -1,1.
x = x*2 - one

! Compute the constant 'PI/2'.
pi_over_2 = atan(one)*2

! Generate known function data on the grid.
y(1:m,1) = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
A(:,0) = one; A(:,1) = x

do i=2, n
 A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
end do

! Solve for the series coefficients.
call lin_sol_lsq(A, y, c)

! Generate an equally spaced grid on the interval.
delta_x = 2/real(m-1,kind(one))
do i=1, m
 x(i) = -one + (i-1)*delta_x
end do

! Evaluate residuals using backward recurrence formulas.
u = zero
v = zero
do i=n, 0, -1
 w = 2*x*u - v + c(i,1)
 v = u

```

```

 u = w
 end do

 y(1:m,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+1 sign changes in the residual curve occur.
 x = one
 x = sign(x,y(1:m,1))

 if (count(x(1:m-1) /= x(2:m)) >= n+1) then
 write (*,*) 'Example 1 for LIN_SOL_LSQ is correct.'
 end if

end

```

## Output

Example 1 for LIN\_SOL\_LSQ is correct.

## Additional Examples

### Example 2: System Solving with the Generalized Inverse

This example solves the same form of the system as Example 1. In this case, the grid of evaluation points is equally spaced. The coefficients are computed using the “smoothing formulas” by rows of the generalized inverse matrix,  $A^\dagger$ , computed using the optional argument “ainv?”. Thus, the coefficients are given by the matrix-vector product  $c = (A^\dagger)y$ , where  $y$  is the vector of values of the function  $y(x)$  evaluated at the grid of points. Also, see `operator_ex10`, supplied with the product examples.

```

 use lin_sol_lsq_int

 implicit none

! This is Example 2 for LIN_SOL_LSQ.

 integer i
 integer, parameter :: m=128, n=8
 real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
 real(kind(1d0)) a(m,0:n), c(0:n,1), pi_over_2, x(m), y(m,1), &
 u(m), v(m), w(m), delta_x, inv(0:n, m)

! Generate an array of equally spaced points on the interval -1,1.

 delta_x = 2/real(m-1,kind(one))
 do i=1, m
 x(i) = -one + (i-1)*delta_x
 end do

! Compute the constant 'PI/2'.

 pi_over_2 = atan(one)*2

! Compute data values on the grid.

```

```

 y(1:m,1) = exp(x) + cos(pi_over_2*x)
! Fill in the least-squares matrix for the Chebyshev polynomials.

 a(:,0) = one
 a(:,1) = x

 do i=2, n
 a(:,i) = 2*x*a(:,i-1) - a(:,i-2)
 end do

! Compute the generalized inverse of the least-squares matrix.

 call lin_sol_lsq(a, y, c, nrhs=0, ainvs=inv)

! Compute the series coefficients using the generalized inverse
! as 'smoothing formulas.'

 c(0:n,1) = matmul(inv(0:n,1:m),y(1:m,1))

! Evaluate residuals using backward recurrence formulas.

 u = zero
 v = zero
 do i=n, 0, -1
 w = 2*x*u - v + c(i,1)
 v = u
 u = w
 end do

 y(1:m,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+2 sign changes in the residual curve occur.
! (This test will fail when n is larger.)

 x = one
 x = sign(x,y(1:m,1))

 if (count(x(1:m-1) /= x(2:m)) == n+2) then
 write (*,*) 'Example 2 for LIN_SOL_LSQ is correct.'
 end if

 end

```

## Output

Example 2 for LIN\_SOL\_LSQ is correct.

### Example 3: Two-Dimensional Data Fitting

This example illustrates the use of radial-basis functions to least-squares fit arbitrarily spaced data points. Let  $m$  data values  $\{y_i\}$  be given at points in the unit square,  $\{p_i\}$ . Each  $p_i$  is a pair of real values. Then,  $n$  points  $\{q_j\}$  are chosen on the unit square. A series of *radial-basis functions* is used to represent the data,

$$f(p) = \sum_{j=1}^n c_j (\|p - q_j\|^2 + \delta^2)^{1/2}$$

where  $\delta^2$  is a parameter. This example uses  $\delta^2 = 1$ , but either larger or smaller values can give a better approximation for user problems. The coefficients  $\{c_j\}$  are obtained by solving the following  $m \times n$  linear least-squares problem:

$$f(p_j) = y_j$$

This example illustrates an effective use of Fortran 90 array operations to eliminate many details required to build the matrix and right-hand side for the  $\{c_j\}$ . For this example, the two sets of points  $\{p_i\}$  and  $\{q_j\}$  are chosen randomly. The values  $\{y_j\}$  are computed from the following formula:

$$y_j = e^{-\|p_j\|^2}$$

The residual function

$$r(p) = e^{-\|p\|^2} - f(p)$$

is computed at an  $N \times N$  square grid of equally spaced points on the unit square. The magnitude of  $r(p)$  may be larger at certain points on this grid than the residuals at the given points,  $\{p_i\}$ . Also, see `operator_ex11`, supplied with the product examples.

```

use lin_sol_lsq_int
use rand_gen_int

implicit none

! This is Example 3 for LIN_SOL_LSQ.

integer i, j
integer, parameter :: m=128, n=32, k=2, n_eval=16
real(kind(1d0)), parameter :: one=1.0d0, delta_sqr=1.0d0
real(kind(1d0)) a(m,n), b(m,1), c(n,1), p(k,m), q(k,n), &
 x(k*m), y(k*n), t(k,m,n), res(n_eval,n_eval), &
 w(n_eval), delta

! Generate a random set of data points in k=2 space.

call rand_gen(x)
p = reshape(x, (/k,m/))

! Generate a random set of center points in k-space.

call rand_gen(y)
q = reshape(y, (/k,n/))

! Compute the coefficient matrix for the least-squares system.
```

```

t = spread(p,3,n)
do j=1, n
 t(1:,:,j) = t(1:,:,j) - spread(q(1:,j),2,m)
end do

a = sqrt(sum(t**2,dim=1) + delta_sqr)

! Compute the right hand side of data values.

b(1:,1) = exp(-sum(p**2,dim=1))

! Compute the solution.

call lin_sol_lsqr(a, b, c)

! Check the results.

if (sum(abs(matmul(transpose(a),b-matmul(a,c)))/sum(abs(a)) &
 <= sqrt(epsilon(one))) then
 write (*,*) 'Example 3 for LIN_SOL_LSQR is correct.'
end if

! Evaluate residuals, known function - approximation at a square
! grid of points. (This evaluation is only for k=2.)

delta = one/real(n_eval-1,kind(one))
do i=1, n_eval
 w(i) = (i-1)*delta
end do
res = exp(-(spread(w,1,n_eval)**2 + spread(w,2,n_eval)**2))
do j=1, n
 res = res - c(j,1)*sqrt((spread(w,1,n_eval) - q(1,j))**2 + &
 (spread(w,2,n_eval) - q(2,j))**2 + delta_sqr)
end do

end

```

## Output

Example 3 for LIN\_SOL\_LSQR is correct.

### Example 4: Least-squares with an Equality Constraint

This example solves a least-squares system  $Ax \cong b$  with the constraint that the solution values have a sum equal to the value 1. To solve this system, one heavily weighted row vector and right-hand side component is added to the system corresponding to this constraint. Note that the weight used is

$$\varepsilon^{-1/2}$$

where  $\varepsilon$  is the machine precision, but any larger value can be used. The fact that `lin_sol_lsqr` performs row pivoting in this case is critical for obtaining an accurate solution to the constrained problem solved using weighting. See Golub and Van Loan (1989, Chapter 12) for more information about this method. Also, see `operator_ex12`, supplied with the product examples.

```

use lin_sol_lsq_int
use rand_gen_int

implicit none

! This is Example 4 for LIN_SOL_LSQ.

integer, parameter :: m=64, n=32
real(kind(1e0)), parameter :: one=1.0e0
real(kind(1e0)) :: a(m+1,n), b(m+1,1), x(n,1), y(m*n)

! Generate a random matrix.

call rand_gen(y)
a(1:m,1:n) = reshape(y, (/m,n/))

! Generate a random right hand side.

call rand_gen(b(1:m,1))

! Heavily weight desired constraint. All variables sum to one.

a(m+1,1:n) = one/sqrt(epsilon(one))
b(m+1,1) = one/sqrt(epsilon(one))

call lin_sol_lsq(a, b, x)

if (abs(sum(x) - one)/sum(abs(x)) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_SOL_LSQ is correct.'
end if

end

```

## Output

Example 4 for LIN\_SOL\_LSQ is correct.

---

## LIN\_SOL\_SVD

Solves a rectangular least-squares system of linear equations  $Ax \cong b$  using singular value decomposition

$$A = USV^T$$

With optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of  $A$ , the orthogonal  $m \times m$  and  $n \times n$  matrices  $U$  and  $V$ , and the  $m \times n$  diagonal matrix of singular values,  $S$ .

## Required Arguments

- A** — Array of size  $m \times n$  containing the matrix. (Input [/Output])  
If the packaged option `lin_sol_svd_overwrite_input` is used, this array is not saved on output.
- B** — Array of size  $m \times nb$  containing the right-hand side matrix. (Input [/Output])  
If the packaged option `lin_sol_svd_overwrite_input` is used, this array is not saved on output.
- X** — Array of size  $n \times nb$  containing the solution matrix. (Output)

## Optional Arguments

- MROWS** =  $m$  (Input)  
Uses array `A(1:m, 1:n)` for the input matrix.  
Default:  $m = \text{size}(A, 1)$
- NCOLS** =  $n$  (Input)  
Uses array `A(1:m, 1:n)` for the input matrix.  
Default:  $n = \text{size}(A, 2)$
- NRHS** =  $nb$  (Input)  
Uses the array `b(1:, 1:nb)` for the input right-hand side matrix.  
Default:  $nb = \text{size}(b, 2)$   
Note that `b` must be a rank-2 array.
- RANK** =  $k$  (Output)  
Number of singular values that are at least as large as the value *Small*. It will satisfy  $k \leq \min(m, n)$ .
- u** = `u(:, :)` (Output)  
Array of the same type and kind as `A(1:m, 1:n)`. It contains the  $m \times m$  orthogonal matrix *U* of the singular value decomposition.
- s** = `s(:)` (Output)  
Array of the same precision as `A(1:m, 1:n)`. This array is real even when the matrix data is complex. It contains the  $m \times n$  diagonal matrix *S* in a rank-1 array. The singular values are nonnegative and ordered non-increasing.
- v** = `v(:, :)` (Output)  
Array of the same type and kind as `A(1:m, 1:n)`. It contains the  $n \times n$  orthogonal matrix *V*.
- iopt** = `iopt(:)` (Input)  
Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>lin_sol_svd</code> |                                          |              |
|-----------------------------------------------|------------------------------------------|--------------|
| Option Prefix = ?                             | Option Name                              | Option Value |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_svd_set_small</code>       | 1            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_svd_overwrite_input</code> | 2            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_svd_safe_reciprocal</code> | 3            |
| <code>s_, d_, c_, z_</code>                   | <code>lin_sol_svd_scan_for_NaN</code>    | 4            |

`iopt(IO) = ?_options(?_lin_sol_svd_set_small, Small)`

Replaces with zero a diagonal term of the matrix  $S$  if it is smaller in magnitude than the value *Small*. This determines the approximate rank of the matrix, which is returned as the “rank=” optional argument. A solution is approximated based on this replacement.

Default: the smallest number that can be safely reciprocated

`iopt(IO) = ?_options(?_lin_sol_svd_overwrite_input, ?_dummy)`

Does not save the input arrays  $A(:, :)$  and  $b(:, :)$ .

`iopt(IO) = ?_options(?_lin_sol_svd_safe_reciprocal, safe)`

Replaces a denominator term with *safe* if it is smaller in magnitude than the value *safe*.

Default: the smallest number that can be safely reciprocated

`iopt(IO) = ?_options(?_lin_sol_svd_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(a(i,j)) .or. isNaN(b(i,j)) ==.true.`

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs

## FORTRAN 90 Interface

Generic: `CALL LIN_SOL_SVD (A, B, X [...])`

Specific: The specific interface names are `S_LIN_SOL_SVD`, `D_LIN_SOL_SVD`, `C_LIN_SOL_SVD`, and `Z_LIN_SOL_SVD`.

## Description

Routine `LIN_SOL_SVD` solves a rectangular system of linear algebraic equations in a least-squares sense. It computes the factorization of  $A$  known as the singular value decomposition. This decomposition has the following form:

$$A = USV^T$$

The matrices  $U$  and  $V$  are orthogonal. The matrix  $S$  is diagonal with the diagonal terms non-increasing. See Golub and Van Loan (1989, Chapters 5.4 and 5.5) for further details.

## Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `LIN_SOL_SVD`. These error messages are numbered 401–412; 421–432; 441–452; 461–472.

### Example 1: Least-squares solution of a Rectangular System

The least-squares solution of a rectangular  $m \times n$  system  $Ax \cong b$  is obtained. The use of `lin_sol_lsq` is more efficient in this case since the matrix is of full rank. This example anticipates a problem where the matrix  $A$  is poorly conditioned or not of full rank; thus, `lin_sol_svd` is the appropriate routine. Also, see `operator_ex13`, [Chapter 10](#).

```
use lin_sol_svd_int
use rand_gen_int

implicit none

! This is Example 1 for LIN_SOL_SVD.

integer, parameter :: m=128, n=32
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)) A(m,n), b(m,1), x(n,1), y(m*n), err

! Generate a random matrix and right-hand side.
call rand_gen(y)
A = reshape(y, (/m,n/))
call rand_gen(b(1:m,1))

! Compute the least-squares solution matrix of Ax=b.
call lin_sol_svd(A, b, x)

! Check that the residuals are orthogonal to the
! column vectors of A.
err = sum(abs(matmul(transpose(A), b-matmul(A, x))))/sum(abs(A))
if (err <= sqrt(epsilon(one))) then

 write (*,*) 'Example 1 for LIN_SOL_SVD is correct.'
end if

end
```

## Output

```
Example 1 for LIN_SOL_SVD is correct.
```

## Additional Examples

### Example 2: Polar Decomposition of a Square Matrix

A polar decomposition of an  $n \times n$  random matrix is obtained. This decomposition satisfies  $A = PQ$ , where  $P$  is orthogonal and  $Q$  is self-adjoint and positive definite.

Given the singular value decomposition

$$A = USV^T$$

the polar decomposition follows from the matrix products

$$P = UV^T \text{ and } Q = VSV^T$$

This example uses the optional arguments “u=”, “s=”, and “v=”, then array intrinsic functions to calculate  $P$  and  $Q$ . Also, see `operator_ex14`, [Chapter 10](#).

```

use lin_sol_svd_int
use rand_gen_int

implicit none

! This is Example 2 for LIN_SOL_SVD.

integer i
integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
real(kind(1d0)) a(n,n), b(n,0), ident(n,n), p(n,n), q(n,n), &
 s_d(n), u_d(n,n), v_d(n,n), x(n,0), y(n*n)

! Generate a random matrix.

call rand_gen(y)
a = reshape(y, (/n,n/))

! Compute the singular value decomposition.

call lin_sol_svd(a, b, x, nrhs=0, s=s_d, &
 u=u_d, v=v_d)

! Compute the (left) orthogonal factor.

p = matmul(u_d, transpose(v_d))

! Compute the (right) self-adjoint factor.

q = matmul(v_d*spread(s_d,1,n), transpose(v_d))

ident=zero
do i=1, n
 ident(i,i) = one
end do

! Check the results.

if (sum(abs(matmul(p, transpose(p)) - ident))/sum(abs(p)) &
 <= sqrt(epsilon(one))) then
 if (sum(abs(a - matmul(p,q))/sum(abs(a)) &
 <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_SOL_SVD is correct.'
 end if
end if

```

```
end
```

## Output

Example 2 for LIN\_SOL\_SVD is correct.

### Example 3: Reduction of an Array of Black and White

An  $n \times n$  array  $A$  contains entries that are either 0 or 1. The entry is chosen so that as a two-dimensional object with origin at the point (1, 1), the array appears as a black circle of radius  $n/4$  centered at the point  $(n/2, n/2)$ .

A singular value decomposition

$$A = USV^T$$

is computed, where  $S$  is of low rank. Approximations using fewer of these nonzero singular values and vectors suffice to reconstruct  $A$ . Also, see `operator_ex15`, supplied with the product examples.

```
use lin_sol_svd_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 3 for LIN_SOL_SVD.

integer i, j, k
integer, parameter :: n=32
real(kind(1e0)), parameter :: half=0.5e0, one=1e0, zero=0e0
real(kind(1e0)) a(n,n), b(n,0), x(n,0), s(n), u(n,n), &
 v(n,n), c(n,n)

! Fill in value one for points inside the circle.
a = zero
do i=1, n
 do j=1, n
 if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) a(i,j) = one
 end do
end do

! Compute the singular value decomposition.
call lin_sol_svd(a, b, x, nrhs=0,&
 s=s, u=u, v=v)

! How many terms, to the nearest integer, exactly
! match the circle?
c = zero; k = count(s > half)
do i=1, k
 c = c + spread(u(1:n,i),2,n)*spread(v(1:n,i),1,n)*s(i)
 if (count(int(c-a) /= 0) == 0) exit
end do

if (i < k) then
 write (*,*) 'Example 3 for LIN_SOL_SVD is correct.'
```

```
end if
end
```

## Output

Example 3 for LIN\_SOL\_SVD is correct.

## Example 4: Laplace Transform Solution

This example illustrates the solution of a linear least-squares system where the matrix is poorly conditioned. The problem comes from solving the integral equation:

$$\int_0^1 e^{-st} f(t) dt = s^{-1} (1 - e^{-s}) = g(s)$$

The unknown function  $f(t) = 1$  is computed. This problem is equivalent to the numerical inversion of the Laplace Transform of the function  $g(s)$  using real values of  $t$  and  $s$ , solving for a function that is nonzero only on the unit interval. The evaluation of the integral uses the following approximate integration rule:

$$\int_0^1 f(t) e^{-st} dt = \sum_{j=1}^n f(t_j) \int_{t_j}^{t_{j+1}} e^{-st} dt$$

The points  $\{t_j\}$  are chosen equally spaced by using the following:

$$t_j = \frac{j-1}{n}$$

The points  $\{s_j\}$  are computed so that the range of  $g(s)$  is uniformly sampled. This requires the solution of  $m$  equations

$$g(s_i) = g_i = \frac{i}{m+1}$$

for  $j = 1, \dots, n$  and  $i = 1, \dots, m$ . Fortran 90 array operations are used to solve for the collocation points  $\{s_i\}$  as a single series of steps. Newton's method,

$$s \leftarrow s - \frac{h}{h'}$$

is applied to the array function

$$h(s) = e^{-s} + sg - 1$$

where the following is true:

$$g = [g_1, \dots, g_m]^T$$

Note the coefficient matrix for the solution values

$$f = [f(t_1), \dots, f(t_n)]^T$$

whose entry at the intersection of row  $i$  and column  $j$  is equal to the value

$$\int_{t_j}^{t_{j+1}} e^{-s_i t} dt$$

is explicitly integrated and evaluated as an array operation. The solution analysis of the resulting linear least-squares system

$$Af \cong g$$

is obtained by computing the singular value decomposition

$$A = USV^T$$

An approximate solution is computed with the transformed right-hand side

$$b = U^T g$$

followed by using as few of the largest singular values as possible to minimize the following squared error residual:

$$\sum_{j=1}^n (1 - f_j)^2$$

This determines an optimal value  $k$  to use in the approximate solution

$$f = \sum_{j=1}^k b_j \frac{v_j}{s_j}$$

Also, see `operator_ex16`, supplied with the product examples.

```
use lin_sol_svd_int
use rand_gen_int
use error_option_packet

implicit none
```

! This is Example 4 for LIN\_SOL\_SVD.

```
integer i, j, k
integer, parameter :: m=64, n=16
real(kind(1e0)), parameter :: one=1e0, zero=0.0e0
real(kind(1e0)) :: g(m), s(m), t(n+1), a(m,n), b(m,1), &
 f(n,1), U_S(m,m), V_S(n,n), S_S(n), &
 rms, oldrms
```

```

real(kind(1e0)) :: delta_g, delta_t

delta_g = one/real(m+1,kind(one))

! Compute which collocation equations to solve.
do i=1,m
 g(i)=i*delta_g
end do

! Compute equally spaced quadrature points.
delta_t =one/real(n,kind(one))
do j=1,n+1
 t(j)=(j-1)*delta_t
end do

! Compute collocation points.
s=m
solve_equations: do
 s=s-(exp(-s)-(one-s*g))/(g-exp(-s))
 if (sum(abs((one-exp(-s))/s - g)) <= &
 epsilon(one)*sum(g)) &
 exit solve_equations
end do solve_equations

! Evaluate the integrals over the quadrature points.
a = (exp(-spread(t(1:n),1,m)*spread(s,2,n)) &
 - exp(-spread(t(2:n+1),1,m)*spread(s,2,n))) / &
 spread(s,2,n)

b(1:,1)=g

! Compute the singular value decomposition.

call lin_sol_svd(a, b, f, nrhs=0, &
 rank=k, u=U_S, v=V_S, s=S_S)

! Singular values that are larger than epsilon determine
! the rank=k.
k = count(S_S > epsilon(one))
oldrms = huge(one)
g = matmul(transpose(U_S), b(1:m,1))

! Find the minimum number of singular values that gives a good
! approximation to f(t) = 1.

do i=1,k
 f(1:n,1) = matmul(V_S(1:,1:i), g(1:i)/S_S(1:i))
 f = f - one
 rms = sum(f**2)/n
 if (rms > oldrms) exit
 oldrms = rms
end do

write (*,"(' Using this number of singular values, ', &
 &i4 / ' the approximate R.M.S. error is ', 1pe12.4)") &
i-1, oldrms

```

```

if (sqrt(olddrms) <= delta_t**2) then
 write (*,*) 'Example 4 for LIN_SOL_SVD is correct.'
end if

end

```

## Output

Example 4 for LIN\_SOL\_SVD is correct.

---

## LIN\_SOL\_TRI

Solves multiple systems of linear equations

$$A_j x_j = y_j, j = 1, \dots, k$$

Each matrix  $A_j$  is tridiagonal with the same dimension,  $n$ . The default solution method is based on  $LU$  factorization computed using cyclic reduction or, optionally, Gaussian elimination with partial pivoting.

### Required Arguments

- C** — Array of size  $2n \times k$  containing the upper diagonals of the matrices  $A_j$ . Each upper diagonal is entered in array locations  $c(1:n-1, j)$ . The data  $C(n, 1:k)$  are not used. (Input [/Output])  
The input data is overwritten. See note below.
- D** — Array of size  $2n \times k$  containing the diagonals of the matrices  $A_j$ . Each diagonal is entered in array locations  $D(1:n, j)$ . (Input [/Output])  
The input data is overwritten. See note below.
- B** — Array of size  $2n \times k$  containing the lower diagonals of the matrices  $A_j$ . Each lower diagonal is entered in array locations  $B(2:n, j)$ . The data  $B(1, 1:k)$  are not used. (Input [/Output])  
The input data is overwritten. See note below.
- Y** — Array of size  $2n \times k$  containing the right-hand sides,  $y_j$ . Each right-hand side is entered in array locations  $Y(1:n, j)$ . The computed solution  $x_j$  is returned in locations  $Y(1:n, j)$ . (Input [/Output])

---

**NOTE:** The required arguments have the Input data overwritten. If these quantities are used later, they must be saved in user-defined arrays. The routine uses each array's locations  $(n+1:2*n, 1:k)$  for scratch storage and intermediate data in the LU factorization. The default values for problem dimensions are  $n = (\text{size}(D, 1))/2$  and  $k = \text{size}(D, 2)$ .

---

## Optional Arguments

`NCOLS = n` (Input)

Uses arrays  $C(1:n-1, 1:k)$ ,  $D(1:n, 1:k)$ , and  $B(2:n, 1:k)$  as the upper, main and lower diagonals for the input tridiagonal matrices. The right-hand sides and solutions are in array  $Y(1:n, 1:k)$ . Note that each of these arrays are rank-2.

Default:  $n = (\text{size}(D, 1))/2$

`NPROB = k` (Input)

The number of systems solved.

Default:  $k = \text{size}(D, 2)$

`iopt = iopt(:)` (Input)

Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>LIN_SOL_TRI</code> |                                         |              |
|-----------------------------------------------|-----------------------------------------|--------------|
| Option Prefix = ?                             | Option Name                             | Option Value |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_set_small</code>      | 1            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_set_jolt</code>       | 2            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_scan_for_NaN</code>   | 3            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_factor_only</code>    | 4            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_solve_only</code>     | 5            |
| <code>s_,d_,c_,z_</code>                      | <code>lin_sol_tri_use_Gauss_elim</code> | 6            |

`iopt(IO) = ?_options(?_lin_sol_tri_set_small, Small)`

Whenever a reciprocation is performed on a quantity smaller than *Small*, it is replaced by that value plus  $2 \times \text{jolt}$ .

Default:  $0.25 \times \text{epsilon}()$

`iopt(IO) = ?_options(?_lin_sol_tri_set_jolt, jolt)`

Default: *epsilon()*, machine precision

`iopt(IO) = ?_options(?_lin_sol_tri_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(C(i,j))` .or.

`isNaN(D(i,j))` .or.

`isNaN(B(i,j))` .or.

`isNaN(Y(i,j)) == .true.`

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs.

`iopt(IO) = ?_options(?_lin_sol_tri_factor_only, ?_dummy)`

Obtain the  $LU$  factorization of the matrices  $A_j$ . Does not solve for a solution.

Default: Factor the matrices and solve the systems.

`iopt(IO) = ?_options(?_lin_sol_tri_solve_only, ?_dummy)`

Solve the systems  $A_j x_j = y_j$  using the previously computed  $LU$  factorization.

Default: Factor the matrices and solve the systems.

`iopt(IO) = ?_options(?_lin_sol_tri_use_Gauss_elim, ?_dummy)`

The accuracy, numerical stability or efficiency of the cyclic reduction algorithm may be inferior to the use of  $LU$  factorization with partial pivoting.

Default: Use cyclic reduction to compute the factorization.

## FORTRAN 90 Interface

Generic:     `CALL LIN_SOL_TRI (C, D, B, Y [, ...])`

Specific:    The specific interface names are `S_LIN_SOL_TRI`, `D_LIN_SOL_TRI`,

`C_LIN_SOL_TRI`, and `Z_LIN_SOL_TRI`.

## Description

Routine `lin_sol_tri` solves  $k$  systems of tridiagonal linear algebraic equations, each problem of dimension  $n \times n$ . No relation between  $k$  and  $n$  is required. See Kershaw, pages 86–88 in Rodrigue (1982) for further details. To deal with poorly conditioned or singular systems, a specific regularizing term is added to each reciprocated value. This technique keeps the factorization process efficient and avoids exceptions from overflow or division by zero. Each occurrence of an array reciprocal  $a^{-1}$  is replaced by the expression  $(a + t)^{-1}$ , where the array temporary  $t$  has the value 0 whenever the corresponding entry satisfies  $|a| > \textit{Small}$ . Alternately,  $t$  has the value  $2 \times \textit{jolt}$ . (Every small denominator gives rise to a finite “jolt”.) Since this tridiagonal solver is used in the routines `lin_svd` and `lin_eig_self` for inverse iteration, regularization is required. Users can reset the values of *Small* and *jolt* for their own needs. Using the default values for these parameters, it is generally necessary to scale the tridiagonal matrix so that the maximum magnitude has value approximately one. This is normally not an issue when the systems are nonsingular.

The routine is designed to use cyclic reduction as the default method for computing the  $LU$  factorization. Using an optional parameter, standard elimination and partial pivoting will be used to compute the factorization. Partial pivoting is numerically stable but is likely to be less efficient than cyclic reduction.

## Fatal, Terminal, and Warning Error Messages

See the `messages.gls` file for error messages for `LIN_SOL_TRI`. These error messages are numbered 1081–1086; 1101–1106; 1121–1126; 1141–1146.

## Example 1: Solution of Multiple Tridiagonal Systems

The upper, main and lower diagonals of  $n$  systems of size  $n \times n$  are generated randomly. A scalar is added to the main diagonal so that the systems are positive definite. A random vector  $x_j$  is

generated and right-hand sides  $y_j = A_j y_j$  are computed. The routine is used to compute the solution, using the  $A_j$  and  $y_j$ . The results should compare closely with the  $x_j$  used to generate the right-hand sides. Also, see `operator_ex17`, supplied with the product examples.

```

use lin_sol_tri_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 1 for LIN_SOL_TRI.

integer i
integer, parameter :: n=128
real(kind(ld0)), parameter :: one=1d0, zero=0d0
real(kind(ld0)) err
real(kind(ld0)), dimension(2*n,n) :: d, b, c, res(n,n), &
 t(n), x, y

! Generate the upper, main, and lower diagonals of the
! n matrices A_i. For each system a random vector x is used
! to construct the right-hand side, Ax = y. The lower part
! of each array remains zero as a result.

c = zero; d=zero; b=zero; x=zero
do i = 1, n
 call rand_gen (c(1:n,i))
 call rand_gen (d(1:n,i))
 call rand_gen (b(1:n,i))
 call rand_gen (x(1:n,i))
end do

! Add scalars to the main diagonal of each system so that
! all systems are positive definite.
t = sum(c+d+b,DIM=1)
d(1:n,1:n) = d(1:n,1:n) + spread(t,DIM=1,NCOPIES=n)

! Set Ax = y. The vector x generates y. Note the use
! of EOSHIFT and array operations to compute the matrix
! product, n distinct ones as one array operation.

y(1:n,1:n)=d(1:n,1:n)*x(1:n,1:n) + &
 c(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=+1,DIM=1) + &
 b(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=-1,DIM=1)

! Compute the solution returned in y. (The input values of c,
! d, b, and y are overwritten by lin_sol_tri.) Check for any
! error messages.

call lin_sol_tri (c, d, b, y)

! Check the size of the residuals, y-x. They should be small,
! relative to the size of values in x.
res = x(1:n,1:n) - y(1:n,1:n)
err = sum(abs(res)) / sum(abs(x(1:n,1:n)))

```

```

if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SOL_TRI is correct.'
end if

end

```

## Output

Example 1 for LIN\_SOL\_TRI is correct.

## Additional Examples

### Example 2: Iterative Refinement and Use of Partial Pivoting

This program unit shows usage that typically gives acceptable accuracy for a large class of problems. Our goal is to use the efficient cyclic reduction algorithm when possible, and keep on using it unless it will fail. In exceptional cases our program switches to the *LU* factorization with partial pivoting. This use of both factorization and solution methods enhances reliability and maintains efficiency on the average. Also, see `operator_ex18`, supplied with the product examples.

```

use lin_sol_tri_int
use rand_gen_int

implicit none

! This is Example 2 for LIN_SOL_TRI.

integer i, nopt
integer, parameter :: n=128
real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
real(kind(1d0)), parameter :: d_one=1d0, d_zero=0d0
real(kind(1e0)), dimension(2*n,n) :: d, b, c, res(n,n), &
 x, y
real(kind(1e0)) change_new, change_old, err
type(s_options) :: iopt(2) = s_options(0,s_zero)
real(kind(1d0)), dimension(n,n) :: d_save, b_save, c_save, &
 x_save, y_save, x_sol
logical solve_only

c = s_zero; d=s_zero; b=s_zero; x=s_zero

! Generate the upper, main, and lower diagonals of the
! matrices A. A random vector x is used to construct the
! right-hand sides: y=A*x.
do i = 1, n
 call rand_gen (c(1:n,i))
 call rand_gen (d(1:n,i))
 call rand_gen (b(1:n,i))
 call rand_gen (x(1:n,i))
end do

! Save double precision copies of the diagonals and the
! right-hand side.
c_save = c(1:n,1:n); d_save = d(1:n,1:n)
b_save = b(1:n,1:n); x_save = x(1:n,1:n)

```

```

y_save(1:n,1:n) = d(1:n,1:n)*x_save + &
 c(1:n,1:n)*EOSHIFT(x_save,SHIFT=+1,DIM=1) + &
 b(1:n,1:n)*EOSHIFT(x_save,SHIFT=-1,DIM=1)

! Iterative refinement loop.
 factorization_choice: do nopt=0, 1

! Set the logical to flag the first time through.

 solve_only = .false.
 x_sol = d_zero
 change_old = huge(s_one)

 iterative_refinement: do

! This flag causes a copy of data to be moved to work arrays
! and a factorization and solve step to be performed.
 if (.not. solve_only) then
 c(1:n,1:n)=c_save; d(1:n,1:n)=d_save
 b(1:n,1:n)=b_save
 end if

! Compute current residuals, $y - A*x$, using current x .
 y(1:n,1:n) = -y_save + &
 d_save*x_sol + &
 c_save*EOSHIFT(x_sol,SHIFT=+1,DIM=1) + &
 b_save*EOSHIFT(x_sol,SHIFT=-1,DIM=1)

 call lin_sol_tri (c, d, b, y, iopt=iopt)

 x_sol = x_sol + y(1:n,1:n)

 change_new = sum(abs(y(1:n,1:n)))

! If size of change is not decreasing, stop the iteration.
 if (change_new >= change_old) exit iterative_refinement

 change_old = change_new
 iopt(nopt+1) = s_options(s_lin_sol_tri_solve_only,s_zero)
 solve_only = .true.

 end do iterative_refinement

! Use Gaussian Elimination if Cyclic Reduction did not get an
! accurate solution.
! It is an exceptional event when Gaussian Elimination is required.
 if (sum(abs(x_sol - x_save)) / sum(abs(x_save)) &
 <= sqrt(epsilon(d_one))) exit factorization_choice

 iopt = s_options(0,s_zero)
 iopt(nopt+1) = s_options(s_lin_sol_tri_use_Gauss_elim,s_zero)

end do factorization_choice

! Check on accuracy of solution.

```

```

res = x(1:n,1:n) - x_save
err = sum(abs(res)) / sum(abs(x_save))
if (err <= sqrt(epsilon(d_one))) then
 write (*,*) 'Example 2 for LIN_SOL_TRI is correct.'
end if

end

```

## Output

Example 2 for LIN\_SOL\_TRI is correct.

### Example 3: Selected Eigenvectors of Tridiagonal Matrices

The eigenvalues  $\lambda_1, \dots, \lambda_n$  of a tridiagonal real, self-adjoint matrix are computed. Note that the computation is performed using the IMSL MATH/LIBRARY FORTRAN 77 interface to routine [EVASE](#). The user may write this interface based on documentation of the arguments (IMSL 2003, p. 480), or use the module *Numerical\_Libraries* as we have done here. The eigenvectors corresponding to  $k < n$  of the eigenvalues are required. These vectors are computed using inverse iteration for all the eigenvalues at one step. See Golub and Van Loan (1989, Chapter 7). The eigenvectors are then orthogonalized. Also, see `operator_ex19`, supplied with the product examples.

```

use lin_sol_tri_int
use rand_gen_int
use Numerical_Libraries

implicit none

! This is Example 3 for LIN_SOL_TRI.

integer i, j, nopt
integer, parameter :: n=128, k=n/4, ncoda=1, lda=2
real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
real(kind(1e0)) A(lda,n), EVAL(k)
type(s_options) :: iopt(2)=s_options(0,s_zero)
real(kind(1e0)) d(n), b(n), d_t(2*n,k), c_t(2*n,k), perf_ratio, &
 b_t(2*n,k), y_t(2*n,k), eval_t(k), res(n,k), temp
logical small

! This flag is used to get the k largest eigenvalues.
small = .false.

! Generate the main diagonal and the co-diagonal of the
! tridiagonal matrix.

call rand_gen (b)
call rand_gen (d)

A(1,1:)=b; A(2,1:)=d

! Use Numerical Libraries routine for the calculation of k
! largest eigenvalues.

```

```

CALL EVASB (N, K, A, LDA, NCODA, SMALL, EVAL)
EVAL_T = EVAL

! Use DNFL tridiagonal solver for inverse iteration
! calculation of eigenvectors.
 factorization_choice: do nopt=0,1

! Create k tridiagonal problems, one for each inverse
! iteration system.
 b_t(1:n,1:k) = spread(b,DIM=2,NCOPIES=k)
 c_t(1:n,1:k) = EOSHIFT(b_t(1:n,1:k),SHIFT=1,DIM=1)
 d_t(1:n,1:k) = spread(d,DIM=2,NCOPIES=k) - &
 spread(EVAL_T,DIM=1,NCOPIES=n)

! Start the right-hand side at random values, scaled downward
! to account for the expected 'blowup' in the solution.
 do i=1, k
 call rand_gen (y_t(1:n,i))
 end do

! Do two iterations for the eigenvectors.
 do i=1, 2
 y_t(1:n,1:k) = y_t(1:n,1:k)*epsilon(s_one)
 call lin_sol_tri(c_t, d_t, b_t, y_t, &
 iopt=iopt)
 iopt(nopt+1) = s_options(s_lin_sol_tri_solve_only,s_zero)
 end do

! Orthogonalize the eigenvectors. (This is the most
! intensive part of the computing.)
 do j=1,k-1 ! Forward sweep of HMGS orthogonalization.
 temp=s_one/sqrt(sum(y_t(1:n,j)**2))
 y_t(1:n,j)=y_t(1:n,j)*temp

 y_t(1:n,j+1:k)=y_t(1:n,j+1:k)+ &
 spread(-matmul(y_t(1:n,j),y_t(1:n,j+1:k)), &
 DIM=1,NCOPIES=n)* spread(y_t(1:n,j),DIM=2,NCOPIES=k-j)
 end do
 temp=s_one/sqrt(sum(y_t(1:n,k)**2))
 y_t(1:n,k)=y_t(1:n,k)*temp

 do j=k-1,1,-1 ! Backward sweep of HMGS.
 y_t(1:n,j+1:k)=y_t(1:n,j+1:k)+ &
 spread(-matmul(y_t(1:n,j),y_t(1:n,j+1:k)), &
 DIM=1,NCOPIES=n)* spread(y_t(1:n,j),DIM=2,NCOPIES=k-j)
 end do

! See if the performance ratio is smaller than the value one.
! If it is not the code will re-solve the systems using Gaussian
! Elimination. This is an exceptional event. It is a necessary
! complication for achieving reliable results.

 res(1:n,1:k) = spread(d,DIM=2,NCOPIES=k)*y_t(1:n,1:k) + &
 spread(b,DIM=2,NCOPIES=k)* &
 EOSHIFT(y_t(1:n,1:k),SHIFT=-1,DIM=1) + &

```

```

EOSHIFT(spread(b,DIM=2,NCOPIES=k)*y_t(1:n,1:k),SHIFT=1) &
-y_t(1:n,1:k)*spread(EVAL_T(1:k),DIM=1,NCOPIES=n)

! If the factorization method is Cyclic Reduction and perf_ratio is
! larger than one, re-solve using Gaussian Elimination. If the
! method is already Gaussian Elimination, the loop exits
! and perf_ratio is checked at the end.
perf_ratio = sum(abs(res(1:n,1:k))) / &
 sum(abs(EVAL_T(1:k))) / &
 epsilon(s_one) / (5*n)
if (perf_ratio <= s_one) exit factorization_choice
iopt(nopt+1) = s_options(s_lin_sol_tri_use_Gauss_elim,s_zero)

end do factorization_choice

if (perf_ratio <= s_one) then
write (*,*) 'Example 3 for LIN_SOL_TRI is correct.'
end if

end

```

## Output

Example 3 for LIN\_SOL\_TRI is correct.

### Example 4: Tridiagonal Matrix Solving within Diffusion Equations

The normalized partial differential equation

$$u_t \equiv \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \equiv u_{xx}$$

is solved for values of  $0 \leq x \leq \pi$  and  $t > 0$ . A boundary value problem consists of choosing the value

$$u(0, t) = u_0$$

such that the equation

$$u(x_1, t_1) = u_1$$

is satisfied. Arbitrary values

$$x_1 = \frac{\pi}{2}, u_1 = \frac{1}{2}$$

and

$$t_1 = 1$$

are used for illustration of the solution process. The one-parameter equation

$$u(x_1, t_1) - u_1 = 0$$

The variables are changed to

$$v(x, t) = u(x, t) - u_0$$

that  $v(0, t) = 0$ . The function  $v(x, t)$  satisfies the differential equation. The one-parameter equation solved is therefore

$$v(x_1, t_1) - (u_1 - u_0) = 0$$

To solve this equation for  $u_0$ , use the standard technique of the *variational equation*,

$$w \equiv \frac{\partial v}{\partial u_0}$$

Thus

$$\frac{\partial w}{\partial t} = \frac{\partial^2 w}{\partial x^2}$$

Since the initial data for

$$v(x, 0) = -u_0$$

the variational equation initial condition is

$$w(x, 0) = -1$$

This model problem illustrates the method of lines and Galerkin principle implemented with the differential-algebraic solver, `D2SPG` (IMSL 2003, pp. 889–911). We use the integrator in “reverse communication” mode for evaluating the required functions, derivatives, and solving linear algebraic equations. See Example 4 of routine `DASPG` for a problem that uses reverse communication. Next see Example 4 of routine `IVPAG` for the development of the piecewise-linear Galerkin discretization method to solve the differential equation. This present example extends parts of both previous examples and illustrates Fortran 90 constructs. It further illustrates how a user can deal with a defect of an integrator that normally functions using only dense linear algebra factorization methods for solving the corrector equations. See the comments in Brenan et al. (1989, esp. p. 137). Also, see `operator_ex20`, supplied with the product examples.

```
use lin_sol_tri_int
use rand_gen_int
use Numerical_Libraries

implicit none

! This is Example 4 for LIN_SOL_TRI.

integer, parameter :: n=1000, ichap=5, iget=1, iput=2, &
 inum=6, irnum=7
real(kind(1e0)), parameter :: zero=0e0, one = 1e0
integer i, ido, in(50), inr(20), iopt(6), ival(7), &
 iwk(35+n)
real(kind(1e0)) hx, pi_value, t, u_0, u_1, atol, rtol, sval(2), &
```

```

 tend, wk(41+11*n), y(n), ypr(n), a_diag(n), &
 a_off(n), r_diag(n), r_off(n), t_y(n), t_ypr(n), &
 t_g(n), t_diag(2*n,1), t_upper(2*n,1), &
 t_lower(2*n,1), t_sol(2*n,1)
type(s_options) :: iopti(2)=s_options(0,zero)

character(2) :: pi(1) = 'pi'
! Define initial data.
t = 0.0e0
u_0 = 1
u_1 = 0.5
tend = one

! Initial values for the variational equation.
y = -one; ypr= zero
pi_value = const(pi)
hx = pi_value/(n+1)

a_diag = 2*hx/3
a_off = hx/6
r_diag = -2/hx
r_off = 1/hx

! Get integer option numbers.
iopt(1) = inum
call iumag ('math', ichap, iget, 1, iopt, in)

! Get floating point option numbers.
iopt(1) = irnum
call iumag ('math', ichap, iget, 1, iopt, inr)

! Set for reverse communication evaluation of the DAE.
iopt(1) = in(26)
ival(1) = 0

! Set for use of explicit partial derivatives.
iopt(2) = in(5)
ival(2) = 1

! Set for reverse communication evaluation of partials.
iopt(3) = in(29)
ival(3) = 0

! Set for reverse communication solution of linear equations.
iopt(4) = in(31)
ival(4) = 0

! Storage for the partial derivative array are not allocated or
! required in the integrator.
iopt(5) = in(34)
ival(5) = 1

! Set the sizes of iwk, wk for internal checking.
iopt(6) = in(35)
ival(6) = 35 + n
ival(7) = 41 + 11*n

! Set integer options:
call iumag ('math', ichap, iput, 6, iopt, ival)
! Reset tolerances for integrator:
atol = 1e-3; rtol= 1e-3
sval(1) = atol; sval(2) = rtol
iopt(1) = inr(5)

```

```

! Set floating point options:
 call sumag ('math', ichap, iput, 1, iopt, sval)
! Integrate ODE/DAE. Use dummy external names for g(y,y')
! and partials.
 ido = 1
 Integration_Loop: do

 call d2spg (n, t, tend, ido, y, ypr, dgspg, djspg, iwk, wk)
! Find where g(y,y') goes. (It only goes in one place here, but can
! vary where divided differences are used for partial derivatives.)
 iopt(1) = in(27)
 call iumag ('math', ichap, iget, 1, iopt, ival)
! Direct user response:
 select case(ido)

 case(1,4)
! This should not occur.
 write (*,*) ' Unexpected return with ido = ', ido
 stop

 case(3)
! Reset options to defaults. (This is good housekeeping but not
! required for this problem.)
 in = -in
 call iumag ('math', ichap, iput, 50, in, ival)
 inr = -inr
 call sumag ('math', ichap, iput, 20, inr, sval)
 exit Integration_Loop
 case(5)
! Evaluate partials of g(y,y').
 t_y = y; t_ypr = ypr

 t_g = r_diag*t_y + r_off*EOSHIFT(t_y,SHIFT=+1) &
 + EOSHIFT(r_off*t_y,SHIFT=-1) &
 - (a_diag*t_ypr + a_off*EOSHIFT(t_ypr,SHIFT=+1) &
 + EOSHIFT(a_off*t_ypr,SHIFT=-1))
! Move data from the assumed size to assumed shape arrays.
 do i=1, n
 wk(ival(1)+i-1) = t_g(i)
 end do
 cycle Integration_Loop

 case(6)
! Evaluate partials of g(y,y').
! Get value of c_j for partials.
 iopt(1) = inr(9)
 call sumag ('math', ichap, iget, 1, iopt, sval)

! Subtract c_j from diagonals to compute (partials for y')*c_j.
! The linear system is tridiagonal.
 t_diag(1:n,1) = r_diag - sval(1)*a_diag
 t_upper(1:n,1) = r_off - sval(1)*a_off
 t_lower = EOSHIFT(t_upper,SHIFT=+1,DIM=1)

 cycle Integration_Loop
 end select
 end do

```

```

 case(7)
! Compute the factorization.
 iopti(1) = s_options(s_lin_sol_tri_factor_only,zero)
 call lin_sol_tri (t_upper, t_diag, t_lower, &
 t_sol, iopt=iopti)
 cycle Integration_Loop

 case(8)
! Solve the system.
 iopti(1) = s_options(s_lin_sol_tri_solve_only,zero)
! Move data from the assumed size to assumed shape arrays.
 t_sol(1:n,1)=wk(ival(1):ival(1)+n-1)

 call lin_sol_tri (t_upper, t_diag, t_lower, &
 t_sol, iopt=iopti)

! Move data from the assumed shape to assumed size arrays.
 wk(ival(1):ival(1)+n-1)=t_sol(1:n,1)

 cycle Integration_Loop

 case(2)
! Correct initial value to reach u_1 at t=tend.
 u_0 = u_0 - (u_0*y(n/2) - (u_1-u_0)) / (y(n/2) + 1)

! Finish up internally in the integrator.
 ido = 3
 cycle Integration_Loop
 end select
end do Integration_Loop

write (*,*) 'The equation u_t = u_xx, with u(0,t) = ', u_0
write (*,*) 'reaches the value ',u_1, ' at time = ', tend, '.'
write (*,*) 'Example 4 for LIN_SOL_TRI is correct.'

end

```

## Output

Example 4 for LIN\_SOL\_TRI is correct.

---

## LIN\_SVD

Computes the singular value decomposition (SVD) of a rectangular matrix,  $A$ . This gives the decomposition

$$A = USV^T$$

where  $V$  is an  $n \times n$  orthogonal matrix,  $U$  is an  $m \times m$  orthogonal matrix, and  $S$  is a real, rectangular diagonal matrix.

## Required Arguments

- A** — Array of size  $m \times n$  containing the matrix. (Input [/Output])  
 If the packaged option `lin_svd_overwrite_input` is used, this array is not saved on output.
- S** — Array of size  $\min(m, n)$  containing the real singular values. These nonnegative values are in non-increasing order. (Output)
- U** — Array of size  $m \times m$  containing the singular vectors,  $U$ . (Output)
- V** — Array of size  $n \times n$  containing the singular vectors,  $V$ . (Output)

## Optional Arguments

- MROWS** =  $m$  (Input)  
 Uses array `A(1:m, 1:n)` for the input matrix.  
 Default: `m = size(A, 1)`
- NCOLS** =  $n$  (Input)  
 Uses array `A(1:m, 1:n)` for the input matrix.  
 Default: `n = size(A, 2)`
- RANK** =  $k$  (Output)  
 Number of singular values that exceed the value *Small*. **RANK** will satisfy  $k \leq \min(m, n)$ .
- iopt** = `iopt(:)` (Input)  
 Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>LIN_SVD</code> |                                      |              |
|-------------------------------------------|--------------------------------------|--------------|
| Option Prefix = ?                         | Option Name                          | Option Value |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_set_small</code>       | 1            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_overwrite_input</code> | 2            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_scan_for_NaN</code>    | 3            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_use_qr</code>          | 4            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_skip_orth</code>       | 5            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_use_gauss_elim</code>  | 6            |
| <code>S_,d_,c_,z_</code>                  | <code>lin_svd_set_perf_ratio</code>  | 7            |

`iopt(IO) = ?_options(?_lin_svd_set_small, Small)`  
 If a singular value is smaller than *Small*, it is defined as zero for the purpose of computing the rank of *A*.  
 Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_svd_overwrite_input, ?_dummy)`  
 Does not save the input array `A(:, :)`.

`iopt(IO) = ?_options(?_lin_svd_scan_for_NaN, ?_dummy)`  
 Examines each input array entry to find the first value such that  
`isNaN(a(i,j)) == .true.`  
 See the `isNaN()` function, [Chapter 10](#).  
 Default: The array is not scanned for NaNs.

`iopt(IO) = ?_options(?_lin_svd_use_qr, ?_dummy)`  
 Uses a rational *QR* algorithm to compute eigenvalues. Accumulate the singular vectors using this algorithm.  
 Default: singular vectors computed using inverse iteration

`iopt(IO) = ?_options(?_lin_svd_skip_Orth, ?_dummy)`  
 If the eigenvalues are computed using inverse iteration, skips the final orthogonalization of the vectors. This method results in a more efficient computation. However, the singular vectors, while a complete set, may not be orthogonal.  
 Default: singular vectors are orthogonalized if obtained using inverse iteration

`iopt(IO) = ?_options(?_lin_svd_use_gauss_elim, ?_dummy)`  
 If the eigenvalues are computed using inverse iteration, uses standard elimination with partial pivoting to solve the inverse iteration problems.  
 Default: singular vectors computed using cyclic reduction

`iopt(IO) = ?_options(?_lin_svd_set_perf_ratio, perf_ratio)`  
 Uses residuals for approximate normalized singular vectors if they have a performance index no larger than *perf\_ratio*. Otherwise an alternate approach is taken and the singular vectors are computed again: Standard elimination is used instead of cyclic reduction, or the standard *QR* algorithm is used as a backup procedure to inverse iteration. Larger values of *perf\_ratio* are less likely to cause these exceptions.  
 Default: *perf\_ratio* = 4

## FORTRAN 90 Interface

Generic:    `CALL LIN_SVD (A, S, U, V [, ...])`

Specific:   The specific interface names are `S_LIN_SVD`, `D_LIN_SVD`, `C_LIN_SVD`, and `Z_LIN_SVD`.

## Description

Routine `lin_svd` is an implementation of the *QR* algorithm for computing the SVD of rectangular matrices. An orthogonal reduction of the input matrix to upper bidiagonal form is performed. Then, the SVD of a real bidiagonal matrix is calculated. The orthogonal decomposition  $AV = US$  results from products of intermediate matrix factors. See Golub and Van Loan (1989, Chapter 8) for details.

## Fatal, Terminal, and Warning Error Messages

See the `messages.gls` file for error messages for `LIN_SVD`. These error messages are numbered 1001–1010; 1021–1030; 1041–1050; 1061–1070.

## Example 1: Computing the SVD

The SVD of a square, random matrix  $A$  is computed. The residuals  $R = AV - US$  are small with respect to working precision. Also, see `operator_ex21`, supplied with the product examples.

```
use lin_svd_int
use rand_gen_int

implicit none

! This is Example 1 for LIN_SVD.

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)) err
real(kind(1d0)), dimension(n,n) :: A, U, V, S(n), y(n*n)

! Generate a random n by n matrix.
call rand_gen(y)
A = reshape(y, (/n,n/))

! Compute the singular value decomposition.
call lin_svd(A, S, U, V)

! Check for small residuals of the expression A*V - U*S.
err = sum(abs(matmul(A,V) - U*spread(S,dim=1,ncopies=n))) &
 / sum(abs(S))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SVD is correct.'
end if
end
```

## Output

```
Example 1 for LIN_SVD is correct.
```

## Additional Examples

### Example 2: Linear Least Squares with a Quadratic Constraint

An  $m \times n$  matrix equation  $Ax \cong b$ ,  $m > n$ , is approximated in a least-squares sense. The matrix  $b$  is size  $m \times k$ . Each of the  $k$  solution vectors of the matrix  $x$  is constrained to have Euclidean length of value  $\alpha_j > 0$ . The value of  $\alpha_j$  is chosen so that the constrained solution is 0.25 the length of the nonregularized or standard least-squares equation. See Golub and Van Loan (1989, Chapter 12) for more details. In the Example 2 code, Newton's method is used to solve for each regularizing parameter of the  $k$  systems. The solution is then computed and its length is checked. Also, see operator\_ex22, supplied with the product examples.

```
use lin_svd_int
use rand_gen_int

implicit none

! This is Example 2 for LIN_SVD.

integer, parameter :: m=64, n=32, k=4
real(kind(1d0)), parameter :: one=1d0, zero=0d0
real(kind(1d0)) a(m,n), s(n), u(m,m), v(n,n), y(m*max(n,k)), &
 b(m,k), x(n,k), g(m,k), alpha(k), lamda(k), &
 delta_lamda(k), t_g(n,k), s_sq(n), phi(n,k), &
 phi_dot(n,k), rand(k), err

! Generate a random matrix for both A and B.
call rand_gen(y)
a = reshape(y, (/m,n/))

call rand_gen(y)
b = reshape(y, (/m,k/))

! Compute the singular value decomposition.
call lin_svd(a, s, u, v)

! Choose alpha so that the lengths of the regularized solutions
! are 0.25 times lengths of the non-regularized solutions.

g = matmul(transpose(u),b)
x = matmul(v,spread(one/s,dim=2,ncopies=k)*g(1:n,1:k))
alpha = 0.25*sqrt(sum(x**2,dim=1))

t_g = g(1:n,1:k)*spread(s,dim=2,ncopies=k)
s_sq = s**2; lamda = zero

solve_for_lamda: do
 x=one/(spread(s_sq,dim=2,ncopies=k)+ &
 spread(lamda,dim=1,ncopies=n))
 phi = (t_g*x)**2; phi_dot = -2*phi*x
 delta_lamda = (sum(phi,dim=1)-alpha**2)/sum(phi_dot,dim=1)

! Make Newton method correction to solve the secular equations for
! lamda.
lamda = lamda - delta_lamda
```

```

 if (sum(abs(delta_lamda)) <= &
 sqrt(epsilon(one))*sum(lamda)) &
 exit solve_for_lamda

! This is intended to fix up negative solution approximations.
 call rand_gen(rand)
 where (lamda < 0) lamda = s(1) * rand

end do solve_for_lamda

! Compute solutions and check lengths.
 x = matmul(v,t_g/(spread(s_sq,dim=2,ncopies=k)+ &
 spread(lamda,dim=1,ncopies=n)))

 err = sum(abs(sum(x**2,dim=1) - alpha**2))/sum(abs(alpha**2))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_SVD is correct.'
 end if

end

```

## Output

Example 2 for LIN\_SVD is correct.

### Example 3: Generalized Singular Value Decomposition

The  $n \times n$  matrices  $A$  and  $B$  are expanded in a Generalized Singular Value Decomposition (GSVD). Two  $n \times n$  orthogonal matrices,  $U$  and  $V$ , and a nonsingular matrix  $X$  are computed such that

$$AX = U \operatorname{diag}(c_1, \dots, c_n)$$

and

$$BX = V \operatorname{diag}(s_1, \dots, s_n)$$

The values  $s_i$  and  $c_i$  are normalized so that

$$s_i^2 + c_i^2 = 1$$

The  $c_i$  are nonincreasing, and the  $s_i$  are nondecreasing. See Golub and Van Loan (1989, Chapter 8) for more details. Our method is based on computing three SVDs as opposed to the  $QR$  decomposition and two SVDs outlined in Golub and Van Loan. As a bonus, an SVD of the matrix  $X$  is obtained, and you can use this information to answer further questions about its conditioning. This form of the decomposition assumes that the matrix

$$D = \begin{bmatrix} A \\ B \end{bmatrix}$$

has all its singular values strictly positive. For alternate problems, where some singular values of  $D$  are zero, the GSVD becomes

$$U^T A = \text{diag}(c_1, \dots, c_n) W$$

and

$$V^T B = \text{diag}(s_1, \dots, s_n) W$$

The matrix  $W$  has the same singular values as the matrix  $D$ . Also, see `operator_ex23`, supplied with the product examples.

```

use lin_svd_int
use rand_gen_int

implicit none

! This is Example 3 for LIN_SVD.

integer, parameter :: n=32
integer i
real(kind(1d0)), parameter :: one=1.0d0
real(kind(1d0)) a(n,n), b(n,n), d(2*n,n), x(n,n), u_d(2*n,2*n), &
 v_d(n,n), v_c(n,n), u_c(n,n), v_s(n,n), u_s(n,n), &
 y(n*n), s_d(n), c(n), s(n), sc_c(n), sc_s(n), &
 err1, err2

! Generate random square matrices for both A and B.

call rand_gen(y)
a = reshape(y, (/n,n/))

call rand_gen(y)
b = reshape(y, (/n,n/))

! Construct D; A is on the top; B is on the bottom.

d(1:n,1:n) = a
d(n+1:2*n,1:n) = b

! Compute the singular value decompositions used for the GSVD.

call lin_svd(d, s_d, u_d, v_d)
call lin_svd(u_d(1:n,1:n), c, u_c, v_c)
call lin_svd(u_d(n+1:,1:n), s, u_s, v_s)

! Rearrange c(:) so it is non-increasing. Move singular
! vectors accordingly. (The use of temporary objects sc_c and
! x is required.)

sc_c = c(n:1:-1); c = sc_c
x = u_c(1:n,n:1:-1); u_c = x
x = v_c(1:n,n:1:-1); v_c = x

! The columns of v_c and v_s have the same span. They are
! equivalent by taking the signs of the largest magnitude values

```

```

! positive.

do i=1, n
 sc_c(i) = sign(one,v_c(sum(maxloc(abs(v_c(1:n,i))))),i))
 sc_s(i) = sign(one,v_s(sum(maxloc(abs(v_s(1:n,i))))),i))
end do

v_c = v_c*spread(sc_c,dim=1,ncopies=n)
u_c = u_c*spread(sc_c,dim=1,ncopies=n)

v_s = v_s*spread(sc_s,dim=1,ncopies=n)
u_s = u_s*spread(sc_s,dim=1,ncopies=n)

! In this form of the GSVD, the matrix X can be unstable if D
! is ill-conditioned.
x = matmul(v_d*spread(one/s_d,dim=1,ncopies=n),v_c)

! Check residuals for GSVD, A*X = u_c*diag(c_1, ..., c_n), and
! B*X = u_s*diag(s_1, ..., s_n).
err1 = sum(abs(matmul(a,x) - u_c*spread(c,dim=1,ncopies=n))) &
 / sum(s_d)
err2 = sum(abs(matmul(b,x) - u_s*spread(s,dim=1,ncopies=n))) &
 / sum(s_d)
if (err1 <= sqrt(epsilon(one)) .and. &
 err2 <= sqrt(epsilon(one))) then

 write (*,*) 'Example 3 for LIN_SVD is correct.'
end if

end

```

#### Example 4: Ridge Regression as Cross-Validation with Weighting

This example illustrates a particular choice for the *ridge regression* problem: The least-squares problem  $Ax \cong b$  is modified by the addition of a regularizing term to become

$$\min_x \left( \|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2 \right)$$

The solution to this problem, with row  $k$  deleted, is denoted by  $x_k(\lambda)$ . Using nonnegative weights  $(w_1, \dots, w_m)$ , the *cross-validation squared error*  $C(\lambda)$  is given by:

$$mC(\lambda) = \sum_{k=1}^m w_k \left( a_k^T x_k(\lambda) - b_k \right)^2$$

With the SVD  $A = USV^T$  and product  $g = U^T b$ , this quantity can be written as

$$mC(\lambda) = \sum_{k=1}^m w_k \left( \frac{\left( b_k - \sum_{j=1}^n u_{kj} g_j \frac{s_j^2}{(s_j^2 + \lambda^2)} \right)}{\left( 1 - \sum_{j=1}^n u_{kj}^2 \frac{s_j^2}{(s_j^2 + \lambda^2)} \right)} \right)^2$$

This expression is minimized. See Golub and Van Loan (1989, Chapter 12) for more details. In the Example 4 code,  $mC(\lambda)$ , at  $p = 10$  grid points are evaluated using a log-scale with respect to  $\lambda$ ,  $0.1s_1 \leq \lambda \leq 10s_1$ . Array operations and intrinsics are used to evaluate the function and then to choose an approximate minimum. Following the computation of the optimum  $\lambda$ , the regularized solutions are computed. Also, see `operator_ex24`, supplied with the product examples.

```

use lin_svd_int
use rand_gen_int

implicit none

! This is Example 4 for LIN_SVD.

integer i
integer, parameter :: m=32, n=16, p=10, k=4
real(kind(ld0)), parameter :: one=1d0
real(kind(ld0)) log_lamda, log_lamda_t, delta_log_lamda
real(kind(ld0)) a(m,n), b(m,k), w(m,k), g(m,k), t(n), s(n), &
 s_sq(n), u(m,m), v(n,n), y(m*max(n,k)), &
 c_lamda(p,k), lamda(k), x(n,k), res(n,k)

! Generate random rectangular matrices for A and right-hand
! sides, b.
call rand_gen(y)
a = reshape(y, (/m,n/))

call rand_gen(y)
b = reshape(y, (/m,k/))

! Generate random weights for each of the right-hand sides.
call rand_gen(y)
w = reshape(y, (/m,k/))

! Compute the singular value decomposition.
call lin_svd(a, s, u, v)

g = matmul(transpose(u),b)
s_sq = s**2

log_lamda = log(10.*s(1)); log_lamda_t=log_lamda
delta_log_lamda = (log_lamda - log(0.1*s(n))) / (p-1)

! Choose lamda to minimize the "cross-validation" weighted
! square error. First evaluate the error at a grid of points,
! uniform in log_scale.

```

```

cross_validation_error: do i=1, p
 t = s_sq/(s_sq+exp(log_lamda))
 c_lamda(i,:) = sum(w*((b-matmul(u(1:m,1:n),g(1:n,1:k))* &
 spread(t,DIM=2,NCOPIES=k)))/ &
 (one-matmul(u(1:m,1:n)**2, &
 spread(t,DIM=2,NCOPIES=k)))**2,DIM=1)
 log_lamda = log_lamda - delta_log_lamda
end do cross_validation_error

! Compute the grid value and lamda corresponding to the minimum.
do i=1, k
 lamda(i) = exp(log_lamda_t - delta_log_lamda* &
 (sum(minloc(c_lamda(1:p,i)))-1))
end do

! Compute the solution using the optimum "cross-validation"
! parameter.
x = matmul(v,g(1:n,1:k)*spread(s,DIM=2,NCOPIES=k)/ &
 (spread(s_sq,DIM=2,NCOPIES=k)+ &
 spread(lamda,DIM=1,NCOPIES=n)))
! Check the residuals, using normal equations.
res = matmul(transpose(a),b-matmul(a,x)) - &
 spread(lamda,DIM=1,NCOPIES=n)*x
if (sum(abs(res))/sum(s_sq) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_SVD is correct.'
end if

end

```

## Output

Example 4 for LIN\_SVD is correct.

---

# Parallel Constrained Least-Squares Solvers

## Solving Constrained Least-Squares Systems

The routine `PARALLEL_NONNEGATIVE_LSQ` is used to solve dense least-squares systems. These are represented by  $Ax \cong b$  where  $A$  is an  $m \times n$  coefficient data matrix,  $b$  is a given right-hand side  $m$ -vector, and  $x$  is the solution  $n$ -vector being computed. Further, there is a constraint requirement,  $x \geq 0$ . The routine `PARALLEL_BOUNDED_LSQ` is used when the problem has lower and upper bounds for the solution,  $\alpha \leq x \leq \beta$ . By making the bounds large, individual constraints can be eliminated. There are no restrictions on the relative sizes of  $m$  and  $n$ . When  $n$  is large, these codes can substantially reduce computer time and storage requirements, compared with using a routine for solving a constrained system and a single processor.

The user provides the matrix partitioned by blocks of columns:

$$A = [A_1 | A_2 | \dots | A_k]$$

An individual block of the partitioned matrix, say  $A_p$ , is located entirely on the processor with rank  $\text{MP\_RANK} = p - 1$ , where  $\text{MP\_RANK}$  is packaged in the module `MPI_SETUP_INT`. This module, and the function `MP_SETUP()`, define the Fortran Library MPI communicator, `MP_LIBRARY_WORLD`. See Chapter 10, [Dense Matrix Parallelism Using MPI](#).

---

## PARALLEL\_NONNEGATIVE\_LSQ




---

For a detailed description of MPI Requirements see “[Dense Matrix Parallelism Using MPI](#)” in Chapter 10 of this manual.

---

Solves a linear, non-negative constrained least-squares system.

### Usage Notes

```
CALL PARALLEL_NONNEGATIVE_LSQ (A, B, X, RNORM, W, INDEX, IPART, IOPT
 = IOPT)
```

### Required Arguments

**$A(I:M,:)$** — (Input/Output) Columns of the matrix with limits given by entries in the array `IPART(1:2, 1:max(1, MP_NPROCS))`. On output  $A_k$  is replaced by the product  $QA_k$ , where  $Q$  is an orthogonal matrix. The value `SIZE(A, 1)` defines the value of  $M$ . Each processor starts and exits with its piece of the partitioned matrix.

**$B(I:M)$**  — (Input/Output) Assumed-size array of length  $M$  containing the right-hand side vector,  $b$ . On output  $b$  is replaced by the product  $Qb$ , where  $Q$  is the orthogonal matrix applied to  $A$ . All processors in the communicator start and exit with the same vector.

**$X(I:N)$**  — (Output) Assumed-size array of length  $N$  containing the solution,  $x \geq 0$ . The value `SIZE(X)` defines the value of  $N$ . All processors exit with the same vector.

**$RNORM$**  — (Output) Scalar that contains the Euclidean or least-squares length of the residual vector,  $\|Ax - b\|$ . All processors exit with the same value.

**$W(I:N)$**  — (Output) Assumed-size array of length  $N$  containing the dual vector,  $w = A^T (b - Ax) \leq 0$ . All processors exit with the same vector.

**INDEX(1:N)** — (Output) Assumed-size array of length *N* containing the *NSETP* indices of columns in the positive solution, and the remainder that are at their constraint. The number of positive components in the solution *x* is given by the Fortran intrinsic function value,  
*NSETP*=COUNT(*X* > 0) . All processors exit with the same array.

**IPART(1:2,1:max(1,MP\_NPROCS))** — (Input) Assumed-size array containing the partitioning describing the matrix *A* . The value *MP\_NPROCS* is the number of processors in the communicator, except when MPI has been finalized with a call to the routine *MP\_SETUP('Final')*. This causes *MP\_NPROCS* to be assigned 0. Normally users will give the partitioning to processor of rank = *MP\_RANK* by setting *IPART*(1,*MP\_RANK*+1) = first column index, and *IPART*(2,*MP\_RANK*+1) = last column index. The number of columns per node is typically based on their relative computing power. To avoid a node with rank *MP\_RANK* doing any work except communication, set *IPART*(1,*MP\_RANK*+1) = 0 and *IPART*(2,*MP\_RANK*+1) = -1. In this exceptional case there is no reference to the array *A(:,:)* at that node.

### Optional Argument

**IOPT(:)**— (Input) Assumed-size array of derived type *S\_OPTIONS* or *D\_OPTIONS*. This argument is used to change internal parameters of the algorithm. Normally users will not be concerned about this argument, so they would not include it in the argument list for the routine.

| Packaged Options for <i>PARALLEL_NONNEGATIVE_LSQ</i> |              |
|------------------------------------------------------|--------------|
| Option Name                                          | Option Value |
| <i>P-NLSQ_SET_TOLERANCE</i>                          | 1            |
| <i>P-NLSQ_SET_MAX_ITERATIONS</i>                     | 2            |
| <i>P-NLSQ_SET_MIN_RESIDUAL</i>                       | 3            |

*IOPT*(*IO*)=?\_OPTIONS(*P-NLSQ\_SET\_TOLERANCE*, *TOLERANCE*) Replaces the default rank tolerance for using a column, from *EPSILON(TOLERANCE)* to *TOLERANCE*. Increasing the value of *TOLERANCE* will cause fewer columns to be moved from their constraints, and may cause the minimum residual *RNORM* to increase.

*IOPT*(*IO*)=?\_OPTIONS(*P-NLSQ\_SET\_MIN\_RESIDUAL*, *RESID*) Replaces the default target for the minimum residual vector length from 0 to *RESID*. Increasing the value of *RESID* can result in fewer iterations and thus increased efficiency. The descent in the optimization will stop at the first point where the minimum residual *RNORM* is smaller than *RESID*. Using this option may result in the dual vector not satisfying its optimality conditions, as noted above.

*IOPT*(*IO*)= *P-NLSQ\_SET\_MAX\_ITERATIONS*

IOPT(IO+1) = NEW\_MAX\_ITERATIONS Replaces the default maximum number of iterations from 3\*N to NEW\_MAX\_ITERATIONS. Note that this option requires two entries in the derived type array.

#### FORTRAN 90 Interface

Generic: CALL PARALLEL\_NONNEGATIVE\_LSQ (A, B, X, RNORM, W, INDEX, IPART [, ...])

Specific: The specific interface names are S\_PARALLEL\_NONNEGATIVE\_LSQ and D\_PARALLEL\_NONNEGATIVE\_LSQ.

### Description

Subroutine PARALLEL\_NONNEGATIVE\_LSQ solves the linear least-squares system  $Ax \cong b$ ,  $x \geq 0$ , using the algorithm *NNLS* found in Lawson and Hanson, (1995), pages 160-161. The code now updates the dual vector  $w$  of Step 2, page 161. The remaining new steps involve exchange of required data, using MPI.

### Example 1: Distributed Linear Inequality Constraint Solver

The program PNLISQ\_EX1 illustrates the computation of the minimum Euclidean length solution of an  $m' \times n'$  system of linear inequality constraints,  $Gy \geq h$ . The solution algorithm is based on Algorithm *LDP*, page 165-166, *loc. cit.* The rows of  $E = [G : h]$  are partitioned and assigned random values. When the minimum Euclidean length solution to the inequalities has been calculated, the residuals  $r = Gy - h \geq 0$  are computed, with the dual variables to the *NNLS* problem indicating the entries of  $r$  that are precisely zero.

The fact that matrix products involving both  $E$  and  $E^T$  are needed to compute the constrained solution  $y$  and the residuals  $r$ , implies that message passing is required. This occurs after the *NNLS* solution is computed.

```

PROGRAM PNLISQ_EX1
! Use Parallel_nonnegative_LSQ to solve an inequality
! constraint problem, Gy >= h. This algorithm uses
! Algorithm LDP of Solving Least Squares Problems,
! page 165. The constraints are allocated to the
! processors, by rows, in columns of the array A(:,:).
 USE PNLISQ_INT
 USE MPI_SETUP_INT
 USE RAND_INT
 USE SHOW_INT

 IMPLICIT NONE
 INCLUDE "mpif.h"

 INTEGER, PARAMETER :: MP=500, NP=400, M=NP+1, N=MP

 REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0
 REAL(KIND(1D0)), ALLOCATABLE :: &

```

```

 A(:, :), B(:), X(:), Y(:), W(:), ASAVE(:, :)
REAL(KIND(1D0)) RNORM
INTEGER, ALLOCATABLE :: INDEX(:), IPART(:, :)

INTEGER K, L, DN, J, JSHIFT, IERROR
LOGICAL :: PRINT=.false.

! Setup for MPI:
MP_NPROCS=MP_SETUP()

DN=N/max(1,max(1,MP_NPROCS))-1
ALLOCATE(IPART(2,max(1,MP_NPROCS)))

! Spread constraint rows evenly to the processors.
IPART(1,1)=1
DO L=2,MP_NPROCS
 IPART(2,L-1)=IPART(1,L-1)+DN
 IPART(1,L)=IPART(2,L-1)+1
END DO
IPART(2,MP_NPROCS)=N

! Define the constraint data using random values.
K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
ALLOCATE(A(M,K), ASAVE(M,K), X(N), W(N), &
 B(M), Y(M), INDEX(N))

! The use of ASAVE can be removed by regenerating
! the data for A(:, :) after the return from
! Parallel_nonnegative_LSQ.
A=rand(A); ASAVE=A
IF(MP_RANK == 0 .and. PRINT) &
 CALL SHOW(IPART, &
 "Partition of the constraints to be solved")

! Set the right-hand side to be one in the last component, zero elsewhere.
B=ZERO;B(M)=ONE

! Solve the dual problem.
CALL Parallel_nonnegative_LSQ &
 (A, B, X, RNORM, W, INDEX, IPART)

! Each processor multiplies its block times the part of
! the dual corresponding to that part of the partition.
Y=ZERO
DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
 JSHIFT=J-IPART(1,MP_RANK+1)+1
 Y=Y+ASAVE(:,JSHIFT)*X(J)
END DO

! Accumulate the pieces from all the processors. Put sum into B(:)
! on rank 0 processor.
B=Y
IF(MP_NPROCS > 1) &
 CALL MPI_REDUCE(Y, B, M, MPI_DOUBLE_PRECISION, &
 MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
IF(MP_RANK == 0) THEN

```

```

! Compute constrained solution at the root.
! The constraints will have no solution if B(M) = ONE.
! All of these example problems have solutions.
 B(M)=B(M)-ONE;B=-B/B(M)
 END IF

! Send the inequality constraint solution to all nodes.
 IF(MP_NPROCS > 1) &
 CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION, &
 0, MP_LIBRARY_WORLD, IERROR)

! For large problems this printing needs to be removed.
 IF(MP_RANK == 0 .and. PRINT) &
 CALL SHOW(B(1:NP), &
 "Minimal length solution of the constraints")

! Compute residuals of the individual constraints.
! If only the solution is desired, the program ends here.
 X=ZERO
 DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
 JSHIFT=J-IPART(1,MP_RANK+1)+1
 X(J)=dot_product(B,ASAVE(:,JSHIFT))
 END DO

! This cleans up residuals that are about rounding
! error unit (times) the size of the constraint
! equation and right-hand side. They are replaced
! by exact zero.
 WHERE(W == ZERO) X=ZERO; W=X

! Each group of residuals is disjoint, per processor.
! We add all the pieces together for the total set of
! constraints.
 IF(MP_NPROCS > 1) &
 CALL MPI_REDUCE(X, W, N, MPI_DOUBLE_PRECISION,&
 MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
 IF(MP_RANK == 0 .and. PRINT) &
 CALL SHOW(W, "Residuals for the constraints")

! See to any errors and shut down MPI.
 MP_NPROCS=MP_SETUP('Final')
 IF(MP_RANK == 0) THEN
 IF(COUNT(W < ZERO) == 0) WRITE(*,*)&
 " Example 1 for PARALLEL_NONNEGATIVE_LSQ is correct."
 END IF
 END

```

## Output

Example 1 for PARALLEL\_NONNEGATIVE\_LSQ is correct.

## Additional Examples

### Example 2: Distributed Non-negative Least-Squares

The program `PNLSQ_EX2` illustrates the computation of the solution to a system of linear least-squares equations with simple constraints:  $a_i^T x \cong b_i, i = 1, \dots, m$ , subject to  $x \geq 0$ . In this example we write the row vectors  $\begin{bmatrix} a_i^T & b_i \end{bmatrix}$  on a file. This illustrates reading the data by rows and arranging the data by columns, as required by `PARALLEL_NONNEGATIVE_LSQ`. After reading the data, the right-hand side vector is broadcast to the group before computing a solution,  $x$ . The block-size is chosen so that each participating processor receives the same number of columns, except any remaining columns sent to the processor with largest rank. This processor contains the right-hand side before the broadcast.

This example illustrates connecting a *BLACS* ‘context’ handle and the Fortran Library MPI communicator, `MP_LIBRARY_WORLD`, described in [Chapter 10](#).

```
PROGRAM PNLSQ_EX2
! Use Parallel_Nonnegative_LSQ to solve a least-squares
! problem, A x = b, with x >= 0. This algorithm uses a
! distributed version of NNLS, found in the book
! Solving Least Squares Problems, page 165. The data is
! read from a file, by rows, and sent to the processors,
! as array columns.

USE PNLSQ_INT
USE SCALAPACK_IO_INT
USE BLACS_INT

USE MPI_SETUP_INT
USE RAND_INT
USE ERROR_OPTION_PACKET

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: M=128, N=32, NP=N+1, NIN=10

real(kind(ld0)), ALLOCATABLE, DIMENSION(:) :: &
 d_A(:, :), A(:, :), B, C, W, X, Y
real(kind(ld0)) RNORM, ERROR
INTEGER, ALLOCATABLE :: INDEX(:), IPART(:, :)

INTEGER I, J, K, L, DN, JSHIFT, IERROR, &
 CONTXT, NPROW, MYROW, MYCOL, DESC_A(9)
TYPE(d_OPTIONS) IOPT(1)

! Routines with the "BLACS_" prefix are from the
! BLACS library.
CALL BLACS_PINFO(MP_RANK, MP_NPROCS)

! Make initialization for BLACS.
CALL BLACS_GET(0, 0, CONTXT)
```

```

! Define processor grid to be 1 by MP_NPROCS.
 NPROW=1
 CALL BLACS_GRIDINIT(CONTXT, 'N/A', NPROW, MP_NPROCS)

! Get this processor's role in the process grid.
 CALL BLACS_GRIDINFO(CONTXT, NPROW, MP_NPROCS, &
 MYROW, MYCOL)

! Connect BLACS context with communicator MP_LIBRARY_WORLD.
 CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

! Setup for MPI:
 MP_NPROCS=MP_SETUP()

 DN=max(1, NP/MP_NPROCS)
 ALLOCATE(IPART(2, MP_NPROCS))

! Spread columns evenly to the processors. Any odd
! number of columns are in the processor with highest
! rank.
 IPART(1, :)=1; IPART(2, :)=0
 DO L=2, MP_NPROCS
 IPART(2, L-1)=IPART(1, L-1)+DN
 IPART(1, L)=IPART(2, L-1)+1
 END DO
 IPART(2, MP_NPROCS)=NP
 IPART(2, :)=min(NP, IPART(2, :))

! Note which processor (L-1) receives the right-hand side.
 DO L=1, MP_NPROCS
 IF(IPART(1, L) <= NP .and. NP <= IPART(2, L)) EXIT
 END DO

 K=max(0, IPART(2, MP_RANK+1)-IPART(1, MP_RANK+1)+1)
 ALLOCATE(d_A(M, K), W(N), X(N), Y(N), &
 B(M), C(M), INDEX(N))

 IF(MP_RANK == 0) THEN
 ALLOCATE(A(M, N))
! Define the matrix data using random values.
 A=rand(A); B=rand(B)

! Write the rows of data to an external file.
 OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')
 DO I=1, M
 WRITE(NIN, *) (A(I, J), J=1, N), B(I)
 END DO
 CLOSE(NIN)
 ELSE

! No resources are used where this array is not saved.
 ALLOCATE(A(M, 0))
 END IF

! Define the matrix descriptor. This includes the
! right-hand side as an additional column. The row

```

```

! block size, on each processor, is arbitrary, but is
! chosen here to match the column block size.
DESC_A=(/1, CONTXT, M, NP, DN+1, DN+1, 0, 0, M/)

! Read the data by rows.
IOPT(1)=ScaLAPACK_READ_BY_ROWS
CALL ScaLAPACK_READ ("Atest.dat", DESC_A, &
d_A, IOPT=IOPT)

! Broadcast the right-hand side to all processors.
JSHIFT=NP-IPART(1,L)+1
IF(K > 0) B=d_A(:,JSHIFT)
IF(MP_NPROCS > 1) &
CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION, L-1, &
MP_LIBRARY_WORLD, IERROR)

! Adjust the partition of columns to ignore the
! last column, which is the right-hand side. It is
! now moved to B(:).
IPART(2,:)=min(N,IPART(2,:))

! Solve the constrained distributed problem.
C=B
CALL Parallel_Nonnegative_LSQ &
(d_A, B, X, RNORM, W, INDEX, IPART)

! Solve the problem on one processor, with data saved
! for a cross-check.
IPART(2,:)=0; IPART(2,1)=N; MP_NPROCS=1

! Since all processors execute this code, all arrays
! must be allocated in the main program.
CALL Parallel_Nonnegative_LSQ &
(A, C, Y, RNORM, W, INDEX, IPART)

! See to any errors.
CALL elpop("Mp_Setup")

! Check the differences in the two solutions. Unique solutions
! may differ in the last bits, due to rounding.
IF(MP_RANK == 0) THEN
ERROR=SUM(ABS(X-Y))/SUM(Y)
IF(ERROR <= sqrt(EPSILON(ERROR))) write(*,*) &
' Example 2 for PARALLEL_NONNEGATIVE_LSQ is correct.'
OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
CLOSE(NIN, STATUS='Delete')
END IF

! Exit from using this process grid.
CALL BLACS_GRIDEXIT(CONTXT)
CALL BLACS_EXIT(0)
END

```

## Output

Example 2 for PARALLEL\_NONNEGATIVE\_LSQ is correct.'

---

## PARALLEL\_BOUNDED\_LSQ



---

For a detailed description of MPI Requirements see “[Dense Matrix Parallelism Using MPI](#)” in Chapter 10 of this manual.

---

Solves a linear least-squares system with bounds on the unknowns.

### Usage Notes

```
CALL PARALLEL_BOUNDED_LSQ (A, B, BND, X, RNORM, W, INDEX, IPART,
 NSETP, NSETZ, IOPT=IOPT)
```

### Required Arguments

**A(I:M,:)**— (Input/Output) Columns of the matrix with limits given by entries in the array `IPART(1:2, 1:max(1,MP_NPROCS))`. On output  $A_k$  is replaced by the product  $QA_k$ , where  $Q$  is an orthogonal matrix. The value `SIZE(A, 1)` defines the value of  $M$ . Each processor starts and exits with its piece of the partitioned matrix.

**B(I:M)** — (Input/Output) Assumed-size array of length  $M$  containing the right-hand side vector,  $b$ . On output  $b$  is replaced by the product  $Q(b - Ag)$ , where  $Q$  is the orthogonal matrix applied to  $A$  and  $g$  is a set of active bounds for the solution. All processors in the communicator start and exit with the same vector.

**BND(I:2,I:N)** — (Input) Assumed-size array containing the bounds for  $x$ . The lower bound  $\alpha_j$  is in `BND(1, J)`, and the upper bound  $\beta_j$  is in `BND(2, J)`.

**X(I:N)** — (Output) Assumed-size array of length  $N$  containing the solution,  $\alpha \leq x \leq \beta$ . The value `SIZE(X)` defines the value of  $N$ . All processors exit with the same vector.

**RNORM** — (Output) Scalar that contains the Euclidean or least-squares length of the residual vector,  $\|Ax - b\|$ . All processors exit with the same value.

**W(I:N)** — (Output) Assumed-size array of length  $N$  containing the dual vector,  $w = A^T(b - Ax)$ . At a solution exactly one of the following is true for each  $j, 1 \leq j \leq n$ ,

- $\alpha_j = x_j = \beta_j$ , and  $w_j$  arbitrary
- $\alpha_j = x_j$ , and  $w_j \leq 0$
- $x_j = \beta_j$ , and  $w_j \geq 0$
- $\alpha_j < x_j < \beta_j$ , and  $w_j = 0$

All processors exit with the same vector.

**INDEX(1:N)** — (Output) Assumed-size array of length  $N$  containing the **NSETP** indices of columns in the solution interior to bounds, and the remainder that are at a constraint. All processors exit with the same array.

**IPART(1:2,1:max(1,MP\_NPROCS))** — (Input) Assumed-size array containing the partitioning describing the matrix  $A$ . The value **MP\_NPROCS** is the number of processors in the communicator, except when **MPI** has been finalized with a call to the routine **MP\_SETUP('Final')**. This causes **MP\_NPROCS** to be assigned 0. Normally users will give the partitioning to processor of rank = **MP\_RANK** by setting **IPART(1,MP\_RANK+1)** = first column index, and **IPART(2,MP\_RANK+1)** = last column index. The number of columns per node is typically based on their relative computing power. To avoid a node with rank **MP\_RANK** doing any work except communication, set **IPART(1,MP\_RANK+1) = 0** and **IPART(2,MP\_RANK+1) = -1**. In this exceptional case there is no reference to the array **A(:,:)** at that node.

**NSETP**— (Output) An **INTEGER** indicating the number of solution components not at constraints. The column indices are output in the array **INDEX(:)**.

**NSETZ**— (Output) An **INTEGER** indicating the solution components held at fixed values. The column indices are output in the array **INDEX(:)**.

## Optional Argument

**IOPT(:)**— (Input) Assumed-size array of derived type **S\_OPTIONS** or **D\_OPTIONS**. This argument is used to change internal parameters of the algorithm. Normally users will not be concerned about this argument, so they would not include it in the argument list for the routine.

| Packaged Options for <b>PARALLEL_BOUNDED_LSQ</b> |              |
|--------------------------------------------------|--------------|
| Option Name                                      | Option Value |
| <b>PBLSQ_SET_TOLERANCE</b>                       | 1            |
| <b>PBLSQ_SET_MAX_ITERATIONS</b>                  | 2            |
| <b>PBLSQ_SET_MIN_RESIDUAL</b>                    | 3            |

IOPT(IO)=?\_OPTIONS(PBLSQ\_SET\_TOLERANCE, TOLERANCE) Replaces the default rank tolerance for using a column, from EPSILON(TOLERANCE) to TOLERANCE. Increasing the value of TOLERANCE will cause fewer columns to be increased from their constraints, and may cause the minimum residual RNORM to increase.

IOPT(IO)=?\_OPTIONS(PBLSQ\_SET\_MIN\_RESIDUAL, RESID) Replaces the default target for the minimum residual vector length from 0 to RESID. Increasing the value of RESID can result in fewer iterations and thus increased efficiency. The descent in the optimization will stop at the first point where the minimum residual RNORM is smaller than RESID. Using this option may result in the dual vector not satisfying its optimality conditions, as noted above.

IOPT(IO) = PBLSQ\_SET\_MAX\_ITERATIONS

IOPT(IO+1) = NEW\_MAX\_ITERATIONS Replaces the default maximum number of iterations from 3\*N to NEW\_MAX\_ITERATIONS. Note that this option requires two entries in the derived type array.

## FORTRAN 90 Interface

Generic: CALL PARALLEL\_BOUNDED\_LSQ (A, B, X [, ...])

Specific: The specific interface names are S\_PARALLEL\_BOUNDED\_LSQ and D\_PARALLEL\_BOUNDED\_LSQ.

## Description

Subroutine PARALLEL\_BOUNDED\_LSQ solves the least-squares linear system

$Ax \cong b$ ,  $\alpha \leq x \leq \beta$ , using the algorithm *BVLS* found in Lawson and Hanson, (1995), pages 279-283. The new steps involve updating the dual vector and exchange of required data, using MPI. The optional changes to default tolerances, minimum residual, and the number of iterations are new features.

## Example 1: Distributed Equality and Inequality Constraint Solver

The program PBLSQ\_EX1 illustrates the computation of the minimum Euclidean length solution of an  $m' \times n'$  system of linear inequality constraints,  $Gy \geq h$ . Additionally the first  $f > 0$  of the constraints are equalities. The solution algorithm is based on Algorithm *LDP*, page 165-166, *loc. cit.* By allowing the dual variables to be free, the constraints become equalities. The rows of  $E = [G : h]$  are partitioned and assigned random values. When the minimum Euclidean length solution to the inequalities has been calculated, the residuals  $r = Gy - h \geq 0$  are computed, with the dual variables to the *BVLS* problem indicating the entries of  $r$  that are exactly zero.

```
PROGRAM PBLSQ_EX1
! Use Parallel_bounded_LSQ to solve an inequality
! constraint problem, Gy >= h. Force F of the constraints
! to be equalities. This algorithm uses LDP of
! Solving Least Squares Problems, page 165.
```

```

! Forcing equality constraints by freeing the dual is
! new here. The constraints are allocated to the
! processors, by rows, in columns of the array A(:,:).
 USE PBLSQ_INT
 USE MPI_SETUP_INT
 USE RAND_INT
 USE SHOW_INT

 IMPLICIT NONE
 INCLUDE "mpif.h"

 INTEGER, PARAMETER :: MP=500, NP=400, M=NP+1, &
 N=MP, F=NP/10

 REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0
 REAL(KIND(1D0)), ALLOCATABLE :: &
 A(:,:), B(:), BND(:,:), X(:), Y(:), &
 W(:), ASAVE(:,:)
 REAL(KIND(1D0)) RNORM
 INTEGER, ALLOCATABLE :: INDEX(:), IPART(:,:)

 INTEGER K, L, DN, J, JSHIFT, IERROR, NSETP, NSETZ
 LOGICAL :: PRINT=.false.

! Setup for MPI:
 MP_NPROCS=MP_SETUP()

 DN=N/max(1,max(1,MP_NPROCS))-1
 ALLOCATE(IPART(2,max(1,MP_NPROCS)))

! Spread constraint rows evenly to the processors.
 IPART(1,1)=1
 DO L=2,MP_NPROCS
 IPART(2,L-1)=IPART(1,L-1)+DN
 IPART(1,L)=IPART(2,L-1)+1
 END DO
 IPART(2,MP_NPROCS)=N

! Define the constraints using random data.
 K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
 ALLOCATE(A(M,K), ASAVE(M,K), BND(2,N), &
 X(N), W(N), B(M), Y(M), INDEX(N))

! The use of ASAVE can be replaced by regenerating the
! data for A(:,:) after the return from
! Parallel_bounded_LSQ
 A=rand(A); ASAVE=A
 IF(MP_RANK == 0 .and. PRINT) &
 call show(IPART,&
 "Partition of the constraints to be solved")

! Set the right-hand side to be one in the last
! component, zero elsewhere.
 B=ZERO;B(M)=ONE

! Solve the dual problem. Letting the dual variable

```

```

! have no constraint forces an equality constraint
! for the primal problem.
 BND(1,1:F)=-HUGE(ONE); BND(1,F+1:)=ZERO
 BND(2,:)=HUGE(ONE)
 CALL Parallel_bounded_LSQ &
 (A, B, BND, X, RNORM, W, INDEX, IPART, &
 NSETP, NSETZ)

! Each processor multiplies its block times the part
! of the dual corresponding to that partition.
 Y=ZERO
 DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
 JSHIFT=J-IPART(1,MP_RANK+1)+1
 Y=Y+ASAVE(:,JSHIFT)*X(J)
 END DO

! Accumulate the pieces from all the processors.
! Put sum into B(:) on rank 0 processor.
 B=Y
 IF(MP_NPROCS > 1) &
 CALL MPI_REDUCE(Y, B, M, MPI_DOUBLE_PRECISION,&
 MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
 IF(MP_RANK == 0) THEN

! Compute constraint solution at the root.
! The constraints will have no solution if B(M) = ONE.
! All of these example problems have solutions.
 B(M)=B(M)-ONE;B=-B/B(M)
 END IF

! Send the inequality constraint or primal solution to all nodes.
 IF(MP_NPROCS > 1) &
 CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION, 0, &
 MP_LIBRARY_WORLD, IERROR)

! For large problems this printing may need to be removed.
 IF(MP_RANK == 0 .and. PRINT) &
 call show(B(1:NP), &
 "Minimal length solution of the constraints")

! Compute residuals of the individual constraints.
 X=ZERO
 DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
 JSHIFT=J-IPART(1,MP_RANK+1)+1
 X(J)=dot_product(B,ASAVE(:,JSHIFT))
 END DO

! This cleans up residuals that are about rounding error
! unit (times) the size of the constraint equation and
! right-hand side. They are replaced by exact zero.
 WHERE(W == ZERO) X=ZERO
 W=X

! Each group of residuals is disjoint, per processor.
! We add all the pieces together for the total set of
! constraints.
 IF(MP_NPROCS > 1) &

```

```

CALL MPI_REDUCE(X, W, N, MPI_DOUBLE_PRECISION, &
 MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
IF(MP_RANK == 0 .and. PRINT) &
 call show(W, "Residuals for the constraints")

! See to any errors and shut down MPI.
MP_NPROCS=MP_SETUP('Final')
IF(MP_RANK == 0) THEN
 IF(COUNT(W < ZERO) == 0 .and.&
 COUNT(W == ZERO) >= F) WRITE(*,*)&
 " Example 1 for PARALLEL_BOUNDED_LSQ is correct."
 END IF
END
END

```

## Output

Example 1 for PARALLEL\_BOUNDED\_LSQ is correct.

## Additional Examples

### Example 2: Distributed Newton-Raphson Method with Step Control

The program `PBLSQ_EX2` illustrates the computation of the solution of a non-linear system of equations. We use a constrained Newton-Raphson method.

This algorithm works with the problem chosen for illustration. The step-size control used here, employing only simple bounds, *may not work* on other non-linear systems of equations. Therefore we do not recommend the simple non-linear solving technique illustrated here for an *arbitrary* problem. The test case is *Brown's Almost Linear Problem*, Moré, et al. (1982). The components are given by:

$$\bullet f_i(x) = x_i + \sum_{j=1}^n x_j - (n+1), i = 1, \dots, n-1$$

$$\bullet f_n(x) = x_1 \dots x_n - 1$$

The functions are zero at the point  $x = (\delta, \dots, \delta, \delta^{1-n})^T$ , where  $\delta > 1$  is a particular root of the polynomial equation  $n\delta^n - (n+1)\delta^{n-1} + 1 = 0$ . To avoid convergence to the local minimum

$x = (0, \dots, 0, n+1)^T$ , we start at the standard point  $x = (1/2, \dots, 1/2, 1/2)^T$  and develop the

Newton method using the linear terms  $f(x-y) \approx f(x) - J(x)y \cong 0$ , where  $J(x)$  is the

Jacobian matrix. The update is constrained so that the first  $n-1$  components satisfy

$x_j - y_j \geq 1/2$ , or  $y_j \leq x_j - 1/2$ . The last component is bounded from both sides,

$0 < x_n - y_n \leq 1/2$ , or  $x_n > y_n \geq (x_n - 1/2)$ . These bounds avoid the local minimum and

allow us to replace the last equation by  $\sum_{j=1}^n \ln(x_j) = 0$ , which is better scaled than the original.

The positive lower bound for  $x_n - y_n$  is replaced by the strict bound, `EPSILON(1D0)`, the arithmetic precision, which restricts the relative accuracy of  $x_n$ . The input for routine `PARALLEL_BOUNDED_LSQ` expects each processor to obtain that part of  $J(x)$  it owns. Those columns of the Jacobian matrix correspond to the partition given in the array `IPART(:, :)`. Here the columns of the matrix are evaluated, in parallel, on the nodes where they are required.

```

PROGRAM PBLSQ_EX2
! Use Parallel_bounded_LSQ to solve a non-linear system
! of equations. The example is an ACM-TOMS test problem,
! except for the larger size. It is "Brown's Almost Linear
! Function."
 USE ERROR_OPTION_PACKET
 USE PBLSQ_INT
 USE MPI_SETUP_INT
 USE SHOW_INT
 USE Numerical_Libraries, ONLY : N1RTY

 IMPLICIT NONE

 INTEGER, PARAMETER :: N=200, MAXIT=5

 REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0, &
 HALF=5D-1, TWO=2D0
 REAL(KIND(1D0)), ALLOCATABLE :: &
 A(:, :), B(:, :), BND(:, :), X(:), Y(:), W(:)
 REAL(KIND(1D0)) RNORM
 INTEGER, ALLOCATABLE :: INDEX(:), IPART(:, :)

 INTEGER K, L, DN, J, JSHIFT, IERROR, NSETP, &
 NSETZ, ITER
 LOGICAL :: PRINT=.false.
 TYPE(D_OPTIONS) IOPT(3)

! Setup for MPI:
 MP_NPROCS=MP_SETUP()

 DN=N/max(1,max(1,MP_NPROCS))-1
 ALLOCATE(IPART(2,max(1,MP_NPROCS)))

! Spread Jacobian matrix columns evenly to the processors.
 IPART(1,1)=1
 DO L=2,MP_NPROCS
 IPART(2,L-1)=IPART(1,L-1)+DN
 IPART(1,L)=IPART(2,L-1)+1
 END DO
 IPART(2,MP_NPROCS)=N

 K=max(0, IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
 ALLOCATE(A(N,K), BND(2,N), &
 X(N), W(N), B(N), Y(N), INDEX(N))

! This is Newton's method on "Brown's almost
! linear function."
 X=HALF

```

```

ITER=0

! Turn off messages and stopping for FATAL class errors.
CALL ERSET (4, 0, 0)

NEWTON_METHOD: DO

! Set bounds for the values after the step is taken.
! All variables are positive and bounded below by HALF,
! except for variable N, which has an upper bound of HALF.
BND(1,1:N-1)=-HUGE(ONE)
BND(2,1:N-1)=X(1:N-1)-HALF
BND(1,N)=X(N)-HALF
BND(2,N)=X(N)-EPSILON(ONE)

! Compute the residual function.
B(1:N-1)=SUM(X)+X(1:N-1)-(N+1)
B(N)=LOG(PRODUCT(X))
if(mp_rank == 0 .and. PRINT) THEN
CALL SHOW(B, &
"Developing non-linear function residual")
END IF
IF (MAXVAL(ABS(B(1:N-1))) <= SQRT(EPSILON(ONE)))&
EXIT NEWTON_METHOD

! Compute the derivatives local to each processor.
A(1:N-1,:)=ONE
DO J=1,N-1
IF(J < IPART(1,MP_RANK+1)) CYCLE
IF(J > IPART(2,MP_RANK+1)) CYCLE
JSHIFT=J-IPART(1,MP_RANK+1)+1
A(J,JSHIFT)=TWO
END DO
A(N,:)=ONE/X(IPART(1,MP_RANK+1):IPART(2,MP_RANK+1))

! Reset the linear independence tolerance.
IOPT(1)=D_OPTIONS(PBLSQ_SET_TOLERANCE,&
sqrt(EPSILON(ONE)))
IOPT(2)=PBLSQ_SET_MAX_ITERATIONS

! If N iterations was not enough on a previous iteration, reset to 2*N.
IF(N1RTY(1) == 0) THEN
IOPT(3)=N
ELSE
IOPT(3)=2*N
CALL E1POP('MP_SETUP')
CALL E1PSH('MP_SETUP')
END IF

CALL parallel_bounded_LSQ &
(A, B, BND, Y, RNORM, W, INDEX, IPART, NSETP, &
NSETZ, IOPT=IOPT)

! The array Y(:) contains the constrained Newton step.
! Update the variables.
X=X-Y

```

```

 IF(mp_rank == 0 .and. PRINT) THEN
 CALL show(BND, "Bounds for the moves")
 CALL SHOW(X, "Developing Solution")
 CALL SHOW((/RNORM/), &
 "Linear problem residual norm")
 END IF

! This is a safety measure for not taking too many steps.
ITER=ITER+1
IF(ITER > MAXIT) EXIT NEWTON_METHOD
END DO NEWTON_METHOD

IF(MP_RANK == 0) THEN
IF(ITER <= MAXIT) WRITE(*,*) &
" Example 2 for PARALLEL_BOUNDED_LSQ is correct."
END IF

! See to any errors and shut down MPI.
MP_NPROCS=MP_SETUP('Final')

END

```

---

## LSARG



Solves a real general system of linear equations with iterative refinement.

### Required Arguments

- A* —  $N$  by  $N$  matrix containing the coefficients of the linear system. (Input)
- B* — Vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X* — Vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

- N* — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .
- LDA* — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .
- IPATH* — Path indicator. (Input)

IPATH = 1 means the system  $AX = B$  is solved.

IPATH = 2 means the system  $A^T X = B$  is solved.

Default: IPATH = 1.

### **FORTRAN 90 Interface**

Generic:     CALL LSARG (A, B, X [, ...])

Specific:    The specific interface names are S\_LSARG and D\_LSARG.

### **FORTRAN 77 Interface**

Single:      CALL LSARG (N, A, LDA, B, IPATH, X)

Double:     The double precision name is DLSARG

### **ScaLAPACK Interface**

Generic:     CALL LSARG (A0, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSARG and D\_LSARG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

Routine LSARG solves a system of linear algebraic equations having a real general coefficient matrix. It first uses routine [LFCRG](#) to compute an  $LU$  factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using the iterative refinement routine [LFIRG](#). The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

LSARG fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. LSARG solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2ARG/DL2ARG. The reference is:

```
CALL L2ARG (N, A, LDA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** — Work vector of length  $N^2$  containing the *LU* factorization of *A* on output.

**IPVT** — Integer work vector of length *N* containing the pivoting information for the *LU* factorization of *A* on output.

**WK** — Work vector of length *N*.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
|---|---|------------------------------------------------------------------------------|

|   |   |                               |
|---|---|-------------------------------|
| 4 | 2 | The input matrix is singular. |
|---|---|-------------------------------|

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — *MXLDA* by *MXCOL* local matrix containing the local portions of the distributed matrix *A*. *A* contains the coefficients of the linear system. (Input)

**B0** — Local vector of length *MXLDA* containing the local portions of the distributed vector *B*. *B* contains the right-hand side of the linear system. (Input)

**X0** — Local vector of length *MXLDA* containing the local portions of the distributed vector *X*. *X* contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, *MXLDA* and *MXCOL* can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of three linear equations is solved. The coefficient matrix has real general form and the right-hand-side vector *b* has three elements.

```
USE LSARG_INT
USE WRRRN_INT
IMPLICIT NONE
```

```
!
```

```
Declare variables
```

```

INTEGER LDA, N
PARAMETER (LDA=3, N=3)

REAL A(LDA,N), B(N), X(N)
! Set values for A and B
A(1,:) = (/ 33.0, 16.0, 72.0/)
A(2,:) = (/ -24.0, -10.0, -57.0/)
A(3,:) = (/ 18.0, -11.0, 7.0/)
!
B = (/129.0, -96.0, 8.5/)

! Solve the system of equations
CALL LSARG (A, B, X)
! Print results
CALL WRRRN ('X', X, 1, N, 1)
END

```

## Output

```

 X
 1 2 3
1.000 1.500 1.000

```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has real general form and the right-hand-side vector  $b$  has three elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LSARG_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
! Declare variables
INTEGER N, DESCA(9), DESCX(9)
INTEGER INFO, MXLDA, MXCOL
REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
PARAMETER (N=3)
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF (MP_RANK .EQ. 0) THEN
 ALLOCATE (A(N,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 33.0, 16.0, 72.0/)
 A(2,:) = (/ -24.0, -10.0, -57.0/)
 A(3,:) = (/ 18.0, -11.0, 7.0/)
!
 B = (/129.0, -96.0, 8.5/)
ENDIF

```

```

! Set up a 1D processor grid and define
! its context id, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! AND MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
! Solve the system of equations
CALL LSARG (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRRRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
1.000 1.500 1.000

```

---

## LSLRG



Solves a real general system of linear equations without iterative refinement.

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficients of the linear system. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations

### Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

$IPATH$  — Path indicator. (Input)  
 $IPATH = 1$  means the system  $AX = B$  is solved.  
 $IPATH = 2$  means the system  $A^T X = B$  is solved.  
Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic:     `CALL LSLRG (A, B, X [, ...])`

Specific:    The specific interface names are `S_LSLRG` and `D_LSLRG`.

### FORTRAN 77 Interface

Single:      `CALL LSLRG (N, A, LDA, B, IPATH, X)`

Double:     The double precision name is `DLSLRG`.

### ScaLAPACK Interface

Generic:     `CALL LSLRG (A0, B0, X0 [, ...])`

Specific:    The specific interface names are `S_LSLRG` and `D_LSLRG`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### Description

Routine `LSLRG` solves a system of linear algebraic equations having a real general coefficient matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. `LSLRG` first uses the routine `LFCRG` to compute an  $LU$  factorization of the coefficient matrix based on Gauss elimination with partial pivoting. Experiments were analyzed to determine efficient implementations on several different computers. For some supercomputers, particularly

those with efficient vendor-supplied BLAS, versions that call Level 1, 2 and 3 BLAS are used. The remaining computers use a factorization method provided to us by Dr. Leonard J. Harding of the University of Michigan. Harding's work involves "loop unrolling and jamming" techniques that achieve excellent performance on many computers. Using an option, `LSLRG` will estimate the condition number of the matrix. The solution of the linear system is then found using `LFSRG`.

The routine `LSLRG` fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This occurs only if  $A$  is close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that small changes in  $A$  can cause large changes in the solution  $x$ . If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that either `LIN_SOL_SVD` or `LSARG` be used.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LRG/DL2LRG`. The reference is:

```
CALL L2LRG (N, A, LDA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing the  $LU$  factorization of  $A$  on output. If  $A$  is not needed,  $A$  and  $FACT$  can share the same storage locations. See Item 3 below to avoid memory bank conflicts.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  on output.

**WK** — Work vector of length  $N$ .

2. Informational errors

| Type | Code | Description                                                                  |
|------|------|------------------------------------------------------------------------------|
| 3    | 1    | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4    | 2    | The input matrix is singular.                                                |

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LRG` the leading dimension of `FACT` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`; respectively, in `LSLRG`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLRG`. Users directly calling `L2LRG` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing

applications that use LSLRG or L2LRG. Default values for the option are  
IVAL(\*) = 1, 16, 0, 1.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSLRG temporarily replaces IVAL(2) by IVAL(1). The routine L2CRG computes the condition number if IVAL(2) = 2. Otherwise L2CRG skips this computation. LSLRG restores the option. Default values for the option are  
IVAL(\*) = 1, 2.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the coefficients of the linear system. (Input)
- B0** — Local vector of length MXLDA containing the local portions of the distributed vector B. B contains the right-hand side of the linear system. (Input)
- X0** — Local vector of length MXLDA containing the local portions of the distributed vector X. X contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example 1

A system of three linear equations is solved. The coefficient matrix has real general form and the right-hand-side vector  $b$  has three elements.

```
USE LSLRG_INT
USE WRRRN_INT
IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)

REAL A(LDA,N), B(N), X(N)
!
! Set values for A and B
A(1,:) = (/ 33.0, 16.0, 72.0/)
A(2,:) = (/ -24.0, -10.0, -57.0/)
A(3,:) = (/ 18.0, -11.0, 7.0/)
!
B = (/129.0 -96.0 8.5/)

!
! Solve the system of equations
CALL LSLRG (A, B, X)
!
! Print results
```

```
CALL WRRRN ('X', X, 1, N, 1)
END
```

## Output

```
 X
 1 2 3
1.000 1.500 1.000
```

## Additional Example

### Example 2

A system of  $N = 16$  linear equations is solved using the routine L2LRG. The option manager is used to eliminate memory bank conflict inefficiencies that may occur when the matrix dimension is a multiple of 16. The leading dimension of  $FACT=A$  is increased from  $N$  to  $N+IVAL(3)=17$ , since  $N=16=IVAL(4)$ . The data used for the test is a nonsymmetric Hadamard matrix and a right-hand side generated by a known solution,  $x_j = j$ ,  $j = 1, \dots, N$ .

```
 USE L2LRG_INT
 USE IUMAG_INT
 USE WRRRN_INT
 USE SGEMV_INT
 IMPLICIT NONE

! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=17, N=16)

! SPECIFICATIONS FOR PARAMETERS
 INTEGER ICHP, IPATH, IPUT, KBANK
 REAL ONE, ZERO
 PARAMETER (ICHP=1, IPATH=1, IPUT=2, KBANK=16, ONE=1.0E0, &
 ZERO=0.0E0)

! SPECIFICATIONS FOR LOCAL VARIABLES
 INTEGER I, IPVT(N), J, K, NN
 REAL A(LDA,N), B(N), WK(N), X(N)

! SPECIFICATIONS FOR SAVE VARIABLES
 INTEGER IOPT(1), IVAL(4)
 SAVE IVAL

! Data for option values.
 DATA IVAL/1, 16, 1, 16/

! Set values for A and B:
 A(1,1) = ONE
 NN = 1

! Generate Hadamard matrix.
 DO 20 K=1, 4
 DO 10 J=1, NN
 DO 10 I=1, NN
 A(NN+I,J) = -A(I,J)
 A(I,NN+J) = A(I,J)
 A(NN+I,NN+J) = A(I,J)
10 CONTINUE
 NN = NN + NN
20 CONTINUE

! Generate right-hand-side.
 DO 30 J=1, N
```

```

 X(J) = J
30 CONTINUE
!
! Set B = A*X.
CALL SGEMV ('N', N, N, ONE, A, LDA, X, 1, ZERO, B, 1)
!
! Clear solution array.
 X = ZERO

!
! Set option to avoid memory
! bank conflicts.
 IOPT(1) = KBANK
CALL IUMAG ('MATH', ICHP, IPUT, 1, IOPT, IVAL)
!
! Solve A*X = B.
CALL L2LRG (N, A, LDA, B, IPATH, X, A, IPVT, WK)
!
! Print results
CALL WRRRN ('X', X, 1, N, 1)
END

```

## Output

```

 X
 1 2 3 4 5 6 7 8 9 10
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00

 11 12 13 14 15 16
11.00 12.00 13.00 14.00 15.00 16.00

```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has real general form and the right-hand-side vector  $b$  has three elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP (see [Chapter 11, “Utilities”](#)) are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LSLRG_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

!
! Declare variables
INTEGER N, DESCA(9), DESCX(9)
INTEGER INFO, MXCOL, MXLDA
REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
PARAMETER (N=3)

!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(N,N), B(N), X(N))
!
! Set values for A and B
 A(1,:) = (/ 33.0, 16.0, 72.0/)
 A(2,:) = (/ -24.0, -10.0, -57.0/)

```

```

 A(3,:) = (/ 18.0, -11.0, 7.0/)
!
 B = (/129.0, -96.0, 8.5/)
ENDIF
!
! Set up a 1D processor grid and define
! its context id, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
!
! Solve the system of equations
CALL LSLRG (A0, B0, X0)
!
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
!
! Print results
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0)CALL WRRRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
1.000 1.500 1.000

```

---

## LFCRG



Computes the  $LU$  factorization of a real general matrix and estimates its  $L_1$  condition number.

### Required Arguments

$A$  —  $N$  by  $N$  matrix to be factored. (Input)

**FACT** —  $N$  by  $N$  matrix containing the  $LU$  factorization of the matrix  $A$ . (Output)  
If  $A$  is not needed,  $A$  and **FACT** can share the same storage locations.

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization.  
(Output)

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of  $A$ .  
(Output)

### Optional Arguments

**$N$**  — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\text{FACT},1)$ .

### FORTRAN 90 Interface

Generic:     CALL LFCRG (A, FACT, IPVT, RCOND, [, ...])

Specific:    The specific interface names are S\_LFCRG and D\_LFCRG.

### FORTRAN 77 Interface

Single:      CALL LFCRG (N, A, LDA, FACT, LDFACT, IPVT, RCOND)

Double:     The double precision name is DLFCRG.

### ScaLAPACK Interface

Generic:     CALL LFCRG (A0, FACT0, IPVT0, RCOND [, ...])

Specific:    The specific interface names are S\_LFCRG and D\_LFCRG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### Description

Routine **LFCRG** performs an  $LU$  factorization of a real general coefficient matrix. It also estimates the condition number of the matrix. The underlying code is based on either LINPACK, LAPACK,

or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “Using ScaLAPACK, LAPACK, LINPACK, and EISPACK” in the Introduction section of this manual. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm. Otherwise, partial pivoting is used.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described in a paper by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCRG fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  either is singular or is very close to a singular matrix.

The  $LU$  factors are returned in a form that is compatible with routines LFIRG, LFSRG and LFD RG. To solve systems of equations with multiple right-hand-side vectors, use LFCRG followed by either LFIRG or LFSRG called once for each right-hand side. The routine LFD RG can be called to compute the determinant of the coefficient matrix after LFCRG has performed the factorization.

Let  $F$  be the matrix FACT and let  $p$  be the vector IPVT. The triangular matrix  $U$  is stored in the upper triangle of  $F$ . The strict lower triangle of  $F$  contains the information needed to reconstruct  $L$  using

$$L^{-1} = L_{N-1} P_{N-1} \dots L_1 P_1$$

where  $P_k$  is the identity matrix with rows  $k$  and  $p_k$  interchanged and  $L_k$  is the identity with  $F_{ik}$  for  $i = k + 1, \dots, N$  inserted below the diagonal. The strict lower half of  $F$  can also be thought of as containing the negative of the multipliers. LFCRG is based on the LINPACK routine SGECO; see Dongarra et al. (1979). SGECO uses unscaled partial pivoting.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2CRG/DL2CRG. The reference is:

```
CALL L2CRG (N, A, LDA, FACT, LDFACT, IPVT, RCOND, WK)
```

The additional argument is

**WK** — Work vector of length  $N$ .

2. Informational errors

Type Code

|   |   |                                               |
|---|---|-----------------------------------------------|
| 3 | 1 | The input matrix is algorithmically singular. |
| 4 | 2 | The input matrix is singular.                 |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the matrix to be factored. (Input)

**FACT0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix FACT. FACT contains the *LU* factorization of the matrix A. (Output)

**IPVT0** — Local vector of length MXLDA containing the local portions of the distributed vector IPVT. IPVT contains the pivoting information for the *LU* factorization. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFCRG is called to factor the matrix and to check for singularity or ill-conditioning. LFIRG is called to determine the columns of the inverse.

```
USE LFCRG_INT
USE UMACH_INT
USE LFIRG_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), J, NOUT
REAL A(LDA,N), AINV(LDA,N), FACT(LDFACT,N), RCOND, &
 RES(N), RJ(N)
!
! Set values for A
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)!
CALL LFCRG (A, FACT, IPVT, RCOND)
!
! Print the reciprocal condition number
! and the L1 condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0
!
! RJ is the J-th column of the identity
! matrix so the following LFIRG
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
CALL LFIRG (A, FACT, IPVT, RJ, AINV(:,J), RES)
```

```

 RJ(J) = 0.0
10 CONTINUE
!
 CALL WRRRN ('AINV', AINV)
!
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < .02
L1 Condition number < 100.0

```

```

 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000
3 -1.000 1.000 0.000

```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. LFCRG is called to factor the matrix and to check for singularity or ill-conditioning. LFIRG is called to determine the columns of the inverse. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFCRG_INT
USE UMACH_INT
USE LFIRG_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA, NOUT
INTEGER, ALLOCATABLE :: IPVT0(:)
REAL, ALLOCATABLE :: A(:, :), AINV(:, :), X0(:), RJ(:)
REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), RES0(:), RJ0(:)
REAL RCOND
PARAMETER (LDA=3, N=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
ENDIF
!
! Set up a 1D processor grid and define
! its context id, MP_ICTXT

```

```

CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), RES0(MXLDA), IPVT0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Call the factorization routine
CALL LFCRG (A0, FACT0, IPVT0, RCOND)
! Print the reciprocal condition number
! and the L1 condition number
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
ENDIF
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0
 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
! RJ is the J-th column of the identity
! matrix so the following LFIRG
! reference computes the J-th column of
! the inverse of A
 CALL LFIRG (A0, FACT0, IPVT0, RJ0, X0, RES0)
 RJ(J) = 0.0
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))
10 CONTINUE
! Print results
! Only Rank=0 has the solution, X.
IF(MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, IPVT0, FACT0, RES0, RJ, RJ0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < .02
L1 Condition number < 100.0

```

```

 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000

```

---

## LFTRG



Computes the  $LU$  factorization of a real general matrix.

### Required Arguments

$A$  —  $N$  by  $N$  matrix to be factored. (Input)

$FACT$  —  $N$  by  $N$  matrix containing the  $LU$  factorization of the matrix  $A$ . (Output)  
If  $A$  is not needed,  $A$  and  $FACT$  can share the same storage locations.

$IPVT$  — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization. (Output)

### Optional Arguments

$N$  — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

$LDFACT$  — Leading dimension of  $FACT$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

### FORTRAN 90 Interface

Generic: `CALL LFTRG (A, FACT, IPVT [, ...])`

Specific: The specific interface names are `S_LFTRG` and `D_LFTRG`.

### FORTRAN 77 Interface

Single: `CALL LFTRG (N, A, LDA, FACT, LDFACT, IPVT)`

Double: The double precision name is `DLFTRG`.

## ScaLAPACK Interface

Generic:     CALL LFTRG (A0, FACT0, IPVT0 [, ...])

Specific:    The specific interface names are S\_LFTRG and D\_LFTRG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFTRG performs an  $LU$  factorization of a real general coefficient matrix. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same norm. Otherwise, partial pivoting is used.

The routine LFTRG fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  is singular or very close to a singular matrix.

The  $LU$  factors are returned in a form that is compatible with routines LFIRG, LFSRG and LFDGR. To solve systems of equations with multiple right-hand-side vectors, use LFTRG followed by either LFIRG or LFSRG called once for each right-hand side. The routine LFDGR can be called to compute the determinant of the coefficient matrix after LFTRG has performed the factorization. Let  $F$  be the matrix FACT and let  $p$  be the vector IPVT. The triangular matrix  $U$  is stored in the upper triangle of  $F$ . The strict lower triangle of  $F$  contains the information needed to reconstruct  $L^{-1}$  using

$$L^{-1} = L_{N-1} P_{N-1} \dots L_1 P_1$$

where  $P_k$  is the identity matrix with rows  $k$  and  $p_k$  interchanged and  $L_k$  is the identity with  $F_{ik}$  for  $i = k + 1, \dots, N$  inserted below the diagonal. The strict lower half of  $F$  can also be thought of as containing the negative of the multipliers.

Routine LFTRG is based on the LINPACK routine SGEFA. See Dongarra et al. (1979). The routine SGEFA uses partial pivoting.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2TRG/ DL2TRG. The reference is:

```
CALL L2TRG (N, A, LDA, FACT, LDFACT, IPVT, WK)
```

The additional argument is:

**WK** — Work vector of length  $N$  used for scaling.

2.    Informational error

| Type | Code |                               |
|------|------|-------------------------------|
| 4    | 2    | The input matrix is singular. |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix to be factored. (Input)

**FACT0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `FACT`. `FACT` contains the *LU* factorization of the matrix `A`. (Output)

**IPVT0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the pivoting information for the *LU* factorization. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A linear system with multiple right-hand sides is solved. Routine `LFTRG` is called to factor the coefficient matrix. The routine `LFSRG` is called to compute the two solutions for the two right-hand sides. In this case, the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFMRG` to perform the factorization, and `LFIRG` to compute the solutions.

```

USE LFTRG_INT
USE LFSRG_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), J
REAL A(LDA,LDA), B(N,2), FACT(LDFACT,LDFACT), X(N,2)
!
! Set values for A and B
!
! A = (1.0 3.0 3.0)
! (1.0 3.0 4.0)
! (1.0 4.0 3.0)
!
! B = (1.0 10.0)
! (4.0 14.0)
! (-1.0 9.0)
!
DATA A/1.0, 1.0, 1.0, 3.0, 3.0, 4.0, 3.0, 4.0, 3.0/
DATA B/1.0, 4.0, -1.0, 10.0, 14.0, 9.0/
!
CALL LFTRG (A, FACT, IPVT)
! Solve for the two right-hand sides

```

```

 DO 10 J=1, 2
 CALL LFSRG (FACT, IPVT, B(:,J), X(:,J))
10 CONTINUE
!
 CALL WRRRN ('X', X)
 END

```

## Output

```

 X
 1 2
1 -2.000 1.000
2 -2.000 -1.000
3 3.000 4.000

```

## ScaLAPACK Example

A linear system with multiple right-hand sides is solved. Routine `LFTRG` is called to factor the coefficient matrix. The routine `LFSRG` is called to compute the two solutions for the two right-hand sides. In this case, the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFRCRG` to perform the factorization, and `LFIRG` to compute the solutions. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFTRG_INT
 USE LFSRG_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, DESCA(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA
 INTEGER, ALLOCATABLE :: IPVT0(:)
 REAL, ALLOCATABLE :: A(:, :), B(:, :), X(:, :), X0(:)
 REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), B0(:)
 PARAMETER (LDA=3, N=3)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N,2), X(N,2))
! Set values for A and B
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
!
 B(1,:) = (/ 1.0, 10.0/)
 B(2,:) = (/ 4.0, 14.0/)
 B(3,:) = (/ -1.0, 9.0/)
 ENDIF
!
! Set up a 1D processor grid and define

```

```

! its context id, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), B0(MXLDA), &
 IPVT0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Call the factorization routine
CALL LFTRG (A0, FACT0, IPVT0)
! Set up the columns of the B
! matrix one at a time in X0
DO 10 J=1, 2
 CALL SCALAPACK_MAP(B(:,j), DESCL, B0)
! Solve for the J-th column of X
 CALL LFSRG (FACT0, IPVT0, B0, X0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X(:,J))
10 CONTINUE
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK.EQ.0) CALL WRRRN ('X', X)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, IPVT0, FACT0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2
1 -2.000 1.000
2 -2.000 -1.000
3 3.000 4.000

```

---

## LFSRG



Solves a real general system of linear equations given the *LU* factorization of the coefficient matrix.

## Required Arguments

**FACT** —  $N$  by  $N$  matrix containing the  $LU$  factorization of the coefficient matrix  $A$  as output from routine `LFCRG` or `LFTRG`. (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from subroutine `LFCRG` or `LFTRG`. (Input).

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

**IPATH** — Path indicator. (Input)

$\text{IPATH} = 1$  means the system  $AX = B$  is solved.

$\text{IPATH} = 2$  means the system  $A^T X = B$  is solved.

Default:  $\text{IPATH} = 1$ .

## FORTRAN 90 Interface

Generic: `CALL LFSRG (FACT, IPVT, B, X [, ...])`

Specific: The specific interface names are `S_LFSRG` and `D_LFSRG`.

## FORTRAN 77 Interface

Single: `CALL LFSRG (N, FACT, LDFACT, IPVT, B, IPATH, X)`

Double: The double precision name is `DLFSRG`.

## ScaLAPACK Interface

Generic: `CALL LFSRG (FACT0, IPVT0, B0, X0 [, ...])`

Specific: The specific interface names are `S_LFSRG` and `D_LFSRG`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFSRG` computes the solution of a system of linear algebraic equations having a real general coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $LU$  factorization. This may be done by calling either `LFMRG` or `LFTRG`. The solution to  $Ax = b$  is found by solving the triangular systems  $Ly = b$  and  $Ux = y$ . The forward elimination step consists of solving the system  $Ly = b$  by applying the same permutations and elimination operations to  $b$  that were applied to the columns of  $A$  in the factorization routine. The backward substitution step consists of solving the triangular system  $Ux = y$  for  $x$ .

`LFSRG` and `LFIRG` both solve a linear system given its  $LU$  factorization. `LFIRG` generally takes more time and produces a more accurate answer than `LFSRG`. Each iteration of the iterative refinement algorithm used by `LFIRG` calls `LFSRG`. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**FACT0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `FACT` as output from routine `LFMRG`. `FACT` contains the  $LU$  factorization of the matrix `A`. (Input)

**IPVT0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the pivoting information for the  $LU$  factorization as output from subroutine `LFMRG` or `LFTRG`/`DLFTRG`. (Input)

**B0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse is computed for a real general  $3 \times 3$  matrix. The input matrix is assumed to be well-conditioned, hence, `LFTRG` is used rather than `LFMRG`.

```
USE LFSRG_INT
USE LFTRG_INT
```

```

 USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER I, IPVT(N), J
REAL A(LDA,LDA), AINV(LDA,LDA), FACT(LDFACT,LDFACT), RJ(N)
!
! Set values for A
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
!
 CALL LFTRG (A, FACT, IPVT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
 RJ = 0.0E0
 DO 10 J=1, N
 RJ(J) = 1.0
!
! RJ is the J-th column of the identity
! matrix so the following LFSRG
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSRG (FACT, IPVT, RJ, AINV(:,J))
 RJ(J) = 0.0
10 CONTINUE
!
! Print results
 CALL WRRRN ('AINV', AINV)
 END

```

## Output

```

 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000
3 -1.000 1.000 0.000

```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. The input matrix is assumed to be well-conditioned, hence, `LFTRG` is used rather than `LFCRG`. `LFSRG` is called to determine the columns of the inverse. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFTRG_INT
 USE UMACH_INT
 USE LFSRG_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'

```

```

!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA
INTEGER, ALLOCATABLE :: IPVT0(:)
REAL, ALLOCATABLE :: A(:, :), AINV(:, :), X0(:), RJ(:)
REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), RJ0(:)
PARAMETER (LDA=3, N=3)

!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
ENDIF

!
! Set up a 1D processor grid and define
! its context id, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), IPVT0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Call the factorization routine
CALL LFTRG (A0, FACT0, IPVT0)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0
 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIRG
! reference computes the J-th column of
! the inverse of A
 CALL LFSRG (FACT0, IPVT0, RJ0, X0)
 RJ(J) = 0.0
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))
10 CONTINUE

!
! Print results
! Only Rank=0 has the solution, AINV.
IF(MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, IPVT0, FACT0, RJ, RJ0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)

!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```
 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000
3 -1.000 1.000 0.000
```

---

## LFIRG



Uses iterative refinement to improve the solution of a real general system of linear equations.

### Required Arguments

- A* —  $N$  by  $N$  matrix containing the coefficient matrix of the linear system. (Input)
- FACT* —  $N$  by  $N$  matrix containing the  $LU$  factorization of the coefficient matrix  $A$  as output from routine [LFCRG](#)/[DLFCRG](#) or [LFTRG](#)/[DLFTRG](#). (Input).
- IPVT* — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from routine [LFCRG](#)/[DLFCRG](#) or [LFTRG](#)/[DLFTRG](#). (Input)
- B* — Vector of length  $N$  containing the right-hand side of the linear system. (Input).
- X* — Vector of length  $N$  containing the solution to the linear system. (Output)
- RES* — Vector of length  $N$  containing the final correction at the improved solution. (Output)

### Optional Arguments

- N* — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .
- LDA* — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .
- LDFACT* — Leading dimension of  $FACT$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .
- IPATH* — Path indicator. (Input)

IPATH = 1 means the system  $A * X = B$  is solved.

IPATH = 2 means the system  $A^T X = B$  is solved.

Default: IPATH = 1.

## FORTRAN 90 Interface

Generic:     CALL LFIRG (A, FACT, IPVT, B, X, RES [...])

Specific:    The specific interface names are S\_LFIRG and D\_LFIRG.

## FORTRAN 77 Interface

Single:      CALL LFIRG (N, A, LDA, FACT, LDFACT, IPVT, B, IPATH, X, RES)

Double:     The double precision name is DLFIRG.

## ScaLAPACK Interface

Generic:     CALL LFIRG (A0, FACT0, IPVT0, B0, X0, RES0 [, ...])

Specific:    The specific interface names are S\_LFIRG and D\_LFIRG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFIRG computes the solution of a system of linear algebraic equations having a real general coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the *Introduction* section of this manual.

To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either LFCRG or LFTRG.

Iterative refinement fails only if the matrix is very ill-conditioned.

Routines LFIRG and LFSRG both solve a linear system given its *LU* factorization. LFIRG generally takes more time and produces a more accurate answer than LFSRG. Each iteration of the iterative refinement algorithm used by LFIRG calls LFSRG.

## Comments

Informational error

Type Code

3 2 The input matrix is too ill-conditioned for iterative refinement to be effective.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficient matrix of the linear system. (Input)
  
- FACT0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `FACT` as output from routine `LFCRG` or `LFTRG`. `FACT` contains the *LU* factorization of the matrix `A`. (Input)
  
- IPVT0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the pivoting information for the *LU* factorization as output from subroutine `LFCRG` or `LFTRG`. (Input)
  
- B0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)
  
- X0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.
  
- RES0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `RES`. `RES` contains the final correction at the improved solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.5 to the second element.

```

USE LFIRG_INT
USE LFCRG_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), NOUT
REAL A(LDA,LDA), B(N), FACT(LDFACT,LDFACT), RCOND, RES(N), X(N)
!
! Set values for A and B
!
! A = (1.0 3.0 3.0)

```

```

! (1.0 3.0 4.0)
! (1.0 4.0 3.0)
!
! B = (-0.5 -1.0 1.5)
!
DATA A/1.0, 1.0, 1.0, 3.0, 3.0, 4.0, 3.0, 4.0, 3.0/
DATA B/-0.5, -1.0, 1.5/
!
CALL LFCRG (A, FACT, IPVT, RCOND)
! Print the reciprocal condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
! Solve the three systems
DO 10 J=1, 3
 CALL LFIRG (A, FACT, IPVT, B, X, RES)
! Print results
 CALL WRRRN ('X', X, 1, N, 1)
! Perturb B by adding 0.5 to B(2)
 B(2) = B(2) + 0.5

10 CONTINUE
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < 0.02
L1 Condition number < 100.0
 X
 1 2 3
-5.000 2.000 -0.500

 X
 1 2 3
-6.500 2.000 0.000

 X
 1 2 3
-8.000 2.000 0.500

```

## ScaLAPACK Example

The same set of linear systems is solved successively as a distributed example. The right-hand side vector is perturbed after solving the system each of the first two times by adding 0.5 to the second element. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFIRG_INT
USE UMACH_INT
USE LFCRG_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT

```

```

IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA, NOUT
INTEGER, ALLOCATABLE :: IPVT0(:)
REAL, ALLOCATABLE :: A(:, :), B(:), X(:), X0(:), AINV(:, :)
REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), RES0(:), B0(:)
REAL RCOND
PARAMETER (LDA=3, N=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
!
 B(:) = (/ -0.5, -1.0, 1.5/)
ENDIF
! Set up a 1D processor grid and define
! its context id, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDA), FACT0(MXLDA,MXCOL), &
 B0(MXLDA), RES0(MXLDA), IPVT0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Call the factorization routine
CALL LFCRG (A0, FACT0, IPVT0, RCOND)
! Print the reciprocal condition number
! and the L1 condition number
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
ENDIF
! Solve the three systems
! one at a time in X
DO 10 J=1, 3
 CALL SCALAPACK_MAP(B, DESCL, B0)
 CALL LFIRG (A0, FACT0, IPVT0, B0, X0, RES0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X)
! Print results
! Only Rank=0 has the solution, X.
 IF(MP_RANK.EQ.0) CALL WRRRN ('X', X, 1, N, 1)
 IF(MP_RANK.EQ.0) B(2) = B(2) + 0.5
10 CONTINUE
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV, B)
DEALLOCATE(A0, B0, IPVT0, FACT0, RES0, X0)

```

```

! Exit ScaLAPACK usage
 CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < 0.02
L1 Condition number < 100.0

```

```

 X
 1 2 3
-5.000 2.000 -0.500

```

```

 X
 1 2 3
-6.500 2.000 0.000

```

```

 X
 1 2 3
-8.000 2.000 0.500

```

---

## LFDRG

Computes the determinant of a real general matrix given the *LU* factorization of the matrix.

### Required Arguments

**FACT** — *N* by *N* matrix containing the *LU* factorization of the matrix *A* as output from routine [LFTRG](#)/[DLFTRG](#) or [LFCRG](#)/[DLFCRG](#). (Input)

**IPVT** — Vector of length *N* containing the pivoting information for the *LU* factorization as output from routine [LFTRG](#)/[DLFTRG](#) or [LFCRG](#)/[DLFCRG](#). (Input).

**DET1** — Scalar containing the mantissa of the determinant. (Output)

The value *DET1* is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or *DET1* = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)

The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**N** — Order of the matrix. (Input)

Default: *N* = size (FACT,2).

**LDFACT** — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDFACT* = size (FACT,1).

## FORTRAN 90 Interface

Generic:    CALL LFDRG (FACT, IPVT, DET1, DET2 [...])

Specific:   The specific interface names are S\_LFDRG and D\_LFDRG.

## FORTRAN 77 Interface

Single:     CALL LFDRG (N, FACT, LDFACT, IPVT, DET1, DET2)

Double:     The double precision name is DLFDRG.

## Description

Routine LFDRG computes the determinant of a real general coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $LU$  factorization. This may be done by calling either [LFDRG](#) or [LFTRG](#). The formula  $\det A = \det L \det U$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements

$$\det U = \prod_{i=1}^N U_{ii}$$

(The matrix  $U$  is stored in the upper triangle of `FACT`.) Since  $L$  is the product of triangular matrices with unit diagonals and of permutation matrices,  $\det L = (-1)^k$  where  $k$  is the number of pivoting interchanges.

Routine LFDRG is based on the LINPACK routine `SGEDI`; see Dongarra et al. (1979)

## Example

The determinant is computed for a real general  $3 \times 3$  matrix.

```
USE LFDRG_INT
USE LFTRG_INT
USE UMACH_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), NOUT
REAL A(LDA,LDA), DET1, DET2, FACT(LDFACT,LDFACT)
!
! Set values for A
! A = (33.0 16.0 72.0)
! (-24.0 -10.0 -57.0)
! (18.0 -11.0 7.0)
!
DATA A/33.0, -24.0, 18.0, 16.0, -10.0, -11.0, 72.0, -57.0, 7.0/
!
CALL LFTRG (A, FACT, IPVT)
! Compute the determinant
CALL LFDRG (FACT, IPVT, DET1, DET2)
! Print the results
CALL UMACH (2, NOUT)
```

```

WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is ', F6.3, ' * 10**', F2.0)
END

```

## Output

The determinant of A is -4.761 \* 10\*\*3.

# LINRG



Computes the inverse of a real general matrix.

## Required Arguments

*A* —  $N$  by  $N$  matrix containing the matrix to be inverted. (Input)

*AINV* —  $N$  by  $N$  matrix containing the inverse of *A*. (Output)

If *A* is not needed, *A* and *AINV* can share the same storage locations.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

*LDAINV* — Leading dimension of *AINV* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDAINV = \text{size}(AINV,1)$ .

## FORTRAN 90 Interface

Generic:    CALL LINRG (A, AINV [,...])

Specific:   The specific interface names are S\_LINRG and D\_LINRG.

## FORTRAN 77 Interface

Single:     CALL LINRG (N, A, LDA, AINV, LDAINV)

Double:     The double precision name is DLINRG.

## ScaLAPACK Interface

Generic:    CALL LINRG (A0, AINV0 [, ...])

Specific:    The specific interface names are S\_LINRG and D\_LINRG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LINRG computes the inverse of a real general matrix. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. LINRG first uses the routine LFCRG to compute an  $LU$  factorization of the coefficient matrix and to estimate the condition number of the matrix. Routine LFCRG computes  $U$  and the information needed to compute  $L^{-1}$ . LINRT is then used to compute  $U^{-1}$ . Finally,  $A^{-1}$  is computed using  $A^{-1} = U^{-1}L^{-1}$ .

The routine LINRG fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. This error occurs only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in  $A^{-1}$ .

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2NRG/DL2NRG. The reference is:

```
CALL L2NRG (N, A, LDA, AINV, LDAINV, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $N + N(N - 1)/2$ .

**IWK** — Integer work vector of length  $N$ .

2.    Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                             |
|---|---|-----------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The inverse might not be accurate. |
|---|---|-----------------------------------------------------------------------------|

|   |   |                               |
|---|---|-------------------------------|
| 4 | 2 | The input matrix is singular. |
|---|---|-------------------------------|

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the matrix to be inverted. (Input)

**AINV0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix AINV. AINV contains the inverse of the matrix A. (Output)  
If A is not needed, A and AINV can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse is computed for a real general  $3 \times 3$  matrix.

```
USE LINRG_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDAINV=3)
INTEGER I, J, NOUT
REAL A(LDA,LDA), AINV(LDAINV,LDAINV)
!
! Set values for A
! A = (1.0 3.0 3.0)
! (1.0 3.0 4.0)
! (1.0 4.0 3.0)
!
DATA A/1.0, 1.0, 1.0, 3.0, 3.0, 4.0, 3.0, 4.0, 3.0/
!
CALL LINRG (A, AINV)
!
! Print results
CALL WRRRN ('AINV', AINV)
END
```

## Output

```
 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000
3 -1.000 1.000 0.000
```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
USE MPI_SETUP_INT
USE LINRG_INT
USE WRRRN_INT
```

```

USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

!
! Declare variables
INTEGER LDA, LDAINV, N, DESCA(9)
INTEGER INFO, MXCOL, MXLDA
REAL, ALLOCATABLE :: A(:,,:), AINV(:,:)
REAL, ALLOCATABLE :: A0(:,,:), AINV0(:,:)
PARAMETER (LDA=3, LDAINV=3, N=3)

!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDAINV,N))
!
! Set values for A
 A(1,:) = (/ 1.0, 3.0, 3.0/)
 A(2,:) = (/ 1.0, 3.0, 4.0/)
 A(3,:) = (/ 1.0, 4.0, 3.0/)
ENDIF

!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), AINV0(MXLDA,MXCOL))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Get the inverse
CALL LINRG (A0, AINV0)

!
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(AINV0, DESCA, AINV)
!
! Print results
! Only Rank=0 has the solution, AINV.
IF(MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, AINV0)

!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 AINV
 1 2 3
1 7.000 -3.000 -3.000
2 -1.000 0.000 1.000
3 -1.000 1.000 0.000

```

---

# LSACG



Solves a complex general system of linear equations with iterative refinement.

## Required Arguments

*A* — Complex  $N$  by  $N$  matrix containing the coefficients of the linear system. (Input)

*B* — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

*X* — Complex vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

*N* — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

*IPATH* — Path indicator. (Input)  
*IPATH* = 1 means the system  $AX = B$  is solved.  
*IPATH* = 2 means the system  $A^H X = B$  is solved  
Default: *IPATH* = 1.

## FORTRAN 90 Interface

Generic:    CALL LSACG (A, B, X [,...])

Specific:    The specific interface names are S\_LSACG and D\_LSACG.

## FORTRAN 77 Interface

Single:     CALL LSACG (N, A, LDA, B, IPATH, X)

Double:     The double precision name is DLSACG.

## ScaLAPACK Interface

Generic:    CALL LSACG (A0, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSACG and D\_LSACG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LSACG` solves a system of linear algebraic equations with a complex general coefficient matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. `LSACG` first uses the routine `LFCCG` to compute an  $LU$  factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using the iterative refinement routine `LFICG`.

`LSACG` fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. `LSACG` solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2ACG/DL2ACG`. The reference is:

```
CALL L2ACG (N, A, LDA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** — Complex work vector of length  $N^2$  containing the  $LU$  factorization of  $A$  on output.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  on output.

**WK** — Complex work vector of length  $N$ .

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
|---|---|------------------------------------------------------------------------------|

|   |   |                               |
|---|---|-------------------------------|
| 4 | 2 | The input matrix is singular. |
|---|---|-------------------------------|

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2ACG` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`; respectively, in `LSACG`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSACG`. Users directly calling `L2ACG` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSACG` or `L2ACG`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSACG` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CCG` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CCG` skips this computation. `LSACG` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficients of the linear system. (Input)
- B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)
- X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of three linear equations is solved. The coefficient matrix has complex general form and the right-hand-side vector  $b$  has three elements.

```

USE LSACG_INT
USE WRCRN_INT
!
! Declare variables
PARAMETER (LDA=3, N=3)
COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (3.0-2.0i 2.0+4.0i 0.0-3.0i)
! (1.0+1.0i 2.0-6.0i 1.0+2.0i)
! (4.0+0.0i -5.0+1.0i 3.0-2.0i)
!
!
```

```

! B = (10.0+5.0i 6.0-7.0i -1.0+2.0i)
!
DATA A/(3.0,-2.0), (1.0,1.0), (4.0,0.0), (2.0,4.0), (2.0,-6.0), &
 (-5.0,1.0), (0.0,-3.0), (1.0,2.0), (3.0,-2.0)/
DATA B/(10.0,5.0), (6.0,-7.0), (-1.0,2.0)/
! Solve AX = B (IPATH = 1)
CALL LSACG (A, B, X)
! Print results
CALL WRCRN ('X', X, 1, N, 1)
END

```

## Output

```

 X
 1 2 3
(1.000,-1.000) (2.000, 1.000) (0.000, 3.000)

```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has complex general form and the right-hand-side vector  $b$  has three elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LSACG_INT
USE WRCRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
! Declare variables
INTEGER LDA, N, DESCA(9), DESCX(9)
INTEGER INFO, MXCOL, MXLDA
COMPLEX, ALLOCATABLE :: A(:,,:), B(:), X(:)
COMPLEX, ALLOCATABLE :: A0(:,,:), B0(:), X0(:)
PARAMETER (LDA=3, N=3)
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ (3.0, -2.0), (2.0, 4.0), (0.0, -3.0)/)
 A(2,:) = (/ (1.0, 1.0), (2.0, -6.0), (1.0, 2.0)/)
 A(3,:) = (/ (4.0, 0.0), (-5.0, 1.0), (3.0, -2.0)/)
!
 B = (/ (10.0, 5.0), (6.0, -7.0), (-1.0, 2.0)/)
ENDIF
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)

```

```

! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
! Solve the system of equations
CALL LSACG (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0)CALL WRCRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
(1.000,-1.000) (2.000, 1.000) (0.000, 3.000)

```

---

## L<sub>S</sub>LCG



Solves a complex general system of linear equations without iterative refinement.

### Required Arguments

- A* — Complex  $N$  by  $N$  matrix containing the coefficients of the linear system. (Input)
- B* — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X* — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If *B* is not needed, *B* and *X* can share the same storage locations)

## Optional Arguments

*N* — Number of equations. (Input)

Default:  $N = \text{size}(A, 2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A, 1)$ .

*IPATH* — Path indicator. (Input)

$IPATH = 1$  means the system  $AX = B$  is solved.

$IPATH = 2$  means the system  $A^H X = B$  is solved

Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic:     CALL LSLCG (A, B, X [, ...])

Specific:    The specific interface names are S\_LSLCG and D\_LSLCG.

## FORTRAN 77 Interface

Single:     CALL LSLCG (N, A, LDA, B, IPATH, X)

Double:     The double precision name is DLSLCG.

## ScaLAPACK Interface

Generic:     CALL LSLCG (A0, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSLCG and D\_LSLCG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LSLCG solves a system of linear algebraic equations with a complex general coefficient matrix. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. LSLCG first uses the routine LFCCG to compute an *LU* factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using LFSCG.

LSLCG fails if *U*, the upper triangular part of the factorization, has a zero diagonal element. This occurs only if *A* either is a singular matrix or is very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that [LSACG](#) be used.

## Comments

1. Workspace may be explicitly provided, if desired, by use of [L2LCG](#)/[DL2LCG](#). The reference is:

```
CALL L2LCG (N, A, LDA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing the  $LU$  factorization of  $A$  on output. If  $A$  is not needed,  $A$  and **FACT** can share the same storage locations.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  on output.

**WK** — Complex work vector of length  $N$ .

2. Informational errors

| Type | Code | Description                                                                  |
|------|------|------------------------------------------------------------------------------|
| 3    | 1    | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4    | 2    | The input matrix is singular.                                                |

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine [L2LCG](#) the leading dimension of **FACT** is increased by **IVAL(3)** when  $N$  is a multiple of **IVAL(4)**. The values **IVAL(3)** and **IVAL(4)** are temporarily replaced by **IVAL(1)** and **IVAL(2)**; respectively, in [LSLCG](#). Additional memory allocation for **FACT** and option value restoration are done automatically in [LSLCG](#). Users directly calling [L2LCG](#) can allocate additional space for **FACT** and set **IVAL(3)** and **IVAL(4)** so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use [LSLCG](#) or [L2LCG](#). Default values for the option are **IVAL(\*)** = 1, 16, 0, 1.

**17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine [LSLCG](#) temporarily replaces **IVAL(2)** by **IVAL(1)**. The routine [L2CCG](#) computes the condition number if **IVAL(2)** = 2. Otherwise [L2CCG](#) skips this computation. [LSLCG](#) restores the option. Default values for the option are **IVAL(\*)** = 1, 2.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix *A*. *A* contains the coefficients of the linear system. (Input)
- B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector *B*. *B* contains the right-hand side of the linear system. (Input)
- X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector *X*. *X* contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

A system of three linear equations is solved. The coefficient matrix has complex general form and the right-hand-side vector *b* has three elements.

```
 USE LSLCG_INT
 USE WRCRN_INT
!
! Declare variables
 PARAMETER (LDA=3, N=3)
 COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (3.0-2.0i 2.0+4.0i 0.0-3.0i)
! (1.0+1.0i 2.0-6.0i 1.0+2.0i)
! (4.0+0.0i -5.0+1.0i 3.0-2.0i)
!
! B = (10.0+5.0i 6.0-7.0i -1.0+2.0i)
!
!
 DATA A/(3.0,-2.0), (1.0,1.0), (4.0,0.0), (2.0,4.0), (2.0,-6.0), &
 (-5.0,1.0), (0.0,-3.0), (1.0,2.0), (3.0,-2.0)/
 DATA B/(10.0,5.0), (6.0,-7.0), (-1.0,2.0)/
!
! Solve AX = B (IPATH = 1)
 CALL LSLCG (A, B, X)
!
! Print results
 CALL WRCRN ('X', X, 1, N, 1)
 END
```



```

! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRCRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
(1.000,-1.000) (2.000, 1.000) (0.000, 3.000)

```

---

## LFCCG



Computes the  $LU$  factorization of a complex general matrix and estimate its  $L_1$  condition number.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix to be factored. (Input)

**FACT** — Complex  $N \times N$  matrix containing the  $LU$  factorization of the matrix **A** (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage locations

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization.  
(Output)

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**.  
(Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDFACT = size (FACT,1)`.

### **FORTRAN 90 Interface**

Generic: `CALL LFCCG (A, FACT, IPVT, RCOND [, ...])`  
Specific: The specific interface names are `S_LFCCG` and `D_LFCCG`.

### **FORTRAN 77 Interface**

Single: `CALL LFCCG (N, A, LDA, FACT, LDFACT, IPVT, RCOND)`  
Double: The double precision name is `DLFCCG`.

### **ScaLAPACK Interface**

Generic: `CALL LFCCG (A0, FACT0, IPVT0, RCOND [, ...])`  
Specific: The specific interface names are `S_LFCCG` and `D_LFCCG`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

Routine `LFCCG` performs an *LU* factorization of a complex general coefficient matrix. It also estimates the condition number of the matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. The *LU* factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by `LINPACK` and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

`LFCCG` fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  either is singular or is very close to a singular matrix.

The *LU* factors are returned in a form that is compatible with routines `LFICG`, `LFSCG` and `LFDCG`. To solve systems of equations with multiple right-hand-side vectors, use `LFCCG` followed by either `LFICG` or `LFSCG` called once for each right-hand side. The routine `LFDCG` can be called to compute the determinant of the coefficient matrix after `LFCCG` has performed the factorization.

Let  $F$  be the matrix `FACT` and let  $p$  be the vector `IPVT`. The triangular matrix  $U$  is stored in the upper triangle of  $F$ . The strict lower triangle of  $F$  contains the information needed to reconstruct  $L$  using

$$L_{11} = L_{N-1} P_{N-1} \dots L_1 P_1$$

where  $P_k$  is the identity matrix with rows  $k$  and  $p_k$  interchanged and  $L_k$  is the identity with  $F_{ik}$  for  $i = k + 1, \dots, N$  inserted below the diagonal. The strict lower half of  $F$  can also be thought of as containing the negative of the multipliers.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2CCG/DL2CCG`. The reference is:

```
CALL L2CCG (N, A, LDA, FACT, LDFACT, IPVT, RCOND, WK)
```

The additional argument is:

**WK** — Complex work vector of length  $N$ .

2. Informational errors

Type Code

3 1 The input matrix is algorithmically singular.

4 2 The input matrix is singular.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix to be factored. (Input)

**FACT0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `FACT`. `FACT` contains the  $LU$  factorization of the matrix `A`. (Output)

**IPVT0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the pivoting information for the  $LU$  factorization. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFCCG is called to factor the matrix and to check for singularity or ill-conditioning. LFICG is called to determine the columns of the inverse.

```
 USE IMSL_LIBRARIES

!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), NOUT
REAL RCOND, THIRD
COMPLEX A(LDA,N), AINV(LDA,N), RJ(N), FACT(LDFACT,N), RES(N)
!
! Declare functions
COMPLEX CMLPX

!
! Set values for A
!
! A = (1.0+1.0i 2.0+3.0i 3.0+3.0i)
! (2.0+1.0i 5.0+3.0i 7.0+4.0i)
! (-2.0+1.0i -4.0+4.0i -5.0+3.0i)
!
DATA A/(1.0,1.0), (2.0,1.0), (-2.0,1.0), (2.0,3.0), (5.0,3.0), &
 (-4.0,4.0), (3.0,3.0), (7.0,4.0), (-5.0,3.0)/
!
! Scale A by dividing by three
THIRD = 1.0/3.0
DO 10 I=1, N
 CALL CSSCAL (N, THIRD, A(:,I), 1)
10 CONTINUE
!
! Factor A
CALL LFCCG (A, FACT, IPVT, RCOND)
!
! Print the L1 condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Set up the columns of the identity
! matrix one at a time in RJ
CALL CSET (N, (0.0,0.0), RJ, 1)
DO 20 J=1, N
 RJ(J) = CMLPX(1.0,0.0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIRG
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFICG (A, FACT, IPVT, RJ, AINV(:,J), RES)
 RJ(J) = CMLPX(0.0,0.0)
20 CONTINUE
!
! Print results
CALL WRCRN ('AINV', AINV)
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END
```

## Output

```
RCOND < .02
L1 Condition number < 100.0
```

|   | AINV             |                  |                  |
|---|------------------|------------------|------------------|
|   | 1                | 2                | 3                |
| 1 | ( 6.400, -2.800) | (-3.800, 2.600)  | (-2.600, 1.200)  |
| 2 | (-1.600, -1.800) | ( 0.200, 0.600)  | ( 0.400, -0.800) |
| 3 | (-0.600, 2.200)  | ( 1.200, -1.400) | ( 0.400, 0.200)  |

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. LFCCG is called to factor the matrix and to check for singularity or ill-conditioning. LFICG is called to determine the columns of the inverse. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFCCG_INT
USE UMACH_INT
USE LFICG_INT
USE WRCRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA, NOUT
INTEGER, ALLOCATABLE :: IPVT0(:)
COMPLEX, ALLOCATABLE :: A(:, :), AINV(:, :), X0(:), RJ(:)
COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), RES0(:), RJ0(:)
REAL RCOND, THIRD
PARAMETER (LDA=3, N=3)

!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ (1.0, 1.0), (2.0, 3.0), (3.0, 3.0)/)
 A(2,:) = (/ (2.0, 1.0), (5.0, 3.0), (7.0, 4.0)/)
 A(3,:) = (/ (-2.0, 1.0), (-4.0, 4.0), (-5.0, 3.0)/)
!
! Scale A by dividing by three
 THIRD = 1.0/3.0
 A = A * THIRD
ENDIF

!
! Set up a 1D processor grid and define
! its context id, MP ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)

!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)

!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP ICTXT, MXLDA, INFO)

```

```

! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), RES0(MXLDA), IPVT0(MXLDA))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Factor A
CALL LFCCG (A0, FACT0, IPVT0, RCOND)
! Print the reciprocal condition number
! and the L1 condition number
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
ENDIF
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0, 0.0)
DO 10 J=1, N
 RJ(J) = (1.0, 0.0)
 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
! RJ is the J-th column of the identity
! matrix so the following LFICG
! reference computes the J-th column of
! the inverse of A
 CALL LFICG (A0, FACT0, IPVT0, RJ0, X0, RES0)
 RJ(J) = (0.0, 0.0)
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))
10 CONTINUE
! Print results
! Only Rank=0 has the solution, AINV.
IF(MP_RANK.EQ.0) CALL WRCRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, FACT0, IPVT0, RJ, RJ0, RES0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

RCOND < .02

L1 Condition number < 100.0

|   | AINV            |                 |                 |
|---|-----------------|-----------------|-----------------|
|   | 1               | 2               | 3               |
| 1 | ( 6.400,-2.800) | (-3.800, 2.600) | (-2.600, 1.200) |
| 2 | (-1.600,-1.800) | ( 0.200, 0.600) | ( 0.400,-0.800) |
| 3 | (-0.600, 2.200) | ( 1.200,-1.400) | ( 0.400, 0.200) |

---

# LFTCG



Computes the *LU* factorization of a complex general matrix.

## Required Arguments

*A* — Complex  $N$  by  $N$  matrix to be factored. (Input)

*FACT* — Complex  $N \times N$  matrix containing the *LU* factorization of the matrix *A*. (Output)  
If *A* is not needed, *A* and *FACT* can share the same storage locations.

*IPVT* — Vector of length  $N$  containing the pivoting information for the *LU* factorization. (Output)

## Optional Arguments

*N* — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

*LDFACT* — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:     `CALL LFTCG (A, FACT, IPVT [...])`

Specific:    The specific interface names are `S_LFTCG` and `D_LFTCG`.

## FORTRAN 77 Interface

Single:     `CALL LFTCG (N, A, LDA, FACT, LDFACT, IPVT)`

Double:     The double precision name is `DLFTCG`.

## ScaLAPACK Interface

Generic:     CALL LFTCG (A0, FACT0, IPVT0 [, ...])

Specific:    The specific interface names are S\_LFTCG and D\_LFTCG.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFTCG performs an  $LU$  factorization of a complex general coefficient matrix. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm.

LFTCG fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  either is singular or is very close to a singular matrix.

The  $LU$  factors are returned in a form that is compatible with routines LFICG, LFSCG and LFDCC. To solve systems of equations with multiple right-hand-side vectors, use LFTCG followed by either LFICG or LFSCG called once for each right-hand side. The routine LFDCC can be called to compute the determinant of the coefficient matrix after LFCCG has performed the factorization.

Let  $F$  be the matrix FACT and let  $p$  be the vector IPVT. The triangular matrix  $U$  is stored in the upper triangle of  $F$ . The strict lower triangle of  $F$  contains the information needed to reconstruct  $L$  using

$$L = L_{N-1}P_{N-1} \dots L_1P_1$$

where  $P_k$  is the identity matrix with rows  $k$  and  $P_k$  interchanged and  $L_k$  is the identity with  $F_{ik}$  for  $i = k + 1, \dots, N$  inserted below the diagonal. The strict lower half of  $F$  can also be thought of as containing the negative of the multipliers.

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.     Workspace may be explicitly provided, if desired, by use of L2TCG/DL2TCG. The reference is:

```
CALL L2TCG (N, A, LDA, FACT, LDFACT, IPVT, WK)
```

The additional argument is:

**WK** — Complex work vector of length  $N$ .

2.     Informational error

Type   Code

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix A. A contains the matrix to be factored. (Input)
- FACT0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix FACT. FACT contains the *LU* factorization of the matrix A. (Output)  
If A is not needed, A and FACT can share the same storage locations.
- IPVT0** — Local vector of length MXLDA containing the local portions of the distributed vector IPVT. IPVT contains the pivoting information for the *LU* factorization. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A linear system with multiple right-hand sides is solved. LFTCG is called to factor the coefficient matrix. LFSCG is called to compute the two solutions for the two right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call LFCCG to perform the factorization, and LFICG to compute the solutions.

```

USE LFTCG_INT
USE LFSCG_INT
USE WRRCRN_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N)
COMPLEX A(LDA,LDA), B(N,2), X(N,2), FACT(LDFACT,LDFACT)
!
! Set values for A
! A = (1.0+1.0i 2.0+3.0i 3.0-3.0i)
! (2.0+1.0i 5.0+3.0i 7.0-5.0i)
! (-2.0+1.0i -4.0+4.0i 5.0+3.0i)
!
DATA A/(1.0,1.0), (2.0,1.0), (-2.0,1.0), (2.0,3.0), (5.0,3.0), &
 (-4.0,4.0), (3.0,-3.0), (7.0,-5.0), (5.0,3.0)/
!
! Set the right-hand sides, B
! B = (3.0+ 5.0i 9.0+ 0.0i)
! (22.0+10.0i 13.0+ 9.0i)
! (-10.0+ 4.0i 6.0+10.0i)
!
DATA B/(3.0,5.0), (22.0,10.0), (-10.0,4.0), (9.0,0.0), &
 (13.0,9.0), (6.0,10.0)/
!

```

```

! Factor A
 CALL LFTCG (A, FACT, IPVT)
! Solve for the two right-hand sides
 DO 10 J=1, 2
 CALL LFSCG (FACT, IPVT, B(:,J), X(:,J))
10 CONTINUE
! Print results
 CALL WRCRN ('X', X)
 END

```

## Output

```

 X
 1 2
1 (1.000,-1.000) (0.000, 2.000)
2 (2.000, 4.000) (-2.000,-1.000)
3 (3.000, 0.000) (1.000, 3.000)

```

## ScaLAPACK Example

The same linear system with multiple right-hand sides is solved as a distributed example. LFTCG is called to factor the matrix. LFSCG is called to compute the two solutions for the two right-hand sides. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFTCG_INT
 USE LFSCG_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
! Declare variables
 INTEGER J, LDA, N, DESCA(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA
 INTEGER, ALLOCATABLE :: IPVT0(:)
 COMPLEX, ALLOCATABLE :: A(:, :), B(:, :), X(:, :), X0(:)
 COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), B0(:)
 PARAMETER (LDA=3, N=3)
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N,2), X(N,2))
! Set values for A and B
 A(1,:) = (/ (1.0, 1.0), (2.0, 3.0), (3.0,-3.0)/)
 A(2,:) = (/ (2.0, 1.0), (5.0, 3.0), (7.0,-5.0)/)
 A(3,:) = (/ (-2.0, 1.0), (-4.0, 4.0), (5.0, 3.0)/)
!
 B(1,:) = (/ (3.0, 5.0), (9.0, 0.0)/)
 B(2,:) = (/ (22.0, 10.0), (13.0, 9.0)/)
 B(3,:) = (/ (-10.0, 4.0), (6.0, 10.0)/)
 ENDIF
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT

```

```

CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), &
 B0(MXLDA), IPVT0(MXLDA))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Factor A
CALL LFTCG (A0, FACT0, IPVT0)
! Solve for the two right-hand sides
DO 10 J=1, 2
 CALL SCALAPACK_MAP(B(:,J), DESCL, B0)
 CALL LFSCG (FACT0, IPVT0, B0, X0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X(:,J))
10 CONTINUE
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK.EQ.0) CALL WRCRN ('X', X)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, FACT0, IPVT0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2
1 (1.000,-1.000) (0.000, 2.000)
2 (2.000, 4.000) (-2.000,-1.000)
3 (3.000, 0.000) (1.000, 3.000)

```

---

## LFSCG



Solves a complex general system of linear equations given the *LU* factorization of the coefficient matrix.

## Required Arguments

- FACT** — Complex  $N$  by  $N$  matrix containing the  $LU$  factorization of the coefficient matrix  $A$  as output from routine `LFCCG/DLFCCG` or `LFTCG/DLFTCG`. (Input)
- IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from routine `LFCCG/DLFCCG` or `LFTCG/DLFTCG`. (Input)
- B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

## Optional Arguments

- N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .
- LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .
- IPATH** — Path indicator. (Input)  
 $\text{IPATH} = 1$  means the system  $AX = B$  is solved.  
 $\text{IPATH} = 2$  means the system  $A^H X = B$  is solved.  
Default:  $\text{IPATH} = 1$ .

## FORTRAN 90 Interface

- Generic: `CALL LFSCG (FACT, IPVT, B, X [, ...])`
- Specific: The specific interface names are `S_LFSCG` and `D_LFSCG`.

## FORTRAN 77 Interface

- Single: `CALL LFSCG (N, FACT, LDFACT, IPVT, B, IPATH, X)`
- Double: The double precision name is `DLFSCG`.

## ScaLAPACK Interface

- Generic: `CALL LFSCG (FACT0, IPVT0, B0, X0 [, ...])`
- Specific: The specific interface names are `S_LFSCG` and `D_LFSCG`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFSCG` computes the solution of a system of linear algebraic equations having a complex general coefficient matrix. To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either `LFCCG` or `LFTCG`. The solution to  $Ax = b$  is found by solving the triangular systems  $Ly = b$  and  $Ux = y$ . The forward elimination step consists of solving the system  $Ly = b$  by applying the same permutations and elimination operations to  $b$  that were applied to the columns of  $A$  in the factorization routine. The backward substitution step consists of solving the triangular system  $Ux = y$  for  $x$ .

Routines `LFSCG` and `LFICG` both solve a linear system given its *LU* factorization. `LFICG` generally takes more time and produces a more accurate answer than `LFSCG`. Each iteration of the iterative refinement algorithm used by `LFICG` calls `LFSCG`.

The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- FACT0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `FACT` as output from routine `LFCCG/DFCCG` or `LFTCG/DLFTCG`. `FACT` contains the *LU* factorization of the matrix `A`. (Input)
- IPVT0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the pivoting information for the *LU* factorization as output from subroutine `LFCCG/DFCCG` or `LFTCG/DLFTCG`. (Input)
- B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)
- X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse is computed for a complex general  $3 \times 3$  matrix. The input matrix is assumed to be well-conditioned, hence `LFTCG` is used rather than `LFCCG`.

```
! USE IMSL_LIBRARIES
! Declare variables
! PARAMETER (LDA=3, LDFACT=3, N=3)
! INTEGER IPVT(N)
```

```

REAL THIRD
COMPLEX A(LDA,LDA), AINV(LDA,LDA), RJ(N), FACT(LDFACT,LDFACT)
!
! Declare functions
COMPLEX CMLPX
!
! Set values for A
!
! A = (1.0+1.0i 2.0+3.0i 3.0+3.0i)
! (2.0+1.0i 5.0+3.0i 7.0+4.0i)
! (-2.0+1.0i -4.0+4.0i -5.0+3.0i)
!
DATA A/(1.0,1.0), (2.0,1.0), (-2.0,1.0), (2.0,3.0), (5.0,3.0), &
 (-4.0,4.0), (3.0,3.0), (7.0,4.0), (-5.0,3.0)/
!
! Scale A by dividing by three
THIRD = 1.0/3.0
DO 10 I=1, N
 CALL CSSCAL (N, THIRD, A(:,I), 1)
10 CONTINUE
!
! Factor A
CALL LFTCG (A, FACT, IPVT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
CALL CSET (N, (0.0,0.0), RJ, 1)
DO 20 J=1, N
 RJ(J) = CMLPX(1.0,0.0)
!
! RJ is the J-th column of the identity
! matrix so the following LFSCG
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSCG (FACT, IPVT, RJ, AINV(:,J))
 RJ(J) = CMLPX(0.0,0.0)
20 CONTINUE
!
! Print results
CALL WRCRN ('AINV', AINV)
END

```

## Output

```

 AINV
 1 2 3
1 (6.400,-2.800) (-3.800, 2.600) (-2.600, 1.200)
2 (-1.600,-1.800) (0.200, 0.600) (0.400,-0.800)
3 (-0.600, 2.200) (1.200,-1.400) (0.400, 0.200)

```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. The input matrix is assumed to be well-conditioned, hence `LFTCG` is used rather than `LFCCG`. `LFSCG` is called to determine the columns of the inverse. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFTCG_INT
USE LFSCG_INT
USE WRCRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA
INTEGER, ALLOCATABLE :: IPVT0(:)
COMPLEX, ALLOCATABLE :: A(:, :), AINV(:, :), X0(:)
COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), RJ(:, :), RJ0(:)
REAL THIRD
PARAMETER (LDA=3, N=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ (1.0, 1.0), (2.0, 3.0), (3.0, 3.0)/)
 A(2,:) = (/ (2.0, 1.0), (5.0, 3.0), (7.0, 4.0)/)
 A(3,:) = (/ (-2.0, 1.0), (-4.0, 4.0), (-5.0, 3.0)/)
!
! Scale A by dividing by three
 THIRD = 1.0/3.0
 A = A * THIRD
ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDA), FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), IPVT0(MXLDA))
!
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Factor A
CALL LFTCG (A0, FACT0, IPVT0)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0, 0.0)
DO 10 J=1, N
 RJ(J) = (1.0, 0.0)
 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
!
! RJ is the J-th column of the identity
! matrix so the following LFICG
! reference computes the J-th column of
! the inverse of A
 CALL LFSCG (FACT0, IPVT0, RJ0, X0)
 RJ(J) = (0.0, 0.0)
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))

```

```

10 CONTINUE
!
! Print results.
! Only Rank=0 has the solution, AINV.
IF (MP_RANK.EQ.0) CALL WRCRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE (A, AINV)
DEALLOCATE (A0, FACT0, IPVT0, RJ, RJ0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT (MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP ('FINAL')
END

```

## Output

```

 AINV
 1 2 3
1 (6.400,-2.800) (-3.800, 2.600) (-2.600, 1.200)
2 (-1.600,-1.800) (0.200, 0.600) (0.400,-0.800)
3 (-0.600, 2.200) (1.200,-1.400) (0.400, 0.200)

```

---

## LFICG



Uses iterative refinement to improve the solution of a complex general system of linear equations.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the linear system. (Input)

**FACT** — Complex  $N$  by  $N$  matrix containing the  $LU$  factorization of the coefficient matrix  $A$  as output from routine LFCCG/DLFCCG or LFTCG/DLFTCG. (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from routine LFCCG/DLFCCG or LFTCG/DLFTCG. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)

**RES** — Complex vector of length  $N$  containing the residual vector at the improved solution. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of  $FACT$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDFACT = \text{size}(FACT,1)$ .

**IPATH** — Path indicator. (Input)

$IPATH = 1$  means the system  $AX = B$  is solved.

$IPATH = 2$  means the system  $A^H X = B$  is solved.

Default:  $IPATH = 1$ .

## **FORTRAN 90 Interface**

Generic: `CALL LFICG (A, FACT, IPVT, B, X, RES [, ...])`

Specific: The specific interface names are `S_LFICG` and `D_LFICG`.

## **FORTRAN 77 Interface**

Single: `CALL LFICG (N, A, LDA, FACT, LDFACT, IPVT, B, IPATH, X, RES)`

Double: The double precision name is `DLFICG`.

## **ScaLAPACK Interface**

Generic: `CALL LFICG (A0, FACT0, IPVT0, B0, X0, RES0 [, ...])`

Specific: The specific interface names are `S_LFICG` and `D_LFICG`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## **Description**

Routine `LFICG` computes the solution of a system of linear algebraic equations having a complex general coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo an  $LU$  factorization. This may be done by calling either `LFCCG`, or `LFTCG`.

Iterative refinement fails only if the matrix is very ill-conditioned. Routines `LFICG` and `LFSCG` both solve a linear system given its  $LU$  factorization. `LFICG` generally takes more time and produces a more accurate answer than `LFSCG`. Each iteration of the iterative refinement algorithm used by `LFICG` calls `LFSCG`.

## Comments

Informational error

Type Code

3 2 The input matrix is too ill-conditioned for iterative refinement to be effective

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix A. A contains the coefficient matrix of the linear system. (Input)

**FACT0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix FACT as output from routine LFCCG or LFTCG. FACT contains the LU factorization of the matrix A. (Input)

**IPVT0** — Local vector of length MXLDA containing the local portions of the distributed vector IPVT. IPVT contains the pivoting information for the LU factorization as output from subroutine LFCCG or LFTCG. (Input)

**B0** — Complex local vector of length MXLDA containing the local portions of the distributed vector B. B contains the right-hand side of the linear system. (Input)

**X0** — Complex local vector of length MXLDA containing the local portions of the distributed vector X. X contains the solution to the linear system. (Output)

**RES0** — Complex local vector of length MXLDA containing the local portions of the distributed vector RES. RES contains the final correction at the improved solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding  $0.5 + 0.5i$  to the second element.

```
USE LFICG_INT
USE LFCCG_INT
USE WRCRN_INT
USE UMACH_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), NOUT
REAL RCOND
COMPLEX A(LDA,LDA), B(N), X(N), FACT(LDFACT,LDFACT), RES(N)
```

```

!
! COMPLEX Cmplx Declare functions
!
! Set values for A
!
! A = (1.0+1.0i 2.0+3.0i 3.0-3.0i)
! (2.0+1.0i 5.0+3.0i 7.0-5.0i)
! (-2.0+1.0i -4.0+4.0i 5.0+3.0i)
!
! DATA A/(1.0,1.0), (2.0,1.0), (-2.0,1.0), (2.0,3.0), (5.0,3.0), &
! (-4.0,4.0), (3.0,-3.0), (7.0,-5.0), (5.0,3.0)/
!
! Set values for B
! B = (3.0+5.0i 22.0+10.0i -10.0+4.0i)
!
! DATA B/(3.0,5.0), (22.0,10.0), (-10.0,4.0)/
!
! Factor A
! CALL LFCCG (A, FACT, IPVT, RCOND)
!
! Print the L1 condition number
! CALL UMACH (2, NOUT)
! WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Solve the three systems
! DO 10 J=1, 3
! CALL LFICG (A, FACT, IPVT, B, X, RES)
!
! Print results
! CALL WRCRN ('X', X, 1, N, 1)
!
! Perturb B by adding 0.5+0.5i to B(2)
! B(2) = B(2) + Cmplx(0.5,0.5)
! 10 CONTINUE
!
! 99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
! END

```

## Output

```

RCOND < 0.025
L1 Condition number < 75.0
 X
 1 2 3
(1.000,-1.000) (2.000, 4.000) (3.000, 0.000)
 X
 1 2 3
(0.910,-1.061) (1.986, 4.175) (3.123, 0.071)
 X
 1 2 3
(0.821,-1.123) (1.972, 4.349) (3.245, 0.142)

```

## ScaLAPACK Example

The same set of linear systems is solved successively as a distributed example. The right-hand-side vector is perturbed after solving the system each of the first two times by adding  $0.5 + 0.5i$  to the second element. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are

used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFICG_INT
USE LFCCG_INT
USE WRCRN_INT
USE UMACH_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA, NOUT
INTEGER, ALLOCATABLE :: IPVT0(:)
COMPLEX, ALLOCATABLE :: A(:, :), B(:), X(:), X0(:), RES(:)
COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), B0(:), RES0(:)
REAL RCOND
PARAMETER (LDA=3, N=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N), RES(N))
!
! Set values for A and B
 A(1,:) = (/ (1.0, 1.0), (2.0, 3.0), (3.0, 3.0)/)
 A(2,:) = (/ (2.0, 1.0), (5.0, 3.0), (7.0, 4.0)/)
 A(3,:) = (/ (-2.0, 1.0), (-4.0, 4.0), (-5.0, 3.0)/)
!
!
 B = (/ (3.0, 5.0), (22.0, 10.0), (-10.0, 4.0)/)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), &
 B0(MXLDA), IPVT0(MXLDA), RES0(MXLDA))
!
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Factor A
CALL LFCCG (A0, FACT0, IPVT0, RCOND)
!
! Print the l1 condition number
IF (MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
ENDIF
!
! Solve the three systems
DO 10 J=1, 3
 CALL SCALAPACK_MAP(B, DESCL, B0)
 CALL LFICG (A0, FACT0, IPVT0, B0, X0, RES0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X)
!
! Print results

```

```

! Only Rank=0 has the solution, X.
 IF (MP_RANK .EQ. 0) CALL WRCRN ('X', X, 1, N, 1)
! Perturb B by adding 0.5+0.5i to B(2)
 IF(MP_RANK .EQ. 0) B(2) = B(2) + (0.5,0.5)
10 CONTINUE
 IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X, RES)
 DEALLOCATE(A0, B0, FACT0, IPVT0, X0, RES0)
! Exit Scalapack usage
 CALL SCALAPACK_EXIT(MP_ICTXT)

! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
 END

```

## Output

```

RCOND < 0.025
L1 Condition number < 75.0
 X
 1 2 3
(1.000,-1.000) (2.000, 4.000) (3.000, 0.000)

 X
 1 2 3
(0.910,-1.061) (1.986, 4.175) (3.123, 0.071)

 X
 1 2 3
(0.821,-1.123) (1.972, 4.349) (3.245, 0.142)

```

---

## LFDCG

Computes the determinant of a complex general matrix given the *LU* factorization of the matrix.

### Required Arguments

**FACT** — Complex *N* by *N* matrix containing the *LU* factorization of the coefficient matrix *A* as output from routine LFCCG/DFCCG or LFTCG/DFTCG. (Input)

**IPVT** — Vector of length *N* containing the pivoting information for the *LU* factorization of *A* as output from routine LFCCG/DFCCG or LFTCG/DFTCG. (Input)

**DET1** — Complex scalar containing the mantissa of the determinant. (Output)  
The value *DET1* is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or *DET1* = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

## Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

$LDFACT$  — Leading dimension of  $\text{FACT}$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\text{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LFDCG (FACT, IPVT, DET1, DET2 [, ...])`

Specific: The specific interface names are `S_LFDCG` and `D_LFDCG`.

## FORTRAN 77 Interface

Single: `CALL LFDCG (N, FACT, LDFACT, IPVT, DET1, DET2)`

Double: The double precision name is `DLFDCG`.

## Description

Routine `LFDCG` computes the determinant of a complex general coefficient matrix. To compute the determinant the coefficient matrix must first undergo an  $LU$  factorization. This may be done by calling either `LFCCG` or `LFTCG`. The formula  $\det A = \det L \det U$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det U = \prod_{i=1}^N U_{ii}$$

(The matrix  $U$  is stored in the upper triangle of  $\text{FACT}$ .) Since  $L$  is the product of triangular matrices with unit diagonals and of permutation matrices,  $\det L = (-1)^k$  where  $k$  is the number of pivoting interchanges.

`LFDCG` is based on the LINPACK routine `CGEDI`; see Dongarra et al. (1979).

## Example

The determinant is computed for a complex general  $3 \times 3$  matrix.

```
USE LFDCG_INT
USE LFTCG_INT
USE UMACH_INT
!
! Declare variables
PARAMETER (LDA=3, LDFACT=3, N=3)
INTEGER IPVT(N), NOUT
REAL DET2
COMPLEX A(LDA,LDA), FACT(LDFACT,LDFACT), DET1
!
! Set values for A
!
! A = (3.0-2.0i 2.0+4.0i 0.0-3.0i)
```

```

! (1.0+1.0i 2.0-6.0i 1.0+2.0i)
! (4.0+0.0i -5.0+1.0i 3.0-2.0i)
!
DATA A/(3.0,-2.0), (1.0,1.0), (4.0,0.0), (2.0,4.0), (2.0,-6.0), &
 (-5.0,1.0), (0.0,-3.0), (1.0,2.0), (3.0,-2.0)/
!
! Factor A
CALL LFTCG (A, FACT, IPVT)
! Compute the determinant for the
! factored matrix
CALL LFDCG (FACT, IPVT, DET1, DET2)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is',3X,' (',F6.3,',',F6.3,&
 ') * 10**',F2.0)
END

```

## Output

The determinant of A is ( 0.700, 1.100) \* 10\*\*1.

---

## LING



Computes the inverse of a complex general matrix.

### Required Arguments

*A* — Complex  $N$  by  $N$  matrix containing the matrix to be inverted. (Input)

*AINV* — Complex  $N$  by  $N$  matrix containing the inverse of *A*. (Output)

If *A* is not needed, *A* and *AINV* can share the same storage locations.

### Optional Arguments

*N* — Number of equations. (Input)

Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

**LDAINV** — Leading dimension of **AINV** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDAINV = \text{size}(\text{AINV}, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LINGC (A, AINV [, ...])`

Specific: The specific interface names are `S_LINGC` and `D_LINGC`.

## FORTRAN 77 Interface

Single: `CALL LINGC (N, A, LDA, AINV, LDAINV)`

Double: The double precision name is `DLINGC`.

## ScaLAPACK Interface

Generic: `CALL LINGC (A0, AINV0 [, ...])`

Specific: The specific interface names are `S_LINGC` and `D_LINGC`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### Description

Routine `LINGC` computes the inverse of a complex general matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

`LINGC` first uses the routine `LFCCG` to compute an  $LU$  factorization of the coefficient matrix and to estimate the condition number of the matrix. `LFCCG` computes  $U$  and the information needed to compute  $L$ . `LINCT` is then used to compute  $U^{-1}$ , i.e. use the inverse of  $U$ . Finally  $A^{-1}$  is computed using  $A^{-1} = U^{-1}L^{-1}$ .

`LINGC` fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. This errors occurs only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in  $A^{-1}$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2NCG/DL2NCG`. The reference is:

```
CALL L2NCG (N, A, LDA, AINV, LDAINV, WK, IWK)
```

The additional arguments are as follows:

**WK** — Complex work vector of length  $N + N(N - 1)/2$ .

**IWK** — Integer work vector of length  $N$ .

2. Informational errors

Type Code

3 1 The input matrix is too ill-conditioned. The inverse might not be accurate.

4 2 The input matrix is singular.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix to be inverted. (Input)

**AINV0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `AINV`. `AINV` contains the inverse of the matrix `A`. (Output)  
If `A` is not needed, `A` and `AINV` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse is computed for a complex general  $3 \times 3$  matrix.

```
USE LINCG_INT
USE WRCRNG_INT
USE CSSCALG_INT
!
! Declare variables
PARAMETER (LDA=3, LDAINV=3, N=3)
REAL THIRD
COMPLEX A(LDA,LDA), AINV(LDAINV,LDAINV)
!
! Set values for A
!
! A = (1.0+1.0i 2.0+3.0i 3.0+3.0i)
! (2.0+1.0i 5.0+3.0i 7.0+4.0i)
! (-2.0+1.0i -4.0+4.0i -5.0+3.0i)
!
DATA A/(1.0,1.0), (2.0,1.0), (-2.0,1.0), (2.0,3.0), (5.0,3.0), &
 (-4.0,4.0), (3.0,3.0), (7.0,4.0), (-5.0,3.0)/
!
! Scale A by dividing by three
```

```

 THIRD = 1.0/3.0
 DO 10 I=1, N
 CALL CSSCAL (N, THIRD, A(:,I), 1)
10 CONTINUE
! Calculate the inverse of A
 CALL LINCG (A, AINV)
! Print results
 CALL WRCRN ('AINV', AINV)
 END

```

## Output

```

 AINV
 1 2 3
1 (6.400,-2.800) (-3.800, 2.600) (-2.600, 1.200)
2 (-1.600,-1.800) (0.200, 0.600) (0.400,-0.800)
3 (-0.600, 2.200) (1.200,-1.400) (0.400, 0.200)

```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LINCG_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
! Declare variables
 INTEGER J, LDA, N, DESCA(9)
 INTEGER INFO, MXCOL, MXLDA, NPROW, NPCOL
 COMPLEX, ALLOCATABLE :: A(:, :), AINV(:, :)
 COMPLEX, ALLOCATABLE :: A0(:, :), AINV0(:, :)
 REAL THIRD
 PARAMETER (LDA=3, N=3)
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
! Set values for A
 A(1,:) = (/ (1.0, 1.0), (2.0, 3.0), (3.0, 3.0)/)
 A(2,:) = (/ (2.0, 1.0), (5.0, 3.0), (7.0, 4.0)/)
 A(3,:) = (/ (-2.0, 1.0), (-4.0, 4.0), (-5.0, 3.0)/)
! Scale A by dividing by three
 THIRD = 1.0/3.0
 A = A * THIRD
 ENDIF
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)

```

```

! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), AINV0(MXLDA,MXCOL))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Factor A
CALL LINGC (A0, AINV0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(AINV0, DESCA, AINV)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK.EQ.0) CALL WRCRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, AINV0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 AINV
 1 2 3
1 (6.400,-2.800) (-3.800, 2.600) (-2.600, 1.200)
2 (-1.600,-1.800) (0.200, 0.600) (0.400,-0.800)
3 (-0.600, 2.200) (1.200,-1.400) (0.400, 0.200)

```

---

## LSLRT



Solves a real triangular system of linear equations.

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix for the triangular linear system. (Input)  
 For a lower triangular system, only the lower triangular part and diagonal of **A** are referenced. For an upper triangular system, only the upper triangular part and diagonal of **A** are referenced.

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

### Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

$IPATH$  — Path indicator. (Input)  
 $IPATH = 1$  means solve  $AX = B$ ,  $A$  lower triangular.  
 $IPATH = 2$  means solve  $AX = B$ ,  $A$  upper triangular.  
 $IPATH = 3$  means solve  $A^T X = B$ ,  $A$  lower triangular.  
 $IPATH = 4$  means solve  $A^T X = B$ ,  $A$  upper triangular.  
Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic: `CALL LSLRT (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLRT` and `D_LSLRT`.

### FORTRAN 77 Interface

Single: `CALL LSLRT (N, A, LDA, B, IPATH, X)`

Double: The double precision name is `DLSLRT`.

### ScaLAPACK Interface

Generic: `CALL LSLRT (A0, B0, X0 [, ...])`

Specific: The specific interface names are `S_LSLRT` and `D_LSLRT`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### Description

Routine `LSLRT` solves a system of linear algebraic equations with a real triangular coefficient matrix. `LSLRT` fails if the matrix  $A$  has a zero diagonal element, in which case  $A$  is singular. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficients of the linear system. (Input)  
For a lower triangular system, only the lower triangular part and diagonal of `A` are referenced. For an upper triangular system, only the upper triangular part and diagonal of `A` are referenced.

**B0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

A system of three linear equations is solved. The coefficient matrix has lower triangular form and the right-hand-side vector, `b`, has three elements.

```
USE LSLRT_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3)
REAL A(LDA,LDA), B(LDA), X(LDA)
!
! Set values for A and B
!
! A = (2.0)
! (2.0 -1.0)
! (-4.0 2.0 5.0)
!
! B = (2.0 5.0 0.0)
!
DATA A/2.0, 2.0, -4.0, 0.0, -1.0, 2.0, 0.0, 0.0, 5.0/
DATA B/2.0, 5.0, 0.0/
!
! Solve AX = B (IPATH = 1)
CALL LSLRT (A, B, X)
!
! Print results
CALL WRRRN ('X', X, 1, 3, 1)
END
```

## Output

```
 X
 1 2 3
1.000 -3.000 2.000
```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has lower triangular form and the right-hand-side vector  $b$  has three elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
USE MPI_SETUP_INT
USE LSLRT_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

! Declare variables
INTEGER LDA, N, DESCA(9), DESCX(9)
INTEGER INFO, MXCOL, MXLDA
REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
PARAMETER (LDA=3, N=3)

! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 2.0, 0.0, 0.0/)
 A(2,:) = (/ 2.0, -1.0, 0.0/)
 A(3,:) = (/ -4.0, 2.0, 5.0/)
!
 B = (/ 2.0, 5.0, 0.0/)
ENDIF

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
! Solve AX = B (IPATH = 1)
CALL LSLRT (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
```

```

! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRRRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit Scalapack usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
1.000 -3.000 2.000

```

---

## LFCRT



Estimates the condition number of a real triangular matrix.

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix for the triangular linear system. (Input)  
 For a lower triangular system, only the lower triangular part and diagonal of **A** are referenced. For an upper triangular system, only the upper triangular part and diagonal of **A** are referenced.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**.  
 (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
 Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means **A** is lower triangular.

IPATH = 2 means A is upper triangular.  
Default: IPATH = 1.

## FORTRAN 90 Interface

Generic:    CALL LFCRT (A, RCOND [, ...])

Specific:    The specific interface names are S\_LFCRT and D\_LFCRT.

## FORTRAN 77 Interface

Single:     CALL LFCRT (N, A, LDA, IPATH, RCOND)

Double:     The double precision name is DLFCRT.

## ScaLAPACK Interface

Generic:    CALL LFCRT (A0, RCOND [, ...])

Specific:    The specific interface names are S\_LFCRT and D\_LFCRT.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFCRT estimates the condition number of a real triangular matrix. The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ .

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2CRT/DL2CRT. The reference is:

```
CALL L2CRT (N, A, LDA, IPATH, RCOND, WK)
```

The additional argument is:

**WK** — Work vector of length  $N$ .

2. Informational error

Type Code

3 1 The input triangular matrix is algorithmically singular.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0**— MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the coefficient matrix for the triangular linear system. (Input)  
For a lower triangular system, only the lower triangular part and diagonal of A are referenced. For an upper triangular system, only the upper triangular part and diagonal of A are referenced.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

An estimate of the reciprocal condition number is computed for a  $3 \times 3$  lower triangular coefficient matrix.

```
USE LFCRT_INT
USE UMACH_INT
!
! Declare variables
PARAMETER (LDA=3)
REAL A(LDA,LDA), RCOND
INTEGER NOUT
!
! Set values for A and B
! A = (2.0)
! (2.0 -1.0)
! (-4.0 2.0 5.0)
!
DATA A/2.0, 2.0, -4.0, 0.0, -1.0, 2.0, 0.0, 0.0, 5.0/
!
! Compute the reciprocal condition
! number (IPATH=1)
CALL LFCRT (A, RCOND)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END
```

## Output

```
RCOND < 0.1
L1 Condition number < 15.0
```

## ScaLAPACK Example

The same lower triangular matrix as in the example above is used in this distributed computing example. An estimate of the reciprocal condition number is computed for the  $3 \times 3$  lower triangular coefficient matrix. `SCALAPACK_MAP` is an IMSL utility routine (see [Chapter 11, “Utilities”](#)) used to map an array to the processor grid. It is used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
USE MPI_SETUP_INT
USE LFCRT_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

! Declare variables
INTEGER LDA, N, NOUT, DESCA(9)
INTEGER INFO, MXCOL, MXLDA
REAL RCOND
REAL, ALLOCATABLE :: A(:, :)
REAL, ALLOCATABLE :: A0(:, :)
PARAMETER (LDA=3, N=3)

! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N))

! Set values for A
 A(1,:) = (/ 2.0, 0.0, 0.0/)
 A(2,:) = (/ 2.0, -1.0, 0.0/)
 A(3,:) = (/ -4.0, 2.0, 5.0/)
ENDIF

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)

! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)

! Set up the array descriptor
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)

! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL))

! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)

! Compute the reciprocal condition
! number (IPATH=1)
CALL LFCRT (A0, RCOND)

! Print results.
! Only Rank=0 has the solution, RCOND.
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
ENDIF
IF (MP_RANK .EQ. 0) DEALLOCATE(A)
```

```

 DEALLOCATE(A0)
! Exit Scalapack usage
 CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
 END

```

## Output

```

RCOND < 0.1
L1 Condition number < 15.0

```

---

# LFDRT

Computes the determinant of a real triangular matrix.

## Required Arguments

*A* — *N* by *N* matrix containing the triangular matrix. (Input)  
The matrix can be either upper or lower triangular.

*DET1* — Scalar containing the mantissa of the determinant. (Output)  
The value *DET1* is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or *DET1* = 0.0.

*DET2* — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

## Optional Arguments

*N* — Number of equations. (Input)  
Default: *N* = size (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = size (*A*,1).

## FORTRAN 90 Interface

Generic:    CALL LFDRT (A, DET1, DET2 [, ...])

Specific:   The specific interface names are *S\_LFDRT* and *D\_LFDRT*.

## FORTRAN 77 Interface

Single:     CALL LFDRT (N, A, LDA, DET1, DET2)

Double:     The double precision name is *DLFDRT*.

## Description

Routine `LFDRT` computes the determinant of a real triangular coefficient matrix. The determinant of a triangular matrix is the product of the diagonal elements

$$\det A = \prod_{i=1}^N A_{ii}$$

`LFDRT` is based on the LINPACK routine `STRDI`; see Dongarra et al. (1979).

## Comments

Informational error

Type Code

3 1 The input triangular matrix is singular.

## Example

The determinant is computed for a  $3 \times 3$  lower triangular matrix.

```
USE LFDRT_INT
USE UMACH_INT
!
! Declare variables
PARAMETER (LDA=3)
REAL A(LDA,LDA), DET1, DET2
INTEGER NOUT
!
! Set values for A
! A = (2.0)
! (2.0 -1.0)
! (-4.0 2.0 5.0)
!
DATA A/2.0, 2.0, -4.0, 0.0, -1.0, 2.0, 0.0, 0.0, 5.0/
!
! Compute the determinant of A
CALL LFDRT (A, DET1, DET2)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
99999 FORMAT (' The determinant of A is ', F6.3, ' * 10**', F2.0)
END
```

## Output

The determinant of A is -1.000 \* 10\*\*1.

---

# LINRT

Computes the inverse of a real triangular matrix.

## Required Arguments

**A** —  $N$  by  $N$  matrix containing the triangular matrix to be inverted. (Input)

For a lower triangular matrix, only the lower triangular part and diagonal of **A** are

referenced. For an upper triangular matrix, only the upper triangular part and diagonal of *A* are referenced.

*AINV* — *N* by *N* matrix containing the inverse of *A*. (Output)

If *A* is lower triangular, *AINV* is also lower triangular. If *A* is upper triangular, *AINV* is also upper triangular. If *A* is not needed, *A* and *AINV* can share the same storage locations.

### Optional Arguments

*N* — Number of equations. (Input)

Default: *N* = size (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = size (*A*,1).

*IPATH* — Path indicator. (Input)

*IPATH* = 1 means *A* is lower triangular.

*IPATH* = 2 means *A* is upper triangular.

Default: *IPATH* = 1.

*LDAINV* — Leading dimension of *AINV* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDAINV* = size (*AINV*,1).

### FORTRAN 90 Interface

Generic:     CALL LINRT (*A*, *AINV* [, ...])

Specific:    The specific interface names are *S\_LINRT* and *D\_LINRT*.

### FORTRAN 77 Interface

Single:     CALL LINRT (*N*, *A*, *LDA*, *IPATH*, *AINV*, *LDAINV*)

Double:     The double precision name is *DLINRT*.

### Description

Routine *LINRT* computes the inverse of a real triangular matrix. It fails if *A* has a zero diagonal element.

### Example

The inverse is computed for a  $3 \times 3$  lower triangular matrix.

```

USE LINRT_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3)
REAL A(LDA,LDA), AINV(LDA,LDA)
!
! Set values for A
! A = (2.0)
! (2.0 -1.0)
! (-4.0 2.0 5.0)
!
DATA A/2.0, 2.0, -4.0, 0.0, -1.0, 2.0, 0.0, 0.0, 5.0/
!
! Compute the inverse of A
CALL LINRT (A, AINV)
!
! Print results
CALL WRRRN ('AINV', AINV)
END

```

## Output

|   | AINV  |        |       |
|---|-------|--------|-------|
|   | 1     | 2      | 3     |
| 1 | 0.500 | 0.000  | 0.000 |
| 2 | 1.000 | -1.000 | 0.000 |
| 3 | 0.000 | 0.400  | 0.200 |

---

## LSLCT



Solves a complex triangular system of linear equations.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the triangular linear system. (Input)

For a lower triangular system, only the lower triangle of  $A$  is referenced. For an upper triangular system, only the upper triangle of  $A$  is referenced.

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means solve  $AX = B$ ,  $A$  lower triangular  
 $IPATH = 2$  means solve  $AX = B$ ,  $A$  upper triangular  
 $IPATH = 3$  means solve  $A^H X = B$ ,  $A$  lower triangular  
 $IPATH = 4$  means solve  $A^H X = B$ ,  $A$  upper triangular  
Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic: `CALL LSLCT (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLCT` and `D_LSLCT`.

## FORTRAN 77 Interface

Single: `CALL LSLCT (N, A, LDA, B, IPATH, X)`

Double: The double precision name is `DLSLCT`.

## ScaLAPACK Interface

Generic: `CALL LSLCT (A0, B0, X0 [, ...])`

Specific: The specific interface names are `S_LSLCT` and `D_LSLCT`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LSLCT` solves a system of linear algebraic equations with a complex triangular coefficient matrix. `LSLCT` fails if the matrix  $A$  has a zero diagonal element, in which case  $A$  is singular. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

Informational error

Type Code

|   |   |                                                                                       |
|---|---|---------------------------------------------------------------------------------------|
| 4 | 1 | The input triangular matrix is singular. Some of its diagonal elements are near zero. |
|---|---|---------------------------------------------------------------------------------------|

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficient matrix of the triangular linear system. (Input)  
For a lower triangular system, only the lower triangular part and diagonal of `A` are referenced. For an upper triangular system, only the upper triangular part and diagonal of `A` are referenced.

**B0** — Local complex vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Local complex vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

A system of three linear equations is solved. The coefficient matrix has lower triangular form and the right-hand-side vector, `b`, has three elements.

```
USE LSLCT_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA
PARAMETER (LDA=3)
COMPLEX A(LDA,LDA), B(LDA), X(LDA)
!
! Set values for A and B
!
! A = (-3.0+2.0i)
! (-2.0-1.0i 0.0+6.0i)
! (-1.0+3.0i 1.0-5.0i -4.0+0.0i)
!
! B = (-13.0+0.0i -10.0-1.0i -11.0+3.0i)
!
DATA A/(-3.0,2.0), (-2.0,-1.0), (-1.0, 3.0), (0.0,0.0), (0.0,6.0), &
 (1.0,-5.0), (0.0,0.0), (0.0,0.0), (-4.0,0.0)/
DATA B/(-13.0,0.0), (-10.0,-1.0), (-11.0,3.0)/
!
! Solve AX = B
CALL LSLCT (A, B, X)
!
! Print results
CALL WRCRN ('X', X, 1, 3, 1)
END
```

## Output

```

 X
 1 2 3
(3.000, 2.000) (1.000, 1.000) (2.000, 0.000)
```

## ScaLAPACK Example

The same lower triangular matrix as in the example above is used in this distributed computing example. The system of three linear equations is solved. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LSLCT_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 COMPLEX, ALLOCATABLE :: A(:, :), B(:), X(:)
 COMPLEX, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
 PARAMETER (LDA=3, N=3)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
!
! Set values for A
 A(1,:) = (/ (-3.0, 2.0), (0.0, 0.0), (0.0, 0.0)/)
 A(2,:) = (/ (-2.0, -1.0), (0.0, 6.0), (0.0, 0.0)/)
 A(3,:) = (/ (-1.0, 3.0), (1.0, -5.0), (-4.0, 0.0)/)
!
 B = (/ (-13.0, 0.0), (-10.0, -1.0), (-11.0, 3.0)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptor
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
 CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
!
! Map input arrays to the processor grid
 CALL SCALAPACK_MAP(A, DESCA, A0)
 CALL SCALAPACK_MAP(B, DESCX, B0)
!
! Solve AX = B
 CALL LSLCT (A0, B0, X0)
!
! Unmap the results from the distributed
```

```

! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRCRN ('X', X, 1, 3, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
(3.000, 2.000) (1.000, 1.000) (2.000, 0.000)

```

---

## LFCCT



Estimates the condition number of a complex triangular matrix.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the triangular matrix. (Input)  
 For a lower triangular system, only the lower triangle of **A** is referenced. For an upper triangular system, only the upper triangle of **A** is referenced.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
 Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means **A** is lower triangular.

IPATH = 2 means A is upper triangular.  
Default: IPATH = 1.

### **FORTRAN 90 Interface**

Generic:     CALL LFCCT (A, RCOND [, ...])

Specific:    The specific interface names are S\_LFCCT and D\_LFCCT.

### **FORTRAN 77 Interface**

Single:      CALL LFCCT (N, A, LDA, IPATH, RCOND)

Double:      The double precision name is DLFCCT.

### **ScaLAPACK Interface**

Generic:     CALL LFCCT (A0, RCOND [, ...])

Specific:    The specific interface names are S\_LFCCT and D\_LFCCT.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

Routine LFCCT estimates the condition number of a complex triangular matrix. The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979). If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

### **Comments**

1.    Workspace may be explicitly provided, if desired, by use of L2CCT/DL2CCT. The reference is:

```
CALL L2CCT (N, A, LDA, IPATH, RCOND, CWK)
```

The additional argument is:

**CWK** — Complex work vector of length N.

2.    Informational error

| Type | Code |                                                          |
|------|------|----------------------------------------------------------|
| 3    | 1    | The input triangular matrix is algorithmically singular. |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix A. A contains the coefficient matrix of the triangular linear system. (Input)  
For a lower triangular system, only the lower triangular part and diagonal of A are referenced. For an upper triangular system, only the upper triangular part and diagonal of A are referenced.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

An estimate of the reciprocal condition number is computed for a  $3 \times 3$  lower triangular coefficient matrix.

```

 USE LFCCT_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=3)
 INTEGER NOUT
 REAL RCOND
 COMPLEX A(LDA,LDA)
!
! Set values for A
!
! A = (-3.0+2.0i
! (-2.0-1.0i 0.0+6.0i
! (-1.0+3.0i 1.0-5.0i -4.0+0.0i)
!
!
 DATA A/(-3.0,2.0), (-2.0,-1.0), (-1.0, 3.0), (0.0,0.0), (0.0,6.0), &
 (1.0,-5.0), (0.0,0.0), (0.0,0.0), (-4.0,0.0)/
!
! Compute the reciprocal condition
! number
 CALL LFCCT (A, RCOND)
!
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
 END

```

## Output

```
RCOND < 0.2
L1 Condition number < 10.0
```

## ScaLAPACK Example

The same lower triangular matrix as in the example above is used in this distributed computing example. An estimate of the reciprocal condition number is computed for a  $3 \times 3$  lower triangular coefficient matrix. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
USE MPI_SETUP_INT
USE LFCCT_INT
USE UMACH_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER LDA, N, NOUT, DESCA(9)
INTEGER INFO, MXCOL, MXLDA
REAL RCOND
COMPLEX, ALLOCATABLE :: A(:, :)
COMPLEX, ALLOCATABLE :: A0(:, :)
PARAMETER (LDA=3, N=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N))
!
! Set values for A
 A(1,:) = (/ (-3.0, 2.0), (0.0, 0.0), (0.0, 0.0)/)
 A(2,:) = (/ (-2.0, -1.0), (0.0, 6.0), (0.0, 0.0)/)
 A(3,:) = (/ (-1.0, 3.0), (1.0, -5.0), (-4.0, 0.0)/)
ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptor
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Compute the reciprocal condition
! number
CALL LFCCT (A0, RCOND)
!
! Print results.
! Only Rank=0 has the solution, RCOND.
IF (MP_RANK .EQ. 0) THEN
```

```

 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
 ENDIF
 IF (MP_RANK .EQ. 0) DEALLOCATE(A)
 DEALLOCATE(A0)
! Exit ScaLAPACK usage
 CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
 END

```

## Output

```

RCOND < 0.2
L1 Condition number < 10.0

```

---

# LFDCT

Computes the determinant of a complex triangular matrix.

## Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the triangular matrix. (Input)

**DET1** — Complex scalar containing the mantissa of the determinant. (Output)

The value **DET1** is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or **DET1** = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)

The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

## Optional Arguments

**N** — Number of equations. (Input)

Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $\text{LDA} = \text{size}(A,1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFDCT (A, DET1, DET2[, ...])

Specific:   The specific interface names are `S_LFDCT` and `D_LFDCT`.

## FORTRAN 77 Interface

Single:     CALL LFDCT (N, A, LDA, DET1, DET2)

Double: The double precision name is DLFDCT.

## Description

Routine LFDCT computes the determinant of a complex triangular coefficient matrix. The determinant of a triangular matrix is the product of the diagonal elements

$$\det A = \prod_{i=1}^N A_{ii}$$

LFDCT is based on the LINPACK routine CTRDI; see Dongarra et al. (1979).

## Comments

Informational error

Type Code

3 1 The input triangular matrix is singular.

## Example

The determinant is computed for a  $3 \times 3$  complex lower triangular matrix.

```
USE LFDCT_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
INTEGER NOUT
REAL DET2
COMPLEX A(LDA,LDA), DET1
!
! Set values for A
!
! A = (-3.0+2.0i)
! (-2.0-1.0i 0.0+6.0i)
! (-1.0+3.0i 1.0-5.0i -4.0+0.0i)
!
DATA A/(-3.0,2.0), (-2.0,-1.0), (-1.0, 3.0), (0.0,0.0), (0.0,6.0), &
 (1.0,-5.0), (0.0,0.0), (0.0,0.0), (-4.0,0.0)/
!
! Compute the determinant of A
CALL LFDCT (A, DET1, DET2)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
99999 FORMAT (' The determinant of A is (',F4.1,',',F4.1,') * 10**', &
 F2.0)
END
```

## Output

The determinant of A is ( 0.5, 0.7) \* 10\*\*2.

---

# LINCT

Computes the inverse of a complex triangular matrix.

## Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the triangular matrix to be inverted. (Input)  
For a lower triangular matrix, only the lower triangle of  $A$  is referenced. For an upper triangular matrix, only the upper triangle of  $A$  is referenced.

**AINV** — Complex  $N$  by  $N$  matrix containing the inverse of  $A$ . (Output)  
If  $A$  is lower triangular, **AINV** is also lower triangular. If  $A$  is upper triangular, **AINV** is also upper triangular. If  $A$  is not needed,  $A$  and **AINV** can share the same storage locations.

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means  $A$  is lower triangular.  
 $IPATH = 2$  means  $A$  is upper triangular.  
Default:  $IPATH = 1$ .

**LDAINV** — Leading dimension of **AINV** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDAINV = \text{size}(AINV,1)$ .

## FORTRAN 90 Interface

Generic: `CALL LINCT (A, AINV [, ...])`

Specific: The specific interface names are `S_LINCT` and `D_LINCT`.

## FORTRAN 77 Interface

Single: `CALL LINCT (N, A, LDA, IPATH, AINV, LDAINV)`

Double: The double precision name is `DLINCT`.

## Description

Routine `LINCT` computes the inverse of a complex triangular matrix. It fails if `A` has a zero diagonal element.

## Comments

Informational error

Type Code

4 1 The input triangular matrix is singular. Some of its diagonal elements are close to zero.

## Example

The inverse is computed for a  $3 \times 3$  lower triangular matrix.

```
USE LINCT_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA
PARAMETER (LDA=3)
COMPLEX A(LDA,LDA), AINV(LDA,LDA)
!
! Set values for A
!
! A = (-3.0+2.0i)
! (-2.0-1.0i 0.0+6.0i)
! (-1.0+3.0i 1.0-5.0i -4.0+0.0i)
!
DATA A/(-3.0,2.0), (-2.0,-1.0), (-1.0, 3.0), (0.0,0.0), (0.0,6.0), &
 (1.0,-5.0), (0.0,0.0), (0.0,0.0), (-4.0,0.0)/
!
! Compute the inverse of A
CALL LINCT (A, AINV)
!
! Print results
CALL WRCRN ('AINV', AINV)
END
```

## Output

```
 AINV
 1 2 3
1 (-0.2308,-0.1538) (0.0000, 0.0000) (0.0000, 0.0000)
2 (-0.0897, 0.0513) (0.0000,-0.1667) (0.0000, 0.0000)
3 (0.2147,-0.0096) (-0.2083,-0.0417) (-0.2500, 0.0000)
```

---

# LSADS



Solves a real symmetric positive definite system of linear equations with iterative refinement.

## Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix of the symmetric positive definite linear system. (Input)  
Only the upper triangle of **A** is referenced.

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

## FORTRAN 90 Interface

Generic:    `CALL LSADS (A, B, X [, ...])`

Specific:    The specific interface names are `S_LSADS` and `D_LSADS`.

## FORTRAN 77 Interface

Single:     `CALL LSADS (N, A, LDA, B, X)`

Double:     The double precision name is `DLSADS`.

## ScaLAPACK Interface

Generic:    `CALL LSADS (A0, B0, X0 [, ...])`

Specific:    The specific interface names are `S_LSADS` and `D_LSADS`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LSADS` solves a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. `LSADS` first uses the routine `LFCD5` to compute an  $R^T R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix. The matrix  $R$  is upper triangular. The solution of the linear system is then found using the iterative refinement routine `LFIDS`. `LSADS` fails if any submatrix of  $R$  is not positive definite, if  $R$  has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is either very close to a singular matrix or a matrix which is not positive definite. If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. `LSADS` solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2ADS/DL2ADS`. The reference is:

```
CALL L2ADS (N, A, LDA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT**— Work vector of length  $N^2$  containing the  $R^T R$  factorization of  $A$  on output.

**WK** — Work vector of length  $N$ .

2. Informational errors

Type Code

3 1 The input matrix is too ill-conditioned. The solution might not be accurate.

4 2 The input matrix is not positive definite.

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2ADS` the leading dimension of `FACT` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSADS`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSADS`. Users directly calling `L2ADS` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no

longer cause inefficiencies. There is no requirement that users change existing applications that use LSADS or L2ADS. Default values for the option are IVAL(\*) = 1, 16, 0, 1.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSADS temporarily replaces IVAL(2) by IVAL(1). The routine L2CDS computes the condition number if IVAL(2) = 2. Otherwise L2CDS skips this computation. LSADS restores the option. Default values for the option are IVAL(\*) = 1, 2.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the coefficient matrix of the symmetric positive definite linear system. (Input)
- B0** — Local vector of length MXLDA containing the local portions of the distributed vector B. B contains the right-hand side of the linear system. (Input)
- X0** — Local vector of length MXLDA containing the local portions of the distributed vector X. X contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of three linear equations is solved. The coefficient matrix has real positive definite form and the right-hand-side vector  $b$  has three elements.

```

USE LSADS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
REAL A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
! B = (27.0 -78.0 64.0)
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
DATA B/27.0, -78.0, 64.0/
!
CALL LSADS (A, B, X)

```

```

! Print results
 CALL WRRRN ('X', X, 1, N, 1)
!
 END

```

## Output

```

 X
 1 2 3
1.000 -4.000 7.000

```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has real positive definite form and the right-hand-side vector  $b$  has three elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LSADS_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'

! Declare variables
 INTEGER LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
 REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
 PARAMETER (LDA=3, N=3)

! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)

!
 B = (/27.0, -78.0, 64.0/)
 ENDIF

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
 CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))

```

```

! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
! Solve the system of equations
CALL LSADS (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0)CALL WRRRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
1.000 -4.000 7.000

```

---

## LSLDS



Solves a real symmetric positive definite system of linear equations without iterative refinement .

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix of the symmetric positive definite linear system. (Input)

Only the upper triangle of **A** is referenced.

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)

Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = size (*A*,1).

### **FORTRAN 90 Interface**

Generic:     CALL LSLDS (A, B, X [, ...])

Specific:    The specific interface names are S\_LSLDS and D\_LSLDS.

### **FORTRAN 77 Interface**

Single:      CALL LSLDS (N, A, LDA, B, X)

Double:      The double precision name is DLSLDS.

### **ScaLAPACK Interface**

Generic:     CALL LSLDS (A0, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSLDS and D\_LSLDS.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

Routine LSLDS solves a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. LSLDS first uses the routine LFCDS to compute an  $R^T R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix. The matrix *R* is upper triangular. The solution of the linear system is then found using the routine LFSDS. LSLDS fails if any submatrix of *R* is not positive definite or if *R* has a zero diagonal element. These errors occur only if *A* either is very close to a singular matrix or to a matrix which is not positive definite. If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. If the coefficient matrix is ill-conditioned, it is recommended that LSADS be used.

### **Comments**

1.    Workspace may be explicitly provided, if desired, by use of L2LDS/DL2LDS. The reference is:

```
CALL L2LDS (N, A, LDA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing the  $R^T R$  factorization of **A** on output. If **A** is not needed, **A** can share the same storage locations as **FACT**.

**WK** — Work vector of length **N**.

2. Informational errors

Type Code

- |   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4 | 2 | The input matrix is not positive definite.                                   |

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine **L2LDS** the leading dimension of **FACT** is increased by **IVAL(3)** when **N** is a multiple of **IVAL(4)**. The values **IVAL(3)** and **IVAL(4)** are temporarily replaced by **IVAL(1)** and **IVAL(2)**, respectively, in **LSLDS**. Additional memory allocation for **FACT** and option value restoration are done automatically in **LSLDS**. Users directly calling **L2LDS** can allocate additional space for **FACT** and set **IVAL(3)** and **IVAL(4)** so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use **LSLDS** or **L2LDS**. Default values for the option are **IVAL(\*) = 1, 16, 0, 1**.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine **LSLDS** temporarily replaces **IVAL(2)** by **IVAL(1)**. The routine **L2CDS** computes the condition number if **IVAL(2) = 2**. Otherwise **L2CDS** skips this computation. **LSLDS** restores the option. Default values for the option are **IVAL(\*) = 1, 2**.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — **MXLDA** by **MXCOL** local matrix containing the local portions of the distributed matrix **A**. **A** contains the coefficient matrix of the symmetric positive definite linear system. (Input)
- B0** — Local vector of length **MXLDA** containing the local portions of the distributed vector **B**. **B** contains the right-hand side of the linear system. (Input)
- X0** — Local vector of length **MXLDA** containing the local portions of the distributed vector **X**. **X** contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, **MXLDA** and **MXCOL** can be obtained through a call

to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of three linear equations is solved. The coefficient matrix has real positive definite form and the right-hand-side vector  $b$  has three elements.

```

 USE LSLDS_INT
 USE WRRRN_INT
!
! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=3, N=3)
 REAL A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
! B = (27.0 -78.0 64.0)
!
 DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
 DATA B/27.0, -78.0, 64.0/
!
 CALL LSLDS (A, B, X)
!
! Print results
 CALL WRRRN ('X', X, 1, N, 1)
!
 END

```

## Output

```

 X
 1 2 3
1.000 -4.000 7.000

```

## ScaLAPACK Example

The same system of three linear equations is solved as a distributed computing example. The coefficient matrix has real positive definite form and the right-hand-side vector  $b$  has three elements. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LSLDS_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'

```

```

!
! Declare variables
INTEGER LDA, N, DESCA(9), DESCX(9)
INTEGER INFO, MXCOL, MXLDA
REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
PARAMETER (LDA=3, N=3)

! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)
!
 B = (/27.0, -78.0, 64.0/)
ENDIF

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
! Solve the system of equations
CALL LSLDS (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0) CALL WRRRN ('X', X, 1, N, 1)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

|       | X     |        |       |
|-------|-------|--------|-------|
|       | 1     | 2      | 3     |
| 1.000 | 1.000 | -4.000 | 7.000 |

---

## LFCDS



Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix and estimate its  $L_1$  condition number.

### Required Arguments

**A** —  $N$  by  $N$  symmetric positive definite matrix to be factored. (Input)  
Only the upper triangle of **A** is referenced.

**FACT** —  $N$  by  $N$  matrix containing the upper triangular matrix  $R$  of the factorization of **A** in the upper triangular part. (Output)  
Only the upper triangle of **FACT** will be used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:    `CALL LFCDS (A, FACT, RCOND [, ...])`

Specific:    The specific interface names are `S_LFCDS` and `D_LFCDS`.

## FORTRAN 77 Interface

Single:      CALL LFCDS (N, A, LDA, FACT, LDFACT, RCOND)

Double:      The double precision name is DLFCDS.

## ScaLAPACK Interface

Generic:      CALL LFCDS (A0, FACT0, RCOND [, ...])

Specific:     The specific interface names are S\_LFCDS and D\_LFCDS.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFCDS computes an  $R^T R$  Cholesky factorization and estimates the condition number of a real symmetric positive definite coefficient matrix. The matrix  $R$  is upper triangular.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\varepsilon$  (where  $\varepsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCDS fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^T R$  factors are returned in a form that is compatible with routines [LFIDS](#), [LFSDS](#) and [LFDSDS](#). To solve systems of equations with multiple right-hand-side vectors, use LFCDS followed by either LFIDS or LFSDS called once for each right-hand side. The routine LFDSDS can be called to compute the determinant of the coefficient matrix after LFCDS has performed the factorization.

## Comments

1.      Workspace may be explicitly provided, if desired, by use of L2CDS/DL2CDS. The reference is:

```
CALL L2CDS (N, A, LDA, FACT, LDFACT, RCOND, WK)
```

The additional argument is:

**WK** — Work vector of length  $N$ .

2.      Informational errors

Type   Code

- 3 1 The input matrix is algorithmically singular.
- 4 2 The input matrix is not positive definite.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the symmetric positive definite matrix to be factored. (Input)

**FACT0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix FACT. FACT contains the upper triangular matrix *R* of the factorization of A in the upper triangular part. (Output)  
Only the upper triangle of FACT will be used. If A is not needed, A and FACT can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFCDS is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. LFIDS is called to determine the columns of the inverse.

```

USE LFCDS_INT
USE UMACH_INT
USE WRRRN_INT
USE LFIDS_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NOUT
PARAMETER (LDA=3, LDFACT=3, N=3)
REAL A(LDA,LDA), AINV(LDA,LDA), RCOND, FACT(LDFACT,LDFACT), &
 RES(N), RJ(N)
!
! Set values for A
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
! Factor the matrix A
CALL LFCDS (A, FACT, RCOND)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFIDS

```

```

! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFIDS (A, FACT, RJ, AINV(:,J), RES)
 RJ(J) = 0.0E0
10 CONTINUE
! Print the results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
 CALL WRRRN ('AINV', AINV)
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F9.3)
 END

```

## Output

```

RCOND < 0.005
L1 Condition number < 875.0

```

```

 AINV
 1 2 3
1 35.00 8.00 -5.00
2 8.00 2.00 -1.00
3 -5.00 -1.00 1.00

```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. `LFCDSD` is called to factor the matrix and to check for singularity or ill-conditioning. `LFIDS` is called to determine the columns of the inverse. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFCDS_INT
 USE UMACH_INT
 USE LFIDS_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
! Declare variables
 INTEGER J, LDA, N, NOUT, DESCA(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL, ALLOCATABLE :: A(:,,:), AINV(:,,:), X0(:), RJ(:)
 REAL, ALLOCATABLE :: A0(:,,:), FACT0(:,,:), RES0(:), RJ0(:)
 REAL RCOND
 PARAMETER (LDA=3, N=3)
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
! Set values for A
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)

```

```

 A(3,:) = (/ 2.0, -5.0, 6.0/)

ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), RES0(MXLDA))
!
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Call the factorization routine
CALL LFCDS (A0, FACT0, RCOND)
!
! Print the reciprocal condition number
! and the L1 condition number
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
ENDIF
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0
!
! Map input array to the processor grid
 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIDS
! reference computes the J-th column of
! the inverse of A
 CALL LFIDS (A0, FACT0, RJ0, X0, RES0)
 RJ(J) = 0.0
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))
10 CONTINUE
!
! Print results.
! Only Rank=0 has the solution, AINV.
IF(MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, FACT0, RJ, RJ0, RES0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F9.3)
END

```

## Output

RCOND < 0.005  
L1 Condition number < 875.0

|   | AINV  |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 35.00 | 8.00  | -5.00 |
| 2 | 8.00  | 2.00  | -1.00 |
| 3 | -5.00 | -1.00 | 1.00  |

---

## LFTDS



Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix.

### Required Arguments

**A** —  $N$  by  $N$  symmetric positive definite matrix to be factored. (Input)  
Only the upper triangle of **A** is referenced.

**FACT** —  $N$  by  $N$  matrix containing the upper triangular matrix  $R$  of the factorization of **A** in the upper triangle, and the lower triangular matrix  $R^T$  in the lower triangle. (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage location.

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDA} = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\mathbf{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:    `CALL LFTDS (A, FACT [, ...])`

Specific:    The specific interface names are `S_LFTDS` and `D_LFTDS`.

## FORTRAN 77 Interface

Single:      CALL LFTDS (N, A, LDA, FACT, LDFACT)

Double:      The double precision name is DLFTDS.

## ScaLAPACK Interface

Generic:      CALL LFTDS (A0, FACT0 [, ...])

Specific:     The specific interface names are S\_LFTDS and D\_LFTDS.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFTDS computes an  $R^T R$  Cholesky factorization of a real symmetric positive definite coefficient matrix. The matrix  $R$  is upper triangular.

LFTDS fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^T R$  factors are returned in a form that is compatible with routines [LFIDS](#), [LFSDS](#) and [LFDDS](#). To solve systems of equations with multiple right-hand-side vectors, use LFTDS followed by either LFIDS or LFSDS called once for each right-hand side. The routine LFDDS can be called to compute the determinant of the coefficient matrix after LFTDS has performed the factorization.

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

Informational error

Type   Code

4      2            The input matrix is not positive definite.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the symmetric positive definite matrix to be factored. (Input)

**FACT0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix FACT. FACT contains the upper triangular matrix  $R$  of the factorization of A in the upper triangular part. (Output)

Only the upper triangle of `FACT` will be used. If `A` is not needed, `A` and `FACT` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse of a  $3 \times 3$  matrix is computed. `LFTDS` is called to factor the matrix and to check for nonpositive definiteness. `LFSDS` is called to determine the columns of the inverse.

```

USE LFTDS_INT
USE LFSDS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=3, LDFACT=3, N=3)
REAL A(LDA,LDA), AINV(LDA,LDA), FACT(LDFACT,LDFACT), RJ(N)
!
! Set values for A
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
! Factor the matrix A
CALL LFTDS (A, FACT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFSDS
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSDS (FACT, RJ, AINV(:,J))
 RJ(J) = 0.0E0
10 CONTINUE
!
! Print the results
CALL WRRRN ('AINV', AINV)
!
END

```

## Output

|   | AINV  |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 35.00 | 8.00  | -5.00 |
| 2 | 8.00  | 2.00  | -1.00 |
| 3 | -5.00 | -1.00 | 1.00  |

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. LFTDS is called to factor the matrix and to check for nonpositive definiteness. LFSDS is called to determine the columns of the inverse. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see Chapter 11, “Utilities”) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
USE MPI_SETUP_INT
USE LFTDS_INT
USE UMACH_INT
USE LFSDS_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'

!
! Declare variables
INTEGER J, LDA, N, DESCA(9), DESCL(9)
INTEGER INFO, MXCOL, MXLDA
REAL, ALLOCATABLE :: A(:, :), AINV(:, :), X0(:)
REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), RES0(:), RJ0(:)
PARAMETER (LDA=3, N=3)

!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)

 ENDIF

!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDA), FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), RES0(MXLDA), IPVT0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Call the factorization routine
CALL LFTDS (A0, FACT0)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0
```

```

 CALL SCALAPACK_MAP(RJ, DESCL, RJ0)
! RJ is the J-th column of the identity
! matrix so the following LFSDS
! reference computes the J-th column of
! the inverse of A
 CALL LFSDS (FACT0, RJ0, X0)
 RJ(J) = 0.0
 CALL SCALAPACK_UNMAP(X0, DESCL, AINV(:,J))
10 CONTINUE
! Print results.
! Only Rank=0 has the solution, AINV.
 IF (MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
 IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
 DEALLOCATE(A0, FACT0, IPVT0, RJ, RJ0, RES0, X0)
! Exit ScaLAPACK usage
 CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
 END

```

## Output

RCOND < 0.005  
L1 Condition number < 875.0

|   | AINV  |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 35.00 | 8.00  | -5.00 |
| 2 | 8.00  | 2.00  | -1.00 |
| 3 | -5.00 | -1.00 | 1.00  |

---

## LFSDS



Solves a real symmetric positive definite system of linear equations given the  $R^T R$  Cholesky factorization of the coefficient matrix.

### Required Arguments

**FACT** —  $N$  by  $N$  matrix containing the  $R^T R$  factorization of the coefficient matrix  $A$  as output from routine LFCDS/DLFCDS or LFTDS/DLFTDS. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

## Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

$LDFACT$  — Leading dimension of  $\text{FACT}$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\text{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic:     `CALL LFSDS (FACT, B, X [, ...])`

Specific:    The specific interface names are `S_LFSDS` and `D_LFSDS`.

## FORTRAN 77 Interface

Single:     `CALL LFSDS (N, FACT, LDFACT, B, X)`

Double:     The double precision name is `DLFSDS`.

## ScaLAPACK Interface

Generic:     `CALL LFSDS (FACT0, B0, X0 [, ...])`

Specific:    The specific interface names are `S_LFSDS` and `D_LFSDS`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFSDS` computes the solution for a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either `LFCD`s or `LFTD`s.  $R$  is an upper triangular matrix.

The solution to  $Ax = b$  is found by solving the triangular systems  $R^T y = b$  and  $Rx = y$ .

`LFSDS` and `LFIDS` both solve a linear system given its  $R^T R$  factorization. `LFIDS` generally takes more time and produces a more accurate answer than `LFSDS`. Each iteration of the iterative refinement algorithm used by `LFIDS` calls `LFSDS`.

The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

Informational error

| Type | Code |                               |
|------|------|-------------------------------|
| 4    | 1    | The input matrix is singular. |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**FACT0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `FACT`. `FACT` contains the  $R^T R$  factorization of the coefficient matrix `A` as output from routine `LFCDSD/DLFCDS` or `LFTDSD/DLFTDS`. (Input)

**B0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A set of linear systems is solved successively. `LFTDSD` is called to factor the coefficient matrix. `LFSDSD` is called to compute the four solutions for the four right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCDSD` to perform the factorization, and `LFIDS` to compute the solutions.

```

USE LFSDS_INT
USE LFTDS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=3, LDFACT=3, N=3)
REAL A(LDA,LDA), B(N,4), FACT(LDFACT,LDFACT), X(N,4)
!
! Set values for A and B
!
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
! B = (-1.0 3.6 -8.0 -9.4)
! (-3.0 -4.2 11.0 17.6)
! (-3.0 -5.2 -6.0 -23.4)
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
DATA B/-1.0, -3.0, -3.0, 3.6, -4.2, -5.2, -8.0, 11.0, -6.0, &
 -9.4, 17.6, -23.4/
!
! Factor the matrix A
CALL LFTDS (A, FACT)

```

```

! Compute the solutions
 DO 10 I=1, 4
 CALL LFSDS (FACT, B(:,I), X(:,I))
10 CONTINUE
! Print solutions
 CALL WRRRN ('The solution vectors are', X)
!
 END

```

## Output

```

 The solution vectors are
 1 2 3 4
1 -44.0 118.4 -162.0 -71.2
2 -11.0 25.6 -36.0 -16.6
3 5.0 -19.0 23.0 6.0

```

## ScaLAPACK Example

The same set of linear systems is solved successively as a distributed example. Routine `LFTDS` is called to factor the coefficient matrix. The routine `LFSDS` is called to compute the four solutions for the four right-hand sides. In this case, the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCD` to perform the factorization, and `LFIDS` to compute the solutions. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFSDS_INT
 USE LFTDS_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, DESCA(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL, ALLOCATABLE :: A(:, :), B(:, :), X(:, :), X0(:)
 REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), B0(:)
 PARAMETER (LDA=3, N=3)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
! Set values for A and B
 ALLOCATE (A(LDA,N), B(N,4), X(N,4))
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)
!
 B(1,:) = (/ -1.0, 3.6, -8.0, -9.4/)
 B(2,:) = (/ -3.0, -4.2, 11.0, 17.6/)
 B(3,:) = (/ -3.0, -5.2, -6.0, -23.4/)
 ENDIF

```

```

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), B0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Call the factorization routine
CALL LFTDS (A0, FACT0)
! Set up the columns of the B
! matrix one at a time in X0
DO 10 J=1, 4
 CALL SCALAPACK_MAP(B(:,j), DESCL, B0)
! Solve for the J-th column of X
 CALL LFSDS (FACT0, B0, X0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X(:,J))
10 CONTINUE
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK.EQ.0) CALL WRRRN ('The solution vectors are', X)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, FACT0, B0, X0)
! Exit Scalapack usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

The solution vectors are
 1 2 3 4
1 -44.0 118.4 -162.0 -71.2
2 -11.0 25.6 -36.0 -16.6
3 5.0 -19.0 23.0 6.0

```

---

## LFIDS



Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.

## Required Arguments

**A** —  $N$  by  $N$  matrix containing the symmetric positive definite coefficient matrix of the linear system. (Input)

Only the upper triangle of **A** is referenced.

**FACT** —  $N$  by  $N$  matrix containing the  $R^T R$  factorization of the coefficient matrix **A** as output from routine LFCDS/DLFCDS or LFTDS/DLFTDS. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, **B** and **X** can share the same storage locations.

**RES** — Vector of length  $N$  containing the residual vector at the improved solution. (Output)

## Optional Arguments

**N** — Number of equations. (Input)

Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic:     `CALL LFIDS (A, FACT, B, X, RES [, ...])`

Specific:    The specific interface names are `S_LFIDS` and `D_LFIDS`.

## FORTRAN 77 Interface

Single:     `CALL LFIDS (N, A, LDA, FACT, LDFACT, B, X, RES)`

Double:     The double precision name is `DLFIDS`.

## ScaLAPACK Interface

Generic:     `CALL LFIDS (A0, FACT0, B0, X0, RES0 [, ...])`

Specific:    The specific interface names are `S_LFIDS` and `D_LFIDS`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFIDS` computes the solution of a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

To compute the solution, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either `LFCD`s or `LFTD`s.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIDS` and `LFSDS` both solve a linear system given its  $R^T R$  factorization. `LFIDS` generally takes more time and produces a more accurate answer than `LFSDS`. Each iteration of the iterative refinement algorithm used by `LFIDS` calls `LFSDS`.

## Comments

Informational error

Type Code

|   |   |                                                                                   |
|---|---|-----------------------------------------------------------------------------------|
| 3 | 2 | The input matrix is too ill-conditioned for iterative refinement to be effective. |
|---|---|-----------------------------------------------------------------------------------|

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the symmetric positive definite coefficient matrix of the linear system. (Input)

**FACT0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `FACT`. `FACT` contains the  $R^T R$  factorization of the coefficient matrix `A` as output from routine `LFCD`s/`DLFCDS` or `LFTD`s/`DLFTDS`. (Input)

**B0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

**RES0** — Local vector of length `MXLDA` containing the local portions of the distributed vector `RES`. `RES` contains the residual vector at the improved solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.2 to the second element.

```

USE LFIDS_INT
USE LFCDS_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=3, LDFACT=3, N=3)
REAL A(LDA,LDA), B(N), RCOND, FACT(LDFACT,LDFACT), RES(N,3), &
 X(N,3)
!
! Set values for A and B
!
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
! B = (1.0 -3.0 2.0)
!
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
DATA B/1.0, -3.0, 2.0/
!
! Factor the matrix A
CALL LFCDS (A, FACT, RCOND)
!
! Print the estimated condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Compute the solutions
DO 10 I=1, 3
 CALL LFIDS (A, FACT, B, X(:,I), RES(:,I))
 B(2) = B(2) + .2E0
10 CONTINUE
!
! Print solutions and residuals
CALL WRRRN ('The solution vectors are', X)
CALL WRRRN ('The residual vectors are', RES)
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F9.3)
END

```

## Output

```

RCOND = 0.001
L1 Condition number = 674.727

The solution vectors are
 1 2 3

```

```

1 1.000 2.600 4.200
2 0.000 0.400 0.800
3 0.000 -0.200 -0.400

```

The residual vectors are

```

 1 2 3
1 0.0000 0.0000 0.0000
2 0.0000 0.0000 0.0000
3 0.0000 0.0000 0.0000

```

## ScaLAPACK Example

The same set of linear systems is solved successively as a distributed example. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.2 to the second element. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFIDS_INT
 USE LFCDS_INT
 USE UMACH_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, NOUT, DESCA(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL RCOND
 REAL, ALLOCATABLE :: A(:, :), B(:), X(:, :), RES(:, :), X0(:)
 REAL, ALLOCATABLE :: A0(:, :), FACT0(:, :), B0(:), RES0(:)
 PARAMETER (LDA=3, N=3)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N,3), RES(N,3))
! Set values for A and B
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)
!
 B = (/ 1.0, -3.0, 2.0/)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP ICTXT, MXLDA, INFO)
 CALL DESCINIT(DESCL, N, 1, MP_MB, 1, 0, 0, MP ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDA), FACT0(MXLDA,MXCOL), B0(MXLDA), &

```

```

 RES0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Call the factorization routine
CALL LFCDS (A0, FACT0, RCOND)
!
! Print the estimated condition number
CALL UMACH (2, NOUT)
IF(MP_RANK .EQ. 0) WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Set up the columns of the B
! matrix one at a time in X0
DO 10 J=1, 3
 CALL SCALAPACK_MAP(B, DESCL, B0)
!
! Solve for the J-th column of X
 CALL LFIDS (A0, FACT0, B0, X0, RES0)
 CALL SCALAPACK_UNMAP(X0, DESCL, X(:,J))
 CALL SCALAPACK_UNMAP(RES0, DESCL, RES(:,J))
 IF(MP_RANK .EQ. 0) B(2) = B(2) + .2E0
10 CONTINUE
!
! Print results.
! Only Rank=0 has the full arrays
IF(MP_RANK.EQ.0) CALL WRRRN ('The solution vectors are', X)
IF(MP_RANK.EQ.0) CALL WRRRN ('The residual vectors are', RES)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X, RES)
DEALLOCATE(A0, B0, FACT0, RES0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_IXTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F9.3)
END

```

## Output

```

RCOND = 0.001
L1 Condition number = 674.727

The solution vectors are
 1 2 3
1 1.000 2.600 4.200
2 0.000 0.400 0.800
3 0.000 -0.200 -0.400

The residual vectors are
 1 2 3
1 0.0000 0.0000 0.0000
2 0.0000 0.0000 0.0000
3 0.0000 0.0000 0.0000

```

---

## LFDDS

Computes the determinant of a real symmetric positive definite matrix given the  $R^T R$  Cholesky factorization of the matrix .

## Required Arguments

**FACT** —  $N$  by  $N$  matrix containing the  $R^T R$  factorization of the coefficient matrix  $A$  as output from routine `LFCDSD/DLFCDS` or `LFTDSD/DLFTDS`. (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)  
The value `DET1` is normalized so that,  $1.0 \leq |\text{DET1}| < 10.0$  or `DET1` = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form,  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

## Optional Arguments

**N** — Number of equations. (Input)  
Default: `N = size (FACT,2)`.

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDFACT = size (FACT,1)`.

## FORTRAN 90 Interface

Generic: `CALL LFDDS (FACT, DET1, DET2 [, ...])`

Specific: The specific interface names are `S_LFDDS` and `D_LFDDS`.

## FORTRAN 77 Interface

Single: `CALL LFDDS (N, FACT, LDFACT, DET1, DET2)`

Double: The double precision name is `DLFDDS`.

## Description

Routine `LFDDS` computes the determinant of a real symmetric positive definite coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either `LFCDSD` or `LFTDSD`. The formula  $\det A = \det R^T \det R = (\det R)^2$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det R = \prod_{i=1}^N R_{ii}$$

(The matrix  $R$  is stored in the upper triangle of `FACT`.)

`LFDDS` is based on the LINPACK routine `SPODI`; see Dongarra et al. (1979).

## Example

The determinant is computed for a real positive definite  $3 \times 3$  matrix.

```

USE LFDDS_INT
USE LFTDS_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, LDFACT, NOUT
PARAMETER (LDA=3, LDFACT=3)
REAL A(LDA,LDA), DET1, DET2, FACT(LDFACT,LDFACT)
!
! Set values for A
! A = (1.0 -3.0 2.0)
! (-3.0 20.0 -5.0)
! (2.0 -5.0 6.0)
!
DATA A/1.0, -3.0, 2.0, -3.0, 20.0, -5.0, 2.0, -5.0, 6.0/
! Factor the matrix
CALL LFTDS (A, FACT)
! Compute the determinant
CALL LFDDS (FACT, DET1, DET2)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is ',F6.3,' * 10**',F2.0)
END

```

## Output

The determinant of A is 2.100 \* 10\*\*1.

---

## LINDS



Computes the inverse of a real symmetric positive definite matrix.

### Required Arguments

*A* — *N* by *N* matrix containing the symmetric positive definite matrix to be inverted. (Input)  
Only the upper triangle of *A* is referenced.

*AINV* — *N* by *N* matrix containing the inverse of *A*. (Output)  
If *A* is not needed, *A* and *AINV* can share the same storage locations.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = size (*A*,2).

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDAINV** — Leading dimension of  $A_{INV}$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDAINV = \text{size}(A_{INV},1)$ .

## FORTRAN 90 Interface

Generic: `CALL LINDS (A, AINV [, ...])`

Specific: The specific interface names are `S_LINDS` and `D_LINDS`.

## FORTRAN 77 Interface

Single: `CALL LINDS (N, A, LDA, AINV, LDAINV)`

Double: The double precision name is `DLINDS`.

## ScaLAPACK Interface

Generic: `CALL LINDS (A0, AINV0 [, ...])`

Specific: The specific interface names are `S_LINDS` and `D_LINDS`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LINDS` computes the inverse of a real symmetric positive definite matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. `LINDS` first uses the routine `LFCD5` to compute an  $R^T R$  factorization of the coefficient matrix and to estimate the condition number of the matrix. `LINRT` is then used to compute  $R^{-1}$ . Finally  $A^{-1}$  is computed using  $A^{-1} = R^{-1} R^T$ .

`LINDS` fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in  $A$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2NDS`/`DL2NDS`. The reference is:

```
CALL L2NDS (N, A, LDA, AINV, LDAINV, WK)
```

The additional argument is:

**WK** — Work vector of length *N*.

2. Informational errors

Type Code

|   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4 | 2 | The input matrix is not positive definite.                                   |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — *MXLDA* by *MXCOL* local matrix containing the local portions of the distributed matrix *A*. *A* contains the symmetric positive definite matrix to be inverted. (Input)

**AINV0** — *MXLDA* by *MXCOL* local matrix containing the local portions of the distributed matrix *AINV*. *AINV* contains the inverse of the matrix *A*. (Output)  
If *A* is not needed, *A* and *AINV* can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, *MXLDA* and *MXCOL* can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse is computed for a real positive definite  $3 \times 3$  matrix.

```
USE LINDS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDAINV
PARAMETER (LDA=3, LDAINV=3)
REAL A(LDA,LDA), AINV(LDAINV,LDAINV)
!
! Set values for A
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
!
CALL LINDS (A, AINV)
! Print results
CALL WRRRN ('AINV', AINV)
!
```

END

## Output

```
 AINV
 1 2 3
1 35.00 8.00 -5.00
2 8.00 2.00 -1.00
3 -5.00 -1.00 1.00
```

## ScaLAPACK Example

The inverse of the same  $3 \times 3$  matrix is computed as a distributed example. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Chapter 11, “Utilities”](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
 USE MPI_SETUP_INT
 USE LINDS_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, LDFACT, N, DESCA(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL, ALLOCATABLE :: A(:, :), AINV(:, :)
 REAL, ALLOCATABLE :: A0(:, :), AINV0(:, :)
 PARAMETER (LDA=3, N=3)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
! Set values for A
 A(1,:) = (/ 1.0, -3.0, 2.0/)
 A(2,:) = (/ -3.0, 10.0, -5.0/)
 A(3,:) = (/ 2.0, -5.0, 6.0/)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
 ALLOCATE(A0(MXLDA,MXCOL), AINV0(MXLDA,MXCOL))
!
! Map input arrays to the processor grid
 CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Call the routine to get the inverse
 CALL LINDS (A0, AINV0)
!
! Unmap the results from the distributed
! arrays back to a nondistributed array.
```

```

! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(AINV0, DESCA, AINV)
! Print results.
! Only Rank=0 has the solution, AINV.
IF (MP_RANK.EQ.0) CALL WRRRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, AINV0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

|   | AINV  |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 35.00 | 8.00  | -5.00 |
| 2 | 8.00  | 2.00  | -1.00 |
| 3 | -5.00 | -1.00 | 1.00  |

---

## LSASF



Solves a real symmetric system of linear equations with iterative refinement.

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix of the symmetric linear system. (Input)  
Only the upper triangle of **A** is referenced.

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

## FORTRAN 90 Interface

Generic:     CALL LSASF (A, B, X [, ...])

Specific:    The specific interface names are S\_LSASF and D\_LSASF.

## FORTRAN 77 Interface

Single:      CALL LSASF (N, A, LDA, B, X)

Double:     The double precision name is DLSASF.

## Description

Routine LSASF solves systems of linear algebraic equations having a real symmetric indefinite coefficient matrix. It first uses the routine LFCSF to compute a  $U D U^T$  factorization of the coefficient matrix and to estimate the condition number of the matrix.  $D$  is a block diagonal matrix with blocks of order 1 or 2, and  $U$  is a matrix composed of the product of a permutation matrix and a unit upper triangular matrix. The solution of the linear system is then found using the iterative refinement routine LFISF.

LSASF fails if a block in  $D$  is singular or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. LSASF solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1.     Workspace may be explicitly provided, if desired, by use of L2ASF/DL2ASF. The reference is

```
CALL L2ASF (N, A, LDA, B, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing information about the  $U D U^T$  factorization of  $A$  on output. If  $A$  is not needed,  $A$  and **FACT** can share the same storage location.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the factorization of  $A$  on output.

**WK** — Work vector of length  $N$ .

2.     Informational errors

Type   Code

- 3 1 The input matrix is too ill-conditioned. The solution might not be accurate.
- 4 2 The input matrix is singular.

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2ASF` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSASF`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSASF`. Users directly calling `L2ASF` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSASF` or `L2ASF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSASF` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CSF` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CSF` skips this computation. `LSASF` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

### Example

A system of three linear equations is solved. The coefficient matrix has real symmetric form and the right-hand-side vector  $b$  has three elements.

```

USE LSASF_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, N=3)
REAL A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
! B = (4.1 -4.7 6.5)
!
!
DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
DATA B/4.1, -4.7, 6.5/
!
CALL LSASF (A, B, X)
!
! Print results
CALL WRRRN ('X', X, 1, N, 1)
END

```

## Output

|  | X      |        |       |
|--|--------|--------|-------|
|  | 1      | 2      | 3     |
|  | -4.100 | -3.500 | 1.200 |

---

## LSSSF



Solves a real symmetric system of linear equations without iterative refinement .

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix of the symmetric linear system. (Input)  
Only the upper triangle of **A** is referenced.

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

### FORTRAN 90 Interface

Generic:    CALL LSSSF (A, B, X [, ...])

Specific:   The specific interface names are S\_LSSSF and D\_LSSSF.

### FORTRAN 77 Interface

Single:     CALL LSSSF (N, A, LDA, B, X)

Double:     The double precision name is DLSSSF.

## Description

Routine `L2LSF` solves systems of linear algebraic equations having a real symmetric indefinite coefficient matrix. It first uses the routine `LFCSF` to compute a  $UDU^T$  factorization of the coefficient matrix.  $D$  is a block diagonal matrix with blocks of order 1 or 2, and  $U$  is a matrix composed of the product of a permutation matrix and a unit upper triangular matrix.

The solution of the linear system is then found using the routine `LFSSF`.

`L2LSF` fails if a block in  $D$  is singular. This occurs only if  $A$  either is singular or is very close to a singular matrix.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LSF/DL2LSF`. The reference is:

```
CALL L2LSF (N, A, LDA, B, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing information about the  $UDU^T$  factorization of  $A$  on output. If  $A$  is not needed,  $A$  and **FACT** can share the same storage locations.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the factorization of  $A$  on output.

**WK** — Work vector of length  $N$ .

2. Informational errors

Type Code

3 1 The input matrix is too ill-conditioned. The solution might not be accurate.

4 2 The input matrix is singular.

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LSF` the leading dimension of **FACT** is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `L2LSF`. Additional memory allocation for **FACT** and option value restoration are done automatically in `L2LSF`. Users directly calling `L2LSF` can allocate additional space for **FACT** and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `L2LSF` or `DL2LSF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `L2CSF` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CSF` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CSF` skips this computation. `L2CSF` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## Example

A system of three linear equations is solved. The coefficient matrix has real symmetric form and the right-hand-side vector  $b$  has three elements.

```

USE L2CSF_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, N=3)
REAL A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
! B = (4.1 -4.7 6.5)
!
DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
DATA B/4.1, -4.7, 6.5/
!
CALL L2CSF (A, B, X)
! Print results
CALL WRRRN ('X', X, 1, N, 1)
END

```

## Output

|  | X      |        |       |
|--|--------|--------|-------|
|  | 1      | 2      | 3     |
|  | -4.100 | -3.500 | 1.200 |

---

## LFCSF



Computes the  $UDU^T$  factorization of a real symmetric matrix and estimate its  $L_1$  condition number.

## Required Arguments

**A** —  $N$  by  $N$  symmetric matrix to be factored. (Input)

Only the upper triangle of **A** is referenced.

**FACT** —  $N$  by  $N$  matrix containing information about the factorization of the symmetric matrix **A**. (Output)

Only the upper triangle of **FACT** is used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization.

(Output)

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**.

(Output)

## Optional Arguments

**N** — Order of the matrix. (Input)

Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic:     CALL LFCSF (A, FACT, IPVT, RCOND [, ...])

Specific:    The specific interface names are S\_LFCSF and D\_LFCSF.

## FORTRAN 77 Interface

Single:      CALL LFCSF (N, A, LDA, FACT, LDFACT, IPVT, RCOND)

Double:      The double precision name is DLFCSF.

## Description

Routine `LFCSF` performs a  $UDU^T$  factorization of a real symmetric indefinite coefficient matrix. It also estimates the condition number of the matrix. The  $UDU^T$  factorization is called the diagonal pivoting factorization.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCSF fails if  $A$  is singular or very close to a singular matrix.

The  $UDU^T$  factors are returned in a form that is compatible with routines [LFISF](#), [LFSSF](#) and [LFDSE](#). To solve systems of equations with multiple right-hand-side vectors, use LFCSF followed by either LFISF or LFSSF called once for each right-hand side. The routine LFDSE can be called to compute the determinant of the coefficient matrix after LFCSF has performed the factorization.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the *Introduction* section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2CSF/DL2CSF. The reference is:

```
CALL L2CSF (N, A, LDA, FACT, LDFACT, IPVT, RCOND, WK)
```

The additional argument is:

**WK** — Work vector of length  $N$ .

2. Informational errors

| Type | Code |                                               |
|------|------|-----------------------------------------------|
| 3    | 1    | The input matrix is algorithmically singular. |
| 4    | 2    | The input matrix is singular.                 |

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFCSF is called to factor the matrix and to check for singularity or ill-conditioning. LFISF is called to determine the columns of the inverse.

```

USE LFCSF_INT
USE UMACH_INT
USE LFISF_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, N=3)
INTEGER IPVT(N), NOUT
REAL A(LDA,LDA), AINV(N,N), FACT(LDA,LDA), RJ(N), RES(N), &
 RCOND
!
! Set values for A

```

```

!
!
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
! DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
! Factor A and return the reciprocal
! condition number estimate
! CALL LFCFSF (A, FACT, IPVT, RCOND)
! Print the estimate of the condition
! number
! CALL UMACH (2, NOUT)
! WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! matrix one at a time in RJ
! RJ = 0.E0
! DO 10 J=1, N
! RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFISF
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
! CALL LFISF (A, FACT, IPVT, RJ, AINV(:,J), RES)
! RJ(J) = 0.0E0
! 10 CONTINUE
! Print the inverse
! CALL WRRRN ('AINV', AINV)
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < 0.05
L1 Condition number < 40.0

```

```

 AINV
 1 2 3
1 -2.500 -2.000 -0.500
2 -2.000 -1.000 0.000
3 -0.500 0.000 0.500

```

---

## LFTSF



Computes the  $UDU^T$  factorization of a real symmetric matrix.

## Required Arguments

**A** —  $N$  by  $N$  symmetric matrix to be factored. (Input)  
Only the upper triangle of **A** is referenced.

**FACT** —  $N$  by  $N$  matrix containing information about the factorization of the symmetric matrix **A**. (Output)  
Only the upper triangle of **FACT** is used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization. (Output)

## Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFTSF (A, FACT, IPVT [, ...])

Specific:   The specific interface names are `S_LFTSF` and `D_LFTSF`.

## FORTRAN 77 Interface

Single:     CALL LFTSF (N, A, LDA, FACT, LDFACT, IPVT)

Double:     The double precision name is `DLFTSF`.

## Description

Routine `LFTSF` performs a  $UDU^T$  factorization of a real symmetric indefinite coefficient matrix. The  $UDU^T$  factorization is called the diagonal pivoting factorization.

`LFTSF` fails if **A** is singular or very close to a singular matrix.

The  $UDU^T$  factors are returned in a form that is compatible with routines `LFISF`, `LFSSF` and `LFDSF`. To solve systems of equations with multiple right-hand-side vectors, use `LFTSF` followed by either `LFISF` or `LFSSF` called once for each right-hand side. The routine `LFDSF` can be called to compute the determinant of the coefficient matrix after `LFTSF` has performed the factorization.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

```

 Informational error
 Type Code
 4 2 The input matrix is singular.

```

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFTSF is called to factor the matrix and to check for singularity. LFSSF is called to determine the columns of the inverse.

```

 USE LFTSF_INT
 USE LFSSF_INT
 USE WRRRN_INT
!
! Declare variables
 PARAMETER (LDA=3, N=3)
 INTEGER IPVT(N)
 REAL A(LDA,LDA), AINV(N,N), FACT(LDA,LDA), RJ(N)
!
! Set values for A
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
 DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
! Factor A
 CALL LFTSF (A, FACT, IPVT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
 RJ = 0.0E0
 DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFSSF
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSSF (FACT, IPVT, RJ, AINV(:,J))
 RJ(J) = 0.0E0
10 CONTINUE
!
! Print the inverse
 CALL WRRRN ('AINV', AINV)
 END

```

## Output

```

 AINV
 1 2 3

```

```
1 -2.500 -2.000 -0.500
2 -2.000 -1.000 0.000
3 -0.500 0.000 0.500
```

---

## LFSSF



Solves a real symmetric system of linear equations given the  $UDU^T$  factorization of the coefficient matrix.

### Required Arguments

**FACT** —  $N$  by  $N$  matrix containing the factorization of the coefficient matrix  $A$  as output from routine `LFCSE/DLFCSE` or `LFTSF/DLFTSF`. (Input)

Only the upper triangle of **FACT** is used.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization of  $A$  as output from routine `LFCSE/DLFCSE` or `LFTSF/DLFTSF`. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, **B** and **X** can share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL LFSSF (FACT, IPVT, B, X [, ...])`

Specific: The specific interface names are `S_LFSSF` and `D_LFSSF`.

### FORTRAN 77 Interface

Single: `CALL LFSSF (N, FACT, LDFACT, IPVT, B, X)`

Double: The double precision name is `DLFSSF`.

## Description

Routine `LFSSF` computes the solution of a system of linear algebraic equations having a real symmetric indefinite coefficient matrix.

To compute the solution, the coefficient matrix must first undergo a  $UDU^T$  factorization. This may be done by calling either `LFCSF` or `LFTSF`.

`LFSSF` and `LFISF` both solve a linear system given its  $UDU^T$  factorization. `LFISF` generally takes more time and produces a more accurate answer than `LFSSF`. Each iteration of the iterative refinement algorithm used by `LFISF` calls `LFSSF`.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the *Introduction* section of this manual.

## Example

A set of linear systems is solved successively. `LFTSF` is called to factor the coefficient matrix. `LFSSF` is called to compute the four solutions for the four right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCSF` to perform the factorization, and `LFISF` to compute the solutions.

```
 USE LFSSF_INT
 USE LFTSF_INT
 USE WRRRN_INT
!
! Declare variables
 PARAMETER (LDA=3, N=3)
 INTEGER IPVT(N)
 REAL A(LDA,LDA), B(N,4), X(N,4), FACT(LDA,LDA)
!
! Set values for A and B
!
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
! B = (-1.0 3.6 -8.0 -9.4)
! (-3.0 -4.2 11.0 17.6)
! (-3.0 -5.2 -6.0 -23.4)
!
!
 DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
 DATA B/-1.0, -3.0, -3.0, 3.6, -4.2, -5.2, -8.0, 11.0, -6.0,&
 -9.4, 17.6, -23.4/
!
! Factor A
 CALL LFTSF (A, FACT, IPVT)
!
! Solve for the four right-hand sides
 DO 10 I=1, 4
 CALL LFSSF (FACT, IPVT, B(:,I), X(:,I))
10 CONTINUE
!
! Print results
 CALL WRRRN ('X', X)
 END
```

## Output

|   | 1     | 2     | 3    | 4     |
|---|-------|-------|------|-------|
| 1 | 10.00 | 2.00  | 1.00 | 0.00  |
| 2 | 5.00  | -3.00 | 5.00 | 1.20  |
| 3 | -1.00 | -4.40 | 1.00 | -7.00 |

---

## LFISF



Uses iterative refinement to improve the solution of a real symmetric system of linear equations.

### Required Arguments

**A** —  $N$  by  $N$  matrix containing the coefficient matrix of the symmetric linear system. (Input)  
Only the upper triangle of **A** is referenced

**FACT** —  $N$  by  $N$  matrix containing the factorization of the coefficient matrix **A** as output from routine **LFCSE/DLFCSE** or **LFTSE/DLFTSE**. (Input)  
Only the upper triangle of **FACT** is used.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization of **A** as output from routine **LFCSE/DLFCSE** or **LFTSE/DLFTSE**. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, **B** and **X** can share the same storage locations.

**RES** — Vector of length  $N$  containing the residual vector at the improved solution. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFISF (A, FACT, IPVT, B, X, RES [, ...])

Specific:   The specific interface names are S\_LFISF and D\_LFISF.

## FORTRAN 77 Interface

Single:     CALL LFISF (N, A, LDA, FACT, LDFACT, IPVT, B, X, RES)

Double:     The double precision name is DLFISF.

## Description

Routine LFISF computes the solution of a system of linear algebraic equations having a real symmetric indefinite coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo a  $UDU^T$  factorization. This may be done by calling either LFCFSF or LFTSFF.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFISF and LFSSE both solve a linear system given its  $UDU^T$  factorization. LFISF generally takes more time and produces a more accurate answer than LFSSE. Each iteration of the iterative refinement algorithm used by LFISF calls LFSSE.

## Comments

Informational error

Type Code

|   |   |                                                                                   |
|---|---|-----------------------------------------------------------------------------------|
| 3 | 2 | The input matrix is too ill-conditioned for iterative refinement to be effective. |
|---|---|-----------------------------------------------------------------------------------|

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.2 to the second element.

```
USE LFISF_INT
USE UMACH_INT
USE LFCFSF_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (LDA=3, N=3)
INTEGER IPVT(N), NOUT
REAL A(LDA,LDA), B(N), X(N), FACT(LDA,LDA), RES(N), RCOND
!
! Set values for A and B
! A = (1.0 -2.0 1.0)
```

```

! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
! B = (4.1 -4.7 6.5)
!
DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
DATA B/4.1, -4.7, 6.5/
!
! Factor A and compute the estimate
! of the reciprocal condition number
CALL LFCSE (A, FACT, IPVT, RCOND)
!
! Print condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Solve, then perturb right-hand side
DO 10 I=1, 3
 CALL LFISF (A, FACT, IPVT, B, X, RES)
!
! Print results
 CALL WRRRN ('X', X, 1, N, 1)
 CALL WRRRN ('RES', RES, 1, N, 1)
 B(2) = B(2) + .20E0
10 CONTINUE
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < 0.035
L1 Condition number < 40.0

 X
 1 2 3
-4.100 -3.500 1.200

 RES
 1 2 3
-2.384E-07 -2.384E-07 0.000E+00

 X
 1 2 3
-4.500 -3.700 1.200

 RES
 1 2 3
-2.384E-07 -2.384E-07 0.000E+00

 X
 1 2 3
-4.900 -3.900 1.200

 RES
 1 2 3
-2.384E-07 -2.384E-07 0.000E+00

```

---

## LFDSF

Computes the determinant of a real symmetric matrix given the  $UDU^T$  factorization of the matrix.

### Required Arguments

**FACT** — N by N matrix containing the factored matrix  $A$  as output from subroutine LFTSF/DLFTSF or LFCSE/DLFCSE. (Input)

**IPVT** — Vector of length N containing the pivoting information for the  $UDU^T$  factorization as output from routine LFTSF/DLFTSF or LFCSE/DLFCSE. (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)  
The value DET1 is normalized so that,  $1.0 \leq |\text{DET1}| < 10.0$  or  $\text{DET1} = 0.0$ .

**DET2** — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form,  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of FACT exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL LFDSF (FACT, IPVT, DET1, DET2 [, ...])

Specific:    The specific interface names are S\_LFDSF and D\_LFDSF.

### FORTRAN 77 Interface

Single:     CALL LFDSF (N, FACT, LDFACT, IPVT, DET1, DET2)

Double:     The double precision name is DLFDSE.

### Description

Routine LFDSF computes the determinant of a real symmetric indefinite coefficient matrix. To compute the determinant, the coefficient matrix must first undergo a  $UDU^T$  factorization. This may be done by calling either LFCSE or LFTSF. Since  $\det U = \pm 1$ , the formula  $\det A = \det U \det D \det U^T = \det D$  is used to compute the determinant. Next  $\det D$  is computed as the product of the determinants of its blocks.

LFDSF is based on the LINPACK routine SSIDI; see Dongarra et al. (1979).

## Example

The determinant is computed for a real symmetric  $3 \times 3$  matrix.

```
 USE LFDSF_INT
 USE LFTSF_INT
 USE UMACH_INT
!
! Declare variables
 PARAMETER (LDA=3, N=3)
 INTEGER IPVT(N), NOUT
 REAL A(LDA,LDA), FACT(LDA,LDA), DET1, DET2
!
! Set values for A
! A = (1.0 -2.0 1.0)
! (-2.0 3.0 -2.0)
! (1.0 -2.0 3.0)
!
 DATA A/1.0, -2.0, 1.0, -2.0, 3.0, -2.0, 1.0, -2.0, 3.0/
! Factor A
 CALL LFTSF (A, FACT, IPVT)
! Compute the determinant
 CALL LFDSF (FACT, IPVT, DET1, DET2)
! Print the results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) DET1, DET2
99999 FORMAT (' The determinant of A is ', F6.3, ' * 10**', F2.0)
 END
```

## Output

The determinant of A is -2.000 \* 10\*\*0.

---

## LSADH



Solves a Hermitian positive definite system of linear equations with iterative refinement.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian positive definite linear system. (Input)  
Only the upper triangle of **A** is referenced.

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution of the linear system. (Output)

## Optional Arguments

$N$  — Number of equations. (Input)

Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

## FORTRAN 90 Interface

Generic:     CALL LSADH (A, B, X [, ...])

Specific:    The specific interface names are S\_LSADH and D\_LSADH.

## FORTRAN 77 Interface

Single:     CALL LSADH (N, A, LDA, B, X)

Double:     The double precision name is DLSADH.

## ScaLAPACK Interface

Generic:     CALL LSADH (A0, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSADH and D\_LSADH.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LSADH solves a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. It first uses the routine [LFCDH](#) to compute an  $R^H R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix. The matrix  $R$  is upper triangular. The solution of the linear system is then found using the iterative refinement routine [LFIDH](#).

LSADH fails if any submatrix of  $R$  is not positive definite, if  $R$  has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  either is very close to a singular matrix or is a matrix that is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. LSADH solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “Using ScaLAPACK, LAPACK, LINPACK, and EISPACK” in the Introduction section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2ADH/DL2ADH. The reference is:

```
CALL L2ADH (N, A, LDA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing the  $R^H R$  factorization of A on output.

**WK** — Complex work vector of length N.

2. Informational errors

| Type | Code | Description                                                                             |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 1    | The input matrix is too ill-conditioned. The solution might not be accurate.            |
| 3    | 4    | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4    | 2    | The input matrix is not positive definite.                                              |
| 4    | 4    | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine L2ADH the leading dimension of FACT is increased by IVAL(3) when N is a multiple of IVAL(4). The values IVAL(3) and IVAL(4) are temporarily replaced by IVAL(1) and IVAL(2), respectively, in LSADH. Additional memory allocation for FACT and option value restoration are done automatically in LSADH. Users directly calling L2ADH can allocate additional space for FACT and set IVAL(3) and IVAL(4) so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use LSADH or L2ADH. Default values for the option are IVAL(\*) = 1, 16, 0, 1.

**17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSADH temporarily replaces IVAL(2) by IVAL(1). The routine L2CDH computes the condition number if IVAL(2) = 2. Otherwise L2CDH skips this computation. LSADH restores the option. Default values for the option are IVAL(\*) = 1, 2.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — Complex `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficient matrix of the Hermitian positive definite linear system. (Input)  
Only the upper triangle of `A` is referenced.
- B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)
- X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

A system of five linear equations is solved. The coefficient matrix has complex positive definite form and the right-hand-side vector  $b$  has five elements.

```
USE LSADH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=5, N=5)
COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i 0.0+0.0i 0.0+0.0i)
! (9.0+0.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
!
! B = (1.0+5.0i 12.0-6.0i 1.0-16.0i -3.0-3.0i 25.0+16.0i)
!
!
! DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
! 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
! (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0)/
! DATA B / (1.0,5.0), (12.0,-6.0), (1.0,-16.0), (-3.0,-3.0), &
! (25.0,16.0)/
!
!
! CALL LSADH (A, B, X)
!
! Print results
!
! CALL WRCRN ('X', X, 1, N, 1)
!
```

END

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)
```

## ScaLAPACK Example

The same system of five linear equations is solved as a distributed computing example. The coefficient matrix has complex positive definite form and the right-hand-side vector  $b$  has five elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LSADH_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 COMPLEX, ALLOCATABLE :: A(:, :), B(:), X(:)
 COMPLEX, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
 PARAMETER (LDA=5, N=5)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
!
! Set values for A and B
 A(1,:) = ((2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0)/)
 A(2,:) = ((0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0)/)
 A(3,:) = ((0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0)/)
 A(4,:) = ((0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0)/)
 A(5,:) = ((0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (9.0, 0.0)/)
!
 B = ((1.0, 5.0), (12.0, -6.0), (1.0, -16.0), (-3.0, -3.0), (25.0, 16.0)/)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
 CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
!
! Map input arrays to the processor grid
```

```

CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0)
!
! Solve the system of equations
CALL LSADH (A0, B0, X0)
!
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
!
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRCRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)

```

---

## LSLDH



Solves a complex Hermitian positive definite system of linear equations without iterative refinement.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian positive definite linear system. (Input)  
Only the upper triangle of  $A$  is referenced.

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

## Optional Arguments

*N* — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

## FORTRAN 90 Interface

Generic:     CALL LSLDH (A, B, X [, ...])

Specific:    The specific interface names are `S_LSLDH` and `D_LSLDH`.

## FORTRAN 77 Interface

Single:      CALL LSLDH (N, A, LDA, B, X)

Double:     The double precision name is `DLSLDH`.

## ScaLAPACK Interface

Generic:     CALL LSLDH (A0, B0, X0 [, ...])

Specific:    The specific interface names are `S_LSLDH` and `D_LSLDH`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LSLDH` solves a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. `LSLDH` first uses the routine `LFCDH` to compute an  $R^H R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix. The matrix *R* is upper triangular. The solution of the linear system is then found using the routine `LFSDH`.

`LSLDH` fails if any submatrix of *R* is not positive definite or if *R* has a zero diagonal element. These errors occur only if *A* is very close to a singular matrix or to a matrix which is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that `LSADH` be used.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2LDH/ DL2LDH. The reference is:

```
CALL L2LDH (N, A, LDA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** —  $N \times N$  work array containing the  $R^H R$  factorization of **A** on output. If **A** is not needed, **A** can share the same storage locations as **FACT**.

**WK** — Complex work vector of length **N**.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
|---|---|------------------------------------------------------------------------------|

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 4 | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
|---|---|-----------------------------------------------------------------------------------------|

|   |   |                                            |
|---|---|--------------------------------------------|
| 4 | 2 | The input matrix is not positive definite. |
|---|---|--------------------------------------------|

|   |   |                                                                                    |
|---|---|------------------------------------------------------------------------------------|
| 4 | 4 | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part. |
|---|---|------------------------------------------------------------------------------------|

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine L2LDH the leading dimension of **FACT** is increased by **IVAL(3)** when **N** is a multiple of **IVAL(4)**. The values **IVAL(3)** and **IVAL(4)** are temporarily replaced by **IVAL(1)** and **IVAL(2)**, respectively, in LSLDH. Additional memory allocation for **FACT** and option value restoration are done automatically in LSLDH. Users directly calling L2LDH can allocate additional space for **FACT** and set **IVAL(3)** and **IVAL(4)** so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use LSLDH or L2LDH. Default values for the option are **IVAL(\*) = 1, 16, 0, 1**.

**17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSLDH temporarily replaces **IVAL(2)** by **IVAL(1)**. The routine L2CDH computes the condition number if **IVAL(2) = 2**. Otherwise L2CDH skips this computation. LSLDH restores the option. Default values for the option are **IVAL(\*) = 1, 2**.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — Complex `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficient matrix of the Hermitian positive definite linear system. (Input)  
Only the upper triangle of `A` is referenced.

**B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)

**X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)  
If `B` is not needed, `B` and `X` can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of five linear equations is solved. The coefficient matrix has complex Hermitian positive definite form and the right-hand-side vector  $b$  has five elements.

```

USE LSLDH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=5, N=5)
COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i 1.0+1.0i 1.0+1.0i)
! (9.0+0.0i 1.0+1.0i 1.0+1.0i 1.0+1.0i)
!
! B = (1.0+5.0i 12.0-6.0i 1.0-16.0i -3.0-3.0i 25.0+16.0i)
!
DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
 (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0)/
DATA B / (1.0,5.0), (12.0,-6.0), (1.0,-16.0), (-3.0,-3.0), &
 (25.0,16.0)/
!
CALL LSLDH (A, B, X)
!
! Print results
CALL WRCRN ('X', X, 1, N, 1)
!
END

```

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)
```

## ScaLAPACK Example

The same system of five linear equations is solved as a distributed computing example. The coefficient matrix has complex positive definite form and the right-hand-side vector  $b$  has five elements. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity.

DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LSLDH_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 COMPLEX, ALLOCATABLE :: A(:, :), B(:), X(:)
 COMPLEX, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
 PARAMETER (LDA=5, N=5)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
!
! Set values for A and B
 A(1,:) = (/ (2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0) /)
 A(2,:) = (/ (0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0) /)
 A(3,:) = (/ (0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0) /)
 A(4,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0) /)
 A(5,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (9.0, 0.0) /)
!
 B = (/ (1.0, 5.0), (12.0, -6.0), (1.0, -16.0), (-3.0, -3.0), (25.0, 16.0) /)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
 CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
 CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDA))
!
! Map input arrays to the processor grid
 CALL SCALAPACK_MAP(A, DESCA, A0)
 CALL SCALAPACK_MAP(B, DESCX, B0)
```

```

! Solve the system of equations
CALL LSLDH (A0, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF (MP_RANK .EQ. 0) CALL WRCRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)

! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)

```

---

## LFCDH



Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix and estimate its  $L_1$  condition number.

### Required Arguments

**A** — Complex  $N$  by  $N$  Hermitian positive definite matrix to be factored. (Input) Only the upper triangle of **A** is referenced.

**FACT** — Complex  $N$  by  $N$  matrix containing the upper triangular matrix  $R$  of the factorization of **A** in the upper triangle. (Output)  
Only the upper triangle of **FACT** will be used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

## Optional Arguments

*N* — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

*LDFACT* --- Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic: `CALL LFCDH (A, FACT, RCOND [, ...])`

Specific: The specific interface names are `S_LFCDH` and `D_LFCDH`.

## FORTRAN 77 Interface

Single: `CALL LFCDH (N, A, LDA, FACT, LDFACT, RCOND)`

Double: The double precision name is `DLFCDH`.

## ScaLAPACK Interface

Generic: `CALL LFCDH (A0, FACT0, RCOND [, ...])`

Specific: The specific interface names are `S_LFCDH` and `D_LFCDH`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFCDH` computes an  $R^H R$  Cholesky factorization and estimates the condition number of a complex Hermitian positive definite coefficient matrix. The matrix *R* is upper triangular.

The  $L_1$  condition number of the matrix *A* is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by `LINPACK` and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. Iterative refinement can sometimes find the solution to such a system.

LFCDH fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^H R$  factors are returned in a form that is compatible with routines [LFIDH](#), [LFSDH](#) and [LFDDH](#). To solve systems of equations with multiple right-hand-side vectors, use LFCDH followed by either LFIDH or LFSDH called once for each right-hand side. The routine LFDDH can be called to compute the determinant of the coefficient matrix after LFCDH has performed the factorization.

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2CDH/DL2CDH. The reference is:

```
CALL L2CDH (N, A, LDA, FACT, LDFACT, RCOND, WK)
```

The additional argument is

**WK** — Complex work vector of length  $N$ .

2. Informational errors

Type Code

- |   |   |                                                                                           |
|---|---|-------------------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is algorithmically singular.                                             |
| 3 | 4 | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part.   |
| 4 | 4 | The input matrix is not Hermitian.                                                        |
| 4 | 2 | The input matrix is not positive definite. It has a diagonal entry with an imaginary part |

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — Complex MXLDA by MXCOL local matrix containing the local portions of the distributed matrix  $A$ .  $A$  contains the Hermitian positive definite matrix to be factored. (Input)

Only the upper triangle of  $A$  is referenced.

**FACT0** — Complex MXLDA by MXCOL local matrix containing the local portions of the distributed matrix  $FACT$ .  $FACT$  contains the upper triangular matrix  $R$  of the factorization of  $A$  in the upper triangle. (Output)

Only the upper triangle of  $FACT$  will be used. If  $A$  is not needed,  $A$  and  $FACT$  can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

The inverse of a  $5 \times 5$  Hermitian positive definite matrix is computed. `LFCDH` is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. `LFIDH` is called to determine the columns of the inverse.

```

USE LFCDH_INT
USE LFIDH_INT
USE UMACH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NOUT
PARAMETER (LDA=5, LDFACT=5, N=5)
REAL RCOND
COMPLEX A(LDA,LDA), AINV(LDA,LDA), FACT(LDFACT,LDFACT), &
 RES(N), RJ(N)
!
! Set values for A
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i)
! (9.0+0.0i)
!
DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
 (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0)/
!
! Factor the matrix A
CALL LFCDH (A, FACT, RCOND)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0, 0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIDH
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFIDH (A, FACT, RJ, AINV(:,J), RES)
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
!
! Print the results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
CALL WRCRN ('AINV', AINV)
!

```

```
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END
```

## Output

```
RCOND < 0.075
L1 Condition number < 25.0
```

```

 AINV
 2
1 (0.7166, 0.0000) (0.2166,-0.2166) (-0.0899,-0.0300) (-0.0207, 0.0622)
2 (0.2166, 0.2166) (0.4332, 0.0000) (-0.0599,-0.1198) (-0.0829, 0.0415)
3 (-0.0899, 0.0300) (-0.0599, 0.1198) (0.1797, 0.0000) (0.0000,-0.1244)
4 (-0.0207,-0.0622) (-0.0829,-0.0415) (0.0000, 0.1244) (0.2592, 0.0000)
5 (0.0092, 0.0046) (0.0138,-0.0046) (-0.0138,-0.0138) (-0.0288, 0.0288)
 5
1 (0.0092,-0.0046)
2 (0.0138, 0.0046)
3 (-0.0138, 0.0138)
4 (-0.0288,-0.0288)
5 (0.1175, 0.0000)
```

## ScaLAPACK Example

The inverse of the same  $5 \times 5$  Hermitian positive definite matrix in the preceding example is computed as a distributed computing example. LFCDH is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. LFIDH (page 187) is called to determine the columns of the inverse. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LFCDH_INT
USE LFIDH_INT
USE WRCRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
 Declare variables
INTEGER J, LDA, N, NOUT, DESCA(9), DESCX(9)
INTEGER INFO, MXCOL, MXLDA
REAL RCOND
COMPLEX, ALLOCATABLE :: A(:, :), AINV(:, :), RJ(:, :), RJ0(:)
COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), RES0(:), X0(:)
PARAMETER (LDA=5, N=5)
!
 Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
!
 Set values for A and B
A(1,:) = (/ (2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0) /)
A(2,:) = (/ (0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0) /)
A(3,:) = (/ (0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0) /)
A(4,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0) /)
```

```

A(5,:) = /(0.0, 0.0),(0.0, 0.0),(0.0, 0.0),(0.0, 0.0),(9.0, 0.0)/)
ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA), RES0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
!
! Factor the matrix A
CALL LFCDH (A0, FACT0, RCOND)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0, 0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
 CALL SCALAPACK_MAP(RJ, DESCX, RJ0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIDH
! reference solves for the J-th column of
! the inverse of A
 CALL LFIDH (A0, FACT0, RJ0, X0, RES0)
!
! Unmap the results from the distributed
! array back to a non-distributed array
 CALL SCALAPACK_UNMAP(X0, DESCX, AINV(:,J))
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
!
! Print the results.
! After the unmap, only Rank=0 has the full
! array.
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
 CALL WRCRN ('AINV', AINV)
ENDIF
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, FACT0, RJ, RJ0, RES0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < 0.075
L1 Condition number < 25.0

```

|   | 1                 | 2                 | 3                 | 4                 |
|---|-------------------|-------------------|-------------------|-------------------|
|   | AINV              |                   |                   |                   |
| 1 | ( 0.7166, 0.0000) | ( 0.2166,-0.2166) | (-0.0899,-0.0300) | (-0.0207, 0.0622) |
| 2 | ( 0.2166, 0.2166) | ( 0.4332, 0.0000) | (-0.0599,-0.1198) | (-0.0829, 0.0415) |
| 3 | (-0.0899, 0.0300) | (-0.0599, 0.1198) | ( 0.1797, 0.0000) | ( 0.0000,-0.1244) |
| 4 | (-0.0207,-0.0622) | (-0.0829,-0.0415) | ( 0.0000, 0.1244) | ( 0.2592, 0.0000) |
| 5 | ( 0.0092, 0.0046) | ( 0.0138,-0.0046) | (-0.0138,-0.0138) | (-0.0288, 0.0288) |
|   | 5                 |                   |                   |                   |
| 1 | ( 0.0092,-0.0046) |                   |                   |                   |
| 2 | ( 0.0138, 0.0046) |                   |                   |                   |
| 3 | (-0.0138, 0.0138) |                   |                   |                   |
| 4 | (-0.0288,-0.0288) |                   |                   |                   |
| 5 | ( 0.1175, 0.0000) |                   |                   |                   |

---

## LFTDH



Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix.

### Required Arguments

**A** — Complex  $N$  by  $N$  Hermitian positive definite matrix to be factored. (Input) Only the upper triangle of **A** is referenced.

**FACT** — Complex  $N$  by  $N$  matrix containing the upper triangular matrix  $R$  of the factorization of **A** in the upper triangle. (Output)

Only the upper triangle of **FACT** will be used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

### FORTRAN 90 Interface

Generic:    `CALL LFTDH (A, FACT [, ...])`

Specific: The specific interface names are `S_LFTDH` and `D_LFTDH`.

## FORTRAN 77 Interface

Single: `CALL LFTDH (N, A, LDA, FACT, LDFACT)`

Double: The double precision name is `DLFTDH`.

## ScaLAPACK Interface

Generic: `CALL LFTDH (A0, FACT0 [, ...])`

Specific: The specific interface names are `S_LFTDH` and `D_LFTDH`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFTDH` computes an  $R^H R$  Cholesky factorization of a complex Hermitian positive definite coefficient matrix. The matrix  $R$  is upper triangular.

`LFTDH` fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^H R$  factors are returned in a form that is compatible with routines `LFIDH`, `LFSDH` and `LFDDH`. To solve systems of equations with multiple right-hand-side vectors, use `LFCDH` followed by either `LFIDH` or `LFSDH` called once for each right-hand side. The IMSL routine `LFDDH` can be called to compute the determinant of the coefficient matrix after `LFCDH` has performed the factorization.

The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

Informational errors

Type Code

3 4 The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part.

4 2 The input matrix is not positive definite.

4 4 The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0**— Complex MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the Hermitian positive definite matrix to be factored. (Input)  
Only the upper triangle of A is referenced.

**FACT0**— Complex MXLDA by MXCOL local matrix containing the local portions of the distributed matrix FACT. FACT contains the upper triangular matrix R of the factorization of A in the upper triangle. (Output)  
Only the upper triangle of FACT will be used. If A is not needed, A and FACT can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

### Example

The inverse of a  $5 \times 5$  matrix is computed. LFTDH is called to factor the matrix and to check for nonpositive definiteness. LFS DH is called to determine the columns of the inverse.

```
USE LFTDH_INT
USE LFS DH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=5, LDFACT=5, N=5)
COMPLEX A(LDA,LDA), AINV(LDA,LDA), FACT(LDFACT,LDFACT), RJ(N)
!
! Set values for A
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i)
! (9.0+0.0i)
!
DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
 (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0)/
!
! Factor the matrix A
CALL LFTDH (A, FACT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFS DH
! reference places the J-th column of
```

```

! the inverse of A in the J-th column
! of AINV
 CALL LFS DH (FACT, RJ, AINV(:,J))
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
! Print the results

 CALL WRCRN ('AINV', AINV, ITRING=1)
!
 END

```

## Output

```

 AINV
 2
1 (0.7166, 0.0000) (0.2166,-0.2166) (-0.0899,-0.0300) (-0.0207, 0.0622)
2 (0.4332, 0.0000) (-0.0599,-0.1198) (-0.0829, 0.0415)
3 (0.1797, 0.0000) (0.0000,-0.1244)
4 (0.2592, 0.0000)
 5
1 (0.0092,-0.0046)
2 (0.0138, 0.0046)
3 (-0.0138, 0.0138)
4 (-0.0288,-0.0288)
5 (0.1175, 0.0000)

```

## ScaLAPACK Example

The inverse of the same  $5 \times 5$  Hermitian positive definite matrix in the preceding example is computed as a distributed computing example. LFTDH is called to factor the matrix and to check for nonpositive definiteness. LFS DH (page 192) is called to determine the columns of the inverse. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFTDH_INT
 USE LFS DH_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 COMPLEX, ALLOCATABLE :: A(:, :), AINV(:, :), RJ(:), RJ0(:)
 COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), X0(:)
 PARAMETER (LDA=5, N=5)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), AINV(LDA,N))
! Set values for A and B

```

```

A(1,:) = (/ (2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0) /)
A(2,:) = (/ (0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0) /)
A(3,:) = (/ (0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0) /)
A(4,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0) /)
A(5,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (9.0, 0.0) /)
ENDIF

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA), FACT0(MXLDA,MXCOL), RJ(N), &
 RJ0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Factor the matrix A
CALL LFTDH (A0, FACT0)
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0, 0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
 CALL SCALAPACK_MAP(RJ, DESCX, RJ0)
! RJ is the J-th column of the identity
! matrix so the following LFIDH
! reference solves for the J-th column of
! the inverse of A
 CALL LFS DH (FACT0, RJ0, X0)
! Unmap the results from the distributed
! array back to a non-distributed array
 CALL SCALAPACK_UNMAP(X0, DESCX, AINV(:,J))
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
! Print the results.
! After the unmap, only Rank=0 has the full
! array.
IF(MP_RANK .EQ. 0) CALL WRCRN ('AINV', AINV)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, AINV)
DEALLOCATE(A0, FACT0, RJ, RJ0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 AINV
 2
 3
 4
1 (0.7166, 0.0000) (0.2166,-0.2166) (-0.0899,-0.0300) (-0.0207, 0.0622)
2 (0.2166, 0.2166) (0.4332, 0.0000) (-0.0599,-0.1198) (-0.0829, 0.0415)
3 (-0.0899, 0.0300) (-0.0599, 0.1198) (0.1797, 0.0000) (0.0000,-0.1244)

```

```

4 (-0.0207,-0.0622) (-0.0829,-0.0415) (0.0000, 0.1244) (0.2592, 0.0000)
5 (0.0092, 0.0046) (0.0138,-0.0046) (-0.0138,-0.0138) (-0.0288, 0.0288)
5
1 (0.0092,-0.0046)
2 (0.0138, 0.0046)
3 (-0.0138, 0.0138)
6 (-0.0288,-0.0288)
7 (0.1175, 0.0000)

```

---

## LFSDH



Solves a complex Hermitian positive definite system of linear equations given the  $R^H R$  factorization of the coefficient matrix.

### Required Arguments

**FACT** — Complex  $N$  by  $N$  matrix containing the factorization of the coefficient matrix  $A$  as output from routine `LFCDH/DFCDH` or `LFTDH/DFTDH`. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, **B** and **X** can share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:    `CALL LFSDH (FACT, B, X [, ...])`

Specific:    The specific interface names are `S_LFSDH` and `D_LFSDH`.

### FORTRAN 77 Interface

Single:    `CALL LFSDH (N, FACT, LDFACT, B, X)`

Double:      The double precision name is DLFSDH.

## ScaLAPACK Interface

Generic:      CALL LFS DH (FACT0, B0, X0 [, ...])

Specific:     The specific interface names are S\_LFS DH and D\_LFS DH.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LFS DH computes the solution for a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either LFCDH or LFTDH.  $R$  is an upper triangular matrix.

The solution to  $Ax = b$  is found by solving the triangular systems  $R^H y = b$  and  $Rx = y$ .

LFS DH and LFIDH both solve a linear system given its  $R^H R$  factorization. LFIDH generally takes more time and produces a more accurate answer than LFS DH. Each iteration of the iterative refinement algorithm used by LFIDH calls LFS DH.

The underlying code is based on either LINPACK , LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “Using ScaLAPACK, LAPACK, LINPACK, and EISPACK” in the Introduction section of this manual.

## Comments

Informational error

Type   Code

4      1            The input matrix is singular.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**FACT0** — MXLDA by MXCOL complex local matrix containing the local portions of the distributed matrix FACT as output from routine LFCDH/DLFCDH or LFTDH/DLFTDH. FACT contains the factorization of the matrix A. (Input)

**B0** — Complex local vector of length MXLDA containing the local portions of the distributed vector B. B contains the right-hand side of the linear system. (Input)

**X0** — Complex local vector of length MXLDA containing the local portions of the distributed vector X. X contains the solution to the linear system. (Output)  
If B is not needed, B and X can share the same storage locations.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` ([Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A set of linear systems is solved successively. `LFTDH` is called to factor the coefficient matrix. `LFS DH` is called to compute the four solutions for the four right-hand sides. In this case, the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCDH` to perform the factorization, and `LFIDH` to compute the solutions.

```

USE LFS DH _INT
USE LFTDH _INT
USE WRCRN _INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=5, LDFACT=5, N=5)
COMPLEX A(LDA,LDA), B(N,3), FACT(LDFACT,LDFACT), X(N,3)
!
! Set values for A and B
!
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i)
! (9.0+0.0i)
!
! B = (3.0+3.0i 4.0+0.0i 29.0-9.0i)
! (5.0-5.0i 15.0-10.0i -36.0-17.0i)
! (5.0+4.0i -12.0-56.0i -15.0-24.0i)
! (9.0+7.0i -12.0+10.0i -23.0-15.0i)
! (-22.0+1.0i 3.0-1.0i -23.0-28.0i)
!
!
! DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
! 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
! (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0) /
! DATA B / (3.0,3.0), (5.0,-5.0), (5.0,4.0), (9.0,7.0), (-22.0,1.0), &
! (4.0,0.0), (15.0,-10.0), (-12.0,-56.0), (-12.0,10.0), &
! (3.0,-1.0), (29.0,-9.0), (-36.0,-17.0), (-15.0,-24.0), &
! (-23.0,-15.0), (-23.0,-28.0) /
!
! Factor the matrix A
! CALL LFTDH (A, FACT)
!
! Compute the solutions
! DO 10 I=1, 3
! CALL LFS DH (FACT, B(:,I), X(:,I))
10 CONTINUE
!
! Print solutions
! CALL WRCRN ('X', X)
!
! END

```

## Output

```

 X
 1 2 3
1 (1.00, 0.00) (3.00, -1.00) (11.00, -1.00)
2 (1.00, -2.00) (2.00, 0.00) (-7.00, 0.00)
3 (2.00, 0.00) (-1.00, -6.00) (-2.00, -3.00)
4 (2.00, 3.00) (2.00, 1.00) (-2.00, -3.00)
5 (-3.00, 0.00) (0.00, 0.00) (-2.00, -3.00)
```

## ScaLAPACK Example

The same set of linear systems as in the preceding example is solved successively as a distributed computing example. `LFTDH` is called to factor the matrix. `LFS DH` is called to compute the four solutions for the four right-hand sides. In this case, the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCDH` to perform the factorization, and `LFIDH` to compute the solutions.

`SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFTDH_INT
 USE LFS DH_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 COMPLEX, ALLOCATABLE :: A(:, :), B(:, :), B0(:), X(:, :)
 COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), X0(:)
 PARAMETER (LDA=5, N=5)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(LDA,3), X(LDA,3))
!
! Set values for A and B
 A(1,:) = (/ (2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0) /)
 A(2,:) = (/ (0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0) /)
 A(3,:) = (/ (0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0) /)
 A(4,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0) /)
 A(5,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (9.0, 0.0) /)
!
 B(1,:) = (/ (3.0, 3.0), (4.0, 0.0), (29.0, -9.0) /)
 B(2,:) = (/ (5.0, -5.0), (15.0, -10.0), (-36.0, -17.0) /)
 B(3,:) = (/ (5.0, 4.0), (-12.0, -56.0), (-15.0, -24.0) /)
 B(4,:) = (/ (9.0, 7.0), (-12.0, 10.0), (-23.0, -15.0) /)
 B(5,:) = (/ (-22.0, 1.0), (3.0, -1.0), (-23.0, -28.0) /)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
```

```

! and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), &
 B0(MXLDA))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Factor the matrix A
CALL LFTDH (A0, FACT0)
! Compute the solutions
DO 10 J=1, 3
 CALL SCALAPACK_MAP(B(:,J), DESCX, B0)
 CALL LFSDH (FACT0, B0, X0)
! Unmap the results from the distributed
! array back to a non-distributed array
 CALL SCALAPACK_UNMAP(X0, DESCX, X(:,J))
10 CONTINUE
! Print the results.
! After the unmap, only Rank=0 has the full
! array.
IF(MP_RANK .EQ. 0) CALL WRCRN ('X', X)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, FACT0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

 X
 1 2 3
1 (1.00, 0.00) (3.00, -1.00) (11.00, -1.00)
2 (1.00, -2.00) (2.00, 0.00) (-7.00, 0.00)
3 (2.00, 0.00) (-1.00, -6.00) (-2.00, -3.00)
4 (2.00, 3.00) (2.00, 1.00) (-2.00, -3.00)
5 (-3.00, 0.00) (0.00, 0.00) (-2.00, -3.00)

```

---

## LFIDH



Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.

## Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the linear system. (Input)  
Only the upper triangle of **A** is referenced.

**FACT** — Complex  $N$  by  $N$  matrix containing the factorization of the coefficient matrix **A** as output from routine `LFCDH/DLFCDH` or `LFTDH/DLFTDH`. (Input)  
Only the upper triangle of **FACT** is used.

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution. (Output)

**RES** — Complex vector of length  $N$  containing the residual vector at the improved solution. (Output)

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic: `CALL LFIDH (A, FACT, B, X, RES [, ...])`

Specific: The specific interface names are `S_LFIDH` and `D_LFIDH`.

## FORTRAN 77 Interface

Single: `CALL LFIDH (N, A, LDA, FACT, LDFACT, B, X, RES)`

Double: The double precision name is `DLFIDH`.

## ScaLAPACK Interface

Generic: `CALL LFIDH (A0, FACT0, B0, X0, RES0 [, ...])`

Specific: The specific interface names are `S_LFIDH` and `D_LFIDH`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine `LFIDH` computes the solution of a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either `LFCDH` or `LFTDH`.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIDH` and `LFSDH` both solve a linear system given its  $R^H R$  factorization. `LFIDH` generally takes more time and produces a more accurate answer than `LFSDH`. Each iteration of the iterative refinement algorithm used by `LFIDH` calls `LFSDH`.

## Comments

Informational error

Type Code

3 3 The input matrix is too ill-conditioned for iterative refinement to be effective.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `A`. `A` contains the coefficient matrix of the linear system. (Input)  
Only the upper triangle of `A` is referenced.
- FACT0** — `MXLDA` by `MXCOL` complex local matrix containing the local portions of the distributed matrix `FACT` as output from routine `LFCDH` or `LFTDH`. `FACT` contains the factorization of the matrix `A`. (Input)  
Only the upper triangle of `FACT` is referenced.
- B0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `B`. `B` contains the right-hand side of the linear system. (Input)
- X0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `X`. `X` contains the solution to the linear system. (Output)
- RES0** — Complex local vector of length `MXLDA` containing the local portions of the distributed vector `RES`. `RES` contains the residual vector at the improved solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA` and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` ([Utilities](#)) after a call to `SCALAPACK_SETUP` (Chapter 11, [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed by adding  $(1 + i)/2$  to the second element after each call to LFIDH.

```
USE LFIDH_INT
USE LFCDH_INT
USE UMACH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N
PARAMETER (LDA=5, LDFACT=5, N=5)
REAL RCOND
COMPLEX A(LDA,LDA), B(N), FACT(LDFACT,LDFACT), RES(N,3), X(N,3)
!
! Set values for A and B
!
! A = (2.0+0.0i -1.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (4.0+0.0i 1.0+2.0i 0.0+0.0i 0.0+0.0i)
! (10.0+0.0i 0.0+4.0i 0.0+0.0i)
! (6.0+0.0i 1.0+1.0i)
! (9.0+0.0i)
!
! B = (3.0+3.0i 5.0-5.0i 5.0+4.0i 9.0+7.0i -22.0+1.0i)
!
! DATA A / (2.0,0.0), 4*(0.0,0.0), (-1.0,1.0), (4.0,0.0), &
! 4*(0.0,0.0), (1.0,2.0), (10.0,0.0), 4*(0.0,0.0), &
! (0.0,4.0), (6.0,0.0), 4*(0.0,0.0), (1.0,1.0), (9.0,0.0)/
! DATA B / (3.0,3.0), (5.0,-5.0), (5.0,4.0), (9.0,7.0), (-22.0,1.0)/
!
! Factor the matrix A
CALL LFCDH (A, FACT, RCOND)
!
! Print the estimated condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Compute the solutions, then perturb B
DO 10 I=1, 3
 CALL LFIDH (A, FACT, B, X(:,I), RES(:,I))
 B(2) = B(2) + (0.5E0,0.5E0)
10 CONTINUE
!
! Print solutions and residuals
CALL WRCRN ('X', X)
CALL WRCRN ('RES', RES)
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END
```

## Output

```
RCOND < 0.07
L1 Condition number < 25.0
```

```

 X
 1 2 3
1 (1.000, 0.000) (1.217, 0.000) (1.433, 0.000)
2 (1.000,-2.000) (1.217,-1.783) (1.433,-1.567)
3 (2.000, 0.000) (1.910, 0.030) (1.820, 0.060)
```

```

4 (2.000, 3.000) (1.979, 2.938) (1.959, 2.876)
5 (-3.000, 0.000) (-2.991, 0.005) (-2.982, 0.009)

```

```

 RES
 1 2 3
1 (1.192E-07, 0.000E+00) (6.592E-08, 1.686E-07) (1.318E-07, 2.010E-14)
2 (1.192E-07, -2.384E-07) (-5.329E-08, -5.329E-08) (1.318E-07, -2.258E-07)
3 (2.384E-07, 8.259E-08) (2.390E-07, -3.309E-08) (2.395E-07, 1.015E-07)
4 (-2.384E-07, 2.814E-14) (-8.240E-08, -8.790E-09) (-1.648E-07, -1.758E-08)
5 (-2.384E-07, -1.401E-08) (-2.813E-07, 6.981E-09) (-3.241E-07, -2.795E-08)

```

## ScaLAPACK Example

As in the preceding example, a set of linear systems is solved successively as a distributed computing example. The right-hand-side vector is perturbed by adding  $(1 + i)/2$  to the second element after each call to LFIDH. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LFCDH_INT
 USE LFIDH_INT
 USE UMACH_INT
 USE WRCRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER J, LDA, N, NOUT, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
 REAL RCOND
 COMPLEX, ALLOCATABLE :: A(:, :), B(:), B0(:), RES(:, :), X(:, :)
 COMPLEX, ALLOCATABLE :: A0(:, :), FACT0(:, :), X0(:), RES0(:)
 PARAMETER (LDA=5, N=5)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
! Set values for A and B
 A(1,:) = (/ (2.0, 0.0), (-1.0, 1.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0) /)
 A(2,:) = (/ (0.0, 0.0), (4.0, 0.0), (1.0, 2.0), (0.0, 0.0), (0.0, 0.0) /)
 A(3,:) = (/ (0.0, 0.0), (0.0, 0.0), (10.0, 0.0), (0.0, 4.0), (0.0, 0.0) /)
 A(4,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (6.0, 0.0), (1.0, 1.0) /)
 A(5,:) = (/ (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (9.0, 0.0) /)
!
 B = (/ (3.0, 3.0), (5.0, -5.0), (5.0, 4.0), (9.0, 7.0), (-22.0, 1.0) /)
 ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(N, N, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
 CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors

```

```

CALL DESCINIT(DESCA, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE(A0(MXLDA,MXCOL), X0(MXLDA),FACT0(MXLDA,MXCOL), &
 B0(MXLDA), RES0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCX, A0)
!
! Factor the matrix A
CALL LFCDH (A0, FACT0, RCOND)
!
! Print the estimated condition number
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
ENDIF
!
! Compute the solutions
DO 10 J=1, 3
 CALL SCALAPACK_MAP(B, DESCX, B0)
 CALL LFIDH (A0, FACT0, B0, X0, RES0)
!
! Unmap the results from the distributed
! array back to a non-distributed array
 CALL SCALAPACK_UNMAP(X0, DESCX, X(:,J))
 CALL SCALAPACK_UNMAP(RES0, DESCX, RES(:,J))
 IF(MP_RANK .EQ. 0) B(2) = B(2) + (0.5E0, 0.5E0)
10 CONTINUE
!
! Print the results.
! After the unmap, only Rank=0 has the full
! array.
IF(MP_RANK .EQ. 0) THEN
 CALL WRCRN ('X', X)
 CALL WRCRN ('RES', RES)
ENDIF
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, RES, X)
DEALLOCATE(A0, B0, FACT0, RES0, X0)

!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

RCOND < 0.07

L1 Condition number < 25.0

```

 X
 1 2 3
1 (1.000, 0.000) (1.217, 0.000) (1.433, 0.000)
2 (1.000,-2.000) (1.217,-1.783) (1.433,-1.567)
3 (2.000, 0.000) (1.910, 0.030) (1.820, 0.060)
4 (2.000, 3.000) (1.979, 2.938) (1.959, 2.876)
5 (-3.000, 0.000) (-2.991, 0.005) (-2.982, 0.009)

```

```

 RES
 1 2 3
1 (1.192E-07, 0.000E+00) (6.592E-08, 1.686E-07) (1.318E-07, 2.010E-14)

```

```

2 (1.192E-07,-2.384E-07) (-5.329E-08,-5.329E-08) (1.318E-07,-2.258E-07)
3 (2.384E-07, 8.259E-08) (2.390E-07,-3.309E-08) (2.395E-07, 1.015E-07)
4 (-2.384E-07, 2.814E-14) (-8.240E-08,-8.790E-09) (-1.648E-07,-1.758E-08)
5 (-2.384E-07,-1.401E-08) (-2.813E-07, 6.981E-09) (-3.241E-07,-2.795E-08)

```

---

## LFDDH

Computes the determinant of a complex Hermitian positive definite matrix given the  $R^H R$  Cholesky factorization of the matrix.

### Required Arguments

**FACT** — Complex  $N$  by  $N$  matrix containing the  $R^H R$  factorization of the coefficient matrix  $A$  as output from routine `LFCDH/DLFCDH` or `LFTDH/DLFTDH`. (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)  
The value `DET1` is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or `DET1` = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**$N$**  — Order of the matrix. (Input)  
Default: `N` = size (FACT,2).

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDFACT` = size (FACT,1).

### FORTRAN 90 Interface

Generic:     CALL LFDDH (FACT, DET1, DET2 [, ...])

Specific:    The specific interface names are `S_LFDDH` and `D_LFDDH`.

### FORTRAN 77 Interface

Single:     CALL LFDDH (N, FACT, LDFACT, DET1, DET2)

Double:     The double precision name is `DLFDDH`.

### Description

Routine `LFDDH` computes the determinant of a complex Hermitian positive definite coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either `LFCDH` or `LFTDH`. The formula  $\det A = \det R^H \det$

$R = (\det R)^2$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det R = \prod_{i=1}^N R_{ii}$$

(The matrix  $R$  is stored in the upper triangle of `FACT`.)

`LFDDH` is based on the `LINPACK` routine `CPODI`; see Dongarra et al. (1979).

## Example

The determinant is computed for a complex Hermitian positive definite  $3 \times 3$  matrix.

```

USE LFDDH_INT
USE LFTDH_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, LDFACT, NOUT
PARAMETER (LDA=3, LDFACT=3)
REAL DET1, DET2
COMPLEX A(LDA,LDA), FACT(LDFACT,LDFACT)
!
! Set values for A
!
! A = (6.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 7.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i 11.0+0.0i)
!
DATA A / (6.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (7.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (11.0,0.0)/
!
! Factor the matrix
CALL LFTDH (A, FACT)
!
! Compute the determinant
CALL LFDDH (FACT, DET1, DET2)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is ',F6.3,' * 10**',F2.0)
END

```

## Output

The determinant of A is 1.400 \* 10\*\*2.

---

# LSAHF



Solves a complex Hermitian system of linear equations with iterative refinement.

## Required Arguments

- A* — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian linear system. (Input)  
Only the upper triangle of *A* is referenced.
- B* — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X* — Complex vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

- N* — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .
- LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

## FORTRAN 90 Interface

- Generic:     CALL LSAHF (A, B, X [, ...])
- Specific:    The specific interface names are `S_LSAHF` and `D_LSAHF`.

## FORTRAN 77 Interface

- Single:     CALL LSAHF (N, A, LDA, B, X)
- Double:     The double precision name is `DLSAHF`.

## Description

Routine `LSAHF` solves systems of linear algebraic equations having a complex Hermitian indefinite coefficient matrix. It first uses the routine `LFCHF` to compute a  $UDU^H$  factorization of the coefficient matrix and to estimate the condition number of the matrix.  $D$  is a block diagonal matrix with blocks of order 1 or 2 and  $U$  is a matrix composed of the product of a permutation matrix and a unit upper triangular matrix. The solution of the linear system is then found using the iterative refinement routine `LFHF`.

`LSAHF` fails if a block in  $D$  is singular or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. `LSAHF` solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2AHF/DL2AHF`. The reference is:

```
CALL L2AHF (N, A, LDA, B, X, FACT, IPVT, CWK)
```

The additional arguments are as follows:

**FACT** — Complex work vector of length  $N^2$  containing information about the  $UDU^H$  factorization of `A` on output.

**IPVT** — Integer work vector of length `N` containing the pivoting information for the factorization of `A` on output.

**CWK** — Complex work vector of length `N`.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is algorithmically singular.                                           |
| 3 | 4 | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4 | 2 | The input matrix singular.                                                              |
| 4 | 4 | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2AHF` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSAHF`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSAHF`. Users directly calling `L2AHF` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSAHF` or `L2AHF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.

**17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSAHF` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CHF` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CHF` skips this computation. `LSAHF` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## Example

A system of three linear equations is solved. The coefficient matrix has complex Hermitian form and the right-hand-side vector  $b$  has three elements.

```
 USE LSAHF_INT
 USE WRCRN_INT
!
! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=3, N=3)
 COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
! B = (7.0+32.0i -39.0-21.0i 51.0+9.0i)
!
 DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
 DATA B/(7.0,32.0), (-39.0,-21.0), (51.0,9.0)/
!
 CALL LSAHF (A, B, X)
!
! Print results
 CALL WRCRN ('X', X, 1, N, 1)
 END
```

## Output

```

 X
 1 2 3
(2.00, 1.00) (-10.00, -1.00) (3.00, 5.00)
```

---

## LSLHF



Solves a complex Hermitian system of linear equations without iterative refinement.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian linear system. (Input)

Only the upper triangle of  $A$  is referenced.

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Complex vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

### FORTRAN 90 Interface

Generic: `CALL LSLHF (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLHF` and `D_LSLHF`.

### FORTRAN 77 Interface

Single: `CALL LSLHF (N, A, LDA, B, X)`

Double: The double precision name is `DL2LHF`.

### Description

Routine `LSLHF` solves systems of linear algebraic equations having a complex Hermitian indefinite coefficient matrix. It first uses the routine `LFCHF` to compute a  $UDU^H$  factorization of the coefficient matrix.  $D$  is a block diagonal matrix with blocks of order 1 or 2 and  $U$  is a matrix composed of the product of a permutation matrix and a unit upper triangular matrix.

The solution of the linear system is then found using the routine `LFSHF`. `LSLHF` fails if a block in  $D$  is singular. This occurs only if  $A$  is singular or very close to a singular matrix. If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that `LSAHF` be used.

### Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LHF/DL2LHF`. The reference is:

```
CALL L2LHF (N, A, LDA, B, X, FACT, IPVT, CWK)
```

The additional arguments are as follows:

**FACT** — Complex work vector of length  $N^2$  containing information about the  $UDU^H$  factorization of  $A$  on output.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the factorization of  $A$  on output.

*CWK* — Complex work vector of length *N*.

2. Informational errors

Type Code

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is algorithmically singular.                                           |
| 3 | 4 | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4 | 2 | The input matrix singular.                                                              |
| 4 | 4 | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LHF` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSLHF`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLHF`. Users directly calling `L2LHF` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSLHF` or `L2LHF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSLHF` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CHF` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CHF` skips this computation. `LSLHF` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

### Example

A system of three linear equations is solved. The coefficient matrix has complex Hermitian form and the right-hand-side vector  $b$  has three elements.

```
USE LSLHF_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
COMPLEX A(LDA,LDA), B(N), X(N)
!
! Set values for A and B
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
```



**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of  $FACT$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## **FORTRAN 90 Interface**

Generic:     CALL `LFCHF` ( $A$ ,  $FACT$ ,  $IPVT$ ,  $RCOND$  [, ...])

Specific:    The specific interface names are `S_LFCHF` and `D_LFCHF`.

## **FORTRAN 77 Interface**

Single:      CALL `LFCHF` ( $N$ ,  $A$ ,  $LDA$ ,  $FACT$ ,  $LDFACT$ ,  $IPVT$ ,  $RCOND$ )

Double:     The double precision name is `DLFCHF`.

## **Description**

Routine `LFCHF` performs a  $UDU^H$  factorization of a complex Hermitian indefinite coefficient matrix. It also estimates the condition number of the matrix. The  $UDU^H$  factorization is called the diagonal pivoting factorization.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

`LFCHF` fails if  $A$  is singular or very close to a singular matrix.

The  $UDU^H$  factors are returned in a form that is compatible with routines `LF1HF`, `LFSHF` and `LFDFH`. To solve systems of equations with multiple right-hand-side vectors, use `LFCHF` followed by either `LF1HF` or `LFSHF` called once for each right-hand side. The routine `LFDFH` can be called to compute the determinant of the coefficient matrix after `LFCHF` has performed the factorization.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## **Comments**

1.    Workspace may be explicitly provided, if desired, by use of `L2CHF/DL2CHF`. The reference is:

```
CALL L2CHF (N, A, LDA, FACT, LDFACT, IPVT, RCOND, CWK)
```

The additional argument is:

**CWK** — Complex work vector of length N.

## 2. Informational errors

| Type | Code | Description                                                                             |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 1    | The input matrix is algorithmically singular.                                           |
| 3    | 4    | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4    | 2    | The input matrix is singular.                                                           |
| 4    | 4    | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

## Example

The inverse of a  $3 \times 3$  complex Hermitian matrix is computed. `LFCHF` is called to factor the matrix and to check for singularity or ill-conditioning. `LFIHF` is called to determine the columns of the inverse.

```

USE LFCHE_INT
USE UMACH_INT
USE LFIHF_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
INTEGER IPVT(N), NOUT
REAL RCOND
COMPLEX A(LDA,LDA), AINV(LDA,N), FACT(LDA,LDA), RJ(N), RES(N)
!
! Set values for A
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
!
! Set output unit number
CALL UMACH (2, NOUT)
!
! Factor A and return the reciprocal
! condition number estimate
CALL LFCHE (A, FACT, IPVT, RCOND)
!
! Print the estimate of the condition
! number
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N

```

```

 RJ(J) = (1.0E0, 0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIHF
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFIHF (A, FACT, IPVT, RJ, AINV(:,J), RES)
 RJ(J) = (0.0E0, 0.0E0)
10 CONTINUE
!
! Print the inverse
 CALL WRCRN ('AINV', AINV)
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
 END

```

## Output

```

RCOND < 0.25
L1 Condition number < 6.0

```

```

 AINV
 1 2 3
1 (0.2000, 0.0000) (0.1200, 0.0400) (0.0800,-0.0400)
2 (0.1200,-0.0400) (0.1467, 0.0000) (-0.1267,-0.0067)
3 (0.0800, 0.0400) (-0.1267, 0.0067) (-0.0267, 0.0000)

```

---

## LFTHF



Computes the  $UDU^H$  factorization of a complex Hermitian matrix.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian linear system.  
(Input)

Only the upper triangle of **A** is referenced.

**FACT** — Complex  $N$  by  $N$  matrix containing the information about the factorization of the Hermitian matrix **A**. (Output)

Only the upper triangle of **FACT** is used. If **A** is not needed, **A** and **FACT** can share the same storage locations.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization.  
(Output)

## Optional Arguments

*N* — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

*LDFACT* — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFTHF (A, FACT, IPVT [, ...])

Specific:   The specific interface names are `S_LFTHF` and `D_LFTHF`.

## FORTRAN 77 Interface

Single:     CALL LFTHF (N, A, LDA, FACT, LDFACT, IPVT)

Double:     The double precision name is `DLFTHF`.

## Description

Routine `LFTHF` performs a  $UDU^H$  factorization of a complex Hermitian indefinite coefficient matrix. The  $UDU^H$  factorization is called the diagonal pivoting factorization.

`LFTHF` fails if *A* is singular or very close to a singular matrix.

The  $UDU^H$  factors are returned in a form that is compatible with routines `LFTHF`, `LFSHF` and `LFDHF`. To solve systems of equations with multiple right-hand-side vectors, use `LFTHF` followed by either `LFTHF` or `LFSHF` called once for each right-hand side. The routine `LFDHF` can be called to compute the determinant of the coefficient matrix after `LFTHF` has performed the factorization.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

Informational errors

Type   Code

3     4     The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part.

4     2     The input matrix is singular.

The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.

### Example

The inverse of a  $3 \times 3$  matrix is computed. LFTHF is called to factor the matrix and check for singularity. LFSHF is called to determine the columns of the inverse.

```

USE LFTHF_INT
USE LFSHF_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
INTEGER IPVT(N)
COMPLEX A(LDA,LDA), AINV(LDA,N), FACT(LDA,LDA), RJ(N)
!
! Set values for A
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
!
! Factor A
CALL LFTHF (A, FACT, IPVT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0, 0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFSHF
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSHF (FACT, IPVT, RJ, AINV(:,J))
 RJ(J) = (0.0E0, 0.0E0)
10 CONTINUE
!
! Print the inverse
CALL WRCRN ('AINV', AINV)
END

```

### Output

```

 AINV
 1 2 3
1 (0.2000, 0.0000) (0.1200, 0.0400) (0.0800,-0.0400)
2 (0.1200,-0.0400) (0.1467, 0.0000) (-0.1267,-0.0067)
3 (0.0800, 0.0400) (-0.1267, 0.0067) (-0.0267, 0.0000)

```

---

# LFSHF



Solves a complex Hermitian system of linear equations given the  $U D U^H$  factorization of the coefficient matrix.

## Required Arguments

**FACT** — Complex  $N$  by  $N$  matrix containing the factorization of the coefficient matrix  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)  
Only the upper triangle of **FACT** is used.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization of  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, **B** and **X** can share the same storage locations.

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LFSHF (FACT, IPVT, B, X [, ...])`

Specific: The specific interface names are `S_LFSHF` and `D_LFSHF`.

## FORTRAN 77 Interface

Single: `CALL LFSHF (N, FACT, LDFACT, IPVT, B, X)`

Double: The double precision name is `DLFSHF`.

## Description

Routine `LFSHF` computes the solution of a system of linear algebraic equations having a complex Hermitian indefinite coefficient matrix.

To compute the solution, the coefficient matrix must first undergo a  $UDU^H$  factorization. This may be done by calling either `LFCHF` or `LFTHF`.

`LFSHF` and `LFIHF` both solve a linear system given its  $UDU^H$  factorization. `LFIHF` generally takes more time and produces a more accurate answer than `LFSHF`. Each iteration of the iterative refinement algorithm used by `LFIHF` calls `LFSHF`.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Example

A set of linear systems is solved successively. `LFTHF` is called to factor the coefficient matrix. `LFSHF` is called to compute the three solutions for the three right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCHF` to perform the factorization, and `LFIHF` to compute the solutions.

```
 USE LFSHF_INT
 USE WRCRN_INT
 USE LFTHF_INT
!
! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=3, N=3)
 INTEGER IPVT(N), I
 COMPLEX A(LDA,LDA), B(N,3), X(N,3), FACT(LDA,LDA)
!
! Set values for A and B
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
! B = (7.0+32.0i -6.0+11.0i -2.0-17.0i)
! (-39.0-21.0i -5.5-22.5i 4.0+10.0i)
! (51.0+ 9.0i 16.0+17.0i -2.0+12.0i)
!
!
 DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
 DATA B/(7.0,32.0), (-39.0,-21.0), (51.0,9.0), (-6.0,11.0), &
 (-5.5,-22.5), (16.0,17.0), (-2.0,-17.0), (4.0,10.0), &
 (-2.0,12.0)/
!
! Factor A
 CALL LFTHF (A, FACT, IPVT)
!
! Solve for the three right-hand sides
 DO 10 I=1, 3
 CALL LFSHF (FACT, IPVT, B(:,I), X(:,I))
10 CONTINUE
!
! Print results
 CALL WRCRN ('X', X)
```

END

## Output

|   | X               |                 |                |
|---|-----------------|-----------------|----------------|
|   | 1               | 2               | 3              |
| 1 | ( 2.00, 1.00)   | ( 1.00, 0.00)   | ( 0.00, -1.00) |
| 2 | (-10.00, -1.00) | ( -3.00, -4.00) | ( 0.00, -2.00) |
| 3 | ( 3.00, 5.00)   | ( -0.50, 3.00)  | ( 0.00, -3.00) |

---

## LFIHF



Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.

### Required Arguments

**A** — Complex  $N$  by  $N$  matrix containing the coefficient matrix of the Hermitian linear system. (Input)

Only the upper triangle of  $A$  is referenced.

**FACT** — Complex  $N$  by  $N$  matrix containing the factorization of the coefficient matrix  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)

Only the upper triangle of **FACT** is used.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization of  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution. (Output)

**RES** — Complex vector of length  $N$  containing the residual vector at the improved solution. (Output)

### Optional Arguments

**N** — Number of equations. (Input)

Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
 Default: `LDFACT = size (FACT,1)`.

### FORTRAN 90 Interface

Generic: `CALL LFIHF (A, FACT, IPVT, B, X, RES [, ...])`

Specific: The specific interface names are `S_LFIHF` and `D_LFIHF`.

### FORTRAN 77 Interface

Single: `CALL LFIHF (N, A, LDA, FACT, LDFACT, IPVT, B, X, RES)`

Double: The double precision name is `DLFIHF`.

### Description

Routine `LFIHF` computes the solution of a system of linear algebraic equations having a complex Hermitian indefinite coefficient matrix.

Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo a  $UDU^H$  factorization. This may be done by calling either `LFCHF` or `LFTHF`.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIHF` and `LFSHF` both solve a linear system given its  $UDU^H$  factorization. `LFIHF` generally takes more time and produces a more accurate answer than `LFSHF`. Each iteration of the iterative refinement algorithm used by `LFIHF` calls `LFSHF`.

### Comments

Informational error

| Type | Code |                                                                                   |
|------|------|-----------------------------------------------------------------------------------|
| 3    | 3    | The input matrix is too ill-conditioned for iterative refinement to be effective. |

### Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding  $0.2 + 0.2i$  to the second element.

```

USE LFIHF_INT
USE UMACH_INT
USE LFCHF_INT
USE WRCRN_INT
!
 Declare variables

```

```

INTEGER LDA, N
PARAMETER (LDA=3, N=3)
INTEGER IPVT(N), NOUT
REAL RCOND
COMPLEX A(LDA,LDA), B(N), X(N), FACT(LDA,LDA), RES(N)

!
!
! Set values for A and B
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
! B = (7.0+32.0i -39.0-21.0i 51.0+9.0i)
!
DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
DATA B/(7.0,32.0), (-39.0,-21.0), (51.0,9.0)/
! Set output unit number
CALL UMACH (2, NOUT)
!
! Factor A and compute the estimate
! of the reciprocal condition number
CALL LFCHE (A, FACT, IPVT, RCOND)
WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
! Solve, then perturb right-hand side
DO 10 I=1, 3
 CALL LFIHF (A, FACT, IPVT, B, X, RES)
! Print results
 WRITE (NOUT,99999) I
 CALL WRCRN ('X', X, 1, N, 1)
 CALL WRCRN ('RES', RES, 1, N, 1)
 B(2) = B(2) + (0.2E0, 0.2E0)
10 CONTINUE
!
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
99999 FORMAT (//, ' For problem ', I1)
END

```

## Output

```

RCOND < 0.25
L1 Condition number < 5.0
For problem 1
 X
 1 2 3
(2.00, 1.00) (-10.00, -1.00) (3.00, 5.00)
 RES
 1 2 3
(2.384E-07,-4.768E-07) (0.000E+00,-3.576E-07) (-1.421E-14, 1.421E-14)
For problem 2
 X
 1 2 3
(2.016, 1.032) (-9.971,-0.971) (2.973, 4.976)
 RES

```

```

 1 2 3
(2.098E-07,-1.764E-07) (6.231E-07,-1.518E-07) (1.272E-07, 4.005E-07)

```

For problem 3

```

 X
 1 2 3
(2.032, 1.064) (-9.941,-0.941) (2.947, 4.952)

```

```

 RES
 1 2 3
(4.196E-07,-3.529E-07) (2.925E-07,-3.632E-07) (2.543E-07, 3.242E-07)

```

## LFDHF

Computes the determinant of a complex Hermitian matrix given the  $UDU^H$  factorization of the matrix.

### Required Arguments

**FACT** — Complex  $N$  by  $N$  matrix containing the factorization of the coefficient matrix  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)  
Only the upper triangle of `FACT` is used.

**IPVT** — Vector of length  $N$  containing the pivoting information for the factorization of  $A$  as output from routine `LFCHF/DLFCHF` or `LFTHF/DLFTHF`. (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)  
The value `DET1` is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or `DET1` = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**N** — Number of equations. (Input)  
Default: `N` = size (`FACT`,2).

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDFACT` = size (`FACT`,1).

### FORTRAN 90 Interface

Generic:    CALL LFDHF (FACT, IPVT, DET1, DET2 [, ...])

Specific:   The specific interface names are `S_LFDHF` and `D_LFDHF`.

## FORTRAN 77 Interface

Single:      CALL LFDHF (N, FACT, LDFACT, IPVT, DET1, DET2)

Double:      The double precision name is DLFDHF.

## Description

Routine LFDHF computes the determinant of a complex Hermitian indefinite coefficient matrix. To compute the determinant, the coefficient matrix must first undergo a  $UDU^H$  factorization. This may be done by calling either LFCFH or LFTHF since  $\det U = \pm 1$ , the formula  $\det A = \det U \det D \det U^H = \det D$  is used to compute the determinant.  $\det D$  is computed as the product of the determinants of its blocks.

LFDHF is based on the LINPACK routine CSIDI; see Dongarra et al. (1979).

## Example

The determinant is computed for a complex Hermitian  $3 \times 3$  matrix.

```
 USE LFDHF_INT
 USE LFTHF_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LDA, N
 PARAMETER (LDA=3, N=3)
 INTEGER IPVT(N), NOUT
 REAL DET1, DET2
 COMPLEX A(LDA,LDA), FACT(LDA,LDA)
!
! Set values for A
!
! A = (3.0+0.0i 1.0-1.0i 4.0+0.0i)
! (1.0+1.0i 2.0+0.0i -5.0+1.0i)
! (4.0+0.0i -5.0-1.0i -2.0+0.0i)
!
 DATA A/(3.0,0.0), (1.0,1.0), (4.0,0.0), (1.0,-1.0), (2.0,0.0), &
 (-5.0,-1.0), (4.0,0.0), (-5.0,1.0), (-2.0,0.0)/
!
! Factor A
 CALL LFTHF (A, FACT, IPVT)
!
! Compute the determinant
 CALL LFDHF (FACT, IPVT, DET1, DET2)
!
! Print the results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant is', F5.1, ' * 10**', F2.0)
 END
```

## Output

The determinant is -1.5 \* 10\*\*2.

---

# LSLTR

Solves a real tridiagonal system of linear equations.

## Required Arguments

- C** — Vector of length  $N$  containing the subdiagonal of the tridiagonal matrix in  $C(2)$  through  $C(N)$ . (Input/Output)  
On output  $C$  is destroyed.
- D** — Vector of length  $N$  containing the diagonal of the tridiagonal matrix. (Input/Output)  
On output  $D$  is destroyed.
- E** — Vector of length  $N$  containing the superdiagonal of the tridiagonal matrix in  $E(1)$  through  $E(N - 1)$ . (Input/Output)  
On output  $E$  is destroyed.
- B** — Vector of length  $N$  containing the right-hand side of the linear system on entry and the solution vector on return. (Input/Output)

## Optional Arguments

- N** — Order of the tridiagonal matrix. (Input)  
Default:  $N = \text{size}(C,1)$ .

## FORTRAN 90 Interface

Generic:     `CALL LSLTR (C, D, E, B [, ...])`

Specific:    The specific interface names are `S_LSLTR` and `D_LSLTR`.

## FORTRAN 77 Interface

Single:     `CALL LSLTR (N, C, D, E, B)`

Double:     The double precision name is `DLSLTR`.

## Description

Routine `LSLTR` factors and solves the real tridiagonal linear system  $Ax = b$ . `LSLTR` is intended just for tridiagonal systems. The coefficient matrix does not have to be symmetric. The algorithm is Gaussian elimination with partial pivoting for numerical stability. See Dongarra (1979), LINPACK subprograms `SGTSL/DGTSL`, for details. When computing on vector or parallel computers the cyclic reduction algorithm, `LSLCR`, should be considered as an alternative method to solve the system.

## Comments

Informational error

Type Code

4 2 An element along the diagonal became exactly zero during execution.

## Example

A system of  $n = 4$  linear equations is solved.

```
USE LSLTR_INT
USE WRRRL_INT
! Declaration of variables
INTEGER N
PARAMETER (N=4)
!
REAL B(N), C(N), D(N), E(N)
CHARACTER CLABEL(1)*6, FMT*8, RLABEL(1)*4
!
DATA FMT/' (E13.6) '/
DATA CLABEL/'NUMBER' /
DATA RLABEL/'NONE' /
! C(*), D(*), E(*), and B(*)
! contain the subdiagonal, diagonal,
! superdiagonal and right hand side.
DATA C/0.0, 0.0, -4.0, 9.0/, D/6.0, 4.0, -4.0, -9.0/
DATA E/-3.0, 7.0, -8.0, 0.0/, B/48.0, -81.0, -12.0, -144.0/
!
!
CALL LSLTR (C, D, E, B)
! Output the solution.
CALL WRRRL ('Solution:', B, RLABEL, CLABEL, 1, N, 1, FMT=FMT)
END
```

## Output

```
Solution:
 1 2 3 4
0.400000E+01 -0.800000E+01 -0.700000E+01 0.900000E+01
```

---

## LSLCR

Computes the  $LDU$  factorization of a real tridiagonal matrix  $A$  using a cyclic reduction algorithm.

### Required Arguments

$C$  — Array of size  $2N$  containing the upper codiagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $C(1), \dots, C(N-1)$ . (Input/Output)

$A$  — Array of size  $2N$  containing the diagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $A(1), \dots, A(N)$ . (Input/Output)

**B** — Array of size  $2N$  containing the lower codiagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $B(1), \dots, B(N - 1)$ . (Input/Output)

**Y** — Array of size  $2N$  containing the right hand side for the system  $Ax = y$  in the order  $Y(1), \dots, Y(N)$ . (Input/Output) The vector  $x$  overwrites  $Y$  in storage.

**U** — Array of size  $2N$  of flags that indicate any singularities of  $A$ . (Output)  
A value  $U(I) = 1$  means that a divide by zero would have occurred during the factoring.  
Otherwise  $U(I) = 0$ .

**IR** — Array of integers that determine the sizes of loops performed in the cyclic reduction algorithm. (Output)

**IS** — Array of integers that determine the sizes of loops performed in the cyclic reduction algorithm. (Output)  
The sizes of **IR** and **IS** must be at least  $\log_2(N) + 3$ .

### Optional Arguments

**N** — Order of the matrix. (Input)  
 $N$  must be greater than zero  
Default:  $N = \text{size}(C,1)$ .

**IJOB** — Flag to direct the desired factoring or solving step. (Input)  
Default:  $IJOB = 1$ .

| <b>IJOB</b> | <b>Action</b>                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------|
| 1           | Factor the matrix $A$ and solve the system $Ax = y$ , where $y$ is stored in array $Y$ .                                  |
| 2           | Do the solve step only. Use $y$ from array $Y$ . (The factoring step has already been done.)                              |
| 3           | Factor the matrix $A$ but do not solve a system.                                                                          |
| 4, 5, 6     | Same meaning as with the value $IJOB = 3$ . For efficiency, no error checking is done on the validity of any input value. |

### FORTRAN 90 Interface

Generic:    CALL LSLCR (C, A, B, Y, U, IR, IS [, ...])

Specific:   The specific interface names are `S_LSLCR` and `D_LSLCR`.

## FORTRAN 77 Interface

Single:       CALL LSLCR (N, C, A, B, IJOB, Y, U, IR, IS)

Double:       The double precision name is DLSLCR.

## Description

Routine LSLCR factors and solves the real tridiagonal linear system  $Ax = y$ . The matrix is decomposed in the form  $A = LDU$ , where  $L$  is unit lower triangular,  $U$  is unit upper triangular, and  $D$  is diagonal. The algorithm used for the factorization is effectively that described in Kershaw (1982). More details, tests and experiments are reported in Hanson (1990).

LSLCR is intended just for tridiagonal systems. The coefficient matrix does not have to be symmetric. The algorithm amounts to Gaussian elimination, with no pivoting for numerical stability, on the matrix whose rows and columns are permuted to a new order. See Hanson (1990) for details. The expectation is that LSLCR will outperform either LSLTR or LSLPB on vector or parallel computers. Its performance may be inferior for small values of  $n$ , on scalar computers, or high-performance computers with non-optimizing compilers.

## Example

A system of  $n = 1000$  linear equations is solved. The coefficient matrix is the symmetric matrix of the second difference operation, and the right-hand-side vector  $y$  is the first column of the identity matrix. Note that  $a_{n,n} = 1$ . The solution vector will be the first column of the inverse matrix of  $A$ . Then a new system is solved where  $y$  is now the last column of the identity matrix. The solution vector for this system will be the last column of the inverse matrix.

```
 USE LSLCR_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LP, N, N2
 PARAMETER (LP=12, N=1000, N2=2*N)
!
 INTEGER I, IJOB, IR(LP), IS(LP), NOUT
 REAL A(N2), B(N2), C(N2), U(N2), Y1(N2), Y2(N2)
!
! Define matrix entries:
 DO 10 I=1, N - 1
 C(I) = -1.E0
 A(I) = 2.E0
 B(I) = -1.E0
 Y1(I+1) = 0.E0
 Y2(I) = 0.E0
10 CONTINUE
 A(N) = 1.E0
 Y1(1) = 1.E0
 Y2(N) = 1.E0
!
! Obtain decomposition of matrix and
! solve the first system:
 IJOB = 1
 CALL LSLCR (C, A, B, Y1, U, IR, IS, IJOB=IJOB)
!
```

```

! Solve the second system with the
! decomposition ready:
IJOB = 2
CALL LSLCR (C, A, B, Y2, U, IR, IS, IJOB=IJOB)
CALL UMACH (2, NOUT)

WRITE (NOUT,*) ' The value of n is: ', N
WRITE (NOUT,*) ' Elements 1, n of inverse matrix columns 1 '//&
 'and n:', Y1(1), Y2(N)
END

```

## Output

```

The value of n is: 1000
Elements 1, n of inverse matrix columns 1 and n: 1.00000 1000.000

```

---

# LSARB



Solves a real system of linear equations in band storage mode with iterative refinement.

## Required Arguments

**A** —  $(NLCA + NUCA + 1)$  by  $N$  array containing the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)

**NLCA** — Number of lower codiagonals of **A**. (Input)

**NUCA** — Number of upper codiagonals of **A**. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)

IPATH = 1 means the system  $AX = B$  is solved.

IPATH = 2 means the system  $A^T X = B$  is solved.

Default: IPATH = 1.

## FORTRAN 90 Interface

Generic:     CALL LSARB (A, NLCA, NUCA, B, X [, ...])

Specific:    The specific interface names are S\_LSARB and D\_LSARB.

## FORTRAN 77 Interface

Single:      CALL LSARB (N, A, LDA, NLCA, NUCA, B, IPATH, X)

Double:      The double precision name is DLSARB.

## Description

Routine LSARB solves a system of linear algebraic equations having a real banded coefficient matrix. It first uses the routine LFCRB to compute an *LU* factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using the iterative refinement routine LFIRB.

LSARB fails if *U*, the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if *A* is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. Iterative refinement can sometimes find the solution to such a system. LSARB solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2ARB/DL2ARB. The reference is:

```
CALL L2ARB (N, A, LDA, NLCA, NUCA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** — Work vector of length  $(2 * NLCA + NUCA + 1) \times N$  containing the *LU* factorization of *A* on output.

**IPVT** — Work vector of length *N* containing the pivoting information for the *LU* factorization of *A* on output.

**WK** — Work vector of length *N*.

2. Informational errors

Type Code

- |   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4 | 2 | The input matrix is singular.                                                |

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2ARB` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSARB`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSARB`. Users directly calling `L2ARB` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSARB` or `L2ARB`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSARB` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CRB` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CRB` skips this computation. `LSARB` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## Example

A system of four linear equations is solved. The coefficient matrix has real banded form with 1 upper and 1 lower codiagonal. The right-hand-side vector  $b$  has four elements.

```
USE LSARB_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, N, NLCA, NUCA
PARAMETER (LDA=3, N=4, NLCA=1, NUCA=1)
REAL A(LDA,N), B(N), X(N)
!
! Set values for A in band form, and B
!
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
! B = (3.0 1.0 11.0 -2.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
DATA B/3.0, 1.0, 11.0, -2.0/
!
CALL LSARB (A, NLCA, NUCA, B, X)
```

```

! Print results
 CALL WRRRN ('X', X, 1, N, 1)
!
 END

```

## Output

```

 X
 1 2 3 4
2.000 1.000 -3.000 4.000

```

## LSLRB



Solves a real system of linear equations in band storage mode without iterative refinement.

### Required Arguments

**A** —  $(NLCA + NUCA + 1)$  by  $N$  array containing the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
 Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means the system  $AX = B$  is solved.  
 $IPATH = 2$  means the system  $A^T X = B$  is solved.  
 Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic:     CALL LSLRB (A, NLCA, NUCA, B, X [, ...])

Specific:    The specific interface names are S\_LSLRB and D\_LSLRB.

## FORTRAN 77 Interface

Single:     CALL LSLRB (N, A, LDA, NLCA, NUCA, B, IPATH, X)

Double:     The double precision name is DLSLRB.

## ScaLAPACK Interface

Generic:     CALL LSLRB (A0, NLCA, NUCA, B0, X0 [, ...])

Specific:    The specific interface names are S\_LSLRB and D\_LSLRB.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Routine LSLRB solves a system of linear algebraic equations having a real banded coefficient matrix. It first uses the routine LFCRB to compute an *LU* factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using LFSRB. LSLRB fails if *U*, the upper triangular part of the factorization, has a zero diagonal element. This occurs only if *A* is singular or very close to a singular matrix. If the estimated condition number is greater than  $1/\varepsilon$  (where  $\varepsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that LSARB be used.

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.     Workspace may be explicitly provided, if desired, by use of L2LRB/DL2LRB. The reference is:

```
CALL L2LRB (N, A, LDA, NLCA, NUCA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $(2 \times \text{NLCA} + \text{NUCA} + 1) \times N$  containing the *LU* factorization of *A* on output. If *A* is not needed, *A* can share the first  $(\text{NLCA} + \text{NUCA} + 1) \times N$  storage locations with *FACT*.

**IPVT** — Work vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  on output.

**WK** — Work vector of length  $N$ .

2. Informational errors

Type Code

- |   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 1 | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4 | 2 | The input matrix is singular.                                                |

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LRB` the leading dimension of `FACT` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSLRB`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLRB`. Users directly calling `L2LRB` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSLRB` or `L2LRB`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSLRB` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CRB` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CRB` skips this computation. `LSLRB` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** —  $(2 * NLCA + 2 * NUCA + 1)$  by `MXCOL` local matrix containing the local portions of the distributed matrix  $A$ .  $A$  contains the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)
- B0** — Local vector of length `MXCOL` containing the local portions of the distributed vector  $B$ .  $B$  contains the right-hand side of the linear system. (Input)
- X0** — Local vector of length `MXCOL` containing the local portions of the distributed vector  $X$ .  $X$  contains the solution to the linear system. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

A system of four linear equations is solved. The coefficient matrix has real banded form with 1 upper and 1 lower codiagonal. The right-hand-side vector  $b$  has four elements.

```
 USE LSLRB_INT
 USE WRRRN_INT
!
! Declare variables
 INTEGER LDA, N, NLCA, NUCA
 PARAMETER (LDA=3, N=4, NLCA=1, NUCA=1)
 REAL A(LDA,N), B(N), X(N)
!
! Set values for A in band form, and B
!
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
! B = (3.0 1.0 11.0 -2.0)
!
!
 DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
 DATA B/3.0, 1.0, 11.0, -2.0/
!
 CALL LSLRB (A, NLCA, NUCA, B, X)
!
! Print results
 CALL WRRRN ('X', X, 1, N, 1)
!
 END
```

## Output

```
 X
 1 2 3 4
2.000 1.000 -3.000 4.000
```

## ScaLAPACK Example

The same system of four linear equations is solved as a distributed computing example. The coefficient matrix has real banded form with 1 upper and 1 lower codiagonal. The right-hand-side vector  $b$  has four elements. `SCALAPACK_MAP` and `SCALAPACK_UNMAP` are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. `DESCINIT` is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```
 USE MPI_SETUP_INT
 USE LSLRB_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER LDA, M, N, NLCA, NUCA, NRA, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA
```

```

REAL, ALLOCATABLE :: A(:, :), B(:), X(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:)
PARAMETER (LDA=3, N=6, NLCA=1, NUCA=1)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,N), B(N), X(N))
! Set values for A and B
 A(1,:) = (/ 0.0, 0.0, -3.0, 0.0, -1.0, -3.0/)
 A(2,:) = (/ 10.0, 10.0, 15.0, 10.0, 1.0, 6.0/)
 A(3,:) = (/ 0.0, 0.0, 0.0, -5.0, 0.0, 0.0/)
!
 B = (/ 10.0, 7.0, 45.0, 33.0, -34.0, 31.0/)
ENDIF
NRA = NLCA + NUCA + 1
M = 2*NLCA + 2*NUCA + 1
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(M, N, .FALSE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(M, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Reset MXLDA to M
MXLDA = M
! Set up the array descriptors
CALL DESCINIT(DESCA,NRA,N,MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, INFO)
CALL DESCINIT(DESCX, 1, N, 1, MP_NB, 0, 0, MP_ICTXT, 1, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXCOL), X0(MXCOL))
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCX, B0, 1, .FALSE.)
! Solve the system of equations
CALL LSLRB (A0, NLCA, NUCA, B0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X, 1, .FALSE.)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0)CALL WRRRN ('X', X, 1, N, 1)
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, X)
DEALLOCATE(A0, B0, X0)
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

|  | 1     | 2     | 3     | 4     | 5      | 6     |
|--|-------|-------|-------|-------|--------|-------|
|  | 1.000 | 1.600 | 3.000 | 2.900 | -4.000 | 5.167 |

---

## LFCRB



Computes the  $LU$  factorization of a real matrix in band storage mode and estimate its  $L_1$  condition number.

### Required Arguments

**A** —  $(NLCA + NUCA + 1)$  by  $N$  array containing the  $N$  by  $N$  matrix in band storage mode to be factored. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**FACT** —  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the  $LU$  factorization of the matrix  $A$ . (Output)

If  $A$  is not needed,  $A$  can share the first  $(NLCA + NUCA + 1) * N$  locations with **FACT**.

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization. (Output)

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of  $A$ . (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT, 1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFCRB (A, NLCA, NUCA, FACT, IPVT, RCOND [, ...])

Specific:    The specific interface names are S\_LFCRB and D\_LFCRB.

## FORTRAN 77 Interface

Single:     CALL LFCRB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, RCOND)

Double:     The double precision name is DLFCRB.

## Description

Routine LFCRB performs an  $LU$  factorization of a real banded coefficient matrix. It also estimates the condition number of the matrix. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm.

The  $L_1$  condition number of the matrix  $A$  is defined to be

$$\kappa(A) = \|A\|_1 \|A^{-1}\|_1$$

Since it is expensive to compute

$$\|A^{-1}\|_1$$

the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\varepsilon$  (where  $\varepsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCRB fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  is singular or very close to a singular matrix. The  $LU$  factors are returned in a form that is compatible with routines LFIRB, LFSRB and LFDRB. To solve systems of equations with multiple right-hand-side vectors, use LFCRB followed by either LFIRB or LFSRB called once for each right-hand side. The routine LFDRB can be called to compute the determinant of the coefficient matrix after LFCRB has performed the factorization.

Let  $F$  be the matrix FACT, let  $m_l = NLCA$  and let  $m_u = NUCA$ . The first  $m_l + m_u + 1$  rows of  $F$  contain the triangular matrix  $U$  in band storage form. The lower  $m_l$  rows of  $F$  contain the multipliers needed to reconstruct  $L^{-1}$ .

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2CRB/DL2CRB. The reference is:

```
CALL L2CRB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, RCOND, WK)
```

The additional argument is:

**WK** — Work vector of length N.

## 2. Informational errors

| Type | Code |                                               |
|------|------|-----------------------------------------------|
| 3    | 1    | The input matrix is algorithmically singular. |
| 4    | 2    | The input matrix is singular.                 |

## Example

The inverse of a  $4 \times 4$  band matrix with one upper and one lower codiagonal is computed. LFCRB is called to factor the matrix and to check for singularity or ill-conditioning. LFIRB is called to determine the columns of the inverse.

```

USE LFCRB_INT
USE UMACH_INT
USE LFIRB_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)

INTEGER IPVT(N)
REAL A(LDA,N), AINV(N,N), FACT(LDFACT,N), RCOND, RJ(N), RES(N)
!
! Set values for A in band form
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
!
CALL LFCRB (A, NLCA, NUCA, FACT, IPVT, RCOND)
!
! Print the reciprocal condition number
! and the l1 condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFIRB
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
CALL LFIRB (A, NLCA, NUCA, FACT, IPVT, RJ, AINV(:,J), RES)

```

```

 RJ(J) = 0.0E0
10 CONTINUE
!
 CALL WRRRN ('AINV', AINV) Print results

!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND < .07
L1 Condition number = 25.0

```

|   | AINV   |        |        |        |
|---|--------|--------|--------|--------|
|   | 1      | 2      | 3      | 4      |
| 1 | -1.000 | -1.000 | 0.400  | -0.800 |
| 2 | -3.000 | -2.000 | 0.800  | -1.600 |
| 3 | 0.000  | 0.000  | -0.200 | 0.400  |
| 4 | 0.000  | 0.000  | 0.400  | 0.200  |

---

## LFTRB



Computes the *LU* factorization of a real matrix in band storage mode.

### Required Arguments

**A** —  $(NLCA + NUCA + 1)$  by  $N$  array containing the  $N$  by  $N$  matrix in band storage mode to be factored. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**FACT** —  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the *LU* factorization of the matrix  $A$ . (Output)

If  $A$  is not needed,  $A$  can share the first  $(NLCA + NUCA + 1) * N$  locations with **FACT**.

**IPVT** — Vector of length  $N$  containing the pivoting information for the *LU* factorization. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
 Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: LDA = size (*A*,1).

**LDFACT** — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default: LDFACT = size (*FACT*,1).

### **FORTRAN 90 Interface**

Generic:     CALL LFTRB (*A*, *NLCA*, *NUCA*, *FACT* [, ...])

Specific:    The specific interface names are *S\_LFTRB* and *D\_LFTRB*.

### **FORTRAN 77 Interface**

Single:      CALL LFTRB (*N*, *A*, *LDA*, *NLCA*, *NUCA*, *FACT*, *LDFACT*, *IPVT*)

Double:     The double precision name is *DLFTRB*.

### **Description**

Routine *LFTRB* performs an *LU* factorization of a real banded coefficient matrix using Gaussian elimination with partial pivoting. A failure occurs if *U*, the upper triangular factor, has a zero diagonal element. This can happen if *A* is close to a singular matrix. The *LU* factors are returned in a form that is compatible with routines *LFIRB*, *LFSRB* and *LFDRB*. To solve systems of equations with multiple right-hand-side vectors, use *LFTRB* followed by either *LFIRE* or *LFSRB* called once for each right-hand side. The routine *LFDRB* can be called to compute the determinant of the coefficient matrix after *LFTRB* has performed the factorization

Let  $m_l = NLCA$ , and let  $m_u = NUCA$ . The first  $m_l + m_u + 1$  rows of *FACT* contain the triangular matrix *U* in band storage form. The next  $m_l$  rows of *FACT* contain the multipliers needed to produce *L*.

The routine *LFTRB* is based on the the blocked *LU* factorization algorithm for banded linear systems given in Du Croz, et al. (1990). Level-3 BLAS invocations were replaced by in-line loops. The blocking factor *nb* has the default value 1 in *LFTRB*. It can be reset to any positive value not exceeding 32.

### **Comments**

1.    Workspace may be explicitly provided, if desired, by use of *L2TRB*/*DL2TRB*. The reference is:

```
CALL L2TRB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, WK)
```

The additional argument is:

**WK** — Work vector of length *N* used for scaling.

## 2 Informational error

Type Code

4 2 The input matrix is singular.

## 3. [Integer Options](#) with Chapter 11 Options Manager

**21** The performance of the  $LU$  factorization may improve on high-performance computers if the blocking factor,  $NB$ , is increased. The current version of the routine allows  $NB$  to be reset to a value no larger than 32. Default value is  $NB = 1$ .

## Example

A linear system with multiple right-hand sides is solved. `LFTRB` is called to factor the coefficient matrix. `LFSRB` is called to compute the two solutions for the two right-hand sides. In this case the coefficient matrix is assumed to be appropriately scaled. Otherwise, it may be better to call routine `LFMRB` to perform the factorization, and `LFIRB` to compute the solutions.

```
USE LFTRB_INT
USE LFSRB_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL A(LDA,N), B(N,2), FACT(LDFACT,N), X(N,2)
! Set values for A in band form, and B
!
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
! B = (12.0 -17.0)
! (-19.0 23.0)
! (6.0 5.0)
! (8.0 5.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0,&
 2.0, 1.0, 0.0/
DATA B/12.0, -19.0, 6.0, 8.0, -17.0, 23.0, 5.0, 5.0/
! Compute factorization
CALL LFTRB (A, NLCA, NUCA, FACT, IPVT)
! Solve for the two right-hand sides
DO 10 J=1, 2
 CALL LFSRB (FACT, NLCA, NUCA, IPVT, B(:,J), X(:,J))
10 CONTINUE
! Print results
CALL WRRRN ('X', X)
!
END
```

## Output

|   | X      |        |
|---|--------|--------|
|   | 1      | 2      |
| 1 | 3.000  | -8.000 |
| 2 | -6.000 | 1.000  |
| 3 | 2.000  | 1.000  |
| 4 | 4.000  | 3.000  |

---

## LFSRB



Solves a real system of linear equations given the *LU* factorization of the coefficient matrix in band storage mode.

### Required Arguments

**FACT** —  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the *LU* factorization of the coefficient matrix *A* as output from routine *LFCRB/DLFCRB* or *LFTRB/DFTRB*. (Input)

**NLCA** — Number of lower codiagonals of *A*. (Input)

**NUCA** — Number of upper codiagonals of *A*. (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the *LU* factorization of *A* as output from routine *LFCRB/DLFCRB* or *LFTRB/DFTRB*. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If *B* is not needed, *B* and *X* can share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

**IPATH** — Path indicator. (Input)  
 $\text{IPATH} = 1$  means the system  $AX = B$  is solved.  
 $\text{IPATH} = 2$  means the system  $A^T X = B$  is solved.  
Default:  $\text{IPATH} = 1$ .

## FORTRAN 90 Interface

Generic:    CALL LFSRB (FACT, NLCA, NUCA, IPVT, B, X [, ...])

Specific:   The specific interface names are S\_LFSRB and D\_LFSRB.

## FORTRAN 77 Interface

Single:     CALL LFSRB (N, FACT, LDFACT, NLCA, NUCA, IPVT, B, IPATH, X)

Double:     The double precision name is DLFSRB.

## Description

Routine LFSRB computes the solution of a system of linear algebraic equations having a real banded coefficient matrix. To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either LFCRB or LFTRB. The solution to  $Ax = b$  is found by solving the banded triangular systems  $Ly = b$  and  $Ux = y$ . The forward elimination step consists of solving the system  $Ly = b$  by applying the same permutations and elimination operations to  $b$  that were applied to the columns of  $A$  in the factorization routine. The backward substitution step consists of solving the banded triangular system  $Ux = y$  for  $x$ .

LFSRB and LFIRB both solve a linear system given its *LU* factorization. LFIRB generally takes more time and produces a more accurate answer than LFSRB. Each iteration of the iterative refinement algorithm used by LFIRB calls LFSRB.

The underlying code is based on either LINPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Example

The inverse is computed for a real banded  $4 \times 4$  matrix with one upper and one lower codiagonal. The input matrix is assumed to be well-conditioned, hence LFTRB is used rather than LFCRB.

```
USE LFSRB_INT
USE LFTRB_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL A(LDA,N), AINV(N,N), FACT(LDFACT,N), RJ(N)
!
! Set values for A in band form
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
!
CALL LFTRB (A, NLCA, NUCA, FACT, IPVT)
! Set up the columns of the identity
```

```

! matrix one at a time in RJ
 RJ = 0.0E0
 DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFSRB
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSRB (FACT, NLCA, NUCA, IPVT, RJ, AINV(:,J))
 RJ(J) = 0.0E0
10 CONTINUE
! Print results
 CALL WRRRN ('AINV', AINV)
!
 END

```

## Output

|   | AINV   |        |        |        |
|---|--------|--------|--------|--------|
|   | 1      | 2      | 3      | 4      |
| 1 | -1.000 | -1.000 | 0.400  | -0.800 |
| 2 | -3.000 | -2.000 | 0.800  | -1.600 |
| 3 | 0.000  | 0.000  | -0.200 | 0.400  |
| 4 | 0.000  | 0.000  | 0.400  | 0.200  |

---

## LFIRB



Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.

### Required Arguments

**A** —  $(NUCA + NLCA + 1)$  by  $N$  array containing the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**FACT** —  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the  $LU$  factorization of the matrix  $A$  as output from routines LFCRB/DLFCRB or LFTRB/DLFTRB. (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from routine LFCRB/DLFCRB or LFTRB/DLFTRB. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

**RES** — Vector of length  $N$  containing the residual vector at the improved solution. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means the system  $AX = B$  is solved.  
 $IPATH = 2$  means the system  $A^T X = B$  is solved.  
Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic: `CALL LFIRB (A, NLCA, NUCA, FACT, IPVT, B, X, RES [, ...])`

Specific: The specific interface names are `S_LFIRB` and `D_LFIRB`.

### FORTRAN 77 Interface

Single: `CALL LFIRB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, B, IPATH, X, RES)`

Double: The double precision name is `DLFIRB`.

### Description

Routine `LFIRB` computes the solution of a system of linear algebraic equations having a real banded coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either `LFCRB` or `LFTRB`.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIRB` and `LFSRB` both solve a linear system given its *LU* factorization. `LFIRB` generally takes more time and produces a more accurate answer than `LFSRB`. Each iteration of the iterative refinement algorithm used by `LFIRB` calls `LFSRB`.

## Comments

Informational error

Type Code

|   |   |                                                                                  |
|---|---|----------------------------------------------------------------------------------|
| 3 | 2 | The input matrix is too ill-conditioned for iterative refinement to be effective |
|---|---|----------------------------------------------------------------------------------|

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.5 to the second element.

```

USE LFIRB_INT
USE LFCRB_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL A(LDA,N), B(N), FACT(LDFACT,N), RCOND, RES(N), X(N)
! Set values for A in band form, and B
!
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
! B = (3.0 5.0 7.0 -9.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
DATA B/3.0, 5.0, 7.0, -9.0/
!
CALL LFCRB (A, NLCA, NUCA, FACT, IPVT, RCOND)
! Print the reciprocal condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
! Solve the three systems
DO 10 J=1, 3
 CALL LFIRB (A, NLCA, NUCA, FACT, IPVT, B, X, RES)
! Print results
 CALL WRRRN ('X', X, 1, N, 1)
! Perturb B by adding 0.5 to B(2)
 B(2) = B(2) + 0.5E0
10 CONTINUE
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```
RCOND < .07
L1 Condition number = 25.0
 X
 1 2 3 4
2.000 1.000 -5.000 1.000

 X
 1 2 3 4
1.500 0.000 -5.000 1.000

 X
 1 2 3 4
1.000 -1.000 -5.000 1.000
```

---

## LFDRB

Computes the determinant of a real matrix in band storage mode given the *LU* factorization of the matrix.

### Required Arguments

**FACT** —  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the *LU* factorization of the matrix  $A$  as output from routine LFTRB/DLFTRB or LFCRB/DLFCRB. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the *LU* factorization as output from routine LFTRB/DLFTRB or LFCRB/DLFCRB. (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)

The value **DET1** is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or **DET1** = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)

The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**N** — Order of the matrix. (Input)

Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic:     CALL LFDRB (FACT, NLCA, NUCA, IPVT, DET1, DET2 [, ...])

Specific:    The specific interface names are S\_LFDRB and D\_LFDRB.

## FORTRAN 77 Interface

Single:     CALL LFDRB (N, FACT, LDFACT, NLCA, NUCA, IPVT, DET1, DET2)

Double:     The double precision name is DLFDRB.

## Description

Routine LFDRB computes the determinant of a real banded coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $LU$  factorization. This may be done by calling either LFCRB or LFTRB. The formula  $\det A = \det L \det U$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det U = \prod_{i=1}^N U_{ii}$$

(The matrix  $U$  is stored in the upper  $NUCA + NLCA + 1$  rows of  $FACT$  as a banded matrix.) Since  $L$  is the product of triangular matrices with unit diagonals and of permutation matrices,  $\det L = (-1)^k$ , where  $k$  is the number of pivoting interchanges.

LFDRB is based on the LINPACK routine CGBDI; see Dongarra et al. (1979).

## Example

The determinant is computed for a real banded  $4 \times 4$  matrix with one upper and one lower codiagonal.

```
USE LFDRB_INT
USE LFTRB_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL A(LDA,N), DET1, DET2, FACT(LDFACT,N)
!
! Set values for A in band form
! A = (0.0 -1.0 -2.0 2.0)
! (2.0 1.0 -1.0 1.0)
! (-3.0 0.0 2.0 0.0)
!
DATA A/0.0, 2.0, -3.0, -1.0, 1.0, 0.0, -2.0, -1.0, 2.0, &
 2.0, 1.0, 0.0/
!
CALL LFTRB (A, NLCA, NUCA, FACT, IPVT)
! Compute the determinant
CALL LFDRB (FACT, NLCA, NUCA, IPVT, DET1, DET2)
```

```

! Print the results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) DET1, DET2
99999 FORMAT (' The determinant of A is ', F6.3, ' * 10**', F2.0)
 END

```

## Output

The determinant of A is 5.000 \* 10\*\*0.

# LSAQS

Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.

## Required Arguments

**A** —  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band coefficient matrix in band symmetric storage mode. (Input)

**NCODA** — Number of upper codiagonals of **A**. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

## FORTRAN 90 Interface

Generic:    CALL LSAQS (A, NCODA, B, X [, ...])

Specific:   The specific interface names are `S_LSAQS` and `D_LSAQS`.

## FORTRAN 77 Interface

Single:     CALL LSAQS (N, A, LDA, NCODA, B, X)

Double:     The double precision name is `DLSAQS`.

## Description

Routine `LSAQS` solves a system of linear algebraic equations having a real symmetric positive definite band coefficient matrix. It first uses the routine `LFCQS` to compute an  $R^T R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix.  $R$  is an upper triangular band matrix. The solution of the linear system is then found using the iterative refinement routine `LFIQS`.

`LSAQS` fails if any submatrix of  $R$  is not positive definite, if  $R$  has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. `LSAQS` solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2AQS/DL2AQS`. The reference is:

```
CALL L2AQS (N, A, LDA, NCODA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** — Work vector of length `NCODA + 1` by `N` containing the  $R^T R$  factorization of  $A$  in band symmetric storage form on output.

**WK** — Work vector of length `N`.

2. Informational errors

Type Code

3 1 The input matrix is too ill-conditioned. The solution might not be accurate.

4 2 The input matrix is not positive definite.

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2AQS` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSAQS`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSAQS`.

Users directly calling L2AQS can allocate additional space for FACT and set IVAL(3) and IVAL(4) so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use LSAQS or L2AQS. Default values for the option are IVAL(\*) = 1, 16, 0, 1.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSAQS temporarily replaces IVAL(2) by IVAL(1). The routine L2CQS computes the condition number if IVAL(2) = 2. Otherwise L2CQS skips this computation. LSAQS restores the option. Default values for the option are IVAL(\*) = 1,2.

### Example

A system of four linear equations is solved. The coefficient matrix has real positive definite band form, and the right-hand-side vector  $b$  has four elements.

```

USE LSAQS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, N, NCODA
PARAMETER (LDA=3, N=4, NCODA=2)
REAL A(LDA,N), B(N), X(N)
!
! Set values for A in band symmetric form, and B
!
! A = (0.0 0.0 -1.0 1.0)
! (0.0 0.0 2.0 -1.0)
! (2.0 4.0 7.0 3.0)
!
! B = (6.0 -11.0 -11.0 19.0)
!
DATA A/2*0.0, 2.0, 2*0.0, 4.0, -1.0, 2.0, 7.0, 1.0, -1.0, 3.0/
DATA B/6.0, -11.0, -11.0, 19.0/
!
! Solve A*X = B
CALL LSAQS (A, NCODA, B, X)
!
! Print results
CALL WRRRN ('X', X, 1, N, 1)
!
END

```

### Output

|       |        |       |       | X |
|-------|--------|-------|-------|---|
| 1     | 2      | 3     | 4     |   |
| 4.000 | -6.000 | 2.000 | 9.000 |   |

---

## LSLQS

Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement.

## Required Arguments

**A** —  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band symmetric coefficient matrix in band symmetric storage mode. (Input)

**NCODA** — Number of upper codiagonals of **A**. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)

## Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

## FORTRAN 90 Interface

Generic:     CALL LSLQS (A, NCODA, B, X [, ...])

Specific:    The specific interface names are S\_LSLQS and D\_LSLQS.

## FORTRAN 77 Interface

Single:     CALL LSLQS (N, A, LDA, NCODA, B, X)

Double:     The double precision name is DLSLQS.

## Description

Routine LSLQS solves a system of linear algebraic equations having a real symmetric positive definite band coefficient matrix. It first uses the routine LFCQS to compute an  $R^T R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix.  $R$  is an upper triangular band matrix. The solution of the linear system is then found using the routine LFSQS.

LSLQS fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that LSAQS be used.



```

PARAMETER (LDA=3, N=4, NCODA=2)
REAL A(LDA,N), B(N), X(N)
!
! Set values for A in band symmetric form, and B
!
! A = (0.0 0.0 -1.0 1.0)
! (0.0 0.0 2.0 -1.0)
! (2.0 4.0 7.0 3.0)
!
! B = (6.0 -11.0 -11.0 19.0)
!
DATA A/2*0.0, 2.0, 2*0.0, 4.0, -1.0, 2.0, 7.0, 1.0, -1.0, 3.0/
DATA B/6.0, -11.0, -11.0, 19.0/
! Solve A*X = B
CALL LSLQS (A, NCODA, B, X)

! Print results
CALL WRRRN ('X', X, 1, N, 1)
END

```

## Output

|       |        | X     |       |  |  |
|-------|--------|-------|-------|--|--|
| 1     | 2      | 3     | 4     |  |  |
| 4.000 | -6.000 | 2.000 | 9.000 |  |  |

---

## LSLPB

Computes the  $R^TDR$  Cholesky factorization of a real symmetric positive definite matrix  $A$  in codiagonal band symmetric storage mode. Solve a system  $Ax = b$ .

### Required Arguments

**A** — Array containing the  $N$  by  $N$  positive definite band coefficient matrix and right hand side in codiagonal band symmetric storage mode. (Input/Output)  
The number of array columns must be at least  $NCODA + 2$ . The number of column is not an input to this subprogram.

On output, **A** contains the solution and factors. See [Comments](#) section for details.

**NCODA** — Number of upper codiagonals of matrix **A**. (Input)

Must satisfy  $NCODA \geq 0$  and  $NCODA < N$ .

**U** — Array of flags that indicate any singularities of **A**, namely loss of positive-definiteness of a leading minor. (Output)

A value  $U(I) = 0$ . means that the leading minor of dimension  $I$  is not positive-definite. Otherwise,  $U(I) = 1$ .

## Optional Arguments

*N* — Order of the matrix. (Input)  
Must satisfy  $N > 0$ .  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Must satisfy  $LDA \geq N + \text{NCODA}$ .  
Default:  $LDA = \text{size}(A,1)$ .

*IJOB* — Flag to direct the desired factorization or solving step. (Input)  
Default:  $IJOB = 1$ .

### *IJOB* Meaning

- 1 factor the matrix *A* and solve the system  $Ax = b$ , where *b* is stored in column  $\text{NCODA} + 2$  of array *A*. The vector *x* overwrites *b* in storage.
- 2 solve step only. Use *b* as column  $\text{NCODA} + 2$  of *A*. (The factorization step has already been done.) The vector *x* overwrites *b* in storage.
- 3 factor the matrix *A* but do not solve a system.
- 4,5,6 same meaning as with the value  $IJOB - 3$ . For efficiency, no error checking is done on values *LDA*, *N*, *NCODA*, and *U*(\*).

## FORTRAN 90 Interface

Generic: `CALL LSLPB (A, NCODA, U [, ...])`

Specific: The specific interface names are `S_LSLPB` and `D_LSLPB`.

## FORTRAN 77 Interface

Single: `CALL LSLPB (N, A, LDA, NCODA, IJOB, U)`

Double: The double precision name is `DSLSPB`.

## Description

Routine `LSLPB` factors and solves the symmetric positive definite banded linear system  $Ax = b$ . The matrix is factored so that  $A = R^TDR$ , where *R* is unit upper triangular and *D* is diagonal. The reciprocals of the diagonal entries of *D* are computed and saved to make the solving step more efficient. Errors will occur if *D* has a non-positive diagonal element. Such events occur only if *A* is very close to a singular matrix or is not positive definite.

`LSLPB` is efficient for problems with a small band width. The particular cases  $\text{NCODA} = 0, 1, 2$  are done with special loops within the code. These cases will give good performance. See Hanson

(1989) for details. When solving tridiagonal systems,  $NCODA = 1$ , the cyclic reduction code `LSLCR` should be considered as an alternative. The expectation is that `LSLCR` will outperform `LSLPB` on vector or parallel computers. It may be inferior on scalar computers or even parallel computers with non-optimizing compilers.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LPB/DL2LPB`. The reference is:

```
CALL L2LPB (N, A, LDA, NCODA, IJOB, U, WK)
```

The additional argument is:

**WK** — Work vector of length  $NCODA$ .

2. If  $IJOB=1, 3, 4,$  or  $6$ ,  $A$  contains the factors  $R$  and  $D$  on output. These are stored in codiagonal band symmetric storage mode. Column 1 of  $A$  contains the reciprocal of diagonal matrix  $D$ . Columns 2 through  $NCODA+1$  contain the upper diagonal values for upper unit diagonal matrix  $R$ . If  $IJOB=1, 2, 4,$  or  $5$ , the last column of  $A$  contains the solution on output, replacing  $b$ .

3. Informational error

Type Code

4 2 The input matrix is not positive definite.

## Example

A system of four linear equations is solved. The coefficient matrix has real positive definite codiagonal band form and the right-hand-side vector  $b$  has four elements.

```
USE LSLPB_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, N, NCODA
PARAMETER (N=4, NCODA=2, LDA=N+NCODA)
!
INTEGER IJOB
REAL A(LDA,NCODA+2), U(N)
REAL R(N,N), RT(N,N), D(N,N), WK(N,N), AA(N,N)
!
!
! Set values for A and right side in
! codiagonal band symmetric form:
!
! A = (* * * *)
! (* * * *)
! (2.0 * * 6.0)
! (4.0 0.0 * -11.0)
```

```

! (7.0 2.0 -1.0 -11.0)
! (3.0 -1.0 1.0 19.0)
!
DATA ((A(I+NCODA,J),I=1,N),J=1,NCODA+2)/2.0, 4.0, 7.0, 3.0, 0.0,&
0.0, 2.0, -1.0, 0.0, 0.0, -1.0, 1.0, 6.0, -11.0, -11.0,&
19.0/
DATA R/16*0.0/, D/16*0.0/, RT/16*0.0/
! Factor and solve A*x = b.
CALL LSLPB(A, NCODA, U)
! Print results
CALL WRRRN ('X', A((NCODA+1):,(NCODA+2):), NRA=1, NCA=N, LDA=1)

END

```

## Output

```

 X
 1 2 3 4
4.000 -6.000 2.000 9.000

```

---

## LFCQS

Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode and estimate its  $L_1$  condition number.

### Required Arguments

**A** —  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band coefficient matrix in band symmetric storage mode to be factored. (Input)

**NCODA** — Number of upper codiagonals of **A**. (Input)

**FACT** —  $NCODA + 1$  by  $N$  array containing the  $R^T R$  factorization of the matrix **A** in band symmetric form. (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage locations.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:     CALL LFCQS (A, NCODA, FACT, RCOND [, ...])

Specific:    The specific interface names are S\_LFCQS and D\_LFCQS.

## FORTRAN 77 Interface

Single:     CALL LFCQS (N, A, LDA, NCODA, FACT, LDFACT, RCOND)

Double:     The double precision name is DLFCQS.

## Description

Routine LFCQS computes an  $R^T R$  Cholesky factorization and estimates the condition number of a real symmetric positive definite band coefficient matrix.  $R$  is an upper triangular band matrix.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCQS fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^T R$  factors are returned in a form that is compatible with routines LFIQS, LFSQS and LFDQS. To solve systems of equations with multiple right-hand-side vectors, use LFCQS followed by either LFIQS or LFSQS called once for each right-hand side. The routine LFDQS can be called to compute the determinant of the coefficient matrix after LFCQS has performed the factorization.

LFCQS is based on the LINPACK routine SPBCO; see Dongarra et al. (1979).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2CQS/DL2CQS. The reference is:

```
CALL L2CQS (N, A, LDA, NCODA, FACT, LDFACT, RCOND, WK)
```

The additional argument is:

**WK** — Work vector of length  $N$ .

2.    Informational errors

Type   Code

- 3 3      The input matrix is algorithmically singular.
- 4 2      The input matrix is not positive definite.

### Example

The inverse of a  $4 \times 4$  symmetric positive definite band matrix with one codiagonal is computed. LFCQS is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. LFIQS is called to determine the columns of the inverse.

```

USE LFCQS_INT
USE LFIQS_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA, NOUT
PARAMETER (LDA=2, LDFACT=2, N=4, NCODA=1)
REAL A(LDA,N), AINV(N,N), RCOND, FACT(LDFACT,N), &
RES(N), RJ(N)
!
! Set values for A in band symmetric form
!
! A = (0.0 1.0 1.0 1.0)
! (2.0 2.5 2.5 2.0)
!
DATA A/0.0, 2.0, 1.0, 2.5, 1.0, 2.5, 1.0, 2.0/
! Factor the matrix A
CALL LFCQS (A, NCODA, FACT, RCOND)
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFIQS
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFIQS (A, NCODA, FACT, RJ, AINV(:,J), RES)
 RJ(J) = 0.0E0
10 CONTINUE
! Print the results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
CALL WRRRN ('AINV', AINV)
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

### Output

```

RCOND = 0.160
L1 Condition number = 6.239
 AINV
 1 2 3 4
1 0.6667 -0.3333 0.1667 -0.0833
2 -0.3333 0.6667 -0.3333 0.1667

```

```
3 0.1667 -0.3333 0.6667 -0.3333
4 -0.0833 0.1667 -0.3333 0.6667
```

---

## LFTQS

Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.

### Required Arguments

**A** —  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band coefficient matrix in band symmetric storage mode to be factored. (Input)

**NCODA** — Number of upper codiagonals of **A**. (Input)

**FACT** —  $NCODA + 1$  by  $N$  array containing the  $R^T R$  factorization of the matrix **A**. (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage locations.

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL LFTQS (A, NCODA, FACT [, ...])

Specific:    The specific interface names are S\_LFTQS and D\_LFTQS.

### FORTRAN 77 Interface

Single:     CALL LFTQS (N, A, LDA, NCODA, FACT, LDFACT)

Double:     The double precision name is DLFTQS.

### Description

Routine LFTQS computes an  $R^T R$  Cholesky factorization of a real symmetric positive definite band coefficient matrix.  $R$  is an upper triangular band matrix.

LFTQS fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  is very close to a singular matrix or to a matrix which is not positive definite.

The  $R^T R$  factors are returned in a form that is compatible with routines LFIQS, LFSQS and LFDQS. To solve systems of equations with multiple right hand-side vectors, use LFTQS followed by either LFIQS or LFSQS called once for each right-hand side. The routine LFDQS can be called to compute the determinant of the coefficient matrix after LFTQS has performed the factorization.

LFTQS is based on the LINPACK routine CPBFA; see Dongarra et al. (1979).

## Comments

Informational error

Type Code

4 2 The input matrix is not positive definite.

## Example

The inverse of a  $3 \times 3$  matrix is computed. LFTQS is called to factor the matrix and to check for nonpositive definiteness. LFSQS is called to determine the columns of the inverse.

```

USE LFTQS_INT
USE WRRRN_INT
USE LFSQS_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA
PARAMETER (LDA=2, LDFACT=2, N=4, NCODA=1)
REAL A(LDA,N), AINV(N,N), FACT(LDFACT,N), RJ(N)
!
! Set values for A in band symmetric form
!
! A = (0.0 1.0 1.0 1.0)
! (2.0 2.5 2.5 2.0)
!
DATA A/0.0, 2.0, 1.0, 2.5, 1.0, 2.5, 1.0, 2.0/
! Factor the matrix A
CALL LFTQS (A, NCODA, FACT)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = 0.0E0
DO 10 J=1, N
 RJ(J) = 1.0E0
!
! RJ is the J-th column of the identity
! matrix so the following LFSQS
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSQS (FACT, NCODA, RJ, AINV(:,J))
 RJ(J) = 0.0E0
10 CONTINUE
!
! Print the results
CALL WRRRN ('AINV', AINV, ITRING=1)
END

```

## Output

```

 AINV
 1 2 3 4
1 0.6667 -0.3333 0.1667 -0.0833
2 0.6667 -0.3333 0.1667
3 0.6667 -0.3333
4 0.6667
```

---

## LFSQS

Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.

### Required Arguments

**FACT** —  $NCODA + 1$  by  $N$  array containing the  $R^T R$  factorization of the positive definite band matrix  $A$  in band symmetric storage mode as output from subroutine LFCQS/DLFCQS or LFTQS/DLFTQS. (Input)

**NCODA** — Number of upper codiagonals of  $A$ . (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL LFSQS (FACT, NCODA, B, X [, ...])

Specific:    The specific interface names are S\_LFSQS and D\_LFSQS.

### FORTRAN 77 Interface

Single:     CALL LFSQS (N, FACT, LDFACT, NCODA, B, X)

Double:     The double precision name is DLFSQS.

## Description

Routine `LFSQS` computes the solution for a system of linear algebraic equations having a real symmetric positive definite band coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either `LFCQS` or `LFTQS`.  $R$  is an upper triangular band matrix.

The solution to  $Ax = b$  is found by solving the triangular systems  $R^T y = b$  and  $Rx = y$ .

`LFSQS` and `LFIQS` both solve a linear system given its  $R^T R$  factorization. `LFIQS` generally takes more time and produces a more accurate answer than `LFSQS`. Each iteration of the iterative refinement algorithm used by `LFIQS` calls `LFSQS`.

`LFSQS` is based on the LINPACK routine `SPBSL`; see Dongarra et al. (1979).

## Comments

Informational error

Type Code

4 1 The factored matrix is singular.

## Example

A set of linear systems is solved successively. `LFTQS` is called to factor the coefficient matrix. `LFSQS` is called to compute the four solutions for the four right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCQS` to perform the factorization, and `LFIQS` to compute the solutions.

```
USE LFSQS_INT
USE LFTQS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA
PARAMETER (LDA=3, LDFACT=3, N=4, NCODA=2)
REAL A(LDA,N), B(N,4), FACT(LDFACT,N), X(N,4)
!
!
! Set values for A in band symmetric form, and B
!
! A = (0.0 0.0 -1.0 1.0)
! (0.0 0.0 2.0 -1.0)
! (2.0 4.0 7.0 3.0)
!
! B = (4.0 -3.0 9.0 -1.0)
! (6.0 10.0 29.0 3.0)
! (15.0 12.0 11.0 6.0)
! (-7.0 1.0 14.0 2.0)
!
!
! DATA A/2*0.0, 2.0, 2*0.0, 4.0, -1.0, 2.0, 7.0, 1.0, -1.0, 3.0/
! DATA B/4.0, 6.0, 15.0, -7.0, -3.0, 10.0, 12.0, 1.0, 9.0, 29.0,&
! 11.0, 14.0, -1.0, 3.0, 6.0, 2.0/
! Factor the matrix A
! CALL LFTQS (A, NCODA, FACT)
! Compute the solutions
! DO 10 I=1, 4
```

```

 CALL LFSQS (FACT, NCODA, B(:,I), X(:,I))
10 CONTINUE
! Print solutions
 CALL WRRRN ('X', X)
!
 END

```

## Output

|   | X      |        |       |       |
|---|--------|--------|-------|-------|
|   | 1      | 2      | 3     | 4     |
| 1 | 3.000  | -1.000 | 5.000 | 0.000 |
| 2 | 1.000  | 2.000  | 6.000 | 0.000 |
| 3 | 2.000  | 1.000  | 1.000 | 1.000 |
| 4 | -2.000 | 0.000  | 3.000 | 1.000 |

---

## LFIQS

Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.

### Required Arguments

**A** —  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band coefficient matrix in band symmetric storage mode. (Input)

**NCODA** — Number of upper codiagonals of **A**. (Input)

**FACT** —  $NCODA + 1$  by  $N$  array containing the  $R^T R$  factorization of the matrix **A** from routine LFCQS/DLFCQS or LFTQS/DLFTQS. (Input)

**B** — Vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Vector of length  $N$  containing the solution to the system. (Output)

**RES** — Vector of length  $N$  containing the residual vector at the improved solution. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT, 1)$ .

## FORTRAN 90 Interface

Generic:    CALL LFIQS (A, NCODA, FACT, B, X, RES [, ...])

Specific:   The specific interface names are S\_LFIQS and D\_LFIQS.

## FORTRAN 77 Interface

Single:     CALL LFIQS (N, A, LDA, NCODA, FACT, LDFACT, B, X, RES)

Double:     The double precision name is DLFIQS.

## Description

Routine LFIQS computes the solution of a system of linear algebraic equations having a real symmetric positive-definite band coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either IMSL routine LFCQS or LFTQS.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFIQS and LFSQS both solve a linear system given its  $R^T R$  factorization. LFIQS generally takes more time and produces a more accurate answer than LFSQS. Each iteration of the iterative refinement algorithm used by LFIQS calls LFSQS.

## Comments

Informational error

Type Code

3    4            The input matrix is too ill-conditioned for iterative refinement to be effective.

## Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding 0.5 to the second element.

```
USE LFIQS_INT
USE UMACH_INT
USE LFCQS_INT
USE WRRRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA, NOUT
PARAMETER (LDA=2, LDFACT=2, N=4, NCODA=1)
REAL A(LDA,N), B(N), RCOND, FACT(LDFACT,N), RES(N,3), &
 X(N,3)
!
! Set values for A in band symmetric form, and B
!
! A = (0.0 1.0 1.0 1.0)
```

```

! (2.0 2.5 2.5 2.0)
!
! B = (3.0 5.0 7.0 4.0)
!
DATA A/0.0, 2.0, 1.0, 2.5, 1.0, 2.5, 1.0, 2.0/
DATA B/3.0, 5.0, 7.0, 4.0/
! Factor the matrix A
CALL LFCQS (A, NCODA, FACT, RCOND)
! Print the estimated condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
! Compute the solutions
DO 10 I=1, 3
 CALL LFIQS (A, NCODA, FACT, B, X(:,I), RES(:,I))
 B(2) = B(2) + 0.5E0
10 CONTINUE
! Print solutions and residuals
CALL WRRRN ('X', X)
CALL WRRRN ('RES', RES)
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

## Output

```

RCOND = 0.160
L1 Condition number = 6.239
 X
 1 2 3
1 1.167 1.000 0.833
2 0.667 1.000 1.333
3 2.167 2.000 1.833
4 0.917 1.000 1.083

 RES
 1 2 3
1 7.947E-08 0.000E+00 9.934E-08
2 7.947E-08 0.000E+00 3.974E-08
3 7.947E-08 0.000E+00 1.589E-07
4 -3.974E-08 0.000E+00 -7.947E-08

```

---

## LFDQS

Computes the determinant of a real symmetric positive definite matrix given the  $R^T R$  Cholesky factorization of the band symmetric storage mode.

### Required Arguments

**FACT** —  $NCODA + 1$  by  $N$  array containing the  $R^T R$  factorization of the positive definite band matrix,  $A$ , in band symmetric storage mode as output from subroutine LFCQS/DLFCQS or LFTQS/DLFTQS. (Input)

**NCODA** — Number of upper codiagonals of  $A$ . (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)

The value `DET1` is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or `DET1` = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)

The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

## Optional Arguments

**N** — Number of equations. (Input)

Default: `N = size (FACT,2)`.

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDFACT = size (FACT,1)`.

## FORTRAN 90 Interface

Generic: `CALL LFDQS (FACT, NCODA, DET1, DET2 [, ...])`

Specific: The specific interface names are `S_LFDQS` and `D_LFDQS`.

## FORTRAN 77 Interface

Single: `CALL LFDQS (N, FACT, LDFACT, NCODA, DET1, DET2)`

Double: The double precision name is `DLFDQS`.

## Description

Routine `LFDQS` computes the determinant of a real symmetric positive-definite band coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $R^T R$  factorization. This may be done by calling either IMSL routine `LFCQS` or `LFTQS`. The formula  $\det A = \det R^T \det R = (\det R)^2$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det R = \prod_{i=1}^N R_{ii}$$

`LFDQS` is based on the LINPACK routine `SPBDI`; see Dongarra et al. (1979).

## Example

The determinant is computed for a real positive definite  $4 \times 4$  matrix with 2 codiagonals.

```
USE LFDQS_INT
USE LFTQS_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA, NOUT
PARAMETER (LDA=3, N=4, LDFACT=3, NCODA=2)
```

```

REAL A(LDA,N), DET1, DET2, FACT(LDFACT,N)
!
! Set values for A in band symmetric form
!
! A = (0.0 0.0 1.0 -2.0)
! (0.0 2.0 1.0 3.0)
! (7.0 6.0 6.0 8.0)
!
DATA A/2*0.0, 7.0, 0.0, 2.0, 6.0, 1.0, 1.0, 6.0, -2.0, 3.0, 8.0/
! Factor the matrix
CALL LFTQS (A, NCODA, FACT)
! Compute the determinant
CALL LFDQS (FACT, NCODA, DET1, DET2)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is ',F6.3,' * 10**',F2.0)
END

```

## Output

The determinant of A is 1.186 \* 10\*\*3.

---

## LSLTQ

Solves a complex tridiagonal system of linear equations.

### Required Arguments

- C* — Complex vector of length *N* containing the subdiagonal of the tridiagonal matrix in *C*(2) through *C*(*N*). (Input/Output)  
On output *C* is destroyed.
- D* — Complex vector of length *N* containing the diagonal of the tridiagonal matrix.  
(Input/Output)  
On output *D* is destroyed.
- E* — Complex vector of length *N* containing the superdiagonal of the tridiagonal matrix in *E*(1) through *E*(*N* - 1). (Input/Output)  
On output *E* is destroyed.
- B* — Complex vector of length *N* containing the right-hand side of the linear system on entry and the solution vector on return. (Input/Output)

### Optional Arguments

- N* — Order of the tridiagonal matrix. (Input)  
Default: *N* = size (*C*,1).

## FORTRAN 90 Interface

Generic:    CALL LSLTQ (C, D, E, B [, ...])

Specific:   The specific interface names are S\_LSLTQ and D\_LSLTQ.

## FORTRAN 77 Interface

Single:     CALL LSLTQ (N, C, D, E, B)

Double:     The double precision name is DLSLTQ.

## Description

Routine LSLTQ factors and solves the complex tridiagonal linear system  $Ax = b$ . LSLTQ is intended just for tridiagonal systems. The coefficient matrix does not have to be symmetric. The algorithm is Gaussian elimination with pivoting for numerical stability. See Dongarra et al. (1979), LINPACK subprograms CGTSL/ZGTSL, for details. When computing on vector or parallel computers the cyclic reduction algorithm, LSLCQ, should be considered as an alternative method to solve the system.

## Comments

Informational error

Type Code

4    2            An element along the diagonal became exactly zero during execution.

## Example

A system of  $n = 4$  linear equations is solved.

```
USE LSLTQ_INT
USE WRCL_INT
!
! Declaration of variables
INTEGER N
PARAMETER (N=4)
!
COMPLEX B(N), C(N), D(N), E(N)
CHARACTER CLABEL(1)*6, FMT*8, RLABEL(1)*4
!
DATA FMT/'(E13.6)'/
DATA CLABEL/'NUMBER'/
DATA RLABEL/'NONE'/
!
! C(*), D(*), E(*) and B(*)
! contain the subdiagonal,
! diagonal, superdiagonal and
! right hand side.
DATA C/(0.0,0.0), (-9.0,3.0), (2.0,7.0), (7.0,-4.0)/
DATA D/(3.0,-5.0), (4.0,-9.0), (-5.0,-7.0), (-2.0,-3.0)/
DATA E/(-9.0,8.0), (1.0,8.0), (8.0,3.0), (0.0,0.0)/
DATA B/(-16.0,-93.0), (128.0,179.0), (-60.0,-12.0), (9.0,-108.0)/
```

```

!
!
CALL LSLTQ (C, D, E, B)
!
! Output the solution.
CALL WRCRL ('Solution:', B, RLABEL, CLABEL, 1, N, 1, FMT=FMT)
END

```

## Output

```

Solution:
 1 2
(-0.400000E+01, -0.700000E+01) (-0.700000E+01, 0.400000E+01)
 3 4
(0.700000E+01, -0.700000E+01) (0.900000E+01, 0.200000E+01)

```

---

## LSLCQ

Computes the *LDU* factorization of a complex tridiagonal matrix *A* using a cyclic reduction algorithm.

### Required Arguments

- C** — Complex array of size  $2N$  containing the upper codiagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $C(1), \dots, C(N - 1)$ . (Input/Output)
- A** — Complex array of size  $2N$  containing the diagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $A(1), \dots, A(N)$ . (Input/Output)
- B** — Complex array of size  $2N$  containing the lower codiagonal of the  $N$  by  $N$  tridiagonal matrix in the entries  $B(1), \dots, B(N - 1)$ . (Input/Output)
- Y** — Complex array of size  $2N$  containing the right-hand side of the system  $Ax = y$  in the order  $Y(1), \dots, Y(N)$ . (Input/Output)  
The vector  $x$  overwrites  $Y$  in storage.
- U** — Real array of size  $2N$  of flags that indicate any singularities of  $A$ . (Output)  
A value  $U(I) = 1$  means that a divide by zero would have occurred during the factoring. Otherwise  $U(I) = 0$ .
- IR** — Array of integers that determine the sizes of loops performed in the cyclic reduction algorithm. (Output)
- IS** — Array of integers that determine the sizes of loops performed in the cyclic reduction algorithm. (Output)  
The sizes of these arrays must be at least  $\log_2(N) + 3$ .

## Optional Arguments

$N$  — Order of the matrix. (Input)  
N must be greater than zero.  
Default:  $N = \text{size}(C,1)$ .

*IJOB* — Flag to direct the desired factoring or solving step. (Input)  
Default:  $IJOB = 1$ .

| <b>IJOB</b> | <b>Action</b>                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------|
| 1           | Factor the matrix $A$ and solve the system $Ax = y$ , where $y$ is stored in array $Y$ .                                  |
| 2           | Do the solve step only. Use $y$ from array $Y$ . (The factoring step has already been done.)                              |
| 3           | Factor the matrix $A$ but do not solve a system.                                                                          |
| 4           | Same meaning as with the value $IJOB = 3$ . For efficiency, no error checking is done on the validity of any input value. |

## FORTRAN 90 Interface

Generic: `CALL LSLCQ (C, A, B, Y, U, IR, IS [, ...])`

Specific: The specific interface names are `S_LSLCQ` and `D_LSLCQ`.

## FORTRAN 77 Interface

Single: `CALL LSLCQ (N, C, A, B, IJOB, Y, U, IR, IS)`

Double: The double precision name is `DSLSCQ`.

## Description

Routine `LSLCQ` factors and solves the complex tridiagonal linear system  $Ax = y$ . The matrix is decomposed in the form  $A = LDU$ , where  $L$  is unit lower triangular,  $U$  is unit upper triangular, and  $D$  is diagonal. The algorithm used for the factorization is effectively that described in Kershaw (1982). More details, tests and experiments are reported in Hanson (1990).

`LSLCQ` is intended just for tridiagonal systems. The coefficient matrix does not have to be Hermitian. The algorithm amounts to Gaussian elimination, with no pivoting for numerical stability, on the matrix whose rows and columns are permuted to a new order. See Hanson (1990) for details. The expectation is that `LSLCQ` will outperform either `LSLTQ` or `LSLQB` on vector or parallel computers. Its performance may be inferior for small values of  $n$ , on scalar computers, or high-performance computers with non-optimizing compilers.

## Example

A real skew-symmetric tridiagonal matrix,  $A$ , of dimension  $n = 1000$  is given by  $c_k = -k$ ,  $a_k = 0$ , and  $b_k = k$ ,  $k = 1, \dots, n - 1$ ,  $a_n = 0$ . This matrix will have eigenvalues that are purely imaginary. The eigenvalue closest to the imaginary unit is required. This number is obtained by using inverse iteration to approximate a complex eigenvector  $y$ . The eigenvalue is approximated by  $\lambda = y^H A y / y^H y$ . (This example is contrived in the sense that the given tridiagonal skew-symmetric matrix eigenvalue problem is essentially equivalent to the tridiagonal symmetric eigenvalue problem where the  $c_k = k$  and the other data are unchanged.)

```
 USE LSLCQ_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LP, N, N2
 PARAMETER (LP=12, N=1000, N2=2*N)
!
 INTEGER I, IJOB, IR(LP), IS(LP), K, NOUT
 REAL AIMAG, U(N2)
 COMPLEX A(N2), B(N2), C(N2), CMPLX, CONJG, S, T, Y(N2)
 INTRINSIC AIMAG, CMPLX, CONJG
!
! Define entries of skew-symmetric
! matrix, A:
 DO 10 I=1, N - 1
 C(I) = -I
!
! This amounts to subtracting the
! positive imaginary unit from the
! diagonal. (The eigenvalue closest
! to this value is desired.)
 A(I) = CMPLX(0.E0,-1.0E0)
 B(I) = I
!
! This initializes the approximate
! eigenvector.
 Y(I) = 1.E0
10 CONTINUE
 A(N) = CMPLX(0.E0,-1.0E0)
 Y(N) = 1.E0
!
! First step of inverse iteration
! follows. Obtain decomposition of
! matrix and solve the first system:
 IJOB = 1
 CALL LSLCQ (C, A, B, Y, U, IR, IS, N=N, IJOB=IJOB)
!
! Next steps of inverse iteration
! follow. Solve the system again with
! the decomposition ready:
 IJOB = 2
 DO 20 K=1, 3
 CALL LSLCQ (C, A, B, Y, U, IR, IS, N=N, IJOB=IJOB)
20 CONTINUE
!
! Compute the Raleigh quotient to
! estimate the eigenvalue closest to
! the positive imaginary unit. After
```

```

! the approximate eigenvector, y, is
! computed, the estimate of the
! eigenvalue is ctrans(y)*A*y/t,
! where t = ctrans(y)*y.
S = -CONJG(Y(1))*Y(2)
T = CONJG(Y(1))*Y(1)
DO 30 I=2, N - 1
 S = S + CONJG(Y(I))*((I-1)*Y(I-1)-I*Y(I+1))
 T = T + CONJG(Y(I))*Y(I)
30 CONTINUE
S = S + CONJG(Y(N))*(N-1)*Y(N-1)
T = T + CONJG(Y(N))*Y(N)
S = S/T
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The value of n is: ', N
WRITE (NOUT,*) ' Value of approximate imaginary eigenvalue:', &
 AIMAG(S)
STOP
END

```

## Output

```

The value of n is: 1000
Value of approximate imaginary eigenvalue: 1.03811

```

---

## LSACB

Solves a complex system of linear equations in band storage mode with iterative refinement.

### Required Arguments

**A** — Complex  $NLCA + NUCA + 1$  by  $N$  array containing the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)

IPATH = 1 means the system  $AX = B$  is solved.

IPATH = 2 means the system  $A^H X = B$  is solved.

Default: IPATH = 1.

## FORTRAN 90 Interface

Generic:     CALL LSACB (A, NLCA, NUCA, B, X [, ...])

Specific:    The specific interface names are S\_LSACB and D\_LSACB.

## FORTRAN 77 Interface

Single:      CALL LSACB (N, A, LDA, NLCA, NUCA, B, IPATH, X)

Double:     The double precision name is DLSACB.

## Description

Routine LSACB solves a system of linear algebraic equations having a complex banded coefficient matrix. It first uses the routine LFCCB to compute an  $LU$  factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using the iterative refinement routine LFICB.

LSACB fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element or if the iterative refinement algorithm fails to converge. These errors occur only if  $A$  is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system. LSACB solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2ACB/DL2ACB. The reference is:

```
CALL L2ACB (N, A, LDA, NLCA, NUCA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** — Complex work vector of length  $(2 * NLCA + NUCA + 1) * N$  containing the  $LU$  factorization of  $A$  on output.

**IPVT** — Integer work vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  on output.

**WK** — Complex work vector of length  $N$ .

2. Informational errors

Type Code

- |   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 3 | 3 | The input matrix is too ill-conditioned. The solution might not be accurate. |
| 4 | 2 | The input matrix is singular.                                                |

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2ACB` the leading dimension of `FACT` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSACB`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSACB`. Users directly calling `L2ACB` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSACB` or `L2ACB`. Default values for the option are `IVAL(*) = 1,16,0,1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSACB` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CCB` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CCB` skips this computation. `LSACB` restores the option. Default values for the option are `IVAL(*) = 1,2`.

## Example

A system of four linear equations is solved. The coefficient matrix has complex banded form with one upper and one lower codiagonal. The right-hand-side vector  $b$  has four elements.

```
USE LSACB_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N, NLCA, NUCA
PARAMETER (LDA=3, N=4, NLCA=1, NUCA=1)
COMPLEX A(LDA,N), B(N), X(N)
!
! Set values for A in band form, and B
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (-2.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 1.0+1.0i 0.0+2.0i 0.0+0.0i)
!
! B = (-10.0-5.0i 9.5+5.5i 12.0-12.0i 0.0+8.0i)
!
DATA A/(0.0,0.0), (-2.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
 (1.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
 (1.0,-1.0), (0.0,0.0)/
DATA B/(-10.0,-5.0), (9.5,5.5), (12.0,-12.0), (0.0,8.0)/
```

```

! Solve A*X = B
CALL LSACB (A, NLCA, NUCA, B, X)
! Print results
CALL WRCRN ('X', X, 1, N, 1)
!
END

```

## Output

```

 1 2 X 3 4
(3.000, 0.000) (-1.000, 1.000) (3.000, 0.000) (-1.000, 1.000)

```

---

## LSLCB

Solves a complex system of linear equations in band storage mode without iterative refinement.

### Required Arguments

**A** — Complex  $NLCA + NUCA + 1$  by  $N$  array containing the  $N$  by  $N$  banded coefficient matrix in band storage mode. (Input)

**NLCA** — Number of lower codiagonals of **A**. (Input)

**NUCA** — Number of upper codiagonals of **A**. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If **B** is not needed, then **B** and **X** may share the same storage locations)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means the system  $AX = B$  is solved.  
 $IPATH = 2$  means the system  $A^H X = B$  is solved.  
Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic:     CALL LSLCB (A, NLCA, NUCA, B, X [, ...])

Specific: The specific interface names are `S_LSLCB` and `D_LSLCB`.

## FORTRAN 77 Interface

Single: `CALL LSLCB (N, A, LDA, NLCA, NUCA, B, IPATH, X)`

Double: The double precision name is `DLSLCB`.

## Description

Routine `LSLCB` solves a system of linear algebraic equations having a complex banded coefficient matrix. It first uses the routine `LFCCB` to compute an *LU* factorization of the coefficient matrix and to estimate the condition number of the matrix. The solution of the linear system is then found using `LFSCB`.

`LSLCB` fails if *U*, the upper triangular part of the factorization, has a zero diagonal element. This occurs only if *A* is singular or very close to a singular matrix.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that `LSACB` be used.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LCB/DL2LCB`. The reference is:

```
CALL L2LCB (N, A, LDA, NLCA, NUCA, B, IPATH, X, FACT, IPVT, WK)
```

The additional arguments are as follows:

**FACT** —  $(2 * NLCA + NUCA + 1) \times N$  complex work array containing the *LU* factorization of *A* on output. If *A* is not needed, *A* can share the first  $(NLCA + NUCA + 1) * N$  locations with **FACT**.

**IPVT** — Integer work vector of length *N* containing the pivoting information for the *LU* factorization of *A* on output.

**WK** — Complex work vector of length *N*.

2. Informational errors

Type Code

3 3 The input matrix is too ill-conditioned. The solution might not be accurate.

4 2 The input matrix is singular.

3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LCB` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSLCB`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLCB`. Users directly calling `L2LCB` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSLCB` or `L2LCB`. Default values for the option are `IVAL(*) = 1,16,0,1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSLCB` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CCB` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CCB` skips this computation. `LSLCB` restores the option. Default values for the option are `IVAL(*) = 1,2`.

### Example

A system of four linear equations is solved. The coefficient matrix has complex banded form with one upper and one lower codiagonal. The right-hand-side vector  $b$  has four elements.

```

USE LSLCB_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, N, NLCA, NUCA
PARAMETER (LDA=3, N=4, NLCA=1, NUCA=1)
COMPLEX A(LDA,N), B(N), X(N)
!
! Set values for A in band form, and B
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (-2.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 1.0+1.0i 0.0+2.0i 0.0+0.0i)
!
! B = (-10.0-5.0i 9.5+5.5i 12.0-12.0i 0.0+8.0i)
!
DATA A/(0.0,0.0), (-2.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
 (1.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
 (1.0,-1.0), (0.0,0.0)/
DATA B/(-10.0,-5.0), (9.5,5.5), (12.0,-12.0), (0.0,8.0)/
!
! Solve A*X = B
CALL LSLCB (A, NLCA, NUCA, B, X)
!
! Print results
CALL WRCRN ('X', X, 1, N, 1)
!
END

```

## Output

X

( 3.000, 0.000)<sup>1</sup> (-1.000, 1.000)<sup>2</sup> ( 3.000, 0.000)<sup>3</sup> (-1.000, 1.000)<sup>4</sup>

---

## LFCCB

Computes the  $LU$  factorization of a complex matrix in band storage mode and estimate its  $L_1$  condition number.

### Required Arguments

**A** — Complex  $NLCA + NUCA + 1$  by  $N$  array containing the  $N$  by  $N$  matrix in band storage mode to be factored. (Input)

**NLCA** — Number of lower codiagonals of **A**. (Input)

**NUCA** — Number of upper codiagonals of **A**. (Input)

**FACT** — Complex  $2 * NLCA + NUCA + 1$  by  $N$  array containing the  $LU$  factorization of the matrix **A**. (Output)

If **A** is not needed, **A** can share the first  $(NLCA + NUCA + 1) * N$  locations with **FACT**.

**IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization. (Output)

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(\mathbf{A}, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic:    CALL LFCCB (A, NLCA, NUCA, FACT, IPVT, RCOND [, ...])

Specific:   The specific interface names are S\_LFCCB and D\_LFCCB.

## FORTRAN 77 Interface

Single:      CALL LFCCB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, RCOND)

Double:      The double precision name is DLFCCB.

## Description

Routine LFCCB performs an  $LU$  factorization of a complex banded coefficient matrix. It also estimates the condition number of the matrix. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCCB fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  is singular or very close to a singular matrix.

The  $LU$  factors are returned in a form that is compatible with IMSL routines LFICB, LFSCB and LFDCEB. To solve systems of equations with multiple right-hand-side vectors, use LFCCB followed by either LFICB or LFSCB called once for each right-hand side. The routine LFDCEB can be called to compute the determinant of the coefficient matrix after LFCCB has performed the factorization.

Let  $F$  be the matrix FACT, let  $m_l = NLCA$  and let  $m_u = NUCA$ . The first  $m_l + m_u + 1$  rows of  $F$  contain the triangular matrix  $U$  in band storage form. The lower  $m_l$  rows of  $F$  contain the multipliers needed to reconstruct  $L$ .

LFCCB is based on the LINPACK routine CGBCO; see Dongarra et al. (1979). CGBCO uses unscaled partial pivoting.

## Comments

1.      Workspace may be explicitly provided, if desired, by use of L2CCB/DL2CCB. The reference is:

```
CALL L2CCB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, RCOND, WK)
```

The additional argument is

**WK** — Complex work vector of length  $N$ .

2.      Informational errors

Type   Code

- 3 1 The input matrix is algorithmically singular.
- 4 2 The input matrix is singular.

### Example

The inverse of a  $4 \times 4$  band matrix with one upper and one lower codiagonal is computed. LFCCB is called to factor the matrix and to check for singularity or ill-conditioning. LFICB is called to determine the columns of the inverse.

```

USE LFCCB_INT
USE UMACH_INT
USE LFICB_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL RCOND
COMPLEX A(LDA,N), AINV(N,N), FACT(LDFACT,N), RJ(N), RES(N)
!
! Set values for A in band form
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (0.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 4.0+1.0i 0.0+2.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), (0.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
 (4.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
 (1.0,-1.0), (0.0,0.0)/
!
CALL LFCCB (A, NLCA, NUCA, FACT, IPVT, RCOND)
!
! Print the reciprocal condition number
! and the L1 condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFICB
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFICB (A, NLCA, NUCA, FACT, IPVT, RJ, AINV(:,J), RES)
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
!
! Print results
CALL WRCRN ('AINV', AINV)
!

```

```

99999 FORMAT (' RCOND = ',F5.3,/,', L1 condition number = ',F6.3)
END

```

## Output

RCOND = 0.022

L1 condition number = 45.933

|   | AINV            |                 |                 |                 |
|---|-----------------|-----------------|-----------------|-----------------|
|   | 1               | 2               | 3               | 4               |
| 1 | ( 0.562, 0.170) | ( 0.125, 0.260) | (-0.385,-0.135) | (-0.239,-1.165) |
| 2 | ( 0.122, 0.421) | (-0.195, 0.094) | ( 0.101,-0.289) | ( 0.874,-0.179) |
| 3 | ( 0.034, 0.904) | (-0.437, 0.090) | (-0.153,-0.527) | ( 1.087,-1.172) |
| 4 | ( 0.938, 0.870) | (-0.347, 0.527) | (-0.679,-0.374) | ( 0.415,-1.759) |

---

## LFTCB

Computes the *LU* factorization of a complex matrix in band storage mode.

### Required Arguments

**A** — Complex  $NLCA + NUCA + 1$  by  $N$  array containing the  $N$  by  $N$  matrix in band storage mode to be factored. (Input)

**NLCA** — Number of lower codiagonals of **A**. (Input)

**NUCA** — Number of upper codiagonals of **A**. (Input)

**FACT** — Complex  $2 * NLCA + NUCA + 1$  by  $N$  array containing the *LU* factorization of the matrix **A**. (Output)

If **A** is not needed, **A** can share the first  $(NLCA + NUCA + 1) * N$  locations with **FACT**.

**IPVT** — Integer vector of length  $N$  containing the pivoting information for the *LU* factorization. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:     CALL LFTCB (A, NLCA, NUCA, FACT, IPVT [, ...])

Specific:    The specific interface names are S\_LFTCB and D\_LFTCB.

## FORTRAN 77 Interface

Single:      CALL LFTCB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT)

Double:     The double precision name is DLFTCB.

## Description

Routine LFTCB performs an  $LU$  factorization of a complex banded coefficient matrix. The  $LU$  factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same  $\infty$ -norm.

LFTCB fails if  $U$ , the upper triangular part of the factorization, has a zero diagonal element. This can occur only if  $A$  is singular or very close to a singular matrix.

The  $LU$  factors are returned in a form that is compatible with routines LFICB, LFSCB and LFDCEB. To solve systems of equations with multiple right-hand-side vectors, use LFTCB followed by either LFICB or LFSCB called once for each right-hand side. The routine LFDCEB can be called to compute the determinant of the coefficient matrix after LFTCB has performed the factorization.

Let  $F$  be the matrix FACT, let  $m_l = NLCA$  and let  $m_u = NUCA$ . The first  $m_l + m_u + 1$  rows of  $F$  contain the triangular matrix  $U$  in band storage form. The lower  $m_l$  rows of  $F$  contain the multipliers needed to reconstruct  $L^{-1}$ . LFTCB is based on the LINPACK routine CGBFA; see Dongarra et al. (1979). CGBFA uses unscaled partial pivoting.

## Comments

1.     Workspace may be explicitly provided, if desired, by use of L2TCB/DL2TCB The reference is:

```
CALL L2TCB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, WK)
```

The additional argument is:

**WK** — Complex work vector of length N used for scaling.

2.     Informational error

Type   Code

4     2            The input matrix is singular.

## Example

A linear system with multiple right-hand sides is solved. `LFTCB` is called to factor the coefficient matrix. `LFSCB` is called to compute the two solutions for the two right-hand sides. In this case the coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCCB` to perform the factorization, and `LFICB` to compute the solutions.

```
 USE LFTCB_INT
 USE LFSCB_INT
 USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
COMPLEX A(LDA,N), B(N,2), FACT(LDFACT,N), X(N,2)
!
! Set values for A in band form, and B
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (0.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 4.0+1.0i 0.0+2.0i 0.0+0.0i)
!
! B = (-4.0-5.0i 16.0-4.0i)
! (9.5+5.5i -9.5+19.5i)
! (9.0-9.0i 12.0+12.0i)
! (0.0+8.0i -8.0-2.0i)
!
! DATA A/(0.0,0.0), (0.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
! (4.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
! (1.0,-1.0), (0.0,0.0)/
! DATA B/(-4.0,-5.0), (9.5,5.5), (9.0,-9.0), (0.0,8.0), &
! (16.0,-4.0), (-9.5,19.5), (12.0,12.0), (-8.0,-2.0)/
!
! CALL LFTCB (A, NLCA, NUCA, FACT, IPVT)
!
! Solve for the two right-hand sides
DO 10 J=1, 2
 CALL LFSCB (FACT, NLCA, NUCA, IPVT, B(:,J), X(:,J))
10 CONTINUE
!
! Print results
CALL WRCRN ('X', X)
!
END
```

## Output

X

|   |                 |   |                |
|---|-----------------|---|----------------|
|   | 1               |   | 2              |
| 1 | ( 3.000, 0.000) | ( | 0.000, 4.000)  |
| 2 | (-1.000, 1.000) | ( | 1.000, -1.000) |
| 3 | ( 3.000, 0.000) | ( | 0.000, 4.000)  |
| 4 | (-1.000, 1.000) | ( | 1.000, -1.000) |

---

## LFSCB

Solves a complex system of linear equations given the *LU* factorization of the coefficient matrix in band storage mode.

### Required Arguments

**FACT** — Complex  $2 * NLCA + NUCA + 1$  by  $N$  array containing the *LU* factorization of the coefficient matrix  $A$  as output from subroutine `LFCCB/DLFCCB` or `LFTCB/DLFTCB`. (Input)

**NLCA** — Number of lower codiagonals of  $A$ . (Input)

**NUCA** — Number of upper codiagonals of  $A$ . (Input)

**IPVT** — Vector of length  $N$  containing the pivoting information for the *LU* factorization of  $A$  as output from subroutine `LFCCB/DLFCCB` or `LFTCB/DLFTCB`. (Input)

**B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

**IPATH** — Path indicator. (Input)  
 $\text{IPATH} = 1$  means the system  $AX = B$  is solved.  
 $\text{IPATH} = 2$  means the system  $A^H X = B$  is solved.  
Default:  $\text{IPATH} = 1$ .

### FORTRAN 90 Interface

Generic: `CALL LFSCB (FACT, NLCA, NUCA, IPVT, B, X [, ...])`

Specific: The specific interface names are `S_LFSCB` and `D_LFSCB`.

## FORTRAN 77 Interface

Single: `CALL LFSCB (N, FACT, LDFACT, NLCA, NUCA, IPVT, B, IPATH, X)`

Double: The double precision name is `DLFSCB`.

## Description

Routine `LFSCB` computes the solution of a system of linear algebraic equations having a complex banded coefficient matrix. To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either `LFCCB` or `LFTCB`. The solution to  $Ax = b$  is found by solving the banded triangular systems  $Ly = b$  and  $Ux = y$ . The forward elimination step consists of solving the system  $Ly = b$  by applying the same permutations and elimination operations to  $b$  that were applied to the columns of  $A$  in the factorization routine. The backward substitution step consists of solving the banded triangular system  $Ux = y$  for  $x$ .

`LFSCB` and `LFICB` both solve a linear system given its *LU* factorization. `LFICB` generally takes more time and produces a more accurate answer than `LFSCB`. Each iteration of the iterative refinement algorithm used by `LFICB` calls `LFSCB`.

`LFSCB` is based on the LINPACK routine `CGBSL`; see Dongarra et al. (1979).

## Example

The inverse is computed for a real banded  $4 \times 4$  matrix with one upper and one lower codiagonal. The input matrix is assumed to be well-conditioned; hence `LFTCB` is used rather than `LFCCB`.

```
USE LFSCB_INT
USE LFTCB_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
COMPLEX A(LDA,N), AINV(N,N), FACT(LDFACT,N), RJ(N)
!
! Set values for A in band form
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (-2.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 1.0+1.0i 0.0+2.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), (-2.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
(1.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
(1.0,-1.0), (0.0,0.0)/
!
CALL LFTCB (A, NLCA, NUCA, FACT, IPVT)
! Set up the columns of the identity
```

```

! matrix one at a time in RJ
 RJ = (0.0E0,0.0E0)
 DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
! RJ is the J-th column of the identity
! matrix so the following LFSCB
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSCB (FACT, NLCA, NUCA, IPVT, RJ, AINV(:,J))
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
! Print results
 CALL WRCRN ('AINV', AINV)
!
 END

```

## Output

|   | 1                | 2                | 3                | 4                |
|---|------------------|------------------|------------------|------------------|
| 1 | ( 0.165, -0.341) | ( 0.376, -0.094) | (-0.282, 0.471)  | (-1.600, 0.000)  |
| 2 | ( 0.588, -0.047) | ( 0.259, 0.235)  | (-0.494, 0.024)  | (-0.800, -1.200) |
| 3 | ( 0.318, 0.271)  | ( 0.012, 0.247)  | (-0.759, -0.235) | (-0.550, -2.250) |
| 4 | ( 0.588, -0.047) | ( 0.259, 0.235)  | (-0.994, 0.524)  | (-2.300, -1.200) |

---

## LFICB

Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.

### Required Arguments

- A** — Complex  $NLCA + NUCA + 1$  by  $N$  array containing the  $N$  by  $N$  coefficient matrix in band storage mode. (Input)
- NLCA** — Number of lower codiagonals of  $A$ . (Input)
- NUCA** — Number of upper codiagonals of  $A$ . (Input)
- FACT** — Complex  $2 * NLCA + NUCA + 1$  by  $N$  array containing the  $LU$  factorization of the matrix  $A$  as output from routine `LFCCB/DFCCB` or `LFTCB/DFTCB`. (Input)
- IPVT** — Vector of length  $N$  containing the pivoting information for the  $LU$  factorization of  $A$  as output from routine `LFCCB/DFCCB` or `LFTCB/DFTCB`. (Input)
- B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X** — Complex vector of length  $N$  containing the solution. (Output)
- RES** — Complex vector of length  $N$  containing the residual vector at the improved solution. (Output)

## Optional Arguments

*N* — Number of equations. (Input)

Default:  $N = \text{size}(A, 2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A, 1)$ .

*LDFACT* — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDFACT = \text{size}(FACT, 1)$ .

*IPATH* — Path indicator. (Input)

$IPATH = 1$  means the system  $AX = B$  is solved.

$IPATH = 2$  means the system  $A^H X = B$  is solved.

Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic:    CALL LFICB (A, NLCA, NUCA, FACT, IPVT, B, X, RES [, ...])

Specific:   The specific interface names are S\_LFICB and D\_LFICB.

## FORTRAN 77 Interface

Single:     CALL LFICB (N, A, LDA, NLCA, NUCA, FACT, LDFACT, IPVT, B, IPATH, X,  
RES)

Double:     The double precision name is DLFICB.

## Description

Routine LFICB computes the solution of a system of linear algebraic equations having a complex banded coefficient matrix. Iterative refinement is performed on the solution vector to improve the accuracy. Usually almost all of the digits in the solution are accurate, even if the matrix is somewhat ill-conditioned.

To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either LFCCB or LFTCB.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFICB and LFSCB both solve a linear system given its *LU* factorization. LFICB generally takes more time and produces a more accurate answer than LFSCB. Each iteration of the iterative refinement algorithm used by LFICB calls LFSCB.

## Comments

Informational error

Type Code

3 3 The input matrix is too ill-conditioned for iterative refinement be effective.

### Example

A set of linear systems is solved successively. The right-hand-side vector is perturbed after solving the system each of the first two times by adding  $(1 + i)/2$  to the second element.

```
USE LFICB_INT
USE LFCCB_INT
USE WRCRN_INT
USE UMACH_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
INTEGER IPVT(N)
REAL RCOND
COMPLEX A(LDA,N), B(N), FACT(LDFACT,N), RES(N), X(N)
!
! Set values for A in band form, and B
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (-2.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 1.0+1.0i 0.0+2.0i 0.0+0.0i)
!
! B = (-10.0-5.0i 9.5+5.5i 12.0-12.0i 0.0+8.0i)
!
DATA A/(0.0,0.0), (-2.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
 (1.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
 (1.0,-1.0), (0.0,0.0)/
DATA B/(-10.0,-5.0), (9.5,5.5), (12.0,-12.0), (0.0,8.0)/
!
CALL LFCCB (A, NLCA, NUCA, FACT, IPVT, RCOND)
!
! Print the reciprocal condition number
CALL UMACH (2, NOUT)
WRITE (NOUT,99998) RCOND, 1.0E0/RCOND
!
! Solve the three systems
DO 10 J=1, 3
 CALL LFICB (A, NLCA, NUCA, FACT, IPVT, B, X, RES)
!
! Print results
 WRITE (NOUT, 99999) J
 CALL WRCRN ('X', X, 1, N, 1)
 CALL WRCRN ('RES', RES, 1, N, 1)
!
! Perturb B by adding 0.5+0.5i to B(2)
 B(2) = B(2) + (0.5E0,0.5E0)
10 CONTINUE
!
99998 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
99999 FORMAT (//, ' For system ',I1)
END
```

## Output

RCOND = 0.014

L1 Condition number = 72.414

For system 1

X  
1 2 3 4  
( 3.000, 0.000) (-1.000, 1.000) ( 3.000, 0.000) (-1.000, 1.000)

RES  
1 2 3  
( 0.000E+00, 0.000E+00) ( 0.000E+00, 0.000E+00) ( 0.000E+00, 5.684E-14)  
4  
( 3.494E-22, -6.698E-22)

For system 2

X  
1 2 3 4  
( 3.235, 0.141) (-0.988, 1.247) ( 2.882, 0.129) (-0.988, 1.247)

RES  
1 2 3  
(-1.402E-08, 6.486E-09) (-7.012E-10, 4.488E-08) (-1.122E-07, 7.188E-09)  
4  
(-7.012E-10, 4.488E-08)

For system 3

X  
1 2 3 4  
( 3.471, 0.282) (-0.976, 1.494) ( 2.765, 0.259) (-0.976, 1.494)

RES  
1 2 3  
(-2.805E-08, 1.297E-08) (-1.402E-09, -2.945E-08) ( 1.402E-08, 1.438E-08)  
4  
(-1.402E-09, -2.945E-08)

---

## LFDCB

Computes the determinant of a complex matrix given the *LU* factorization of the matrix in band storage mode.

### Required Arguments

**FACT** — Complex  $(2 * NLCA + NUCA + 1)$  by  $N$  array containing the *LU* factorization of the matrix *A* as output from routine LFTCB/DLFTCB or LFCCB/DLFCCB. (Input)

**NLCA** — Number of lower codiagonals in matrix *A*. (Input)

*NUCA* — Number of upper codiagonals in matrix *A*. (Input)

*IPVT* — Vector of length *N* containing the pivoting information for the *LU* factorization as output from routine *LFTCB/DLFTCB* or *LFCCB/DLFCCB*. (Input)

*DET1* — Complex scalar containing the mantissa of the determinant. (Output)  
The value *DET1* is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or *DET1* = 0.0.

*DET2* — Scalar containing the exponent of the determinant. (Output)  
The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

*N* — Order of the matrix. (Input)  
Default: *N* = size(*FACT*,2).

*LDFACT* — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDFACT* = size(*FACT*,1).

### FORTRAN 90 Interface

Generic:     CALL *LFDCB* (*FACT*, *NLCA*, *NUCA*, *IPVT*, *DET1*, *DET2* [, ...])

Specific:    The specific interface names are *S\_LFDCB* and *D\_LFDCB*.

### FORTRAN 77 Interface

Single:     CALL *LFDCB* (*N*, *FACT*, *LDFACT*, *NLCA*, *NUCA*, *IPVT*, *DET1*, *DET2*)

Double:     The double precision name is *DLFDCB*.

### Description

Routine *LFDCB* computes the determinant of a complex banded coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either *LFCCB* or *LFTCB*. The formula  $\det A = \det L \det U$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det U = \prod_{i=1}^N U_{ii}$$

(The matrix *U* is stored in the upper *NUCA* + *NLCA* + 1 rows of *FACT* as a banded matrix.) Since *L* is the product of triangular matrices with unit diagonals and of permutation matrices,  $\det L = (-1)^k$ , where *k* is the number of pivoting interchanges.

*LFDCB* is based on the LINPACK routine *CGBDI*; see Dongarra et al. (1979).

## Example

The determinant is computed for a complex banded  $4 \times 4$  matrix with one upper and one lower codiagonal.

```
 USE LFDCEB_INT
 USE LFTCB_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LDA, LDFACT, N, NLCA, NUCA, NOUT
 PARAMETER (LDA=3, LDFACT=4, N=4, NLCA=1, NUCA=1)
 INTEGER IPVT(N)
 REAL DET2
 COMPLEX A(LDA,N), DET1, FACT(LDFACT,N)
!
! Set values for A in band form
!
! A = (0.0+0.0i 4.0+0.0i -2.0+2.0i -4.0-1.0i)
! (-2.0-3.0i -0.5+3.0i 3.0-3.0i 1.0-1.0i)
! (6.0+1.0i 1.0+1.0i 0.0+2.0i 0.0+0.0i)
!
 DATA A/(0.0,0.0), (-2.0,-3.0), (6.0,1.0), (4.0,0.0), (-0.5,3.0), &
 (1.0,1.0), (-2.0,2.0), (3.0,-3.0), (0.0,2.0), (-4.0,-1.0), &
 (1.0,-1.0), (0.0,0.0)/
!
 CALL LFTCB (A, NLCA, NUCA, FACT, IPVT)
!
! Compute the determinant
 CALL LFDCEB (FACT, NLCA, NUCA, IPVT, DET1, DET2)
!
! Print the results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is (', F6.3, ', ', F6.3, ') * 10**', &
 F2.0)
 END
```

## Output

The determinant of A is ( 2.500,-1.500) \* 10\*\*1.

---

## LSAQH

Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.

### Required Arguments

*A* — Complex  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band Hermitian coefficient matrix in band Hermitian storage mode. (Input)

*NCODA* – Number of upper or lower codiagonals of *A*. (Input)

*B* — Complex vector of length *N* containing the right-hand side of the linear system. (Input)

*X* — Complex vector of length *N* containing the solution to the linear system. (Output)

### Optional Arguments

*N* — Number of equations. (Input)  
Default:  $N = \text{size}(A, 2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL LSAQH (A, NCODA, B, X [, ...])

Specific:    The specific interface names are S\_LSAQH and D\_LSAQH.

### FORTRAN 77 Interface

Single:      CALL LSAQH (N, A, LDA, NCODA, B, X)

Double:     The double precision name is DLSAQH.

### Description

Routine LSAQH solves a system of linear algebraic equations having a complex Hermitian positive definite band coefficient matrix. It first uses the IMSL routine LFCQH to compute an  $R^H R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix. *R* is an upper triangular band matrix. The solution of the linear system is then found using the iterative refinement IMSL routine LFIQH.

LSAQH fails if any submatrix of *R* is not positive definite, if *R* has a zero diagonal element, or if the iterative refinement algorithm fails to converge. These errors occur only if the matrix *A* either is very close to a singular matrix or is a matrix that is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in the solution *x*. Iterative refinement can sometimes find the solution to such a system. LSAQH solves the problem that is represented in the computer; however, this problem may differ from the problem whose solution is desired.

### Comments

1.     Workspace may be explicitly provided, if desired, by use of L2AQH/DL2AQH. The reference is:

```
CALL L2AQH (N, A, LDA, NCODA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** — Complex work vector of length  $(\text{NCODA} + 1) * N$  containing the  $R^H R$  factorization of **A** in band Hermitian storage form on output.

**WK** — Complex work vector of length  $N$ .

2. Informational errors

| Type | Code |                                                                                         |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 3    | The input matrix is too ill-conditioned. The solution might not be accurate.            |
| 3    | 4    | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4    | 2    | The input matrix is not positive definite.                                              |
| 4    | 4    | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2AQH` the leading dimension of `FACT` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSAQH`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSAQH`. Users directly calling `L2AQH` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSAQH` or `L2AQH`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.

**17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSAQH` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CQH` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CQH` skips this computation. `LSAQH` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

### Example

A system of five linear equations is solved. The coefficient matrix has complex Hermitian positive definite band form with one codiagonal and the right-hand-side vector  $b$  has five elements.

```
USE LSAQH_INT
USE WRCRN_INT
! Declare variables
```

```

INTEGER LDA, N, NCODA
PARAMETER (LDA=2, N=5, NCODA=1)
COMPLEX A(LDA,N), B(N), X(N)
!
! Set values for A in band Hermitian form, and B
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
! B = (1.0+5.0i 12.0-6.0i 1.0-16.0i -3.0-3.0i 25.0+16.0i)
!
DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
DATA B/(1.0,5.0), (12.0,-6.0), (1.0,-16.0), (-3.0,-3.0), &
 (25.0,16.0)/
!
! Solve A*X = B
CALL LSAQH (A, NCODA, B, X)
!
! Print results
CALL WRCRN ('X', X, 1, N, 1)
!
END

```

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)

```

---

## LSLQH

Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode without iterative refinement.

### Required Arguments

- A* — Complex  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band Hermitian coefficient matrix in band Hermitian storage mode. (Input)
- NCODA* — Number of upper or lower codiagonals of *A*. (Input)
- B* — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X* — Complex vector of length  $N$  containing the solution to the linear system. (Output)

### Optional Arguments

- N* — Number of equations. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

### FORTRAN 90 Interface

Generic:     CALL LSLQH (A, NCODA, B, X [, ...])

Specific:    The specific interface names are S\_LSLQH and D\_LSLQH.

### FORTRAN 77 Interface

Single:      CALL LSLQH (N, A, LDA, NCODA, B, X)

Double:      The double precision name is DLSLQH.

### Description

Routine LSLQH solves a system of linear algebraic equations having a complex Hermitian positive definite band coefficient matrix. It first uses the routine LFCQH to compute an  $R^H R$  Cholesky factorization of the coefficient matrix and to estimate the condition number of the matrix.  $R$  is an upper triangular band matrix. The solution of the linear system is then found using the routine LFSQH.

LSLQH fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  either is very close to a singular matrix or is a matrix that is not positive definite.

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . If the coefficient matrix is ill-conditioned or poorly scaled, it is recommended that LSAQH be used.

### Comments

1.    Workspace may be explicitly provided, if desired, by use of L2LQH/DL2LQH. The reference is:

```
CALL L2LQH (N, A, LDA, NCODA, B, X, FACT, WK)
```

The additional arguments are as follows:

**FACT** —  $(NCODA + 1) \times N$  complex work array containing the  $R^H R$  factorization of  $A$  in band Hermitian storage form on output. If  $A$  is not needed,  $A$  and **FACT** can share the same storage locations.

**WK** — Complex work vector of length  $N$ .

## 2. Informational errors

| Type | Code |                                                                                         |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 3    | The input matrix is too ill-conditioned. The solution might not be accurate.            |
| 3    | 4    | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4    | 2    | The input matrix is not positive definite.                                              |
| 4    | 4    | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

## 3. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2LQH` the leading dimension of `FACT` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSLQH`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLQH`. Users directly calling `L2LQH` can allocate additional space for `FACT` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSLQH` or `L2LQH`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSLQH` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CQH` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CQH` skips this computation. `LSLQH` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## Example

A system of five linear equations is solved. The coefficient matrix has complex Hermitian positive definite band form with one codiagonal and the right-hand-side vector  $b$  has five elements.

```
USE LSLQH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER N, NCODA, LDA
PARAMETER (N=5, NCODA=1, LDA=NCODA+1)
COMPLEX A(LDA,N), B(N), X(N)
!
! Set values for A in band Hermitian form, and B
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
! B = (1.0+5.0i 12.0-6.0i 1.0-16.0i -3.0-3.0i 25.0+16.0i)
!
```

```

DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
DATA B/(1.0,5.0), (12.0,-6.0), (1.0,-16.0), (-3.0,-3.0), &
 (25.0,16.0)/
!
! Solve A*X = B
CALL LSLQH (A, NCODA, B, X)
! Print results
CALL WRCRN ('X', X, 1, N, 1)
!
END

```

## Output

```

 X
 1 2 3 4
(2.000, 1.000) (3.000, 0.000) (-1.000,-1.000) (0.000,-2.000)
 5
(3.000, 2.000)

```

---

## LSLQB

Computes the  $R^H$  DR Cholesky factorization of a complex Hermitian positive-definite matrix  $A$  in codiagonal band Hermitian storage mode. Solve a system  $Ax = b$ .

### Required Arguments

$A$  — Array containing the  $N$  by  $N$  positive-definite band coefficient matrix and the right hand side in codiagonal band Hermitian storage mode. (Input/Output)  
The number of array columns must be at least  $2 * NCODA + 3$ . The number of columns is not an input to this subprogram.

$NCODA$  — Number of upper codiagonals of matrix  $A$ . (Input)

Must satisfy  $NCODA \geq 0$  and  $NCODA < N$ .

$U$  — Array of flags that indicate any singularities of  $A$ , namely loss of positive-definiteness of a leading minor. (Output)

A value  $U(I) = 0$ . means that the leading minor of dimension  $I$  is not positive-definite. Otherwise,  $U(I) = 1$ .

### Optional Arguments

$N$  — Order of the matrix. (Input)

Must satisfy  $N > 0$ .

Default:  $N = \text{size}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Must satisfy  $LDA \geq N + NCODA$ .  
Default:  $LDA = \text{size}(A,1)$ .

**IJOB** — flag to direct the desired factorization or solving step. (Input)  
Default:  $IJOB = 1$ .

IJOB Meaning

- 1 factor the matrix  $A$  and solve the system  $Ax = b$ ; where the real part of  $b$  is stored in column  $2 * NCODA + 2$  and the imaginary part of  $b$  is stored in column  $2 * NCODA + 3$  of array  $A$ . The real and imaginary parts of  $b$  are overwritten by the real and imaginary parts of  $x$ .
- 2 solve step only. Use the real part of  $b$  as column  $2 * NCODA + 2$  and the imaginary part of  $b$  as column  $2 * NCODA + 3$  of  $A$ . (The factorization step has already been done.) The real and imaginary parts of  $b$  are overwritten by the real and imaginary parts of  $x$ .
- 3 factor the matrix  $A$  but do not solve a system.
- 4,5,6 same meaning as with the value  $IJOB = 3$ . For efficiency, no error checking is done on values  $LDA$ ,  $N$ ,  $NCODA$ , and  $U(*)$ .

## FORTRAN 90 Interface

Generic: `CALL LSLQB (A, NCODA, U [, ...])`

Specific: The specific interface names are `S_LSLQB` and `D_LSLQB`.

## FORTRAN 77 Interface

Single: `CALL LSLQB (N, A, LDA, NCODA, IJOB, U)`

Double: The double precision name is `DLSLQB`.

## Description

Routine `LSLQB` factors and solves the Hermitian positive definite banded linear system  $Ax = b$ . The matrix is factored so that  $A = R^H DR$ , where  $R$  is unit upper triangular and  $D$  is diagonal and real. The reciprocals of the diagonal entries of  $D$  are computed and saved to make the solving step more efficient. Errors will occur if  $D$  has a nonpositive diagonal element. Such events occur only if  $A$  is very close to a singular matrix or is not positive definite.

`LSLQB` is efficient for problems with a small band width. The particular cases  $NCODA = 0, 1$  are done with special loops within the code. These cases will give good performance. See Hanson (1989) for more on the algorithm. When solving tridiagonal systems,  $NCODA = 1$ , the cyclic reduction code `LSLCQ` should be considered as an alternative. The expectation is that `LSLCQ` will outperform `LSLQB` on vector or parallel computers. It may be inferior on scalar computers or even parallel computers with non-optimizing compilers.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2LQB/DL2LQB. The reference is:

```
CALL L2LQB (N, A, LDA, NCODA, IJOB, U, WK1, WK2)
```

The additional arguments are as follows:

**WK1** — Work vector of length *NCODA*.

**WK2** — Work vector of length *NCODA*.

2. Informational error

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                            |
|---|---|--------------------------------------------|
| 4 | 2 | The input matrix is not positive definite. |
|---|---|--------------------------------------------|

## Example

A system of five linear equations is solved. The coefficient matrix has real positive definite codiagonal Hermitian band form and the right-hand-side vector *b* has five elements.

```
USE LSLQB_INT
USE WRRRN_INT
INTEGER LDA, N, NCODA
PARAMETER (N=5, NCODA=1, LDA=N+NCODA)
!
INTEGER I, IJOB, J
REAL A(LDA, 2*NCODA+3), U(N)
!
! Set values for A and right hand side
! in codiagonal band Hermitian form:
!
! (* * * * *)
! (2.0 * * 1.0 5.0)
! A = (4.0 -1.0 1.0 12.0 -6.0)
! (10.0 1.0 2.0 1.0 -16.0)
! (6.0 0.0 4.0 -3.0 -3.0)
! (9.0 1.0 1.0 25.0 16.0)
!
DATA ((A(I+NCODA, J), I=1, N), J=1, 2*NCODA+3) / 2.0, 4.0, 10.0, 6.0, &
9.0, 0.0, -1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 2.0, 4.0, 1.0, &
1.0, 12.0, 1.0, -3.0, 25.0, 5.0, -6.0, -16.0, -3.0, 16.0 /
!
! Factor and solve A*x = b.
!
IJOB = 1
CALL LSLQB (A, NCODA, U)
!
! Print results
!
CALL WRRRN ('REAL(X)', A((NCODA+1):, (2*NCODA+2):), 1, N, 1)
```

```
CALL WRRRN ('IMAG(X)', A((NCODA+1):(2*NCODA+3):), 1, N, 1)
END
```

## Output

```

 REAL(X)
 1 2 3 4 5
2.000 3.000 -1.000 0.000 3.000
```

```

 IMAG(X)
 1 2 3 4 5
1.000 0.000 -1.000 -2.000 2.000
```

## LFCQH

Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode and estimate its  $L_1$  condition number.

### Required Arguments

**A** — Complex  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band Hermitian matrix to be factored in band Hermitian storage mode. (Input)

**NCODA** — Number of upper or lower codiagonals of **A**. (Input)

**FACT** — Complex  $NCODA + 1$  by  $N$  array containing the  $R^H R$  factorization of the matrix **A**. (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage locations.

**RCOND** — Scalar containing an estimate of the reciprocal of the  $L_1$  condition number of **A**. (Output)

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A, 2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL LFCQH (A, NCODA, FACT, RCOND [, ...])`

Specific: The specific interface names are `S_LFCQH` and `D_LFCQH`.

## FORTRAN 77 Interface

Single:      CALL LFCQH (N, A, LDA, NCODA, FACT, LDFACT, RCOND)

Double:      The double precision name is DLFCQH.

## Description

Routine LFCQH computes an  $R^H R$  Cholesky factorization and estimates the condition number of a complex Hermitian positive definite band coefficient matrix.  $R$  is an upper triangular band matrix.

The  $L_1$  condition number of the matrix  $A$  is defined to be  $\kappa(A) = \|A\|_1 \|A^{-1}\|_1$ . Since it is expensive to compute  $\|A^{-1}\|_1$ , the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

If the estimated condition number is greater than  $1/\epsilon$  (where  $\epsilon$  is machine precision), a warning error is issued. This indicates that very small changes in  $A$  can cause very large changes in the solution  $x$ . Iterative refinement can sometimes find the solution to such a system.

LFCQH fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  either is very close to a singular matrix or is a matrix which is not positive definite.

The  $R^H R$  factors are returned in a form that is compatible with routines LFIQH, LFSQH and LFDQH. To solve systems of equations with multiple right-hand-side vectors, use LFCQH followed by either LFIQH or LFSQH called once for each right-hand side. The routine LFDQH can be called to compute the determinant of the coefficient matrix after LFCQH has performed the factorization.

LFCQH is based on the LINPACK routine CPBCO; see Dongarra et al. (1979).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of L2CQH/DL2CQH. The reference is:

```
CALL L2CQH (N, A, LDA, NCODA, FACT, LDFACT, RCOND, WK)
```

The additional argument is:

**WK** — Complex work vector of length  $N$ .

2.    Informational errors

Type   Code

3      1      The input matrix is algorithmically singular.

3      4      The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part.

4      2      The input matrix is not positive definite.

**Example**

The inverse of a  $5 \times 5$  band Hermitian matrix with one codiagonal is computed. LFCQH is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. LFIQH is called to determine the columns of the inverse.

```

USE LFCQH_INT
USE LFIQH_INT
USE UMACH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER N, NCODA, LDA, LDFACT, NOUT
PARAMETER (N=5, NCODA=1, LDA=NCODA+1, LDFACT=LDA)
REAL RCOND
COMPLEX A(LDA,N), AINV(N,N), FACT(LDFACT,N), RES(N), RJ(N)
!
! Set values for A in band Hermitian form
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
!
! Factor the matrix A
CALL LFCQH (A, NCODA, FACT, RCOND)
!
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
!
! RJ is the J-th column of the identity
! matrix so the following LFIQH
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFIQH (A, NCODA, FACT, RJ, AINV(:,J), RES)
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
!
! Print the results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) RCOND, 1.0E0/RCOND
CALL WRCRN ('AINV', AINV)
!
99999 FORMAT (' RCOND = ',F5.3,/, ' L1 Condition number = ',F6.3)
END

```

**Output**

```

RCOND = 0.067
L1 Condition number = 14.961

```

```

AINV

```

```

1 (0.7166, 0.0000) (0.2166,-0.2166) (-0.0899,-0.0300) (-0.0207, 0.0622)
2 (0.2166, 0.2166) (0.4332, 0.0000) (-0.0599,-0.1198) (-0.0829, 0.0415)
3 (-0.0899, 0.0300) (-0.0599, 0.1198) (0.1797, 0.0000) (0.0000,-0.1244)
4 (-0.0207,-0.0622) (-0.0829,-0.0415) (0.0000, 0.1244) (0.2592, 0.0000)
5 (0.0092, 0.0046) (0.0138,-0.0046) (-0.0138,-0.0138) (-0.0288, 0.0288)
1 (0.0092,-0.0046)
2 (0.0138, 0.0046)
3 (-0.0138, 0.0138)
4 (-0.0288,-0.0288)
5 (0.1175, 0.0000)

```

---

## LFTQH

Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.

### Required Arguments

**A** — Complex  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band Hermitian matrix to be factored in band Hermitian storage mode. (Input)

**NCODA** — Number of upper or lower codiagonals of **A**. (Input)

**FACT** — Complex  $NCODA + 1$  by  $N$  array containing the  $R^H R$  factorization of the matrix **A**. (Output)  
If **A** is not needed, **A** and **FACT** can share the same storage locations.

### Optional Arguments

**N** — Order of the matrix. (Input)  
Default:  $N = \text{size}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

### FORTRAN 90 Interface

Generic:    CALL LFTQH (A,NCODA,FACT [, ...])

Specific:   The specific interface names are `S_LFTQH` and `D_LFTQH`.

## FORTRAN 77 Interface

Single:      CALL LFTQH (N, A, LDA, NCODA, FACT, LDFACT)

Double:      The double precision name is DLFTQH.

## Description

Routine LFTQH computes an  $R^H R$  Cholesky factorization of a complex Hermitian positive definite band coefficient matrix.  $R$  is an upper triangular band matrix.

LFTQH fails if any submatrix of  $R$  is not positive definite or if  $R$  has a zero diagonal element. These errors occur only if  $A$  either is very close to a singular matrix or is a matrix which is not positive definite.

The  $R^H R$  factors are returned in a form that is compatible with routines LFIQH, LFSQH and LFDQH. To solve systems of equations with multiple right-hand-side vectors, use LFTQH followed by either LFIQH or LFSQH called once for each right-hand side. The routine LFDQH can be called to compute the determinant of the coefficient matrix after LFTQH has performed the factorization.

LFTQH is based on the LINPACK routine SPBFA; see Dongarra et al. (1979).

## Comments

Informational errors

Type Code

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 4 | The input matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4 | 2 | The input matrix is not positive definite.                                              |
| 4 | 4 | The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

## Example

The inverse of a  $5 \times 5$  band Hermitian matrix with one codiagonal is computed. LFTQH is called to factor the matrix and to check for nonpositive definiteness. LFSQH is called to determine the columns of the inverse.

```
USE LFTQH_INT
USE LFSQH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA
PARAMETER (LDA=2, LDFACT=2, N=5, NCODA=1)
COMPLEX A(LDA,N), AINV(N,N), FACT(LDFACT,N), RJ(N)
!
! Set values for A in band Hermitian form
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
```

```

 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
! Factor the matrix A
CALL LFTQH (A, NCODA, FACT)
! Set up the columns of the identity
! matrix one at a time in RJ
RJ = (0.0E0,0.0E0)
DO 10 J=1, N
 RJ(J) = (1.0E0,0.0E0)
! RJ is the J-th column of the identity
! matrix so the following LFSQH
! reference places the J-th column of
! the inverse of A in the J-th column
! of AINV
 CALL LFSQH (FACT, NCODA, RJ, AINV(:,J))
 RJ(J) = (0.0E0,0.0E0)
10 CONTINUE
! Print the results
CALL WRCRN ('AINV', AINV)
!
END

```

## Output

```

 AINV
 1 2 3 4
1 (0.7166, 0.0000) (0.2166,-0.2166) (-0.0899,-0.0300) (-0.0207, 0.0622)
2 (0.2166, 0.2166) (0.4332, 0.0000) (-0.0599,-0.1198) (-0.0829, 0.0415)
3 (-0.0899, 0.0300) (-0.0599, 0.1198) (0.1797, 0.0000) (0.0000,-0.1244)
4 (-0.0207,-0.0622) (-0.0829,-0.0415) (0.0000, 0.1244) (0.2592, 0.0000)
5 (0.0092, 0.0046) (0.0138,-0.0046) (-0.0138,-0.0138) (-0.0288, 0.0288)
 5
1 (0.0092,-0.0046)
2 (0.0138, 0.0046)
3 (-0.0138, 0.0138)
4 (-0.0288,-0.0288)
5 (0.1175, 0.0000)

```

---

## LFSQH

Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode.

### Required Arguments

**FACT** — Complex  $NCODA + 1$  by  $N$  array containing the  $R^H R$  factorization of the Hermitian positive definite band matrix  $A$ . (Input)

FACT is obtained as output from routine LFCQH/DLFCQH or LFTQH/DLFTQH.

**NCODA** — Number of upper or lower codiagonals of  $A$ . (Input)

**B** — Complex vector of length  $N$  containing the right-hand-side of the linear system. (Input)

$X$  — Complex vector of length  $N$  containing the solution to the linear system. (Output)  
If  $B$  is not needed,  $B$  and  $X$  can share the same storage locations.

### Optional Arguments

$N$  — Number of equations. (Input)  
Default:  $N = \text{size}(\text{FACT}, 2)$ .

$LDFACT$  — Leading dimension of  $\text{FACT}$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL LFSQH (FACT, NCODA, B, X [, ...])`

Specific: The specific interface names are `S_LFSQH` and `D_LFSQH`.

### FORTRAN 77 Interface

Single: `CALL LFSQH (N, FACT, LDFACT, NCODA, B, X)`

Double: The double precision name is `DLFSQH`.

### Description

Routine `LFSQH` computes the solution for a system of linear algebraic equations having a complex Hermitian positive definite band coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either IMSL routine `LFCQH` or `LFTQH`.  $R$  is an upper triangular band matrix.

The solution to  $Ax = b$  is found by solving the triangular systems  $R^H y = b$  and  $Rx = y$ .

`LFSQH` and `LFIQH` both solve a linear system given its  $R^H R$  factorization. `LFIQH` generally takes more time and produces a more accurate answer than `LFSQH`. Each iteration of the iterative refinement algorithm used by `LFIQH` calls `LFSQH`.

`LFSQH` is based on the LINPACK routine `CPBSL`; see Dongarra et al. (1979).

### Comments

Informational error

Type Code

4 1 The factored matrix has a diagonal element close to zero.

### Example

A set of linear systems is solved successively. `LFTQH` is called to factor the coefficient matrix. `LFSQH` is called to compute the three solutions for the three right-hand sides. In this case the

coefficient matrix is assumed to be well-conditioned and correctly scaled. Otherwise, it would be better to call `LFCQH` to perform the factorization, and `LFIQH` to compute the solutions.

```

USE LFSQH_INT
USE LFTQH_INT
USE WRCRN_INT
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA
PARAMETER (LDA=2, LDFACT=2, N=5, NCODA=1)
COMPLEX A(LDA,N), B(N,3), FACT(LDFACT,N), X(N,3)
!
! Set values for A in band Hermitian form, and B
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
! B = (3.0+3.0i 4.0+0.0i 29.0-9.0i)
! (5.0-5.0i 15.0-10.0i -36.0-17.0i)
! (5.0+4.0i -12.0-56.0i -15.0-24.0i)
! (9.0+7.0i -12.0+10.0i -23.0-15.0i)
! (-22.0+1.0i 3.0-1.0i -23.0-28.0i)
!
DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0),&
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
DATA B/(3.0,3.0), (5.0,-5.0), (5.0,4.0), (9.0,7.0), (-22.0,1.0),&
 (4.0,0.0), (15.0,-10.0), (-12.0,-56.0), (-12.0,10.0),&
 (3.0,-1.0), (29.0,-9.0), (-36.0,-17.0), (-15.0,-24.0),&
 (-23.0,-15.0), (-23.0,-28.0)/
!
! Factor the matrix A
CALL LFTQH (A, NCODA, FACT)
!
! Compute the solutions
DO 10 I=1, 3
 CALL LFSQH (FACT, NCODA, B(:,I), X(:,I))
10 CONTINUE
!
! Print solutions
CALL WRCRN ('X', X)
END

```

## Output

```

 X
 1 2 3
1 (1.00, 0.00) (3.00, -1.00) (11.00, -1.00)
2 (1.00, -2.00) (2.00, 0.00) (-7.00, 0.00)
3 (2.00, 0.00) (-1.00, -6.00) (-2.00, -3.00)
4 (2.00, 3.00) (2.00, 1.00) (-2.00, -3.00)
5 (-3.00, 0.00) (0.00, 0.00) (-2.00, -3.00)

```

---

## LFIQH

Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode.

## Required Arguments

- A** — Complex  $NCODA + 1$  by  $N$  array containing the  $N$  by  $N$  positive definite band Hermitian coefficient matrix in band Hermitian storage mode. (Input)
- NCODA** — Number of upper or lower codiagonals of **A**. (Input)
- FACT** — Complex  $NCODA + 1$  by  $N$  array containing the  $R^H R$  factorization of the matrix **A** as output from routine `LFCQH/DLFCQH` or `LFTQH/DLFTQH`. (Input)
- B** — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)
- X** — Complex vector of length  $N$  containing the solution to the linear system. (Output)
- RES** — Complex vector of length  $N$  containing the residual vector at the improved solution. (Output)

## Optional Arguments

- N** — Number of equations. (Input)  
Default:  $N = \text{size}(\mathbf{A}, 2)$ .
- LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .
- LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(\mathbf{FACT}, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LFIQH (A, NCODA, FACT, B, X, RES [, ...])`

Specific: The specific interface names are `S_LFIQH` and `D_LFIQH`.

## FORTRAN 77 Interface

Single: `CALL LFIQH (N, A, LDA, NCODA, FACT, LDFACT, B, X, RES)`

Double: The double precision name is `DLFIQH`.

## Description

Routine `LFIQH` computes the solution for a system of linear algebraic equations having a complex Hermitian positive definite band coefficient matrix. To compute the solution, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either IMSL routine `LFCQH` or `LFTQH`.  $R$  is an upper triangular band matrix.

The solution to  $Ax = b$  is found by solving the triangular systems  $R^H y = b$  and  $Rx = y$ .

LFSQH and LFIQH both solve a linear system given its  $R^H R$  factorization. LFIQH generally takes more time and produces a more accurate answer than LFSQH. Each iteration of the iterative refinement algorithm used by LFIQH calls LFSQH.

## Comments

Informational error

Type Code

4 1 The factored matrix has a diagonal element close to zero.

## Example

A set of linear systems is solved successively. The right-hand side vector is perturbed after solving the system each of the first two times by adding  $(1 + i)/2$  to the second element.

```

USE IMSL_LIBRARIES
!
! Declare variables
INTEGER LDA, LDFACT, N, NCODA
PARAMETER (LDA=2, LDFACT=2, N=5, NCODA=1)
REAL RCOND
COMPLEX A(LDA,N), B(N), FACT(LDFACT,N), RES(N,3), X(N,3)
!
! Set values for A in band Hermitian form, and B
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
! B = (3.0+3.0i 5.0-5.0i 5.0+4.0i 9.0+7.0i -22.0+1.0i)
!
DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
DATA B/(3.0,3.0), (5.0,-5.0), (5.0,4.0), (9.0,7.0), (-22.0,1.0)/
!
! Factor the matrix A
CALL LFCQH (A, NCODA, FACT, RCOND=RCOND)
!
! Print the estimated condition number
CALL UMACH (2, NOUT)
WRITE (NOUT, 99999) RCOND, 1.0E0/RCOND
!
! Compute the solutions
DO 10 I=1, 3
 CALL LFIQH (A, NCODA, FACT, B, X(:,I), RES(:,I))
 B(2) = B(2) + (0.5E0, 0.5E0)
10 CONTINUE
!
! Print solutions
CALL WRCRN ('X', X)
CALL WRCRN ('RES', RES)
99999 FORMAT (' RCOND = ', F5.3, '/', ' L1 Condition number = ', F6.3)
END

```

## Output

```

 X
 1 2 3
1 (1.00, 0.00) (3.00, -1.00) (11.00, -1.00)
2 (1.00, -2.00) (2.00, 0.00) (-7.00, 0.00)
3 (2.00, 0.00) (-1.00, -6.00) (-2.00, -3.00)
4 (2.00, 3.00) (2.00, 1.00) (-2.00, -3.00)
5 (-3.00, 0.00) (0.00, 0.00) (-2.00, -3.00)
```

---

## LFDQH

Computes the determinant of a complex Hermitian positive definite matrix given the  $R^H R$  Cholesky factorization in band Hermitian storage mode.

### Required Arguments

**FACT** — Complex  $NCODA + 1$  by  $N$  array containing the  $R^H R$  factorization of the Hermitian positive definite band matrix  $A$ . (Input)

**FACT** is obtained as output from routine `LFCQH/DLFCQH` or `LFTQH/DLFTQH`.

**NCODA** — Number of upper or lower codiagonals of  $A$ . (Input)

**DET1** — Scalar containing the mantissa of the determinant. (Output)

The value **DET1** is normalized so that  $1.0 \leq |\text{DET1}| < 10.0$  or **DET1** = 0.0.

**DET2** — Scalar containing the exponent of the determinant. (Output)

The determinant is returned in the form  $\det(A) = \text{DET1} * 10^{\text{DET2}}$ .

### Optional Arguments

**N** — Number of equations. (Input)

Default:  $N = \text{size}(\text{FACT}, 2)$ .

**LDFACT** — Leading dimension of **FACT** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $\text{LDFACT} = \text{size}(\text{FACT}, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL LFDQH (FACT, NCODA, DET1, DET2 [, ...])`

Specific: The specific interface names are `S_LFDQH` and `D_LFDQH`.

### FORTRAN 77 Interface

Single: `CALL LFDQH (N, FACT, LDFACT, NCODA, DET1, DET2)`

Double: The double precision name is `DLFDQH`.

## Description

Routine `LFDQH` computes the determinant of a complex Hermitian positive definite band coefficient matrix. To compute the determinant, the coefficient matrix must first undergo an  $R^H R$  factorization. This may be done by calling either `LFCQH` or `LFTQH`. The formula  $\det A = \det R^H \det R = (\det R)^2$  is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det R = \prod_{i=1}^N R_{ii}$$

`LFDQH` is based on the LINPACK routine `CPBDI`; see Dongarra et al. (1979).

## Example

The determinant is computed for a  $5 \times 5$  complex Hermitian positive definite band matrix with one codiagonal.

```
 USE LFDQH_INT
 USE LFTQH_INT
 USE UMACH_INT
!
! Declare variables
 INTEGER LDA, LDFACT, N, NCODA, NOUT
 PARAMETER (LDA=2, N=5, LDFACT=2, NCODA=1)
 REAL DET1, DET2
 COMPLEX A(LDA,N), FACT(LDFACT,N)
!
! Set values for A in band Hermitian form
!
! A = (0.0+0.0i -1.0+1.0i 1.0+2.0i 0.0+4.0i 1.0+1.0i)
! (2.0+0.0i 4.0+0.0i 10.0+0.0i 6.0+0.0i 9.0+0.0i)
!
 DATA A/(0.0,0.0), (2.0,0.0), (-1.0,1.0), (4.0, 0.0), (1.0,2.0), &
 (10.0,0.0), (0.0,4.0), (6.0,0.0), (1.0,1.0), (9.0,0.0)/
!
! Factor the matrix
 CALL LFTQH (A, NCODA, FACT)
!
! Compute the determinant
 CALL LFDQH (FACT, NCODA, DET1, DET2)
!
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) DET1, DET2
!
99999 FORMAT (' The determinant of A is ',F6.3,' * 10**',F2.0)
 END
```

## Output

The determinant of A is 1.736 \* 10\*\*3.

---

# LSLXG

Solves a sparse system of linear algebraic equations by Gaussian elimination.

## Required Arguments

- A** — Vector of length **NZ** containing the nonzero coefficients of the linear system. (Input)
- IROW** — Vector of length **NZ** containing the row numbers of the corresponding elements in **A**. (Input)
- JCOL** — Vector of length **NZ** containing the column numbers of the corresponding elements in **A**. (Input)
- B** — Vector of length **N** containing the right-hand side of the linear system. (Input)
- X** — Vector of length **N** containing the solution to the linear system. (Output)

## Optional Arguments

- N** — Number of equations. (Input)  
Default:  $N = \text{size}(B,1)$ .
- NZ** — The number of nonzero coefficients in the linear system. (Input)  
Default:  $NZ = \text{size}(A,1)$ .
- IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means the system  $Ax = b$  is solved.  
 $IPATH = 2$  means the system  $A^T x = b$  is solved.  
Default:  $IPATH = 1$ .
- IPARAM** — Parameter vector of length 6. (Input/Output)  
Set  $IPARAM(1)$  to zero for default values of **IPARAM** and **RPARAM**.  
Default:  $IPARAM(1) = 0$ .  
See Comment 3.
- RPARAM** — Parameter vector of length 5. (Input/Output)  
See Comment 3.

## FORTRAN 90 Interface

- Generic:    `CALL LSLXG (A, IROW, JCOL, B, X [, ...])`
- Specific:    The specific interface names are `S_LSLXG` and `D_LSLXG`.

## FORTRAN 77 Interface

- Single:    `CALL LSLXG (N, NZ, A, IROW, JCOL, B, IPATH, IPARAM, RPARAM, X)`
- Double:    The double precision name is `DLSLXG`.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a  $n \times n$  sparse matrix. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array  $a$  contains all the nonzeros in  $A$ . Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column numbers for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

with all other entries in  $A$  zero.

The routine `L2LXG` solves a system of linear algebraic equations having a real sparse coefficient matrix. It first uses the routine `LFTXG` to perform an  $LU$  factorization of the coefficient matrix. The solution of the linear system is then found using `LFSXG`.

The routine `LFTXG` by default uses a *symmetric Markowitz strategy* (Crowe et al. 1990) to choose pivots that most likely would reduce fill-ins while maintaining numerical stability. Different strategies are also provided as options for row oriented or column oriented problems. The algorithm can be expressed as

$$P A Q = L U$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively.

Finally, the solution  $x$  is obtained by the following calculations:

$$1) Lz = Pb$$

$$2) Uy = z$$

$$3) x = Qy$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LXG/DL2LXG`. The reference is:

```
CALL L2LXG (N, NZ, A, IROW, JCOL, B, IPATH, IPARAM, RPARAM, X, WK, LWK, IWK,
LIWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length `LWK`.

**LWK** — The length of `WK`, `LWK` should be at least  $2N + \text{MAXNZ}$ .

**IWK** — Integer work vector of length `LIWK`.

**LIWK** — The length of `IWK`, `LIWK` should be at least  $17N + 4 * \text{MAXNZ}$ .

The workspace limit is determined by `MAXNZ`, where

$$\text{MAXNZ} = \text{MINO}(\text{LWK}-2\text{N}, \text{INT}(0.25(\text{LIWK}-17\text{N})))$$

2. Informational errors

Type Code

- |   |   |                                                             |
|---|---|-------------------------------------------------------------|
| 3 | 1 | The coefficient matrix is numerically singular.             |
| 3 | 2 | The growth factor is too large to continue.                 |
| 3 | 3 | The matrix is too ill-conditioned for iterative refinement. |

3. If the default parameters are desired for `LSLXG`, then set `IPARAM(1)` to zero and call the routine `LSLXG`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `LSLXG`.

```
CALL L4LXG (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `L4LXG` will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above.

---

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = The pivoting strategy

| <b><code>IPARAM(2)</code></b> | <b>Action</b>              |
|-------------------------------|----------------------------|
| 1                             | Markowitz row search       |
| 2                             | Markowitz column search    |
| 3                             | Symmetric Markowitz search |

Default: 3.

`IPARAM(3)` = The number of rows which have least numbers of nonzero elements that will be searched for a pivotal element.

Default: 3.

`IPARAM(4)` = The maximal number of nonzero elements in `A` at any stage of the Gaussian elimination. (Output)

`IPARAM(5)` = The workspace limit.

| <b><code>IPARAM(5)</code></b> | <b>Action</b>                                            |
|-------------------------------|----------------------------------------------------------|
| 0                             | Default limit, see Comment 1.                            |
| <i>integer</i>                | This integer value replaces the default workspace limit. |

When `L2LXG` is called, the values of `LWK` and `LIWK` are used instead of `IPARAM(5)`.  
Default: 0.

`IPARAM(6)` = Iterative refinement is done when this is nonzero.  
Default: 0.

***RPARAM*** — Real vector of length 5.

`RPARAM(1)` = The upper limit on the growth factor. The computation stops when the growth factor exceeds the limit.  
Default:  $10^{16}$

`RPARAM(2)` = The stability factor. The absolute value of the pivotal element must be bigger than the largest element in absolute value in its row divided by `RPARAM(2)`.  
Default: 10.0.

`RPARAM(3)` = Drop-tolerance. Any element in the lower triangular factor `L` will be removed if its absolute value becomes smaller than the drop-tolerance at any stage of the Gaussian elimination.  
Default: 0.0.

`RPARAM(4)` = The growth factor. It is calculated as the largest element in absolute value in `A` at any stage of the Gaussian elimination divided by the largest element in absolute value in the original `A` matrix. (Output)  
Large value of the growth factor indicates that an appreciable error in the computed solution is possible.

`RPARAM(5)` = The value of the smallest pivotal element in absolute value.  
(Output)

If double precision is required, then `DL4LXG` is called and `RPARAM` is declared double precision.

## Example

As an example consider the  $6 \times 6$  linear system:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 1 & 50 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let  $x^T = (1, 2, 3, 4, 5, 6)$  so that  $Ax = (10, 7, 45, 33, -34, 31)^T$ . The number of nonzeros in  $A$  is  $nz = 15$ . The sparse coordinate form for  $A$  is given by:

|      |   |    |    |    |    |    |    |    |    |   |    |    |    |    |    |
|------|---|----|----|----|----|----|----|----|----|---|----|----|----|----|----|
| irow | 6 | 2  | 3  | 2  | 4  | 4  | 5  | 5  | 5  | 5 | 1  | 6  | 6  | 2  | 4  |
| jcol | 6 | 2  | 3  | 3  | 4  | 5  | 1  | 6  | 4  | 4 | 1  | 1  | 2  | 4  | 1  |
| a    | 6 | 10 | 15 | -3 | 10 | -1 | -1 | -3 | -5 | 1 | 10 | -1 | -2 | -1 | -2 |

```

USE LSLXG_INT
USE WRRRN_INT
USE L4LXG_INT
INTEGER N, NZ
PARAMETER (N=6, NZ=15)
!
INTEGER IPARAM(6), IROW(NZ), JCOL(NZ)
REAL A(NZ), B(N), RPARAM(5), X(N)
!
DATA A/6., 10., 15., -3., 10., -1., -1., -3., -5., 1., 10., -1., &
 -2., -1., -2./
DATA B/10., 7., 45., 33., -34., 31./
DATA IROW/6, 2, 3, 2, 4, 4, 5, 5, 5, 5, 1, 6, 6, 2, 4/
DATA JCOL/6, 2, 3, 3, 4, 5, 1, 6, 4, 5, 1, 1, 2, 4, 1/
!
! Change a default parameter
CALL L4LXG (IPARAM, RPARAM)
IPARAM(5) = 203
!
! Solve for X
CALL LSLXG (A, IROW, JCOL, B, X, IPARAM=IPARAM)
!

```

```
CALL WRRRN (' x ', X, 1, N, 1)
END
```

## Output

```
 x
 1 2 3 4 5 6
1.000 2.000 3.000 4.000 5.000 6.000
```

---

## LFTXG

Computes the  $LU$  factorization of a real general sparse matrix..

### Required Arguments

**A** — Vector of length `NZ` containing the nonzero coefficients of the linear system. (Input)

**IROW** — Vector of length `NZ` containing the row numbers of the corresponding elements in **A**. (Input)

**JCOL** — Vector of length `NZ` containing the column numbers of the corresponding elements in **A**. (Input)

**NL** — The number of nonzero coefficients in the triangular matrix  $L$  excluding the diagonal elements. (Output)

**NFAC** — On input, the dimension of vector **FACT**. (Input/Output)  
On output, the number of nonzero coefficients in the triangular matrix  $L$  and  $U$ .

**FACT** — Vector of length `NFAC` containing the nonzero elements of  $L$  (excluding the diagonals) in the first `NL` locations and the nonzero elements of  $U$  in `NL + 1` to `NFAC` locations. (Output)

**IRFAC** — Vector of length `NFAC` containing the row numbers of the corresponding elements in **FACT**. (Output)

**JCFAC** — Vector of length `NFAC` containing the column numbers of the corresponding elements in **FACT**. (Output)

**IPVT** — Vector of length `N` containing the row pivoting information for the  $LU$  factorization. (Output)

**JPVT** — Vector of length `N` containing the column pivoting information for the  $LU$  factorization. (Output)

## Optional Arguments

***N*** — Number of equations. (Input)  
Default:  $N = \text{size}(\text{IPVT}, 1)$ .

***NZ*** — The number of nonzero coefficients in the linear system. (Input)  
Default:  $NZ = \text{size}(A, 1)$ .

***IPARAM*** — Parameter vector of length 6. (Input/Output)  
Set  $\text{IPARAM}(1)$  to zero for default values of  $\text{IPARAM}$  and  $\text{RPARAM}$ .  
Default:  $\text{IPARAM}(1) = 0$ .  
See Comment 3.

***RPARAM*** — Parameter vector of length 5. (Input/Output)  
See Comment 3.

## FORTRAN 90 Interface

Generic:    CALL LFTXG (A, IROW, JCOL, NL, NFAC, FACT, IRFAC, JCFAC, IPVT,  
                  JPVT [, ...])

Specific:   The specific interface names are `S_LFTXG` and `D_LFTXG`.

## FORTRAN 77 Interface

Single:     CALL LFTXG (N, NZ, A, IROW, JCOL, IPARAM, RPARAM, NFAC, NL, FACT,  
                  IRFAC, JCFAC, IPVT, JPVT)

Double:     The double precision name is `DLFTXG`.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a  $n \times n$  sparse matrix. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array `a` contains all the nonzeros in  $A$ . Let the number of nonzeros be `nz`. The two integer arrays `irow` and `jcol`, each of length `nz`, contain the row and column numbers for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

with all other entries in  $A$  zero.

The routine `LFTXG` performs an  $LU$  factorization of the coefficient matrix  $A$ . It by default uses a *symmetric Markowitz strategy* (Crowe et al. 1990) to choose pivots that most likely would reduce fillins while maintaining numerical stability. Different strategies are also provided as options for row oriented or column oriented problems. The algorithm can be expressed as

$$P A Q = LU$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively.

Finally, the solution  $x$  is obtained using `LFSXG` by the following calculations:

- 1)  $Lz = Pb$
- 2)  $Uy = z$
- 3)  $x = Qy$

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2TXG/DL2TXG`. The reference is:

```
CALL L2TXG (N, NZ, A, IROW, JCOL, IPARAM, RPARAM, NFAC, NL, FACT,
 IRFAC, JCFAC, IPVT, JPVT, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length `LWK`.

**LWK** — The length of `WK`, `LWK` should be at least `MAXNZ`.

**IWK** — Integer work vector of length `LIWK`.

**LIWK** — The length of `IWK`, `LIWK` should be at least  $15N + 4 * \text{MAXNZ}$ .

The workspace limit is determined by `MAXNZ`, where

```
MAXNZ = MIN0 (LWK, INT (0.25 (LIWK-15N)))
```

2. Informational errors

Type Code

3 1 The coefficient matrix is numerically singular.

3 2 The growth factor is too large to continue.

3. If the default parameters are desired for `LFTXG`, then set `IPARAM(1)` to zero and call the routine `LFTXG`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `LFTXG`.

```
CALL L4LXG (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `L4LXG` will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above.

---

The arguments are as follows:

**IPARAM** — Integer vector of length 6.

IPARAM(1) = Initialization flag.

IPARAM(2) = The pivoting strategy.

| <b>IPARAM(2)</b> | <b>Action</b>              |
|------------------|----------------------------|
| 1                | Markowitz row search       |
| 2                | Markowitz column search    |
| 3                | Symmetric Markowitz search |

Default: 3.

IPARAM(3) = The number of rows which have least numbers of nonzero elements that will be searched for a pivotal element.

Default: 3.

IPARAM(4) = The maximal number of nonzero elements in A at any stage of the Gaussian elimination. (Output)

IPARAM(5) = The workspace limit.

| <b>IPARAM(5)</b> | <b>Action</b>                                            |
|------------------|----------------------------------------------------------|
| 0                | Default limit, see Comment 1.                            |
| <i>integer</i>   | This integer value replaces the default workspace limit. |

When L2TXG is called, the values of LWK and LIWK are used instead of IPARAM(5).

IPARAM(6) = Not used in LFTXG.

**RPARAM** — Real vector of length 5.

RPARAM(1) = The upper limit on the growth factor. The computation stops when the growth factor exceeds the limit.  
Default: 10.

RPARAM(2) = The stability factor. The absolute value of the pivotal element must be bigger than the largest element in absolute value in its row divided by RPARAM(2).  
Default: 10.0.

RPARAM(3) = Drop-tolerance. Any element in the lower triangular factor L will be removed if its absolute value becomes smaller than the drop-tolerance at any stage of the Gaussian elimination.  
Default: 0.0.

RPARAM(4) = The growth factor. It is calculated as the largest element in absolute value in A at any stage of the Gaussian elimination divided by the largest element in absolute value in the original A matrix. (Output)  
 Large value of the growth factor indicates that an appreciable error in the computed solution is possible.

RPARAM(5) = The value of the smallest pivotal element in absolute value.  
 (Output)

If double precision is required, then DL4LXG is called and RPARAM is declared double precision.

### Example

As an example, consider the  $6 \times 6$  matrix of a linear system:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

The sparse coordinate form for A is given by:

|      |   |    |    |    |    |    |    |    |    |   |    |    |    |    |    |
|------|---|----|----|----|----|----|----|----|----|---|----|----|----|----|----|
| irow | 6 | 2  | 3  | 2  | 4  | 4  | 5  | 5  | 5  | 5 | 1  | 6  | 6  | 2  | 4  |
| jcol | 6 | 2  | 3  | 3  | 4  | 5  | 1  | 6  | 4  | 5 | 1  | 1  | 2  | 4  | 1  |
| a    | 6 | 10 | 15 | -3 | 10 | -1 | -1 | -3 | -5 | 1 | 10 | -1 | -2 | -1 | -2 |

```

USE LFTXG_INT
USE WRRRN_INT
USE WRIRN_INT
INTEGER N, NZ
PARAMETER (N=6, NZ=15)
INTEGER IROW(NZ), JCOL(NZ), NFAC, NL, &
 IRFAC(3*NZ), JCFAC(3*NZ), IPVT(N), JPVTV(N)
REAL A(NZ), FACT(3*NZ)
!
DATA A/6., 10., 15., -3., 10., -1., -1., -3., -5., 1., 10., -1., &
 -2., -1., -2./
DATA IROW/6, 2, 3, 2, 4, 4, 5, 5, 5, 5, 1, 6, 6, 2, 4/
DATA JCOL/6, 2, 3, 3, 4, 5, 1, 6, 4, 5, 1, 1, 2, 4, 1/
!
NFAC = 3*NZ
!
 Use default options
CALL LFTXG (A, IROW, JCOL, NL, NFAC, FACT, IRFAC, JCFAC, IPVT, JPVTV)
!
```

```

CALL WRRRN (' fact ', FACT, 1, NFAC, 1)
CALL WRIRN (' irfac ', IRFAC, 1, NFAC, 1)
CALL WRIRN (' jcfac ', JCFAC, 1, NFAC, 1)
CALL WRIRN (' p ', IPVT, 1, N, 1)
CALL WRIRN (' q ', JPV, 1, N, 1)

```

!

END

## Output

```

 fact
 1 2 3 4 5 6 7 8 9 10
-0.10 -5.00 -0.20 -0.10 -0.10 -1.00 -0.20 4.90 -5.10 1.00
 11 12 13 14 15 16
-1.00 30.00 6.00 -2.00 10.00 15.00

 irfac
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
 3 4 4 5 5 6 6 6 5 5 4 4 3 3 2 1

 jcfac
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
 2 3 1 4 2 5 2 6 6 5 6 4 4 3 2 1

 p
 1 2 3 4 5 6
 3 1 6 2 5 4

 q
 1 2 3 4 5 6
 3 1 2 6 5 4

```

---

## LFSXG

Solves a sparse system of linear equations given the *LU* factorization of the coefficient matrix..

### Required Arguments

**NFAC** — The number of nonzero coefficients in *FACT* as output from subroutine LFTXG/DLFTXG. (Input)

**NL** — The number of nonzero coefficients in the triangular matrix *L* excluding the diagonal elements as output from subroutine LFTXG/DLFTXG. (Input)

**FACT** — Vector of length *NFAC* containing the nonzero elements of *L* (excluding the diagonals) in the first *NL* locations and the nonzero elements of *U* in *NL* + 1 to *NFAC* locations as output from subroutine LFTXG/DLFTXG. (Input)

**IRFAC** — Vector of length *NFAC* containing the row numbers of the corresponding elements in *FACT* as output from subroutine LFTXG/DLFTXG. (Input)

**JCFAC** — Vector of length `NFAC` containing the column numbers of the corresponding elements in `FACT` as output from subroutine `LFTXG/DLFTXG`. (Input)

**IPVT** — Vector of length `N` containing the row pivoting information for the *LU* factorization as output from subroutine `LFTXG/DLFTXG`. (Input)

**JPVT** — Vector of length `N` containing the column pivoting information for the *LU* factorization as output from subroutine `LFTXG/DLFTXG`. (Input)

**B** — Vector of length `N` containing the right-hand side of the linear system. (Input)

**X** — Vector of length `N` containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)

Default: `N = size (B,1)`.

**IPATH** — Path indicator. (Input)

`IPATH = 1` means the system  $Ax = B$  is solved.

`IPATH = 2` means the system  $A^T x = B$  is solved.

Default: `IPATH = 1`.

### FORTRAN 90 Interface

Generic: `CALL LFSXG (NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT, B, X [, ...])`

Specific: The specific interface names are `S_LFSXG` and `D_LFSXG`.

### FORTRAN 77 Interface

Single: `CALL LFSXG (N, NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT, B, IPATH, X)`

Double: The double precision name is `DLFSXG`.

### Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a  $n \times n$  sparse matrix. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array `a` contains all the nonzeros in  $A$ . Let the number of nonzeros be `nz`. The two integer arrays `irow` and `jcol`, each of length `nz`, contain the row and column numbers for these entries in  $A$ . That is

$$A_{\text{irow}(i) \text{jcol}(i)} = a(i), \quad i = 1, \dots, \text{nz}$$

with all other entries in  $A$  zero. The routine `LFSXG` computes the solution of the linear equation given its  $LU$  factorization. The factorization is performed by calling `LFTXG`. The solution of the linear system is then found by the forward and backward substitution. The algorithm can be expressed as

$$P A Q = L U$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively. Finally, the solution  $x$  is obtained by the following calculations:

- 1)  $Lz = Pb$
- 2)  $Uy = z$
- 3)  $x = Qy$

For more details, see Crowe et al. (1990).

### Example

As an example, consider the  $6 \times 6$  linear system:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let

$$x_1^T = (1, 2, 3, 4, 5, 6)$$

so that  $Ax_1 = (10, 7, 45, 33, -34, 31)^T$ , and

$$x_2^T = (6, 5, 4, 3, 2, 1)$$

so that  $Ax_2 = (60, 35, 60, 16, -22, 10)^T$ . The sparse coordinate form for  $A$  is given by:

```

irow 6 2 3 2 4 4 5 5 5 5 1 6 6 2 4
jcol 6 2 3 3 4 5 1 6 4 5 1 1 2 4 1
a 6 10 15 -3 10 -1 -1 -3 -5 1 10 -1 -2 -1 -2

```

```

USE LFSXG_INT
USE WRRRL_INT
USE LFTXG_INT
INTEGER N, NZ
PARAMETER (N=6, NZ=15)
INTEGER IPATH, IROW(NZ), JCOL(NZ), NFAC, &
NL, IRFAC(3*NZ), JCFAC(3*NZ), IPVT(N), JPVTV(N)

```

```

REAL X(N), A(NZ), B(N,2), FACT(3*NZ)
CHARACTER TITLE(2)*2, RLABEL(1)*4, CLABEL(1)*6
DATA RLABEL(1)/'NONE'/, CLABEL(1)/'NUMBER'/
!
DATA A/6., 10., 15., -3., 10., -1., -1., -3., -5., 1., 10., -1., &
 -2., -1., -2./
DATA B/10., 7., 45., 33., -34., 31., &
 60., 35., 60., 16., -22., -10./
DATA IROW/6, 2, 3, 2, 4, 4, 5, 5, 5, 5, 1, 6, 6, 2, 4/
DATA JCOL/6, 2, 3, 3, 4, 5, 1, 6, 4, 5, 1, 1, 2, 4, 1/
DATA TITLE/'x1', 'x2'/
!
NFAC = 3*NZ
!
! Perform LU factorization
CALL LFTXG (A, IROW, JCOL, NL, NFAC, FACT, IRFAC, JCFAC, IPVT, JPVT)
!
DO 10 I = 1, 2
!
! Solve A * X(i) = B(i)
CALL LFSXG (NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT, B(:,I), X)
!
CALL WRRRL (TITLE(I), X, RLABEL, CLABEL, 1, N, 1)
10 CONTINUE
END

```

## Output

```

 x1
 1 2 3 4 5 6
1.0 2.0 3.0 4.0 5.0 6.0

 x2
 1 2 3 4 5 6
6.0 5.0 4.0 3.0 2.0 1.0

```

---

## LSLZG

Solves a complex sparse system of linear equations by Gaussian elimination.

### Required Arguments

**A** — Complex vector of length *NZ* containing the nonzero coefficients of the linear system. (Input)

**IROW** — Vector of length *NZ* containing the row numbers of the corresponding elements in **A**. (Input)

**JCOL** — Vector of length *NZ* containing the column numbers of the corresponding elements in **A**. (Input)

**B** — Complex vector of length *N* containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length *N* containing the solution to the linear system. (Output)

## Optional Arguments

*N* — Number of equations. (Input)  
Default:  $N = \text{size}(B,1)$ .

*NZ* — The number of nonzero coefficients in the linear system. (Input)  
Default:  $NZ = \text{size}(A,1)$ .

*IPATH* — Path indicator. (Input)  
 $IPATH = 1$  means the system  $Ax = b$  is solved.  
 $IPATH = 2$  means the system  $A^H x = b$  is solved.  
Default:  $IPATH = 1$ .

*IPARAM* — Parameter vector of length 6. (Input/Output)  
Set  $IPARAM(1)$  to zero for default values of *IPARAM* and *RPARAM*. See Comment 3.  
Default:  $IPARAM = 0$ .

*RPARAM* — Parameter vector of length 5. (Input/Output)  
See Comment 3

## FORTRAN 90 Interface

Generic:    `CALL LSLZG (A, IROW, JCOL, B, X [, ...])`

Specific:    The specific interface names are `S_LSLZG` and `D_LSLZG`.

## FORTRAN 77 Interface

Single:    `CALL LSLZG (N, NZ, A, IROW, JCOL, B, IPATH, IPARAM, RPARAM, X)`

Double:    The double precision name is `DLSLZG`.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a  $n \times n$  complex sparse matrix. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array `a` contains all the nonzeros in  $A$ . Let the number of nonzeros be `nz`. The two integer arrays `irow` and `jcol`, each of length `nz`, contain the row and column numbers for these entries in  $A$ . That is

$$A_{irow(i) icol(i)} = a(i), \quad i = 1, \dots, nz$$

with all other entries in  $A$  zero.

The subroutine `LSLZG` solves a system of linear algebraic equations having a complex sparse coefficient matrix. It first uses the routine `LFTZG` to perform an  $LU$  factorization of the coefficient matrix. The solution of the linear system is then found using `LFSZG`. The routine `LFTZG` by default

uses a *symmetric Markowitz strategy* (Crowe et al. 1990) to choose pivots that most likely would reduce fill-ins while maintaining numerical stability. Different strategies are also provided as options for row oriented or column oriented problems. The algorithm can be expressed as

$$P A Q = L U$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively. Finally, the solution  $x$  is obtained by the following calculations:

$$1) Lz = Pb$$

$$2) Uy = z$$

$$3) x = Qy$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2LZG/DL2LZG. The reference is:

```
CALL L2LZG (N, NZ, A, IROW, JCOL, B, IPATH, IPARAM, RPARAM, X, WK,
 LWK, IWK, LIWK)
```

The additional arguments are as follows:

**WK** — Complex work vector of length LWK.

**LWK** — The length of WK, LWK should be at least  $2N + \text{MAXNZ}$ .

**IWK** — Integer work vector of length LIWK.

**LIWK** — The length of IWK, LIWK should be at least  $17N + 4 * \text{MAXNZ}$ .

The workspace limit is determined by MAXNZ, where

$$\text{MAXNZ} = \text{MIN0}(\text{LWK} - 2N, \text{INT}(0.25(\text{LIWK} - 17N)))$$

2. Informational errors

| Type | Code |                                                             |
|------|------|-------------------------------------------------------------|
| 3    | 1    | The coefficient matrix is numerically singular.             |
| 3    | 2    | The growth factor is too large to continue.                 |
| 3    | 3    | The matrix is too ill-conditioned for iterative refinement. |

3. If the default parameters are desired for LSLZG, then set IPARAM(1) to zero and call the routine LSLZG. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling LSLZG.

```
CALL L4LZG (IPARAM, RPARAM)
```

Set nondefault values for desired *IPARAM*, *RPARAM* elements.

---

Note that the call to *L4LZG* will set *IPARAM* and *RPARAM* to their default values, so only nondefault values need to be set above. The arguments are as follows:

---

***IPARAM*** — Integer vector of length 6.

*IPARAM*(1) = Initialization flag.

*IPARAM*(2) = The pivoting strategy.

| <b><i>IPARAM</i>(2)</b> | <b>Action</b>              |
|-------------------------|----------------------------|
| 1                       | Markowitz row search       |
| 2                       | Markowitz column search    |
| 3                       | Symmetric Markowitz search |

Default: 3.

*IPARAM*(3) = The number of rows which have least numbers of nonzero elements that will be searched for a pivotal element.  
Default: 3.

*IPARAM*(4) = The maximal number of nonzero elements in *A* at any stage of the Gaussian elimination. (Output)

*IPARAM*(5) = The workspace limit.

| <b><i>IPARAM</i>(5)</b> | <b>Action</b>                                            |
|-------------------------|----------------------------------------------------------|
| 0                       | Default limit, see Comment 1.                            |
| <i>integer</i>          | This integer value replaces the default workspace limit. |

When *L2LZG* is called, the values of *LWK* and *LIWK* are used instead of *IPARAM*(5).

Default: 0.

*IPARAM*(6) = Iterative refinement is done when this is nonzero.  
Default: 0.

***RPARAM*** — Real vector of length 5.

*RPARAM*(1) = The upper limit on the growth factor. The computation stops when the growth factor exceeds the limit.  
Default: 10.

*RPARAM*(2) = The stability factor. The absolute value of the pivotal element must be bigger than the largest element in absolute value in its row

divided by `RPARAM(2)`.  
 Default: 10.0.

`RPARAM(3)` = Drop-tolerance. Any element in `A` will be removed if its absolute value becomes smaller than the drop-tolerance at any stage of the Gaussian elimination.  
 Default: 0.0.

`RPARAM(4)` = The growth factor. It is calculated as the largest element in absolute value in `A` at any stage of the Gaussian elimination divided by the largest element in absolute value in the original `A` matrix. (Output)  
 Large value of the growth factor indicates that an appreciable error in the computed solution is possible.

`RPARAM(5)` = The value of the smallest pivotal element in absolute value.  
 (Output)

If double precision is required, then `DL4LZG` is called and `RPARAM` is declared double precision.

### Example

As an example, consider the  $6 \times 6$  linear system:

$$A = \begin{bmatrix} 10+7i & 0 & 0 & 0 & 0 & 0 \\ 0 & 3+2i & -3+0i & -1+2i & 0 & 0 \\ 0 & 0 & 4+2i & 0 & 0 & 0 \\ -2-4i & 0 & 0 & 1+6i & -1+3i & 0 \\ -5+4i & 0 & 0 & -5+0i & 12+2i & -7+7i \\ -1+12i & -2+8i & 0 & 0 & 0 & 3+7i \end{bmatrix}$$

Let

$$x^T = (1+i, 2+2i, 3+3i, 4+4i, 5+5i, 6+6i)$$

so that

$$Ax = (3+17i, -19+5i, 6+18i, -38+32i, -63+49i, -57+83i)^T$$

The number of nonzeros in `A` is `nz = 15`. The sparse coordinate form for `A` is given by:

```
irow 6 2 2 4 3 1 5 4 6 5 5 6 4 2 5
jcol 6 2 3 5 3 1 1 4 1 4 5 2 1 4 6
```

```
USE LSLZG_INT
USE WRCRN_INT
INTEGER N, NZ
PARAMETER (N=6, NZ=15)
```

```

!
INTEGER IROW(NZ), JCOL(NZ)
COMPLEX A(NZ), B(N), X(N)
!
DATA A/(3.0,7.0), (3.0,2.0), (-3.0,0.0), (-1.0,3.0), (4.0,2.0), &
 (10.0,7.0), (-5.0,4.0), (1.0,6.0), (-1.0,12.0), (-5.0,0.0), &
 (12.0,2.0), (-2.0,8.0), (-2.0,-4.0), (-1.0,2.0), (-7.0,7.0)/
DATA B/(3.0,17.0), (-19.0,5.0), (6.0,18.0), (-38.0,32.0), &
 (-63.0,49.0), (-57.0,83.0)/
DATA IROW/6, 2, 2, 4, 3, 1, 5, 4, 6, 5, 5, 6, 4, 2, 5/
DATA JCOL/6, 2, 3, 5, 3, 1, 1, 4, 1, 4, 5, 2, 1, 4, 6/
!
! Use default options
CALL LSLZG (A, IROW, JCOL, B, X)
!
CALL WRCRN ('X', X)
END

```

## Output

```

 X
1 (1.000, 1.000)
2 (2.000, 2.000)
3 (3.000, 3.000)
4 (4.000, 4.000)
5 (5.000, 5.000)
6 (6.000, 6.000)

```

---

## LFTZG

Computes the *LU* factorization of a complex general sparse matrix.

### Required Arguments

**A** — Complex vector of length *NZ* containing the nonzero coefficients of the linear system. (Input)

**IROW** — Vector of length *NZ* containing the row numbers of the corresponding elements in *A*. (Input)

**JCOL** — Vector of length *NZ* containing the column numbers of the corresponding elements in *A*. (Input)

**NFAC** — On input, the dimension of vector *FACT*. (Input/Output)  
On output, the number of nonzero coefficients in the triangular matrix *L* and *U*.

**NL** — The number of nonzero coefficients in the triangular matrix *L* excluding the diagonal elements. (Output)

**FACT** — Complex vector of length *NFAC* containing the nonzero elements of *L* (excluding the diagonals) in the first *NL* locations and the nonzero elements of *U* in *NL* + 1 to *NFAC* locations. (Output)

**IRFAC** — Vector of length *NFAC* containing the row numbers of the corresponding elements in *FACT*. (Output)

**JCFAC** — Vector of length *NFAC* containing the column numbers of the corresponding elements in *FACT*. (Output)

**IPVT** — Vector of length *N* containing the row pivoting information for the *LU* factorization. (Output)

**JPVT** — Vector of length *N* containing the column pivoting information for the *LU* factorization. (Output)

### Optional Arguments

***N*** — Number of equations. (Input)  
Default: *N* = size (*IPVT*,1).

***NZ*** — The number of nonzero coefficients in the linear system. (Input)  
Default: *NZ* = size (*A*,1).

***IPARAM*** — Parameter vector of length 6. (Input/Output)  
Set *IPARAM*(1) to zero for default values of *IPARAM* and *RPARAM*. See Comment 3.  
Default: *IPARAM* = 0.

***RPARAM*** — Parameter vector of length 5. (Input/Output)  
See Comment 3.

### FORTRAN 90 Interface

Generic:    CALL LFTZG (A, IROW, JCOL, NFAC, NL, FACT, IRFAC, JCFAC, IPVT,  
                  JPVT [, ...])

Specific:    The specific interface names are *S\_LFTZG* and *D\_LFTZG*.

### FORTRAN 77 Interface

Single:     CALL LFTZG (N, NZ, A, IROW, JCOL, IPARAM, RPARAM, NFAC, NL, FACT,  
                  IRFAC, JCFAC, IPVT, JPVT)

Double:     The double precision name is *DLFTZG*.

### Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a complex  $n \times n$  sparse matrix. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array  $a$  contains all the nonzeros in  $A$ . Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

with all other entries in  $A$  zero.

The routine `LFTZG` performs an  $LU$  factorization of the coefficient matrix  $A$ . It uses by default a *symmetric Markowitz strategy* (Crowe et al. 1990) to choose pivots that most likely would reduce fill-ins while maintaining numerical stability. Different strategies are also provided as options for row oriented or column oriented problems. The algorithm can be expressed as

$$P A Q = L U$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively.

Finally, the solution  $x$  is obtained using `LFSZG` by the following calculations:

- 1)  $Lz = Pb$
- 2)  $Uy = z$
- 3)  $x = Qy$

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2TZG/DL2TZG`. The reference is:

```
CALL L2TZG (N, NZ, A, IROW, JCOL, IPARAM, RPARAM, NFAC, NL, FACT,
 IRFAC, JCFAC, IPVT, JPVT, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

**WK** — Complex work vector of length `LWK`.

**LWK** — The length of `WK`, `LWK` should be at least `MAXNZ`.

**IWK** — Integer work vector of length `LIWK`.

**LIWK** — The length of `IWK`, `LIWK` should be at least  $15N + 4 * \text{MAXNZ}$ .

The workspace limit is determined by `MAXNZ`, where

$$\text{MAXNZ} = \text{MIN0}(\text{LWK}, \text{INT}(0.25(\text{LIWK} - 15N)))$$

2. Informational errors

Type Code

3 1 The coefficient matrix is numerically singular.

3      2            The growth factor is too large to continue.

3. If the default parameters are desired for `LFTZG`, then set `IPARAM(1)` to zero and call the routine `LFTZG`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `LFTZG`:

```
CALL L4LZG (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `L4LZG` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above. The arguments are as follows:

---

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = The pivoting strategy.

| <b><i>IPARAM(2)</i></b> | <b>Action</b>              |
|-------------------------|----------------------------|
| 1                       | Markowitz row search       |
| 2                       | Markowitz column search    |
| 3                       | Symmetric Markowitz search |

`IPARAM(3)` = The number of rows which have least numbers of nonzero elements that will be searched for a pivotal element.

Default: 3.

`IPARAM(4)` = The maximal number of nonzero elements in A at any stage of the Gaussian elimination. (Output)

`IPARAM(5)` = The workspace limit.

| <b><i>IPARAM(5)</i></b> | <b>Action</b>                                                                                                                                                                                 |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0                       | Default limit, see Comment 1.                                                                                                                                                                 |
| <i>integer</i>          | This integer value replaces the default workspace limit. When <code>L2TZG</code> is called, the values of <code>LWK</code> and <code>LIWK</code> are used instead of <code>IPARAM(5)</code> . |

Default: 0.

`IPARAM(6)` = Not used in `LFTZG`.

***RPARAM*** — Real vector of length 5.

RPARAM(1) = The upper limit on the growth factor. The computation stops when the growth factor exceeds the limit.

Default: 10.

RPARAM(2) = The stability factor. The absolute value of the pivotal element must be bigger than the largest element in absolute value in its row divided by RPARAM(2).

Default: 10.0.

RPARAM(3) = Drop-tolerance. Any element in the lower triangular factor  $L$  will be removed if its absolute value becomes smaller than the drop-tolerance at any stage of the Gaussian elimination.

Default: 0.0.

RPARAM(4) = The growth factor. It is calculated as the largest element in absolute value in  $A$  at any stage of the Gaussian elimination divided by the largest element in absolute value in the original  $A$  matrix. (Output) Large value of the growth factor indicates that an appreciable error in the computed solution is possible.

RPARAM(5) = The value of the smallest pivotal element in absolute value. (Output)

If double precision is required, then DL4LZG is called and RPARAM is declared double precision.

## Example

As an example, the following  $6 \times 6$  matrix is factorized, and the outcome is printed:

$$A = \begin{bmatrix} 10+7i & 0 & 0 & 0 & 0 & 0 \\ 0 & 3+2i & -3+0i & -1+2i & 0 & 0 \\ 0 & 0 & 4+2i & 0 & 0 & 0 \\ -2-4i & 0 & 0 & 1+6i & -1+3i & 0 \\ -5+4i & 0 & 0 & -5+0i & 12+2i & -7+7i \\ -1+12i & -2+8i & 0 & 0 & 0 & 3+7i \end{bmatrix}$$

The sparse coordinate form for  $A$  is given by:

```

irow 6 2 2 4 3 1 5 4 6 5 5 6 4 2 5
jcol 6 2 3 5 3 1 1 4 1 4 5 2 1 4 6

```

```

USE LFTZG_INT
USE WRCRN_INT
USE WRIRN_INT
INTEGER N, NFAC, NZ

```

```

PARAMETER (N=6, NZ=15)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IPVT(N), IRFAC(45), IROW(NZ), JCFAC(45), &
 JCOL(NZ), JPVT(N), NL
COMPLEX A(NZ), FAC(45)
!
DATA A/(3.0,7.0), (3.0,2.0), (-3.0,0.0), (-1.0,3.0), (4.0,2.0), &
 (10.0,7.0), (-5.0,4.0), (1.0,6.0), (-1.0,12.0), (-5.0,0.0), &
 (12.0,2.0), (-2.0,8.0), (-2.0,-4.0), (-1.0,2.0), (-7.0,7.0)/
DATA IROW/6, 2, 2, 4, 3, 1, 5, 4, 6, 5, 5, 6, 4, 2, 5/
DATA JCOL/6, 2, 3, 5, 3, 1, 1, 4, 1, 4, 5, 2, 1, 4, 6/
DATA NFAC/45/
!
! Use default options
CALL LFTZG (A, IROW, JCOL, NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT)
!
CALL WRCRN ('fact',FACT, 1, NFAC, 1)
CALL WRIRN (' irfac ',IRFAC, 1, NFAC, 1)
CALL WRIRN (' jcfac ',JCFAC, 1, NFAC, 1)
CALL WRIRN (' p ',IPVT, 1, N, 1)
CALL WRIRN (' q ',JPVT, 1, N, 1)
!
END

```

## Output

```

fact
1 (0.50, 0.85)
2 (0.15, -0.41)
3 (-0.60, 0.30)
4 (2.23, -1.97)
5 (-0.15, 0.50)
6 (-0.04, 0.26)
7 (-0.32, -0.17)
8 (-0.92, 7.46)
9 (-6.71, -6.42)
10 (12.00, 2.00)
11 (-1.00, 2.00)
12 (-3.32, 0.21)
13 (3.00, 7.00)
14 (-2.00, 8.00)
15 (10.00, 7.00)
16 (4.00, 2.00)

irfac
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
3 4 4 5 5 6 6 6 5 5 4 4 3 3 2 1

jcfac
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 1 4 2 5 2 6 6 5 6 4 4 3 2 1

p
1 2 3 4 5 6

```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 3 | 1 | 6 | 2 | 5 | 4 |
|   |   |   | q |   |   |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 3 | 1 | 2 | 6 | 5 | 4 |

---

## LFSZG

Solves a complex sparse system of linear equations given the  $LU$  factorization of the coefficient matrix.

### Required Arguments

**NFAC** — The number of nonzero coefficients in **FACT** as output from subroutine **LFTZG/DLFTZG**. (Input)

**NL** — The number of nonzero coefficients in the triangular matrix  $L$  excluding the diagonal elements as output from subroutine **LFTZG/DLFTZG**. (Input)

**FACT** — Complex vector of length **NFAC** containing the nonzero elements of  $L$  (excluding the diagonals) in the first **NL** locations and the nonzero elements of  $U$  in **NL+ 1** to **NFAC** locations as output from subroutine **LFTZG/DLFTZG**. (Input)

**IRFAC** — Vector of length **NFAC** containing the row numbers of the corresponding elements in **FACT** as output from subroutine **LFTZG/DLFTZG**. (Input)

**JCFAC** — Vector of length **NFAC** containing the column numbers of the corresponding elements in **FACT** as output from subroutine **LFTZG/DLFTZG**. (Input)

**IPVT** — Vector of length **N** containing the row pivoting information for the  $LU$  factorization as output from subroutine **LFTZG/DLFTZG**. (Input)

**JPVT** — Vector of length **N** containing the column pivoting information for the  $LU$  factorization as output from subroutine **LFTZG/DLFTZG**. (Input)

**B** — Complex vector of length **N** containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length **N** containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(B,1)$ .

**IPATH** — Path indicator. (Input)  
 $IPATH = 1$  means the system  $Ax = b$  is solved.  
 $IPATH = 2$  means the system  $A^H x = b$  is solved.  
Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic:     CALL LFSZG (NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT, B, X [, ...])

Specific:    The specific interface names are S\_LFSZG and D\_LFSZG.

## FORTRAN 77 Interface

Single:     CALL LFSZG (N, NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT, B, IPATH, X)

Double:     The double precision name is DLFSZG.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is a complex  $n \times n$  sparse matrix. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array  $a$  contains all the nonzeros in  $A$ . Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column numbers for these entries in  $A$ . That is

$$A_{irow(i)jcol(i)} = a(i), \quad i = 1, \dots, nz$$

with all other entries in  $A$  zero.

The routine LFSZG computes the solution of the linear equation given its  $LU$  factorization. The factorization is performed by calling LFTZG. The solution of the linear system is then found by the forward and backward substitution. The algorithm can be expressed as

$$P A Q = LU$$

where  $P$  and  $Q$  are the row and column permutation matrices determined by the Markowitz strategy (Duff et al. 1986), and  $L$  and  $U$  are lower and upper triangular matrices, respectively.

Finally, the solution  $x$  is obtained by the following calculations:

$$1) Lz = Pb$$

$$2) Uy = z$$

$$3) x = Qy$$

For more details, see Crowe et al. (1990).

## Example

As an example, consider the  $6 \times 6$  linear system:

$$A = \begin{bmatrix} 10+7i & 0 & 0 & 0 & 0 & 0 \\ 0 & 3+2i & -3+0i & -1+2i & 0 & 0 \\ 0 & 0 & 4+2i & 0 & 0 & 0 \\ -2-4i & 0 & 0 & 1+6i & -1+3i & 0 \\ -5+4i & 0 & 0 & -5+0i & 12+2i & -7+7i \\ -1+12i & -2+8i & 0 & 0 & 0 & 3+7i \end{bmatrix}$$

Let

$$x_1^T = (1+i, 2+2i, 3+3i, 4+4i, 5+5i, 6+6i)$$

so that

$$Ax_1 = (3 + 17i, -19 + 5i, 6 + 18i, -38 + 32i, -63 + 49i, -57 + 83i)^T$$

and

$$x_2^T = (6+6i, 5+5i, 4+4i, 3+3i, 2+2i, 1+i)$$

so that

$$Ax_2 = (18 + 102i, -16 + 16i, 8 + 24i, -11 - 11i, -63 + 7i, -132 + 106i)^T$$

The sparse coordinate form for  $A$  is given by:

|      |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| irow | 6 | 2 | 2 | 4 | 3 | 1 | 5 | 4 | 6 | 5 | 5 | 6 | 4 | 2 | 5 |
| jcol | 6 | 2 | 3 | 5 | 3 | 1 | 1 | 4 | 1 | 4 | 5 | 2 | 1 | 4 | 6 |

```

USE LFSZG_INT
USE WRCRN_INT
USE LFTZG_INT
INTEGER N, NZ
PARAMETER (N=6, NZ=15)
!
INTEGER IPATH, IPV(T(N), IRFAC(3*NZ), IROW(NZ), &
 JCFAC(3*NZ), JCOL(NZ), JPVT(N), NFAC, NL
COMPLEX A(NZ), B(N,2), FACT(3*NZ), X(N)
CHARACTER TITLE(2)*2
!
DATA A/(3.0,7.0), (3.0,2.0), (-3.0,0.0), (-1.0,3.0), (4.0,2.0), &
 (10.0,7.0), (-5.0,4.0), (1.0,6.0), (-1.0,12.0), (-5.0,0.0), &
 (12.0,2.0), (-2.0,8.0), (-2.0,-4.0), (-1.0,2.0), (-7.0,7.0)/
DATA B/(3.0,17.0), (-19.0,5.0), (6.0,18.0), (-38.0,32.0), &
 (-63.0,49.0), (-57.0,83.0), (18.0,102.0), (-16.0,16.0), &
 (8.0,24.0), (-11.0,-11.0), (-63.0,7.0), (-132.0,106.0)/
DATA IROW/6, 2, 2, 4, 3, 1, 5, 4, 6, 5, 5, 6, 4, 2, 5/
DATA JCOL/6, 2, 3, 5, 3, 1, 1, 4, 1, 4, 5, 2, 1, 4, 6/
DATA TITLE/'x1', 'x2'/
!
NFAC = 3*NZ
!
 Perform LU factorization
CALL LFTZG (A, IROW, JCOL, NFAC, NL, FACT, IRFAC, JCFAC, IPV(T, JPVT)

```

```

!
 IPATH = 1
 DO 10 I = 1,2
!
! Solve A * X(i) = B(i)
 CALL LFSZG (NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPV, &
 B(:,I), X)
 CALL WRCRN (TITLE(I), X)
10 CONTINUE
!
 END

```

## Output

```

 x1
1 (1.000, 1.000)
2 (2.000, 2.000)
3 (3.000, 3.000)
4 (4.000, 4.000)
5 (5.000, 5.000)
6 (6.000, 6.000)

```

```

 x2
1 (6.000, 6.000)
2 (5.000, 5.000)
3 (4.000, 4.000)
4 (3.000, 3.000)
5 (2.000, 2.000)
6 (1.000, 1.000)

```

---

## LSLXD

Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.

### Required Arguments

**A** — Vector of length *NZ* containing the nonzero coefficients in the lower triangle of the linear system. (Input)

The sparse matrix has nonzeros only in entries (*IROW*(*i*), *JCOL*(*i*)) for *i* = 1 to *NZ*, and at this location the sparse matrix has value *A*(*i*).

**IROW** — Vector of length *NZ* containing the row numbers of the corresponding elements in the lower triangle of *A*. (Input)

Note *IROW*(*i*) ≥ *JCOL*(*i*), since we are only indexing the lower triangle.

**JCOL** — Vector of length *NZ* containing the column numbers of the corresponding elements in the lower triangle of *A*. (Input)

**B** — Vector of length *N* containing the right-hand side of the linear system. (Input)

**X** — Vector of length *N* containing the solution to the linear system. (Output)

## Optional Arguments

*N* — Number of equations. (Input)  
Default:  $N = \text{size}(B,1)$ .

*NZ* — The number of nonzero coefficients in the lower triangle of the linear system. (Input)  
Default:  $NZ = \text{size}(A,1)$ .

*ITWKSP* — The total workspace needed. (Input)  
If the default is desired, set *ITWKSP* to zero.  
Default:  $ITWKSP = 0$ .

## FORTRAN 90 Interface

Generic:     CALL LSLXD (A, IROW, JCOL, B, X [, ...])

Specific:    The specific interface names are *S\_LSLXD* and *D\_LSLXD*.

## FORTRAN 77 Interface

Single:     CALL LSLXD (N, NZ, A, IROW, JCOL, B, ITWKSP, X)

Double:     The double precision name is *DLSLXD*.

## Description

Consider the linear equation

$$Ax = b$$

where *A* is sparse, positive definite and symmetric. The sparse coordinate format for the matrix *A* requires one real and two integer vectors. The real array *a* contains all the nonzeros in the *lower triangle* of *A* including the diagonal. Let the number of nonzeros be *nz*. The two integer arrays *irow* and *jcol*, each of length *nz*, contain the row and column indices for these entries in *A*. That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of *A* zero.

The routine *LSLXD* solves a system of linear algebraic equations having a real, sparse and positive definite coefficient matrix. It first uses the routine *LSCXD* to compute a symbolic factorization of a permutation of the coefficient matrix. It then calls *LNFXD* to perform the numerical factorization. The solution of the linear system is then found using *LFSXD*.

The routine *LSCXD* computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor, *L*. Then the routine *LNFXD* produces the numerical entries in *L* so that we have

$$P A P^T = L L^T$$

Here  $P$  is the permutation matrix determined by the ordering.

The numerical computations can be carried out in one of two ways. The first method performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

Finally, the solution  $x$  is obtained by the following calculations:

$$1) Ly_1 = Pb$$

$$2) L^T y_2 = y_1$$

$$3) x = P^T y_2$$

The routine `LFSXD` accepts  $b$  and the permutation vector which determines  $P$ . It then returns  $x$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LXD/DL2LXD`. The reference is:

```
CALL L2LXD (N, NZ, A, IROW, JCOL, B, X, IPER, IPARAM,
 RPARAM, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

***IPER*** — Vector of length  $N$  containing the ordering.

***IPARAM*** — Integer vector of length 4. See Comment 3.

***RPARAM*** — Real vector of length 2. See Comment 3.

***WK*** — Real work vector of length  $LWK$ .

***LWK*** — The length of  $WK$ ,  $LWK$  should be at least  $2N + 6NZ$ .

***IWK*** — Integer work vector of length  $LIWK$ .

***LIWK*** — The length of  $IWK$ ,  $LIWK$  should be at least  $15N + 15NZ + 9$ .

Note that the parameter `ITWKS` is not an argument to this routine.

2. Informational errors

Type Code

4 1 The coefficient matrix is not positive definite.

4 2 A column without nonzero elements has been found in the coefficient matrix.

3. If the default parameters are desired for L2LXD, then set IPARAM(1) to zero and call the routine L2LXD. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling L2LXD.

```
CALL L4LXD (IPARAM, RPARAM)
```

Set nondefault values for desired IPARAM, RPARAM elements.

---

Note that the call to L4LXD will set IPARAM and RPARAM to their default values, so only nondefault values need to be set above. The arguments are as follows:

---

**IPARAM** — Integer vector of length 4.

IPARAM(1) = Initialization flag.

IPARAM(2) = The numerical factorization method.

| IPARAM(2) | Action                  |
|-----------|-------------------------|
| 0         | Multifrontal            |
| 1         | Markowitz column search |

Default: 0.

IPARAM(3) = The ordering option.

| IPARAM(3) | Action                            |
|-----------|-----------------------------------|
| 0         | Minimum degree ordering           |
| 1         | User's ordering specified in IPER |

Default: 0.

IPARAM(4) = The total number of nonzeros in the factorization matrix.

**RPARAM** — Real vector of length 2.

RPARAM(1) = The value of the largest diagonal element in the Cholesky factorization.

RPARAM(2) = The value of the smallest diagonal element in the Cholesky factorization.

If double precision is required, then DL4LXD is called and RPARAM is declared double precision.

## Example

As an example consider the  $5 \times 5$  linear system:

$$A = \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

Let  $x^T = (1, 2, 3, 4, 5)$  so that  $Ax = (23, 55, 107, 197, 278)^T$ . The number of nonzeros in the lower triangle of  $A$  is  $nz = 10$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

```
irow 1 2 3 3 4 4 5 5 5 5
jcol 1 2 1 3 3 4 1 2 4 5
a 10 20 1 30 4 40 2 3 5 50
```

or equivalently by

```
irow 4 5 5 5 1 2 3 3 4 5
jcol 4 1 2 4 1 2 1 3 3 5
a 40 2 3 5 10 20 1 30 4 50
```

```
USE LSLXD_INT
USE WRRRN_INT
INTEGER N, NZ
PARAMETER (N=5, NZ=10)
!
INTEGER IROW(NZ), JCOL(NZ)
REAL A(NZ), B(N), X(N)
!
DATA A/10., 20., 1., 30., 4., 40., 2., 3., 5., 50./
DATA B/23., 55., 107., 197., 278./
DATA IROW/1, 2, 3, 3, 4, 4, 5, 5, 5, 5/
DATA JCOL/1, 2, 1, 3, 3, 4, 1, 2, 4, 5/
!
Solve A * X = B
CALL LSLXD (A, IROW, JCOL, B, X)
!
Print results
CALL WRRRN (' x ', X, 1, N, 1)
END
```

## Output

```
 x
 1 2 3 4 5
1.000 2.000 3.000 4.000 5.000
```

---

## LSCXD

Performs the symbolic Cholesky factorization for a sparse symmetric matrix using a minimum degree ordering or a user-specified ordering, and set up the data structure for the numerical Cholesky factorization

### Required Arguments

**IROW** — Vector of length  $NZ$  containing the row subscripts of the nonzeros in the lower triangular part of the matrix including the nonzeros on the diagonal. (Input)

**JCOL** — Vector of length  $NZ$  containing the column subscripts of the nonzeros in the lower triangular part of the matrix including the nonzeros on the diagonal. (Input)  
( $IROW(K)$ ,  $JCOL(K)$ ) gives the row and column indices of the  $k$ -th nonzero element of the matrix stored in coordinate form. Note,  $IROW(K) \geq JCOL(K)$ .

**NZSUB** — Vector of length  $MAXSUB$  containing the row subscripts for the off-diagonal nonzeros in the Cholesky factor in compressed format. (Output)

**INZSUB** — Vector of length  $N + 1$  containing pointers for  $NZSUB$ . The row subscripts for the off-diagonal nonzeros in column  $J$  are stored in  $NZSUB$  from location  $INZSUB(J)$  to  $INZSUB(J + (ILNZ(J + 1) - ILNZ(J) - 1))$ . (Output)

**MAXNZ** — Total number of off-diagonal nonzeros in the Cholesky factor. (Output)

**ILNZ** — Vector of length  $N + 1$  containing pointers to the Cholesky factor. The off-diagonal nonzeros in column  $J$  of the factor are stored from location  $ILNZ(J)$  to  $ILNZ(J + 1) - 1$ . (Output)  
( $ILNZ$ ,  $NZSUB$ ,  $INZSUB$ ) sets up the data structure for the off-diagonal nonzeros of the Cholesky factor in column ordered form using compressed subscript format.

**INVPER** — Vector of length  $N$  containing the inverse permutation. (Output)  
 $INVPER(K) = I$  indicates that the original row  $K$  is the new row  $I$ .

### Optional Arguments

**N** — Number of equations. (Input)  
Default:  $N = \text{size}(INVPER, 1)$ .

**NZ** — Total number of the nonzeros in the lower triangular part of the symmetric matrix, including the nonzeros on the diagonal. (Input)  
Default:  $NZ = \text{size}(IROW, 1)$ .

**IJOB** — Integer parameter selecting an ordering to permute the matrix symmetrically. (Input)  
 $IJOB = 0$  selects the user ordering specified in  $IPER$  and reorders it so that the multifrontal method can be used in the numerical factorization.  
 $IJOB = 1$  selects the user ordering specified in  $IPER$ .

IJOB = 2 selects a minimum degree ordering.  
IJOB = 3 selects a minimum degree ordering suitable for the multifrontal method in the numerical factorization.  
Default: IJOB = 3.

**ITWKSP** — The total workspace needed. (Input)  
If the default is desired, set ITWKSP to zero.  
Default: ITWKSP = 0.

**MAXSUB** — Number of subscripts contained in array NZSUB. (Input/Output)  
On input, MAXSUB gives the size of the array NZSUB.  
Note that when default workspace (ITWKSP = 0) is used, set MAXSUB = 3 \* NZ.  
Otherwise (ITWKSP > 0), set MAXSUB = (ITWKSP - 10 \* N - 7) / 4. On output, MAXSUB gives the number of subscripts used by the compressed subscript format.  
Default: MAXSUB = 3\*NZ.

**IPER** — Vector of length N containing the ordering specified by IJOB. (Input/Output)  
IPER (I) = K indicates that the original row K is the new row I.

**ISPACE** — The storage space needed for stack of frontal matrices. (Output)

## FORTRAN 90 Interface

Generic: Because the Fortran compiler cannot determine the precision desired from the required arguments, there is no generic Fortran 90 Interface for this routine. The specific Fortran 90 Interfaces are:

Single:      CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER [, ...])

Or

CALL S\_LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER [, ...])

Double:      CALL DLSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER [, ...])

Or

CALL D\_LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER [, ...])

## FORTRAN 77 Interface

Single:      CALL LSCXD (N, NZ, IROW, JCOL, IJOB, ITWKSP, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE)

Double:      The double precision name is DLSCXD.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and symmetric. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array  $a$  contains all the nonzeros in the *lower triangle* of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero.

The routine `LSCXD` computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor,  $L$ . Then the routine `LNFXD` produces the numerical entries in  $L$  so that we have

$$PAP^T = LL^T$$

Here,  $P$  is the permutation matrix determined by the ordering.

The numerical computations can be carried out in one of two ways. The first method performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2CXD`. The reference is:

```
CALL L2CXD (N, NZ, IROW, JCOL, IJOB, MAXSUB, NZSUB,
 INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, LIWK,
 IWK)
```

The additional arguments are as follows:

**LIWK** — The length of `IWK`, `LIWK` should be at least  $10N + 12NZ + 7$ . Note that the argument `MAXSUB` should be set to  $(LIWK - 10N - 7)/4$ .

**IWK** — Integer work vector of length `LIWK`.

Note that the parameter `ITWKSP` is not an argument to this routine.

2. Informational errors

Type Code

4 1 The matrix is structurally singular.

### Example

As an example, the following matrix is symbolically factorized, and the result is printed:

$$A = \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

The number of nonzeros in the lower triangle of  $A$  is  $nz = 10$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

```
 irow 1 2 3 3 4 4 5 5 5 5
 jcol 1 2 1 3 3 4 1 2 4 5
```

or equivalently by

```
 irow 4 5 5 5 1 2 3 3 4 5
 jcol 4 1 2 4 1 2 1 3 3 5
```

```
 USE LSCXD_INT
 USE WRIRN_INT
 INTEGER N, NZ
 PARAMETER (N=5, NZ=10)
!
 INTEGER ILNZ(N+1), INVPER(N), INZSUB(N+1), IPER(N), &
 IROW(NZ), ISPACE, JCOL(NZ), MAXNZ, MAXSUB, &
 NZSUB(3*NZ)
!
 DATA IROW/1, 2, 3, 3, 4, 4, 5, 5, 5, 5/
 DATA JCOL/1, 2, 1, 3, 3, 4, 1, 2, 4, 5/
 MAXSUB = 3 * NZ
 CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER, &
 MAXSUB=MAXSUB, IPER=IPER)
!
 Print results
 CALL WRIRN (' iper ', IPER, 1, N, 1)
 CALL WRIRN (' invper ', INVPER, 1, N, 1)
 CALL WRIRN (' nzsub ', NZSUB, 1, MAXSUB, 1)
 CALL WRIRN (' inzsub ', INZSUB, 1, N+1, 1)
 CALL WRIRN (' ilnz ', ILNZ, 1, N+1, 1)
 END
```

### Output

```
 iper
1 2 3 4 5
```

```

2 1 5 4 3

 invper
1 2 3 4 5
2 1 5 4 3

 nzsub
1 2 3 4
3 5 4 5

 inzsub
1 2 3 4 5 6
1 1 3 4 4 4

 ilnz
1 2 3 4 5 6
1 2 4 6 7 7

```

---

## LNF XD

Computes the numerical Cholesky factorization of a sparse symmetrical matrix  $A$ .

### Required Arguments

- $A$**  — Vector of length  $NZ$  containing the nonzero coefficients of the lower triangle of the linear system. (Input)
- $IROW$**  — Vector of length  $NZ$  containing the row numbers of the corresponding elements in the lower triangle of  $A$ . (Input)
- $JCOL$**  — Vector of length  $NZ$  containing the column numbers of the corresponding elements in the lower triangle of  $A$ . (Input)
- $MAXSUB$**  — Number of subscripts contained in array  $NZSUB$  as output from subroutine `LSCXD/DLSCXD`. (Input)
- $NZSUB$**  — Vector of length  $MAXSUB$  containing the row subscripts for the nonzeros in the Cholesky factor in compressed format as output from subroutine `LSCXD/DLSCXD`. (Input)
- $INZSUB$**  — Vector of length  $N + 1$  containing pointers for  $NZSUB$  as output from subroutine `LSCXD/DLSCXD`. (Input)  
The row subscripts for the nonzeros in column  $J$  are stored from location  $INZSUB(J)$  to  $INZSUB(J + 1) - 1$ .
- $MAXNZ$**  — Length of  $RLNZ$  as output from subroutine `LSCXD/DLSCXD`. (Input)
- $ILNZ$**  — Vector of length  $N + 1$  containing pointers to the Cholesky factor as output from subroutine `LSCXD/DLSCXD`. (Input)  
The row subscripts for the nonzeros in column  $J$  of the factor are stored from location

$ILNZ(J)$  to  $ILNZ(J + 1) - 1$ . ( $ILNZ$ ,  $NZSUB$ ,  $INZSUB$ ) sets up the compressed data structure in column ordered form for the Cholesky factor.

***IPER*** — Vector of length  $N$  containing the permutation as output from subroutine `LSCXD/DLSCXD`. (Input)

***INVPER*** — Vector of length  $N$  containing the inverse permutation as output from subroutine `LSCXD/DLSCXD`. (Input)

***ISPACE*** — The storage space needed for the stack of frontal matrices as output from subroutine `LSCXD/DLSCXD`. (Input)

***DIAGNL*** — Vector of length  $N$  containing the diagonal of the factor. (Output)

***RLNZ*** — Vector of length  $MAXNZ$  containing the strictly lower triangle nonzeros of the Cholesky factor. (Output)

***RPARAM*** — Parameter vector containing factorization information. (Output)  
`RPARAM(1)` = smallest diagonal element.  
`RPARAM(2)` = largest diagonal element.

## Optional Arguments

***N*** — Number of equations. (Input)  
Default:  $N = \text{size}(IPER, 1)$ .

***NZ*** — The number of nonzero coefficients in the linear system. (Input)  
Default:  $NZ = \text{size}(A, 1)$ .

***IJOB*** — Integer parameter selecting factorization method. (Input)  
`IJOB = 1` yields factorization in sparse column format.  
`IJOB = 2` yields factorization using multifrontal method.  
Default: `IJOB = 1`.

***ITWKSP*** — The total workspace needed. (Input)  
If the default is desired, set `ITWKSP` to zero.  
Default: `ITWKSP = 0`.

## FORTRAN 90 Interface

Generic: `CALL LNF XD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, DIAGNL, RLNZ, RPARAM [, ...])`

Specific: The specific interface names are `S_LNF XD` and `D_LNF XD`.

## FORTRAN 77 Interface

Single:      CALL LNF XD (N, NZ, A, IROW, JCOL, IJOB, ITWKSP, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, ITWKSP, DIAGNL, RLNZ, RPARAM)

Double:      The double precision name is DLNF XD.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and symmetric. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array  $a$  contains all the nonzeros in the *lower triangle* of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero. The routine LNF XD produces the Cholesky factorization of  $PA P^T$  given the symbolic factorization of  $A$  which is computed by LSC XD. That is, this routine computes  $L$  which satisfies

$$PA P^T = LL^T$$

The diagonal of  $L$  is stored in DIAGNL and the strictly lower triangular part of  $L$  is stored in compressed subscript form in  $R = RLNZ$  as follows. The nonzeros in the  $j$ -th column of  $L$  are stored in locations  $R(i), \dots, R(i+k)$  where  $i = ILNZ(j)$  and  $k = ILNZ(j+1) - ILNZ(j) - 1$ . The row subscripts are stored in the vector NZSUB from locations INZSUB( $j$ ) to INZSUB( $j$ ) +  $k$ .

The numerical computations can be carried out in one of two ways. The first method (when IJOB = 2) performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method (when IJOB = 1) is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

## Comments

1.      Workspace may be explicitly provided by use of L2FXD/DL2FXD . The reference is:

```
CALL L2FXD (N, NZ, A, IROW, JCOL, IJOB, MAXSUB, NZSUB,
 INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, DIAGNL,
 RLNZ, RPARAM, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length `LWK`.

**LWK** — The length of `WK`, `LWK` should be at least  $N + 3NZ$ .

**IWK** — Integer work vector of length `LIWK`.

**LIWK** — The length of `IWK`, `LIWK` should be at least  $2N$ .

Note that the parameter `ITWKSP` is not an argument to this routine.

2. Informational errors

Type Code

4 1 The coefficient matrix is not positive definite.

4 2 A column without nonzero elements has been found in the coefficient matrix.

### Example

As an example, consider the  $5 \times 5$  linear system:

$$A = \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

The number of nonzeros in the lower triangle of  $A$  is  $nz = 10$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

```
irow 1 2 3 3 4 4 5 5 5 5
jcol 1 2 1 3 3 4 1 2 4 5
a 10 20 1 30 4 40 2 3 5 50
```

or equivalently by

```
irow 4 5 5 5 1 2 3 3 4 5
jcol 4 1 2 4 1 2 1 3 3 5
a 40 2 3 5 10 20 1 30 4 50
```

We first call `LSCXD` to produce the symbolic information needed to pass on to `LNFXD`. Then call `LNFXD` to factor this matrix. The results are displayed below.

```
USE LNFXD_INT
USE LSCXD_INT
```

```

USE WRRRN_INT
INTEGER N, NZ, NRLNZ
PARAMETER (N=5, NZ=10, NRLNZ=10)
!
INTEGER IJOB, ILNZ(N+1), INVPER(N), INZSUB(N+1), IPER(N), &
IROW(NZ), ISPACE, JCOL(NZ), MAXNZ, MAXSUB, &
NZSUB(3*NZ)
REAL A(NZ), DIAGNL(N), RLNZ(NRLNZ), RPARAM(2), R(N,N)
!
DATA A/10., 20., 1., 30., 4., 40., 2., 3., 5., 50./
DATA IROW/1, 2, 3, 3, 4, 4, 5, 5, 5, 5/
DATA JCOL/1, 2, 1, 3, 3, 4, 1, 2, 4, 5/
!
! Select minimum degree ordering
! for multifrontal method
IJOB = 3
!
! Use default workspace
MAXSUB = 3*NZ
CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER, &
MAXSUB=MAXSUB)
!
! Check if NRLNZ is large enough
IF (NRLNZ .GE. MAXNZ) THEN
!
! Choose multifrontal method
IJOB = 2
CALL LNFCD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, &
ILNZ, IPER, INVPER, ISPACE, DIAGNL, RLNZ, RPARAM, &
IJOB=IJOB)
!
! Print results
CALL WRRRN ('diagnl ', DIAGNL, NRA=1, NCA=N, LDA=1)
CALL WRRRN ('rlnz ', RLNZ, NRA=1, NCA=MAXNZ, LDA=1)
END IF
!
! Construct L matrix
DO I=1,N
!
! Diagonal
R(I,I) = DIAG(I)
IF (ILNZ(I) .GT. MAXNZ) GO TO 50
!
! Find elements of RLNZ for this column
ISTRT = ILNZ(I)
ISTOP = ILNZ(I+1) - 1
!
! Get starting index for NZSUB
K = INZSUB(I)
DO J=ISTRT, ISTOP
!
! NZSUB(K) is the row for this element of
! RLNZ
R((NZSUB(K)),I) = RLNZ(J)
K = K + 1
END DO
END DO
50 CONTINUE
CALL WRRRN ('L', R, NRA=N, NCA=N)
END

```

## Output

```

 diagnl
1 2 3 4 5

```

```

4.472 3.162 7.011 6.284 5.430

 rlnz
 1 2 3 4 5 6
0.6708 0.6325 0.3162 0.7132 -0.0285 0.6398

 L
 1 2 3 4 5
1 4.472 0.000 0.000 0.000 0.000
2 0.000 3.162 0.000 0.000 0.000
3 0.671 0.632 7.011 0.000 0.000
4 0.000 0.000 0.713 6.284 0.000
5 0.000 0.316 -0.029 0.640 5.430

```

---

## LFSXD

Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

### Required Arguments

*N* — Number of equations. (Input)

*MAXSUB* — Number of subscripts contained in array *NZSUB* as output from subroutine *LSCXD/DLSCXD*. (Input)

*NZSUB* — Vector of length *MAXSUB* containing the row subscripts for the off-diagonal nonzeros in the factor as output from subroutine *LSCXD/DLSCXD*. (Input)

*INZSUB* — Vector of length *N + 1* containing pointers for *NZSUB* as output from subroutine *LSCXD/DLSCXD*. (Input)  
The row subscripts of column *J* are stored from location *INZSUB(J)* to *INZSUB(J + 1) - 1*.

*MAXNZ* — Total number of off-diagonal nonzeros in the Cholesky factor as output from subroutine *LSCXD/DLSCXD*. (Input)

*RLNZ* — Vector of length *MAXNZ* containing the off-diagonal nonzeros in the factor in column ordered format as output from subroutine *LNFXD/DLNFXD*. (Input)

*ILNZ* — Vector of length *N + 1* containing pointers to *RLNZ* as output from subroutine *LSCXD/DLSCXD*. The nonzeros in column *J* of the factor are stored from location *ILNZ(J)* to *ILNZ(J + 1) - 1*. (Input)  
The values (*RLNZ*, *ILNZ*, *NZSUB*, *INZSUB*) give the off-diagonal nonzeros of the factor in a compressed subscript data format.

*DIAGNL* — Vector of length *N* containing the diagonals of the Cholesky factor as output from subroutine *LNFXD/DLNFXD*. (Input)

**IPER** — Vector of length  $N$  containing the ordering as output from subroutine LSCXD/DLSCXD. (Input)  
 $IPER(I) = K$  indicates that the original row  $K$  is the new row  $I$ .

**B** — Vector of length  $N$  containing the right-hand side. (Input)

**X** — Vector of length  $N$  containing the solution. (Output)

### FORTRAN 90 Interface

Generic:    CALL LFSXD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL,  
                  IPER, B, X)

Specific:    The specific interface names are S\_LFSXD and D\_LFSXD.

### FORTRAN 77 Interface

Single:     CALL LFSXD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL,  
                  IPER, B, X)

Double:     The double precision name is DLFSXD.

### Description

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and symmetric. The sparse coordinate format for the matrix  $A$  requires one real and two integer vectors. The real array  $a$  contains all the nonzeros in the *lower triangle* of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$
$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero.

The routine LFSXD computes the solution of the linear system given its Cholesky factorization. The factorization is performed by calling LSCXD followed by LNF XD. The routine LSCXD computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor,  $L$ . Then the routine LNF XD produces the numerical entries in  $L$  so that we have

$$PAP^T = LL^T$$

Here  $P$  is the permutation matrix determined by the ordering.

The numerical computations can be carried out in one of two ways. The first method performs the factorization using a multifrontal technique. This option requires more storage but in certain cases

will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

Finally, the solution  $x$  is obtained by the following calculations:

$$1) Ly_1 = Pb$$

$$2) L^T y_2 = y_1$$

$$3) x = P^T y_2$$

## Comments

Informational error

Type Code

4 1 The input matrix is numerically singular.

## Example

As an example, consider the  $5 \times 5$  linear system:

$$A = \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

Let

$$x_1^T = (1, 2, 3, 4, 5)$$

so that  $Ax_1 = (23, 55, 107, 197, 278)^T$ , and

$$x_2^T = (5, 4, 3, 2, 1)$$

so that  $Ax_2 = (55, 83, 103, 97, 82)^T$ . The number of nonzeros in the lower triangle of  $A$  is  $n_z = 10$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

|      |    |    |   |    |   |    |   |   |    |
|------|----|----|---|----|---|----|---|---|----|
| irow | 1  | 2  | 3 | 3  | 4 | 4  | 5 | 5 | 5  |
| jcol | 1  | 2  | 1 | 3  | 3 | 4  | 1 | 2 | 4  |
| a    | 10 | 20 | 1 | 30 | 4 | 40 | 2 | 3 | 50 |

or equivalently by

```

irow 4 5 5 5 1 2 3 3 4 5
jcol 4 1 2 4 1 2 1 3 3 5
a 40 2 3 5 10 20 1 30 4 50

```

```

USE LFSXD_INT
USE LNF XD_INT
USE LSCXD_INT
USE WRRRN_INT
INTEGER N, NZ, NRLNZ
PARAMETER (N=5, NZ=10, NRLNZ=10)
!
INTEGER IJOB, ILNZ(N+1), INVPER(N), INZSUB(N+1), IPER(N), &
IROW(NZ), ISPACE, ITWKSP, JCOL(NZ), MAXNZ, MAXSUB, &
NZSUB(3*NZ)
REAL A(NZ), B1(N), B2(N), DIAGNL(N), RLNZ(NRLNZ), RPARAM(2), &
X(N)
!
DATA A/10., 20., 1., 30., 4., 40., 2., 3., 5., 50./
DATA B1/23., 55., 107., 197., 278./
DATA B2/55., 83., 103., 97., 82./
DATA IROW/1, 2, 3, 3, 4, 4, 5, 5, 5, 5/
DATA JCOL/1, 2, 1, 3, 3, 4, 1, 2, 4, 5/
!
! Select minimum degree ordering
! for multifrontal method
IJOB = 3
!
! Use default workspace
ITWKSP = 0
MAXSUB = 3*NZ
CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER, &
MAXSUB=MAXSUB, IPER=IPER, ISPACE=ISPACE)
!
! Check if NRLNZ is large enough
IF (NRLNZ .GE. MAXNZ) THEN
!
! Choose multifrontal method
IJOB = 2
CALL LNF XD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, &
IPER, INVPER, ISPACE, DIAGNL, RLNZ, RPARAM, IJOB=IJOB)
!
! Solve A * X1 = B1
CALL LFSXD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL, &
IPER, B1, X)
!
! Print X1
CALL WRRRN (' x1 ', X, 1, N, 1)
!
! Solve A * X2 = B2
CALL LFSXD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, &
DIAGNL, IPER, B2, X)
!
! Print X2
CALL WRRRN (' x2 ' X, 1, N, 1)

END IF
!
END

```

## Output

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
|       |       | x1    |       |       |
| 1     | 2     | 3     | 4     | 5     |
| 1.000 | 2.000 | 3.000 | 4.000 | 5.000 |

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
|       |       | x2    |       |       |
| 1     | 2     | 3     | 4     | 5     |
| 5.000 | 4.000 | 3.000 | 2.000 | 1.000 |

## LSLZD

Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.

### Required Arguments

**A** — Complex vector of length *NZ* containing the nonzero coefficients in the lower triangle of the linear system. (Input)

The sparse matrix has nonzeros only in entries (*IROW*(*i*), *JCOL*(*i*)) for *i* = 1 to *NZ*, and at this location the sparse matrix has value *A*(*i*).

**IROW** — Vector of length *NZ* containing the row numbers of the corresponding elements in the lower triangle of *A*. (Input)

Note *IROW*(*i*) ≥ *JCOL*(*i*), since we are only indexing the lower triangle.

**JCOL** — Vector of length *NZ* containing the column numbers of the corresponding elements in the lower triangle of *A*. (Input)

**B** — Complex vector of length *N* containing the right-hand side of the linear system. (Input)

**X** — Complex vector of length *N* containing the solution to the linear system. (Output)

### Optional Arguments

**N** — Number of equations. (Input)

Default: *N* = size (*B*,1).

**NZ** — The number of nonzero coefficients in the lower triangle of the linear system. (Input)

Default: *NZ* = size (*A*,1).

**ITWKSP** — The total workspace needed. (Input)

If the default is desired, set *ITWKSP* to zero.

Default: *ITWKSP* = 0.

### FORTRAN 90 Interface

Generic:     CALL LSLZD (A, IROW, JCOL, B, X [, ...])

Specific:    The specific interface names are *S\_LSLZD* and *D\_LSLZD*.

## FORTRAN 77 Interface

Single:      CALL LSLZD (N, NZ, A, IROW, JCOL, B, ITWKSP, X)

Double:      The double precision name is DLSLZD.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and Hermitian. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array  $a$  contains all the nonzeros in the lower triangle of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero.

The routine LSLZD solves a system of linear algebraic equations having a complex, sparse, Hermitian and positive definite coefficient matrix. It first uses the routine LSCXD to compute a symbolic factorization of a permutation of the coefficient matrix. It then calls LNFZD to perform the numerical factorization. The solution of the linear system is then found using LFSZD.

The routine LSCXD computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor,  $L$ . Then the routine LNFZD produces the numerical entries in  $L$  so that we have

$$PAP^T = LL^H$$

Here  $P$  is the permutation matrix determined by the ordering.

The numerical computations can be carried out in one of two ways. The first method performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

Finally, the solution  $x$  is obtained by the following calculations:

$$1) \quad Ly_1 = Pb$$

$$2) \quad L^H y_2 = y_1$$

$$3) \quad x = P^T y_2$$

The routine LFSZD accepts  $b$  and the permutation vector which determines  $P$ . It then returns  $x$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2LZD/DL2LZD. The reference is:

```
CALL L2LZD (N, NZ, A, IROW, JCOL, B, X, IPER, IPARAM,
 RPARAM, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

**IPER** — Vector of length  $N$  containing the ordering.

**IPARAM** — Integer vector of length 4. See Comment 3.

**RPARAM** — Real vector of length 2. See Comment 3.

**WK** — Complex work vector of length  $LWK$ .

**LWK** — The length of  $WK$ ,  $LWK$  should be at least  $2N + 6NZ$ .

**IWK** — Integer work vector of length  $LIWK$ .

**LIWK** — The length of  $IWK$ ,  $LIWK$  should be at least  $15N + 15NZ + 9$ .

Note that the parameter `ITWKSP` is not an argument for this routine.

2. Informational errors

Type Code

4 1 The coefficient matrix is not positive definite.

4 2 A column without nonzero elements has been found in the coefficient matrix.

3. If the default parameters are desired for L2LZD, then set `IPARAM(1)` to zero and call the routine L2LZD. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling L2LZD.

```
CALL L4LZD (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to L4LZD will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above. The arguments are as follows:

---

**IPARAM** — Integer vector of length 4.

`IPARAM(1)` = Initialization flag.

IPARAM(2) = The numerical factorization method.

| IPARAM(2) | Action        |
|-----------|---------------|
| 0         | Multifrontal  |
| 1         | Sparse column |

Default: 0.

IPARAM(3) = The ordering option.

| IPARAM(3) | Action                            |
|-----------|-----------------------------------|
| 0         | Minimum degree ordering           |
| 1         | User's ordering specified in IPER |

Default: 0.

IPARAM(4) = The total number of nonzeros in the factorization matrix.

**RPARAM** — Real vector of length 2.

RPARAM(1) = The absolute value of the largest diagonal element in the Cholesky factorization.

RPARAM(2) = The absolute value of the smallest diagonal element in the Cholesky factorization.

If double precision is required, then DL4LZD is called and RPARAM is declared double precision.

## Example

As an example, consider the  $3 \times 3$  linear system:

$$A = \begin{bmatrix} 2+0i & -1+i & 0 \\ -1-i & 4+0i & 1+2i \\ 0 & 1-2i & 10+0i \end{bmatrix}$$

Let  $x^T = (1 + i, 2 + 2i, 3 + 3i)$  so that  $Ax = (-2 + 2i, 5 + 15i, 36 + 28i)^T$ . The number of nonzeros in the lower triangle of  $A$  is  $nz = 5$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

|      |        |        |         |        |        |
|------|--------|--------|---------|--------|--------|
| irow | 1      | 2      | 3       | 2      | 3      |
| jcol | 1      | 2      | 3       | 1      | 2      |
| a    | $2+0i$ | $4+0i$ | $10+0i$ | $-1-i$ | $1-2i$ |

or equivalently by

```

irow 3 2 3 1 2
jcol 3 1 2 1 2
a 10+0i -1-i 1-2i 2+0i 4+0i

```

```

USE LSLZD_INT
USE WRCRN_INT
INTEGER N, NZ
PARAMETER (N=3, NZ=5)
!
INTEGER IROW(NZ), JCOL(NZ)
COMPLEX A(NZ), B(N), X(N)
!
DATA A/(2.0,0.0), (4.0,0.0), (10.0,0.0), (-1.0,-1.0), (1.0,-2.0)/
DATA B/(-2.0,2.0), (5.0,15.0), (36.0,28.0)/
DATA IROW/1, 2, 3, 2, 3/
DATA JCOL/1, 2, 3, 1, 2/
!
 Solve A * X = B
CALL LSLZD (A, IROW, JCOL, B, X)
!
 Print results
CALL WRCRN (' x ', X, 1, N, 1)
END

```

## Output

```

 x
 1 2 3
(1.000, 1.000) (2.000, 2.000) (3.000, 3.000)

```

---

## LNFZD

Computes the numerical Cholesky factorization of a sparse Hermitian matrix  $A$ .

### Required Arguments

$A$  — Complex vector of length  $NZ$  containing the nonzero coefficients of the lower triangle of the linear system. (Input)

$IROW$  — Vector of length  $NZ$  containing the row numbers of the corresponding elements in the lower triangle of  $A$ . (Input)

$JCOL$  — Vector of length  $NZ$  containing the column numbers of the corresponding elements in the lower triangle of  $A$ . (Input)

$MAXSUB$  — Number of subscripts contained in array  $NZSUB$  as output from subroutine `LSCXD/DLSCXD`. (Input)

**NZSUB** — Vector of length  $\text{MAXSUB}$  containing the row subscripts for the nonzeros in the Cholesky factor in compressed format as output from subroutine `LSCXD/DLSCXD`.  
(Input)

**INZSUB** — Vector of length  $N + 1$  containing pointers for `NZSUB` as output from subroutine `LSCXD/DLSCXD`. (Input)  
The row subscripts for the nonzeros in column  $J$  are stored from location `INZSUB(J)` to `INZSUB(J + 1) - 1`.

**MAXNZ** — Length of `RLNZ` as output from subroutine `LSCXD/DLSCXD`. (Input)

**ILNZ** — Vector of length  $N + 1$  containing pointers to the Cholesky factor as output from subroutine `LSCXD/DLSCXD`. (Input)  
The row subscripts for the nonzeros in column  $J$  of the factor are stored from location `ILNZ(J)` to `ILNZ(J + 1) - 1`.  
(`ILNZ`, `NZSUB`, `INZSUB`) sets up the compressed data structure in column ordered form for the Cholesky factor.

**IPER** — Vector of length  $N$  containing the permutation as output from subroutine `LSCXD/DLSCXD`. (Input)

**INVPER** — Vector of length  $N$  containing the inverse permutation as output from subroutine `LSCXD/DLSCXD`. (Input)

**ISPACE** — The storage space needed for the stack of frontal matrices as output from subroutine `LSCXD/DLSCXD`. (Input)

**DIAGNL** — Complex vector of length  $N$  containing the diagonal of the factor. (Output)

**RLNZ** — Complex vector of length  $\text{MAXNZ}$  containing the strictly lower triangle nonzeros of the Cholesky factor. (Output)

**RPARAM** — Parameter vector containing factorization information. (Output)  
`RPARAM(1)` = smallest diagonal element in absolute value.  
`RPARAM(2)` = largest diagonal element in absolute value.

## Optional Arguments

**$N$**  — Number of equations. (Input)  
Default:  $N = \text{size}(\text{IPER}, 1)$ .

**$NZ$**  — The number of nonzero coefficients in the linear system. (Input)  
Default:  $NZ = \text{size}(A, 1)$ .

**$IJOB$**  — Integer parameter selecting factorization method. (Input)  
`IJOB = 1` yields factorization in sparse column format.  
`IJOB = 2` yields factorization using multifrontal method.  
Default: `IJOB = 1`.

**ITWKSP** — The total workspace needed. (Input)

If the default is desired, set **ITWKSP** to zero. See Comment 1 for the default.

Default: **ITWKSP** = 0.

## **FORTRAN 90 Interface**

Generic:     CALL LNFZD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, IPER,  
                  INVPER, ISPACE, DIAGNL, RLNZ, RPARAM [, ...])

Specific:    The specific interface names are **S\_LNFZD** and **D\_LNFZD**.

## **FORTRAN 77 Interface**

Single:      CALL LNFZD (N, NZ, A, IROW, JCOL, IJOB, MAXSUB, NZSUB, INZSUB, MAXNZ,  
                  ILNZ, IPER, INVPER, ISPACE, ITWKSP, DIAGNL, RLNZ, RPARAM)

Double:      The double precision name is **DLNFZD**.

## **Description**

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and Hermitian. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array  $a$  contains all the nonzeros in the *lower triangle* of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i)jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero.

The routine **LNFZD** produces the Cholesky factorization of  $PA P^T$  given the symbolic factorization of  $A$  which is computed by **LSCXD**. That is, this routine computes  $L$  which satisfies

$$P A P^T = L L^H$$

The diagonal of  $L$  is stored in **DIAGNL** and the strictly lower triangular part of  $L$  is stored in compressed subscript form in  $R = \text{RLNZ}$  as follows. The nonzeros in the  $j$ th column of  $L$  are stored in locations  $R(i), \dots, R(i+k)$  where  $i = \text{ILNZ}(j)$  and  $k = \text{ILNZ}(j+1) - \text{ILNZ}(j) - 1$ . The row subscripts are stored in the vector **NZSUB** from locations **INZSUB**( $j$ ) to **INZSUB**( $j$ ) +  $k$ .

The numerical computations can be carried out in one of two ways. The first method (when **IJOB** = 2) performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983,

1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method (when  $IJOB = 1$ ) is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme.

## Comments

1. Workspace may be explicitly provided by use of L2FZD/DL2FZD. The reference is:

```
CALL L2FZD (N, NZ, A, IROW, JCOL, IJOB, MAXSUB, NZSUB,
 INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, DIAGNL,
 RLNZ, RPARAM, WK, LWK, IWK, LIWK)
```

The additional arguments are as follows:

**WK** — Complex work vector of length  $LWK$ .

**LWK** — The length of  $WK$ ,  $LWK$  should be at least  $N + 3NZ$ .

**IWK** — Integer work vector of length  $LIWK$ .

**LIWK** — The length of  $IWK$ ,  $LIWK$  should be at least  $2N$ .

Note that the parameter  $ITWKSP$  is not an argument to this routine.

2. Informational errors

Type Code

|   |   |                                                                             |
|---|---|-----------------------------------------------------------------------------|
| 4 | 1 | The coefficient matrix is not positive definite.                            |
| 4 | 2 | A column without nonzero elements has been found in the coefficient matrix. |

## Example

As an example, consider the  $3 \times 3$  linear system:

$$A = \begin{bmatrix} 2+0i & -1+i & 0 \\ -1-i & 4+0i & 1+2i \\ 0 & 1-2i & 10+0i \end{bmatrix}$$

The number of nonzeros in the lower triangle of  $A$  is  $nz = 5$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

|      |        |        |         |        |        |
|------|--------|--------|---------|--------|--------|
| irow | 1      | 2      | 3       | 2      | 3      |
| jcol | 1      | 2      | 3       | 1      | 2      |
| a    | $2+0i$ | $4+0i$ | $10+0i$ | $-1-i$ | $1-2i$ |

or equivalently by

|      |         |        |        |        |        |
|------|---------|--------|--------|--------|--------|
| irow | 3       | 2      | 3      | 1      | 2      |
| jcol | 3       | 1      | 2      | 1      | 2      |
| a    | $10+0i$ | $-1-i$ | $1-2i$ | $2+0i$ | $4+0i$ |

We first call LSCXD to produce the symbolic information needed to pass on to LNFZD. Then call LNFZD to factor this matrix. The results are displayed below.

```

USE LNFZD_INT
USE LSCXD_INT
USE WRCRN_INT
INTEGER N, NZ, NRLNZ
PARAMETER (N=3, NZ=5, NRLNZ=5)
!
INTEGER IJOB, ILNZ(N+1), INVPER(N), INZSUB(N+1), IPER(N), &
IROW(NZ), ISPACE, JCOL(NZ), MAXNZ, MAXSUB, &
NZSUB(3*NZ)
REAL RPARAM(2)
COMPLEX A(NZ), DIAGNL(N), RLNZ(NRLNZ)
!
DATA A/(2.0,0.0), (4.0,0.0), (10.0,0.0), (-1.0,-1.0), (1.0,-2.0)/
DATA IROW/1, 2, 3, 2, 3/
DATA JCOL/1, 2, 3, 1, 2/
!
! Select minimum degree ordering
! for multifrontal method
IJOB = 3
MAXSUB = 3*NZ
CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER, &
IJOB=IJOB, MAXSUB=MAXSUB)
!
! Check if NRLNZ is large enough
IF (NRLNZ .GE. MAXNZ) THEN
!
! Choose multifrontal method
IJOB = 2
CALL LNFZD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, &
ILNZ, IPER, INVPER, ISPACE, DIAGNL, RLNZ, RPARAM, &
IJOB=IJOB)
!
! Print results
CALL WRCRN (' diagnl ', DIAGNL, 1, N, 1)
CALL WRCRN (' rlnz ', RLNZ, 1, MAXNZ, 1)
END IF
!
END

```

## Output

```

 diagnl
 1 2 3
(1.414, 0.000) (1.732, 0.000) (2.887, 0.000)

 rlnz
 1 2
(-0.707,-0.707) (0.577,-1.155)

```

---

# LFSZD

Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

## Required Arguments

*N* — Number of equations. (Input)

*MAXSUB* — Number of subscripts contained in array *NZSUB* as output from subroutine *LSCXD/DLSCXD*. (Input)

*NZSUB* — Vector of length *MAXSUB* containing the row subscripts for the off-diagonal nonzeros in the factor as output from subroutine *LSCXD/DLSCXD*. (Input)

*INZSUB* — Vector of length  $N + 1$  containing pointers for *NZSUB* as output from subroutine *LSCXD/DLSCXD*. (Input)  
The row subscripts of column *J* are stored from location *INZSUB*(*J*) to *INZSUB*(*J* + 1) - 1.

*MAXNZ* — Total number of off-diagonal nonzeros in the Cholesky factor as output from subroutine *LSCXD/DLSCXD*. (Input)

*RLNZ* — Complex vector of length *MAXNZ* containing the off-diagonal nonzeros in the factor in column ordered format as output from subroutine *LNFDZ/DLNFDZ*. (Input)

*ILNZ* — Vector of length  $N + 1$  containing pointers to *RLNZ* as output from subroutine *LSCXD/DLSCXD*. The nonzeros in column *J* of the factor are stored from location *ILNZ*(*J*) to *ILNZ*(*J* + 1) - 1. (Input)  
The values (*RLNZ*, *ILNZ*, *NZSUB*, *INZSUB*) give the off-diagonal nonzeros of the factor in a compressed subscript data format.

*DIAGNL* — Complex vector of length *N* containing the diagonals of the Cholesky factor as output from subroutine *LNFDZ/DLNFDZ*. (Input)

*IPER* — Vector of length *N* containing the ordering as output from subroutine *LSCXD/DLSCXD*. (Input)  
*IPER*(*I*) = *K* indicates that the original row *K* is the new row *I*.

*B* — Complex vector of length *N* containing the right-hand side. (Input)

*X* — Complex vector of length *N* containing the solution. (Output)

## FORTRAN 90 Interface

Generic:   CALL LFSZD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL,  
                  IPER, B, X)

Specific: The specific interface names are `S_LFSZD` and `D_LFSZD`.

## FORTRAN 77 Interface

Single: `CALL LFSZD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL, IPER, B, X)`

Double: The double precision name is `DLFSZD`.

## Description

Consider the linear equation

$$Ax = b$$

where  $A$  is sparse, positive definite and Hermitian. The sparse coordinate format for the matrix  $A$  requires one complex and two integer vectors. The complex array  $a$  contains all the nonzeros in the *lower triangle* of  $A$  including the diagonal. Let the number of nonzeros be  $nz$ . The two integer arrays  $irow$  and  $jcol$ , each of length  $nz$ , contain the row and column indices for these entries in  $A$ . That is

$$A_{irow(i) jcol(i)} = a(i), \quad i = 1, \dots, nz$$

$$irow(i) \geq jcol(i) \quad i = 1, \dots, nz$$

with all other entries in the lower triangle of  $A$  zero.

The routine `LFSZD` computes the solution of the linear system given its Cholesky factorization. The factorization is performed by calling `LSCXD` followed by `LNFDZ`. The routine `LSCXD` computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor,  $L$ . Then the routine `LNFDZ` produces the numerical entries in  $L$  so that we have

$$PAP^T = LL^H$$

Here  $P$  is the permutation matrix determined by the ordering.

The numerical computations can be carried out in one of two ways. The first method performs the factorization using a multifrontal technique. This option requires more storage but in certain cases will be faster. The multifrontal method is based on the routines in Liu (1987). For detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989). The second method is fully described in George and Liu (1981). This is just the standard factorization method based on the sparse compressed storage scheme. Finally, the solution  $x$  is obtained by the following calculations:

$$1) Ly_1 = Pb$$

$$2) L^H y_2 = y_1$$

$$3) x = P^T y_2$$

## Comments

Informational error

Type Code

4 1 The input matrix is numerically singular.

## Example

As an example, consider the  $3 \times 3$  linear system:

$$A = \begin{bmatrix} 2+0i & -1+i & 0 \\ -1-i & 4+0i & 1+2i \\ 0 & 1-2i & 10+0i \end{bmatrix}$$

Let

$$x_1^T = (1+i, 2+2i, 3+3i)$$

so that  $Ax_1 = (-2+2i, 5+15i, 36+28i)^T$ , and

$$x_2^T = (3+3i, 2+2i, 1+i)$$

so that  $Ax_2 = (2+6i, 7-5i, 16+8i)^T$ . The number of nonzeros in the lower triangle of  $A$  is  $nz = 5$ . The sparse coordinate form for the lower triangle of  $A$  is given by:

|      |        |        |         |        |        |
|------|--------|--------|---------|--------|--------|
| irow | 1      | 2      | 3       | 2      | 3      |
| jcol | 1      | 2      | 3       | 1      | 2      |
| a    | $2+0i$ | $4+0i$ | $10+0i$ | $-1-i$ | $1-2i$ |

or equivalently by

|      |         |        |        |        |        |
|------|---------|--------|--------|--------|--------|
| irow | 3       | 2      | 3      | 1      | 2      |
| jcol | 3       | 1      | 2      | 1      | 2      |
| a    | $10+0i$ | $-1-i$ | $1-2i$ | $2+0i$ | $4+0i$ |

```
USE IMSL_LIBRARIES
INTEGER N, NZ, NRLNZ
PARAMETER (N=3, NZ=5, NRLNZ=5)
!
INTEGER IJOB, ILNZ(N+1), INVPER(N), INZSUB(N+1), IPER(N), &
 IROW(NZ), ISPACE, JCOL(NZ), MAXNZ, MAXSUB, &
 NZSUB(3*NZ)
COMPLEX A(NZ), B1(N), B2(N), DIAGNL(N), RLNZ(NRLNZ), X(N)
REAL RPARAM(2)
!
DATA A/(2.0,0.0), (4.0,0.0), (10.0,0.0), (-1.0,-1.0), (1.0,-2.0)/
DATA B1/(-2.0,2.0), (5.0,15.0), (36.0,28.0)/
DATA B2/(2.0,6.0), (7.0,5.0), (16.0,8.0)/
DATA IROW/1, 2, 3, 2, 3/
```

```

DATA JCOL/1, 2, 3, 1, 2/
! Select minimum degree ordering
! for multifrontal method
IJOB = 3
! Use default workspace
MAXSUB = 3*NZ
CALL LSCXD (IROW, JCOL, NZSUB, INZSUB, MAXNZ, ILNZ, INVPER, &
 IJOB=IJOB, MAXSUB=MAXSUB, IPER=IPER, ISPACE=ISPACE)
! Check if NRLNZ is large enough
IF (NRLNZ .GE. MAXNZ) THEN
! Choose multifrontal method
 IJOB = 2
 CALL LNFZD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, &
 MAXNZ, ILNZ, IPER, INVPER, ISPACE, DIAGNL, &
 RLNZ, RPARAM, IJOB=IJOB)
! Solve A * X1 = B1
 CALL LFSZD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL, &
 IPER, B1, X)
! Print X1
 CALL WRCRN (' x1 ', X, 1, N,1)
! Solve A * X2 = B2
 CALL LFSZD (N, MAXSUB, NZSUB, INZSUB, MAXNZ, RLNZ, ILNZ, DIAGNL, &
 IPER, B2, X)
! Print X2
 CALL WRCRN (' x2 ', X, 1, N,1)
END IF
!
END

```

## Output

```

 x1
 1 2 3
(1.000, 1.000) (2.000, 2.000) (3.000, 3.000)

 x2
 1 2 3
(3.000, 3.000) (2.000, 2.000) (1.000, 1.000)

```

---

## LSLTO

Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

### Required Arguments

**A** — Real vector of length  $2N - 1$  containing the first row of the coefficient matrix followed by its first column beginning with the second element. (Input)  
See Comment 2.

**B** — Real vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Real vector of length  $N$  containing the solution of the linear system. (Output)  
If  $B$  is not needed then  $B$  and  $X$  may share the same storage locations.

### Optional Arguments

$N$  — Order of the matrix represented by  $A$ . (Input)  
Default:  $N = (\text{size}(A,1) + 1)/2$

$IPATH$  — Integer flag. (Input)  
 $IPATH = 1$  means the system  $Ax = B$  is solved.  
 $IPATH = 2$  means the system  $A^T x = B$  is solved.  
Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic: `CALL LSLTO (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLTO` and `D_LSLTO`.

### FORTRAN 77 Interface

Single: `CALL LSLTO (N, A, B, IPATH, X)`

Double: The double precision name is `DLSLTO`.

### Description

*Toeplitz matrices* have entries that are constant along each diagonal, for example,

$$A = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_{-1} & p_0 & p_1 & p_2 \\ p_{-2} & p_{-1} & p_0 & p_1 \\ p_{-3} & p_{-2} & p_{-1} & p_0 \end{bmatrix}$$

The routine `LSLTO` is based on the routine `TSLS` in the `TOEPLITZ` package, see Arushanian et al. (1983). It is based on an algorithm of Trench (1964). This algorithm is also described by Golub and van Loan (1983), pages 125–133.

### Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LTO/DL2LTO`. The reference is:

```
CALL L2LTO (N, A, B, IPATH, X, WK)
```

The additional argument is:

**WK** — Work vector of length  $2N - 2$ .

2. Because of the special structure of Toeplitz matrices, the first row and the first column of a Toeplitz matrix completely characterize the matrix. Hence, only the elements  $A(1, 1), \dots, A(1, N), A(2, 1), \dots, A(N, 1)$  need to be stored.

### Example

A system of four linear equations is solved. Note that only the first row and column of the matrix  $A$  are entered.

```
USE LSLTO_INT
USE WRRRN_INT
!
! Declare variables
INTEGER N
PARAMETER (N=4)
REAL A(2*N-1), B(N), X(N)
!
! Set values for A, and B
!
! A = (2 -3 -1 6)
! (1 2 -3 -1)
! (4 1 2 -3)
! (3 4 1 2)
!
! B = (16 -29 -7 5)
!
DATA A/2.0, -3.0, -1.0, 6.0, 1.0, 4.0, 3.0/
DATA B/16.0, -29.0, -7.0, 5.0/
!
! Solve AX = B
CALL LSLTO (A, B, X)
!
! Print results
CALL WRRRN ('X', X, 1, N, 1)
END
```

### Output

```

 X
 1 2 3 4
-2.000 -1.000 7.000 4.000
```

---

## LSLTC

Solves a complex Toeplitz linear system.

### Required Arguments

- $A$  — Complex vector of length  $2N - 1$  containing the first row of the coefficient matrix followed by its first column beginning with the second element. (Input)  
See Comment 2.

$B$  — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Complex vector of length  $N$  containing the solution of the linear system. (Output)

### Optional Arguments

$N$  — Order of the matrix represented by  $A$ . (Input)

Default:  $N = \text{size}(A,1)$ .

$IPATH$  — Integer flag. (Input)

$IPATH = 1$  means the system  $Ax = B$  is solved.

$IPATH = 2$  means the system  $A^T x = B$  is solved.

Default:  $IPATH = 1$ .

### FORTRAN 90 Interface

Generic: `CALL LSLTC (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLTC` and `D_LSLTC`.

### FORTRAN 77 Interface

Single: `CALL LSLTC (N, A, B, IPATH, X)`

Double: The double precision name is `DLSLTC`.

### Description

*Toeplitz matrices* have entries which are constant along each diagonal, for example,

$$A = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_{-1} & p_0 & p_1 & p_2 \\ p_{-2} & p_{-1} & p_0 & p_1 \\ p_{-3} & p_{-2} & p_{-1} & p_0 \end{bmatrix}$$

The routine `LSLTC` is based on the routine `TSLC` in the `TOEPLITZ` package, see Arushanian et al. (1983). It is based on an algorithm of Trench (1964). This algorithm is also described by Golub and van Loan (1983), pages 125–133.

### Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LTC/DL2LTC`. The reference is:

```
CALL L2LTC (N, A, B, IPATH, X, WK)
```

The additional argument is

**WK** — Complex work vector of length  $2N - 2$ .

- Because of the special structure of Toeplitz matrices, the first row and the first column of a Toeplitz matrix completely characterize the matrix. Hence, only the elements  $A(1, 1), \dots, A(1, N), A(2, 1), \dots, A(N, 1)$  need to be stored.

### Example

A system of four complex linear equations is solved. Note that only the first row and column of the matrix  $A$  are entered.

```

USE LSLTC_INT
USE WRCRN_INT
!
! Declare variables
PARAMETER (N=4)
COMPLEX A(2*N-1), B(N), X(N)
!
! Set values for A and B
!
! A = (2+2i -3 1+4i 6-2i)
! (i 2+2i -3 1+4i)
! (4+2i i 2+2i -3)
! (3-4i 4+2i i 2+2i)
!
! B = (6+65i -29-16i 7+i -10+i)
!
DATA A/(2.0,2.0), (-3.0,0.0), (1.0,4.0), (6.0,-2.0), (0.0,1.0), &
 (4.0,2.0), (3.0,-4.0)/
DATA B/(6.0,65.0), (-29.0,-16.0), (7.0,1.0), (-10.0,1.0)/
!
! Solve AX = B
CALL LSLTC (A, B, X)
!
! Print results
CALL WRCRN ('X', X, 1, N, 1)
END

```

### Output

```

 X
1 2 3 4
(-2.000, 0.000) (-1.000,-5.000) (7.000, 2.000) (0.000, 4.000)

```

## LSLCC



Solves a complex circulant linear system.

## Required Arguments

$A$  — Complex vector of length  $N$  containing the first row of the coefficient matrix. (Input)

$B$  — Complex vector of length  $N$  containing the right-hand side of the linear system. (Input)

$X$  — Complex vector of length  $N$  containing the solution of the linear system. (Output)

## Optional Arguments

$N$  — Order of the matrix represented by  $A$ . (Input)

Default:  $N = \text{size}(A,1)$ .

$IPATH$  — Integer flag. (Input)

$IPATH = 1$  means the system  $Ax = B$  is solved.

$IPATH = 2$  means the system  $A^T x = B$  is solved.

Default:  $IPATH = 1$ .

## FORTRAN 90 Interface

Generic: `CALL LSLCC (A, B, X [, ...])`

Specific: The specific interface names are `S_LSLCC` and `D_LSLCC`.

## FORTRAN 77 Interface

Single: `CALL LSLCC (N, A, B, IPATH, X)`

Double: The double precision name is `DLSLCC`.

## Description

*Circulant matrices* have the property that each row is obtained by shifting the row above it one place to the right. Entries that are shifted off at the right re-enter at the left. For example,

$$A = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_4 & p_1 & p_2 & p_3 \\ p_3 & p_4 & p_1 & p_2 \\ p_2 & p_3 & p_4 & p_1 \end{bmatrix}$$

If  $q_k = p_{-k}$  and the subscripts on  $p$  and  $q$  are interpreted modulo  $N$ , then

$$(Ax)_j = \sum_{i=1}^N p_{i-j+1} x_i = \sum_{i=1}^N q_{j-i+1} x_i = (q * x)_j$$

where  $q * x$  is the convolution of  $q$  and  $x$ . By the convolution theorem, if  $q * x = b$ , then

$$\hat{q} \otimes \hat{x} = \hat{b}, \text{ where } \hat{q}$$

is the discrete Fourier transform of  $q$  as computed by the IMSL routine `FFTCF` and  $\otimes$  denotes elementwise multiplication. By division,

$$\hat{x} = \hat{b} \oslash \hat{q}$$

where  $\oslash$  denotes elementwise division. The vector  $x$  is recovered from

$$\hat{x}$$

through the use of IMSL routine `FFTCB`.

To solve  $A^T x = b$ , use the vector  $p$  instead of  $q$  in the above algorithm.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LCC/DL2LCC`. The reference is:

```
CALL L2LCC (N, A, B, IPATH, X, ACOPI, WK)
```

The additional arguments are as follows:

**ACOPY** — Complex work vector of length  $N$ . If  $A$  is not needed, then  $A$  and  $ACOPY$  may be the same.

**WK** — Work vector of length  $6N + 15$ .

2. Informational error

Type Code

4 2 The input matrix is singular.

3. Because of the special structure of circulant matrices, the first row of a circulant matrix completely characterizes the matrix. Hence, only the elements  $A(1, 1), \dots, A(1, N)$  need to be stored.

## Example

A system of four linear equations is solved. Note that only the first row of the matrix  $A$  is entered.

```
USE LSLCC_INT
USE WRCRN_INT
!
! Declare variables
INTEGER N
PARAMETER (N=4)
COMPLEX A(N), B(N), X(N)
!
! Set values for A, and B
!
```

```

! A = (2+2i -3+0i 1+4i 6-2i)
!
! B = (6+65i -41-10i -8-30i 63-3i)
!
DATA A/(2.0,2.0), (-3.0,0.0), (1.0,4.0), (6.0,-2.0)/
DATA B/(6.0,65.0), (-41.0,-10.0), (-8.0,-30.0), (63.0,-3.0)/
! Solve AX = B (IPATH = 1)
CALL LSLCC (A, B, X)
! Print results
CALL WRCRN ('X', X, 1, N, 1)
END

```

## Output

```

 1 2 3 4
(-2.000, 0.000) (-1.000,-5.000) (7.000, 2.000) (0.000, 4.000)

```

---

## PCGRC

Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication.

### Required Arguments

**IDO** — Flag indicating task to be done. (Input/Output)

On the initial call `IDO` must be 0. If the routine returns with `IDO = 1`, then set  $Z = AP$ , where  $A$  is the matrix, and call `PCGRC` again. If the routine returns with `IDO = 2`, then set  $Z$  to the solution of the system  $MZ = R$ , where  $M$  is the preconditioning matrix, and call `PCGRC` again. If the routine returns with `IDO = 3`, then the iteration has converged and  $X$  contains the solution.

**X** — Array of length  $N$  containing the solution. (Input/Output)

On input,  $X$  contains the initial guess of the solution. On output,  $X$  contains the solution to the system.

**P** — Array of length  $N$ . (Output)

Its use is described under `IDO`.

**R** — Array of length  $N$ . (Input/Output)

On initial input, it contains the right-hand side of the linear system. On output, it contains the residual.

**Z** — Array of length  $N$ . (Input)

When `IDO = 1`, it contains  $AP$ , where  $A$  is the linear system. When `IDO = 2`, it contains the solution of  $MZ = R$ , where  $M$  is the preconditioning matrix. When `IDO = 0`, it is ignored. Its use is described under `IDO`.

### Optional Arguments

**N** — Order of the linear system. (Input)

Default:  $N = \text{size}(X,1)$ .

**RELERR** — Relative error desired. (Input)

Default: RELERR = 1.e-5 for single precision and 1.d-10 for double precision.

**ITMAX** — Maximum number of iterations allowed. (Input)

Default: ITMAX = N.

## FORTRAN 90 Interface

Generic: CALL PCGRC (IDO, X, P, R, Z [, ...])

Specific: The specific interface names are S\_PCGRC and D\_PCGRC.

## FORTRAN 77 Interface

Single: CALL PCGRC (IDO, N, X, P, R, Z, RELERR, ITMAX)

Double: The double precision name is DPCGRC.

## Description

Routine PCGRC solves the symmetric definite linear system  $Ax = b$  using the preconditioned conjugate gradient method. This method is described in detail by Golub and Van Loan (1983, Chapter 10), and in Hageman and Young (1981, Chapter 7).

The *preconditioning matrix*,  $M$ , is a matrix that approximates  $A$ , and for which the linear system  $Mz = r$  is easy to solve. These two properties are in conflict; balancing them is a topic of much current research.

The number of iterations needed depends on the matrix and the error tolerance RELERR. As a rough guide,  $ITMAX = N^{1/2}$  is often sufficient when  $N \gg 1$ . See the references for further information.

Let  $M$  be the preconditioning matrix, let  $b, p, r, x$  and  $z$  be vectors and let  $\tau$  be the desired relative error. Then the algorithm used is as follows.

$$\lambda = -1$$

$$p_0 = x_0$$

$$r_1 = b - Ap$$

For  $k = 1, \dots, itmax$

$$z_k = M^{-1}r_k$$

If  $k = 1$  then

$$\beta_k = 1$$

$$p_k = z_k$$

Else

$$\beta_k = z_k^T r_k / z_{k-1}^T r_{k-1}$$

$$p_k = z_k + \beta_k p_k$$

End if

$$z_k = Ap$$

$$\alpha_k = z_{k-1}^T r_{k-1} / z_k^T p_k$$

$$x_k = x_k + \alpha_k p_k$$

$$r_k = r_k - \alpha_k z_k$$

If ( $\|z_k\|_2 \leq \tau(1 - \lambda)\|x_k\|_2$ ) Then

Recompute  $\lambda$

If ( $\|z_k\|_2 \leq \tau(1 - \lambda)\|x_k\|_2$ ) Exit

End if

End loop

Here  $\lambda$  is an estimate of  $\lambda_{\max}(G)$ , the largest eigenvalue of the iteration matrix  $G = I - M^{-1}A$ . The stopping criterion is based on the result (Hageman and Young, 1981, pages 148–151)

$$\frac{\|x_k - x\|_M}{\|x\|_M} \leq \frac{1}{1 - \lambda_{\max}(G)} \frac{\|z_k\|_M}{\|x_k\|_M}$$

Where

$$\|x\|_M^2 = x^T Mx$$

It is known that

$$\lambda_{\max}(T_1) \leq \lambda_{\max}(T_2) \leq \dots \leq \lambda_{\max}(G) < 1$$

where the  $T_n$  are the symmetric, tridiagonal matrices

$$T_n = \begin{bmatrix} \mu_1 & \omega_2 & & & \\ \omega_2 & \mu_2 & \omega_3 & & \\ & \omega_3 & \mu_3 & \omega_4 & \\ & & & \ddots & \ddots & \ddots \\ & & & & & & \ddots \end{bmatrix}$$

with

$$\mu_k = 1 - \beta_k / \alpha_{k-1} - 1 / \alpha_k, \mu_1 = 1 - 1 / \alpha_1$$

and

$$\omega_k = \sqrt{\beta_k} / \alpha_{k-1}$$

The largest eigenvalue of  $T_k$  is found using the routine `EVASB`. Usually this eigenvalue computation is needed for only a few of the iterations.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `P2GRC/DP2GRC`. The reference is:

```
CALL P2GRC (IDO, N, X, P, R, Z, RELERR, ITMAX, TRI, WK,
 IWK)
```

The additional arguments are as follows:

**TRI** — Workspace of length  $2 * ITMAX$  containing a tridiagonal matrix (in band symmetric form) whose largest eigenvalue is approximately the same as the largest eigenvalue of the iteration matrix. The workspace arrays `TRI`, `WK` and `IWK` should not be changed between the initial call with `IDO = 0` and `PCGRC/DPCGRC` returning with `IDO = 3`.

**WK** — Workspace of length  $5 * ITMAX$ .

**IWK** — Workspace of length `ITMAX`.

2. Informational errors

| Type | Code |                                                     |
|------|------|-----------------------------------------------------|
| 4    | 1    | The preconditioning matrix is singular.             |
| 4    | 2    | The preconditioning matrix is not definite.         |
| 4    | 3    | The linear system is not definite.                  |
| 4    | 4    | The linear system is singular.                      |
| 4    | 5    | No convergence after <code>ITMAX</code> iterations. |

## Example

In this example, the solution to a linear system is found. The coefficient matrix  $A$  is stored as a full matrix. The preconditioning matrix is the diagonal of  $A$ . This is called the *Jacobi preconditioner*. It is also used by the IMSL routine [JCGRC](#).

```
 USE PCGRC_INT
 USE MURRV_INT
 USE WRRRN_INT
 USE SCOPY_INT
 INTEGER LDA, N
 PARAMETER (N=3, LDA=N)
!
 INTEGER IDO, ITMAX, J
 REAL A(LDA,N), B(N), P(N), R(N), X(N), Z(N)
!
! (1, -3, 2)
! A = (-3, 10, -5)
! (2, -5, 6)
 DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
! B = (27.0, -78.0, 64.0)
 DATA B/27.0, -78.0, 64.0/
!
! Set R to right side
 CALL SCOPY (N, B, 1, R, 1)
!
! Initial guess for X is B
 CALL SCOPY (N, B, 1, X, 1)
!
 ITMAX = 100
 IDO = 0
10 CALL PCGRC (IDO, X, P, R, Z, ITMAX=ITMAX)
 IF (IDO .EQ. 1) THEN
!
! Set z = Ap
 CALL MURRV (A, P, Z)
 GO TO 10
 ELSE IF (IDO .EQ. 2) THEN
!
! Use diagonal of A as the
! preconditioning matrix M
! and set z = inv(M)*r
 DO 20 J=1, N
 Z(J) = R(J)/A(J,J)
20 CONTINUE
 GO TO 10
 END IF
!
! Print the solution
 CALL WRRRN ('Solution', X)
!
 END
```

## Output

```
Solution
1 1.001
2 -4.000
3 7.000
```



```

-1.0, 0.0, -1.0, 4.0, 3*0.0/
! Set PRECND in band symmetric form
DATA PRECND/0.0, 4.0, -1.0, 4.0, -1.0, 4.0, -1.0, 4.0, -1.0, 4.0, &
-1.0, 4.0, -1.0, 4.0, -1.0, 4.0, -1.0, 4.0/
! Right side is (1, ..., 1)
R = 1.0E0
! Initial guess for X is 0
X = 0.0E0
! Factor the preconditioning matrix
CALL LFCQS (PRECND, NCOPRE, PREFAC, RCOND)
!
ITMAX = 100
RELERR = 1.0E-4
IDO = 0
10 CALL PCGRC (IDO, X, P, R, Z, RELERR=RELERR, ITMAX=ITMAX)
IF (IDO .EQ. 1) THEN
! Set z = Ap
CALL MURBV (A, NCODA, NCODA, P, Z)
GO TO 10
ELSE IF (IDO .EQ. 2) THEN
! Solve PRECND*z = r for r
CALL LSLQS (PREFAC, NCOPRE, R, Z)
GO TO 10
END IF
! Print the solution
CALL WRRRN ('Solution', X)
!
END

```

## Output

```

Solution
1 0.955
2 1.241
3 1.349
4 1.578
5 1.660
6 1.578
7 1.349
8 1.241
9 0.955

```

---

## JCGRC

Solves a real symmetric definite linear system using the Jacobi-preconditioned conjugate gradient method with reverse communication.

### Required Arguments

**IDO** — Flag indicating task to be done. (Input/Output)

On the initial call IDO must be 0. If the routine returns with IDO = 1, then set

$Z = A * P$ , where A is the matrix, and call JCGRC again. If the routine returns with

IDO = 2, then the iteration has converged and X contains the solution.

**DIAGNL** — Vector of length  $N$  containing the diagonal of the matrix. (Input)  
Its elements must be all strictly positive or all strictly negative.

**X** — Array of length  $N$  containing the solution. (Input/Output)  
On input,  $X$  contains the initial guess of the solution. On output,  $X$  contains the solution to the system.

**P** — Array of length  $N$ . (Output)  
Its use is described under `IDO`.

**R** — Array of length  $N$ . (Input/Output)  
On initial input, it contains the right-hand side of the linear system. On output, it contains the residual.

**Z** — Array of length  $N$ . (Input)  
When `IDO = 1`, it contains  $AP$ , where  $A$  is the linear system. When `IDO = 0`, it is ignored. Its use is described under `IDO`.

### Optional Arguments

**N** — Order of the linear system. (Input)  
Default: `N = size(X,1)`.

**RELERR** — Relative error desired. (Input)  
Default: `RELERR = 1.e-5` for single precision and `1.d-10` for double precision.

**ITMAX** — Maximum number of iterations allowed. (Input)  
Default: `ITMAX = 100`.

### FORTRAN 90 Interface

Generic: `CALL JCGRC (IDO, DIAGNL, X, P, R, Z [, ...])`

Specific: The specific interface names are `S_JCGRC` and `D_JPCGRC`.

### FORTRAN 77 Interface

Single: `CALL JCGRC (IDO, N, DIAGNL, X, P, R, Z, RELERR, ITMAX)`

Double: The double precision name is `DJCGRC`.

### Description

Routine `JCGRC` solves the symmetric definite linear system  $Ax = b$  using the Jacobi conjugate gradient method. This method is described in detail by Golub and Van Loan (1983, Chapter 10), and in Hageman and Young (1981, Chapter 7).

This routine is a special case of the routine `PCGRC`, with the diagonal of the matrix `A` used as the preconditioning matrix. For details of the algorithm see `PCGRC`.

The number of iterations needed depends on the matrix and the error tolerance `RELERR`. As a rough guide, `ITMAX = N` is often sufficient when  $N \gg 1$ . See the references for further information.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `J2GRC/DJ2GRC`. The reference is:

```
CALL J2GRC (IDO, N, DIAGNL, X, P, R, Z, RELERR, ITMAX, TRI, WK, IWK)
```

The additional arguments are as follows:

**TRI** — Workspace of length  $2 * ITMAX$  containing a tridiagonal matrix (in band symmetric form) whose largest eigenvalue is approximately the same as the largest eigenvalue of the iteration matrix. The workspace arrays `TRI`, `WK` and `IWK` should not be changed between the initial call with `IDO = 0` and `JCGRC/DJCGRC` returning with `IDO = 2`.

**WK** — Workspace of length  $5 * ITMAX$ .

**IWK** — Workspace of length `ITMAX`.

2. Informational errors

| Type | Code | Description                                         |
|------|------|-----------------------------------------------------|
| 4    | 1    | The diagonal contains a zero.                       |
| 4    | 2    | The diagonal elements have different signs.         |
| 4    | 3    | No convergence after <code>ITMAX</code> iterations. |
| 4    | 4    | The linear system is not definite.                  |
| 4    | 5    | The linear system is singular.                      |

## Example

In this example, the solution to a linear system is found. The coefficient matrix `A` is stored as a full matrix.

```
USE IMSL_LIBRARIES
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
!
INTEGER IDO, ITMAX
REAL A(LDA,N), B(N), DIAGNL(N), P(N), R(N), X(N), &
Z(N)
!
!
!
DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
```

```

! B = (27.0, -78.0, 64.0)
DATA B/27.0, -78.0, 64.0/
! Set R to right side
CALL SCOPY (N, B, 1, R, 1)
! Initial guess for X is B
CALL SCOPY (N, B, 1, X, 1)
! Copy diagonal of A to DIAGNL
CALL SCOPY (N, A(:, 1), LDA+1, DIAGNL, 1)
! Set parameters
ITMAX = 100
IDO = 0
10 CALL JCGRC (IDO, DIAGNL, X, P, R, Z, ITMAX=ITMAX)
IF (IDO .EQ. 1) THEN
! Set z = Ap
CALL MURRV (A, P, Z)
GO TO 10
END IF
! Print the solution
CALL WRRRN ('Solution', X)
!
END

```

## Output

```

Solution
1 1.001
2 -4.000
3 7.000

```

---

## GMRES

Uses the Generalized Minimal Residual Method with reverse communication to generate an approximate solution of  $Ax = b$ .

### Required Arguments

**IDO**— Flag indicating task to be done. (Input/Output)

On the initial call `IDO` must be 0. If the routine returns with `IDO = 1`, then set  $Z = AP$ , where  $A$  is the matrix, and call `GMRES` again. If the routine returns with `IDO = 2`, then set  $Z$  to the solution of the system  $MZ = P$ , where  $M$  is the preconditioning matrix, and call `GMRES` again. If the routine returns with `IDO = 3`, set  $Z = AM^{-1}P$ , and call `GMRES` again. If the routine returns with `IDO = 4`, the iteration has converged, and `X` contains the approximate solution to the linear system.

**X**— Array of length `N` containing an approximate solution. (Input/Output)

On input, `X` contains an initial guess of the solution. On output, `X` contains the approximate solution.

**P**— Array of length `N`. (Output)

Its use is described under `IDO`.

**R** — Array of length  $N$ . (Input/Output)  
On initial input, it contains the right-hand side of the linear system. On output, it contains the residual,  $b - Ax$ .

**Z** — Array of length  $N$ . (Input)  
When  $IDO = 1$ , it contains  $AP$ , where  $A$  is the coefficient matrix. When  $IDO = 2$ , it contains  $M^T P$ . When  $IDO = 3$ , it contains  $AM^T P$ . When  $IDO = 0$ , it is ignored.

**TOL** — Stopping tolerance. (Input/Output)  
The algorithm attempts to generate a solution  $x$  such that  $|b - Ax| \leq TOL * |b|$ . On output, **TOL** contains the final residual norm.

### Optional Arguments

**N** — Order of the linear system. (Input)  
Default:  $N = \text{size}(X,1)$ .

### FORTRAN 90 Interface

Generic:     CALL GMRES (IDO, X, P, R, Z, TOL [, ...])

Specific:    The specific interface names are `S_GMRES` and `D_GMRES`.

### FORTRAN 77 Interface

Single:      CALL GMRES (IDO, N, X, P, R, Z, TOL)

Double:     The double precision name is `DGMRES`.

### Description

The routine `GMRES` implements restarted `GMRES` with reverse communication to generate an approximate solution to  $Ax = b$ . It is based on `GMRES` by Homer Walker.

There are four distinct `GMRES` implementations, selectable through the parameter vector `INFO`. The first Gram-Schmidt implementation, `INFO(1) = 1`, is essentially the original algorithm by Saad and Schultz (1986). The second Gram-Schmidt implementation, developed by Homer Walker and Lou Zhou, is simpler than the first implementation. The least squares problem is constructed in upper-triangular form and the residual vector updating at the end of a `GMRES` cycle is cheaper. The first Householder implementation is algorithm 2.2 of Walker (1988), but with more efficient correction accumulation at the end of each `GMRES` cycle. The second Householder implementation is algorithm 3.1 of Walker (1988). The products of Householder transformations are expanded as sums, allowing most work to be formulated as large scale matrix-vector operations. Although BLAS are used wherever possible, extensive use of Level 2 BLAS in the second Householder implementation may yield a performance advantage on certain computing environments.

The Gram-Schmidt implementations are less expensive than the Householder, the latter requiring about twice as much arithmetic beyond the coefficient matrix/vector products. However, the Householder implementations may be more reliable near the limits of residual reduction. See

Walker (1988) for details. Issues such as the cost of coefficient matrix/vector products, availability of effective preconditioners, and features of particular computing environments may serve to mitigate the extra expense of the Householder implementations.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `G2RES/DG2RES`. The reference is:

```
CALL G2RES (IDO, N, X, P, R, Z, TOL, INFO, USRNPR,
 USRNRN, WORK)
```

The additional arguments are as follows:

**INFO** — Integer vector of length 10 used to change parameters of `GMRES`. (Input/Output).

For any components `INFO(1) ... INFO(7)` with value zero on input, the default value is used.

`INFO(1) = IMP`, the flag indicating the desired implementation.

| IMP | Action                             |
|-----|------------------------------------|
| 1   | first Gram-Schmidt implementation  |
| 2   | second Gram-Schmidt implementation |
| 3   | first Householder implementation   |
| 4   | second Householder implementation  |

Default: `IMP = 1`

`INFO(2) = KDMAX`, the maximum Krylov subspace dimension, i.e., the maximum allowable number of `GMRES` iterations before restarting. It must satisfy  $1 \leq \text{KDMAX} \leq N$ .  
Default: `KDMAX = min(N, 20)`

`INFO(3) = ITMAX`, the maximum number of `GMRES` iterations allowed.  
Default: `ITMAX = 1000`

`INFO(4) = IRP`, the flag indicating whether right preconditioning is used. If `IRP = 0`, no right preconditioning is performed. If `IRP = 1`, right preconditioning is performed. If `IRP = 0`, then `IDO = 2` or `3` will not occur.  
Default: `IRP = 0`

`INFO(5) = IRESUP`, the flag that indicates the desired residual vector updating prior to restarting or on termination.

| IRESUP | Action                                                   |
|--------|----------------------------------------------------------|
| 1      | update by linear combination, restarting only            |
| 2      | update by linear combination, restarting and termination |
| 3      | update by direct evaluation, restarting only             |
| 4      | update by direct evaluation, restarting and termination  |

Updating by direct evaluation requires an otherwise unnecessary matrix-vector product. The alternative is to update by forming a linear combination of various available vectors. This may or may not be cheaper and may be less reliable if the residual vector has been greatly reduced. If IRESUP = 2 or 4, then the residual vector is returned in WORK(1), ..., WORK(N). This is useful in some applications but costs another unnecessary residual update. It is recommended that IRESUP = 1 or 2 be used, unless matrix-vector products are inexpensive or great residual reduction is required. In this case use IRESUP = 3 or 4. The meaning of “inexpensive” varies with IMP as follows:

|     |                           |
|-----|---------------------------|
| IMP | ≤                         |
| 1   | (KDMAX + 1) * N flops     |
| 2   | N flops                   |
| 3   | (2 * KDMAX + 1) * N flops |
| 4   | (2 * KDMAX + 1) * N flops |

“Great residual reduction” means that TOL is only a few orders of magnitude larger than machine epsilon.

Default: IRESUP = 1

INFO(6) = flag for indicating the inner product and norm used in the Gram-Schmidt implementations. If INFO(6) = 0, sdot and snrm2, from BLAS, are used. If INFO(6) = 1, the user must provide the routines, as specified under arguments USRNPR and USRNRM.

Default: INFO(6) = 0

INFO(7) = IPRINT, the print flag. If IPRINT = 0, no printing is performed. If IPRINT = 1, print the iteration numbers and residuals.

Default: IPRINT = 0

INFO(8) = the total number of GMRES iterations on output.

INFO(9) = the total number of matrix-vector products in GMRES on output.

INFO(10) = the total number of right preconditioner solves in GMRES on output if IRP = 1.

**USRNPR** — User-supplied FUNCTION to use as the inner product in the Gram-Schmidt implementation, if `INFO(6) = 1`. If `INFO(6) = 0`, the dummy function `G8RES/DG8RES` may be used. The usage is

```
REAL FUNCTION USRNPR (N, SX, INCX, SY, INCY)
```

`N` — Length of vectors `X` and `Y`. (Input)

`SX` — Real vector of length `MAX(N*IABS(INCX), 1)`. (Input)

`INCX` — Displacement between elements of `SX`. (Input)

`X(I)` is defined to be `SX(1+(I-1)*INCX)` if `INCX` is greater than 0,  
or  
`SX(1+(I-N)*INCX)` if `INCX` is less than 0.

`SY` — Real vector of length `MAX(N*IABS(INCY), 1)`. (Input)

`INCY` — Displacement between elements of `SY`. (Input)

`Y(I)` is defined to be `SY(1+(I-1)*INCY)` if `INCY` is greater than 0, or  
`SY(1+(I-N)*INCY)` if `INCY` is less than zero.  
`USRNPR` must be declared `EXTERNAL` in the calling program.

**USRNRM** — User-supplied FUNCTION to use as the norm  $\|X\|$  in the Gram-Schmidt implementation, if `INFO(6) = 1`. If `INFO(6) = 0`, the dummy function `G9RES/DG9RES` may be used. The usage is

```
REAL FUNCTION USRNRM (N, SX, INCX)
```

`N` — Length of vectors `X` and `Y`. (Input)

`SX` — Real vector of length `MAX(N*IABS(INCX), 1)`. (Input)

`INCX` — Displacement between elements of `SX`. (Input)

`X(I)` is defined to be `SX(1+(I-1)*INCX)` if `INCX` is greater than 0, or  
`SX(1+(I-N)*INCX)` if `INCX` is less than 0.  
`USRNRM` must be declared `EXTERNAL` in the calling program.

**WORK** — Work array whose length is dependent on the chosen implementation.

| IMP | length of <i>WORK</i>                        |
|-----|----------------------------------------------|
| 1   | $N * (KDMAX + 2) + KDMAX**2 + 3 * KDMAX + 2$ |
| 2   | $N * (KDMAX + 2) + KDMAX**2 + 2 * KDMAX + 1$ |
| 3   | $N * (KDMAX + 2) + 3 * KDMAX + 2$            |
| 4   | $N * (KDMAX + 2) + KDMAX**2 + 2 * KDMAX + 2$ |

## Example 1

This is a simple example of GMRES usage. A solution to a small linear system is found. The coefficient matrix  $A$  is stored as a full matrix, and no preconditioning is used. Typically, preconditioning is required to achieve convergence in a reasonable number of iterations.

```

 USE IMSL_LIBRARIES
! Declare variables
 INTEGER LDA, N
 PARAMETER (N=3, LDA=N)
!
! Specifications for local variables
 INTEGER IDO, NOUT
 REAL P(N), TOL, X(N), Z(N)
 REAL A(LDA,N), R(N)
 SAVE A, R
!
! Specifications for intrinsics
 INTRINSIC SQRT
 REAL SQRT
!
! (33.0 16.0 72.0)
! A = (-24.0 -10.0 -57.0)
! (18.0 -11.0 7.0)
!
! B = (129.0 -96.0 8.5)
!
 DATA A/33.0, -24.0, 18.0, 16.0, -10.0, -11.0, 72.0, -57.0, 7.0/
 DATA R/129.0, -96.0, 8.5/
!
 CALL UMACH (2, NOUT)
!
! Initial guess = (0 ... 0)
!
 X = 0.0E0
!
! Set stopping tolerance to
! square root of machine epsilon
 TOL = AMACH(4)
 TOL = SQRT(TOL)
 IDO = 0
10 CONTINUE
 CALL GMRES (IDO, X, P, R, Z, TOL)
 IF (IDO .EQ. 1) THEN
! Set z = A*p
 CALL MURRV (A, P, Z)
 GO TO 10
 END IF
!
 CALL WRRRN ('Solution', X, 1, N, 1)
 WRITE (NOUT, '(A11, E15.5)') 'Residual = ', TOL
 END

```

## Output

```
 Solution
 1 2 3
1.000 1.500 1.000
Residual = 0.29746E-05
```

## Additional Examples

### Example 2

This example solves a linear system with a coefficient matrix stored in coordinate form, the same problem as in the document example for [LSLXG](#). Jacobi preconditioning is used, i.e. the preconditioning matrix  $M$  is the diagonal matrix with  $M_{ii} = A_{ii}$ , for  $i = 1, \dots, n$ .

```
 USE IMSL_LIBRARIES
 INTEGER N, NZ

 PARAMETER (N=6, NZ=15)

!
! Specifications for local variables
 INTEGER IDO, INFO(10), NOUT
 REAL P(N), TOL, WORK(1000), X(N), Z(N)
 REAL DIAGIN(N), R(N)
!
! Specifications for intrinsics
 INTRINSIC SQRT
 REAL SQRT
!
! Specifications for subroutines
 EXTERNAL AMULTP
!
! Specifications for functions
 EXTERNAL G8RES, G9RES
!
 DATA DIAGIN/0.1, 0.1, 0.0666667, 0.1, 1.0, 0.1666667/
 DATA R/10.0, 7.0, 45.0, 33.0, -34.0, 31.0/
!
 CALL UMACH (2, NOUT)
!
! Initial guess = (1 ... 1)
 X = 1.0E0
!
! Set up the options vector INFO
! to use preconditioning
 INFO = 0
 INFO(4) = 1
!
! Set stopping tolerance to
! square root of machine epsilon
 TOL = AMACH(4)
 TOL = SQRT(TOL)
 IDO = 0
10 CONTINUE
 CALL G2RES (IDO, N, X, P, R, Z, TOL, INFO, G8RES, G9RES, WORK)
 IF (IDO .EQ. 1) THEN
!
! Set z = A*p
 CALL AMULTP (P, Z)
 GO TO 10
 ELSE IF (IDO .EQ. 2) THEN
```

```

!
!
! Set z = inv(M)*p
! The diagonal of inv(M) is stored
! in DIAGIN
!
 CALL SHPROD (N, DIAGIN, 1, P, 1, Z, 1)
 GO TO 10
ELSE IF (IDO .EQ. 3) THEN
!
! Set z = A*inv(M)*p
!
 CALL SHPROD (N, DIAGIN, 1, P, 1, Z, 1)
 P = Z
 CALL AMULTP (P, Z)
 GO TO 10
END IF
!
CALL WRRRN ('Solution', X)
WRITE (NOUT, '(A11, E15.5)') 'Residual = ', TOL
END
!
SUBROUTINE AMULTP (P, Z)
USE IMSL_LIBRARIES
INTEGER NZ
PARAMETER (NZ=15)
!
REAL P(*), Z(*)
!
INTEGER N
PARAMETER (N=6)
!
INTEGER I
INTEGER IROW(NZ), JCOL(NZ)
REAL A(NZ)
SAVE A, IROW, JCOL
!
! SPECIFICATIONS FOR ARGUMENTS
! SPECIFICATIONS FOR PARAMETERS
! SPECIFICATIONS FOR LOCAL VARIABLES
! SPECIFICATIONS FOR SUBROUTINES
! Define the matrix A
!
DATA A/6.0, 10.0, 15.0, -3.0, 10.0, -1.0, -1.0, -3.0, -5.0, 1.0, &
 10.0, -1.0, -2.0, -1.0, -2.0/
DATA IROW/6, 2, 3, 2, 4, 4, 5, 5, 5, 5, 1, 6, 6, 2, 4/
DATA JCOL/6, 2, 3, 3, 4, 5, 1, 6, 4, 5, 1, 1, 2, 4, 1/
!
CALL SSET(N, 0.0, Z, 1)
!
! Accumulate the product A*p in z
DO 10 I=1, NZ
 Z(IROW(I)) = Z(IROW(I)) + A(I)*P(JCOL(I))
10 CONTINUE
RETURN
END

```

## Output

```

Solution
1 1.000
2 2.000
3 3.000

```

```

4 4.000
5 5.000
6 6.000
Residual = 0.25882E-05

```

### Example 3

The coefficient matrix in this example corresponds to the five-point discretization of the 2-d Poisson equation with the Dirichlet boundary condition. Assuming the natural ordering of the unknowns, and moving all boundary terms to the right hand side, we obtain the block tridiagonal matrix

$$A = \begin{bmatrix} T & -I & & & \\ -I & \ddots & \ddots & & \\ & \ddots & \ddots & -I & \\ & & -I & T & \end{bmatrix}$$

where

$$T = \begin{bmatrix} 4 & -1 & & & \\ -1 & \ddots & \ddots & & \\ & \ddots & \ddots & -1 & \\ & & -1 & 4 & \end{bmatrix}$$

and  $I$  is the identity matrix. Discretizing on a  $k \times k$  grid implies that  $T$  and  $I$  are both  $k \times k$ , and thus the coefficient matrix  $A$  is  $k^2 \times k^2$ .

The problem is solved twice, with discretization on a  $50 \times 50$  grid. During both solutions, use the second Householder implementation to take advantage of the large scale matrix/vector operations done in Level 2 BLAS. Also choose to update the residual vector by direct evaluation since the small tolerance will require large residual reduction.

The first solution uses no preconditioning. For the second solution, we construct a block diagonal preconditioning matrix

$$M = \begin{bmatrix} T & & \\ & \ddots & \\ & & T \end{bmatrix}$$

$M$  is factored once, and these factors are used in the forward solves and back substitutions necessary when GMRES returns with `IDO = 2` or `3`.

Timings are obtained for both solutions, and the ratio of the time for the solution with no preconditioning to the time for the solution with preconditioning is printed. Though the exact results are machine dependent, we see that the savings realized by faster convergence from using a preconditioner exceed the cost of factoring  $M$  and performing repeated forward and back solves.

```

USE IMSL_LIBRARIES
INTEGER K, N
PARAMETER (K=50, N=K*K)

!
! Specifications for local variables
INTEGER IDO, INFO(10), IR(20), IS(20), NOUT
REAL A(2*N), B(2*N), C(2*N), G8RES, G9RES, P(2*N), R(N), &
 TNOPRE, TOL, TPRES, U(2*N), WORK(100000), X(N), &
 Y(2*N), Z(2*N)

!
! Specifications for subroutines
EXTERNAL AMULTP, G8RES, G9RES

!
! Specifications for functions
CALL UMACH (2, NOUT)

!
! Right hand side and initial guess
! to (1 ... 1)
R = 1.0E0
X = 1.0E0

!
! Use the 2nd Householder
! implementation and update the
! residual by direct evaluation
INFO = 0
INFO(1) = 4
INFO(5) = 3
TOL = AMACH(4)
TOL = 100.0*TOL
IDO = 0

!
! Time the solution with no
! preconditioning
TNOPRE = CPSEC()
10 CONTINUE
CALL G2RES (IDO, N, X, P, R, Z, TOL, INFO, G8RES, G9RES, WORK)
IF (IDO .EQ. 1) THEN

!
! Set z = A*p
!
CALL AMULTP (K, P, Z)
GO TO 10
END IF
TNOPRE = CPSEC() - TNOPRE

!
WRITE (NOUT, '(A32, I4)') 'Iterations, no preconditioner = ', &
INFO(8)

!
! Solve again using the diagonal blocks
! of A as the preconditioning matrix M
R = 1.0E0
X = 1.0E0

!
! Define M
CALL SSET (N-1, -1.0, B, 1)
CALL SSET (N-1, -1.0, C, 1)
CALL SSET (N, 4.0, A, 1)
INFO(4) = 1
TOL = AMACH(4)
TOL = 100.0*TOL
IDO = 0
TPRES = CPSEC()

!
! Compute the LDU factorization of M

```



```

CALL SAXPY (N-K, -1.0, P((K+1):(N)), 1, Z, 1)
CALL SAXPY (N-K, -1.0, P, 1, Z((K+1):(N)), 1)
!
RETURN
END

```

## Output

```

Iterations, no preconditioner = 329
Iterations, with preconditioning = 192
(Precondition time)/(No precondition time) = 0.66278

```

---

## ARPACK\_SVD

Computes some singular values and left and right singular vectors of a real rectangular matrix  $A_{M \times N} = USV^T$ . There is no restriction on the relative sizes,  $M$  and  $N$ . The user supplies matrix-vector products  $y = Ax$  and  $y = A^T x$  for the iterative method. This routine calls [ARPACK\\_SYMMETRIC](#). Descriptions for both [ARPACK\\_SVD](#) and [ARPACK\\_SYMMETRIC](#) are found in Chapter 2, “Eigensystem Analysis”.

---

## LSQRR



Solves a linear least-squares problem without iterative refinement.

### Required Arguments

- A** — NRA by NCA matrix containing the coefficient matrix of the least-squares system to be solved. (Input)
- B** — Vector of length NRA containing the right-hand side of the least-squares system. (Input)
- X** — Vector of length NCA containing the solution vector with components corresponding to the columns not used set to zero. (Output)
- RES** — Vector of length NRA containing the residual vector  $B - A * X$ . (Output)
- KBASIS** — Scalar containing the number of columns used in the solution.

### Optional Arguments

- NRA** — Number of rows of A. (Input)  
Default: NRA = size (A,1).

*NCA* — Number of columns of *A*. (Input)

Default: *NCA* = size (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = size (*A*,1).

*TOL* — Scalar containing the nonnegative tolerance used to determine the subset of columns of *A* to be included in the solution. (Input)

If *TOL* is zero, a full complement of min(*NRA*, *NCA*) columns is used. See [Comments](#).

Default: *TOL* = 0.0

### **FORTRAN 90 Interface**

Generic:     CALL LSQRR (A, B, X, RES, KBASIS [, ...])

Specific:    The specific interface names are S\_LSQRR and D\_LSQRR.

### **FORTRAN 77 Interface**

Single:     CALL LSQRR (NRA, NCA, A, LDA, B, TOL, X, RES, KBASIS)

Double:     The double precision name is DLSQRR.

### **ScaLAPACK Interface**

Generic:     CALL LSQRR (A0, B0, X0, RES0, KBASIS [, ...])

Specific:    The specific interface names are S\_LSQRR and D\_LSQRR.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

Routine LSQRR solves the linear least-squares problem. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. The routine LQRRR is first used to compute the *QR* decomposition of *A*. Pivoting, with all rows free, is used. Column *k* is in the basis if

$$|R_{kk}| \leq \tau |R_{11}|$$

with  $\tau = \text{TOL}$ . The truncated least-squares problem is then solved using IMSL routine LQRSL. Finally, the components in the solution, with the same index as columns that are not in the basis, are set to zero; and then, the permutation determined by the pivoting in IMSL routine LQRRR is applied.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2QRR/DL2QRR. The reference is:

```
CALL L2QRR (NRA, NCA, A, LDA, B, TOL, X, RES, KBASIS, QR,
 QRAUX, IPVT, WORK)
```

The additional arguments are as follows:

**QR** — Work vector of length  $NRA * NCA$  representing an  $NRA$  by  $NCA$  matrix that contains information from the  $QR$  factorization of  $A$ . The upper trapezoidal part of  $QR$  contains the upper trapezoidal part of  $R$  with its diagonal elements ordered in decreasing magnitude. The strict lower trapezoidal part of  $QR$  contains information to recover the orthogonal matrix  $Q$  of the factorization. If  $A$  is not needed,  $QR$  can share the same storage locations as  $A$ .

**QRAUX** — Work vector of length  $NCA$  containing information about the orthogonal factor of the  $QR$  factorization of  $A$ .

**IPVT** — Integer work vector of length  $NCA$  containing the pivoting information for the  $QR$  factorization of  $A$ .

**WORK** — Work vector of length  $2 * NCA - 1$ .

2. Routine LSQRR calculates the  $QR$  decomposition with pivoting of a matrix  $A$  and tests the diagonal elements against a user-supplied tolerance  $TOL$ . The first integer  $KBASIS = k$  is determined for which

$$|r_{k+1,k+1}| \leq TOL * |r_{11}|$$

In effect, this condition implies that a set of columns with a condition number approximately bounded by  $1.0/TOL$  is used. Then, LQRSL performs a truncated fit of the first  $KBASIS$  columns of the permuted  $A$  to an input vector  $B$ . The coefficient of this fit is unscrambled to correspond to the original columns of  $A$ , and the coefficients corresponding to unused columns are set to zero. It may be helpful to scale the rows and columns of  $A$  so that the error estimates in the elements of the scaled matrix are roughly equal to  $TOL$ .

3. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine L2QRR the leading dimension of  $QR$  is increased by  $IVAL(3)$  when  $N$  is a multiple of  $IVAL(4)$ . The values  $IVAL(3)$  and  $IVAL(4)$  are temporarily replaced by  $IVAL(1)$  and  $IVAL(2)$ , respectively, in LSQRR. Additional memory allocation for  $QR$  and option value restoration are done automatically in LSQRR. Users directly calling L2QRR can allocate additional space for  $QR$  and set  $IVAL(3)$  and  $IVAL(4)$  so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing

applications that use LSQRR or L2QRR. Default values for the option are IVAL(\*) = 1, 16, 0, 1.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSQRR temporarily replaces IVAL(2) by IVAL(1). The routine L2CRG computes the condition number if IVAL(2) = 2. Otherwise L2CRG skips this computation. LSQRR restores the option. Default values for the option are IVAL(\*) = 1, 2.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix A. A contains the coefficient matrix of the least squares system to be solved. (Input)
- B0** — Local vector of length MXLDA containing the local portions of the distributed vector B. B contains the right-hand side of the least squares system. (Input)
- X0** — Local vector of length MXLDX containing the local portions of the distributed vector X. X contains the solution vector with components corresponding to the columns not used set to zero. (Output)
- RES0** — Local vector of length MXLDA containing the local portions of the distributed vector RES. RES contains the residual vector  $B - A * X$ . (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA, MXLDX, and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

Consider the problem of finding the coefficients  $c_i$  in

$$f(x) = c_0 + c_1x + c_2x^2$$

given data at  $x = 1, 2, 3$  and  $4$ , using the method of least squares. The row of the matrix  $A$  contains the value of  $1, x$  and  $x^2$  at the data points. The vector  $b$  contains the data, chosen such that

$c_0 \approx 1, c_1 \approx 2$  and  $c_2 \approx 0$ . The routine LSQRR solves this least-squares problem.

```

USE LSQRR_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (NRA=4, NCA=3, LDA=NRA)
REAL A(LDA,NCA), B(NRA), X(NCA), RES(NRA), TOL
!
! Set values for A
!
! A = (1 2 4)
!

```

```

! (1 4 16)
! (1 6 36)
! (1 8 64)
!
DATA A/4*1.0, 2.0, 4.0, 6.0, 8.0, 4.0, 16.0, 36.0, 64.0/
!
! Set values for B
!
DATA B/ 4.999, 9.001, 12.999, 17.001 /
!
! Solve the least squares problem
TOL = 1.0E-4
CALL LSQRR (A, B, X, RES, KBASIS, TOL=TOL)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'KBASIS = ', KBASIS
CALL WRRRN ('X', X, 1, NCA, 1)
CALL WRRRN ('RES', RES, 1, NRA, 1)
!
END

```

## Output

```

KBASIS = 3

 X
 1 2 3
0.999 2.000 0.000

 RES
 1 2 3 4
-0.000400 0.001200 -0.001200 0.000400

```

## ScaLAPACK Example

The previous example is repeated here as a distributed computing example. Consider the problem of finding the coefficients  $c_i$  in

$$f(x) = c_0 + c_1x + c_2x^2$$

given data at  $x = 1, 2, 3$  and  $4$ , using the method of least squares. The row of the matrix  $A$  contains the value of  $1, x$  and  $x^2$  at the data points. The vector  $b$  contains the data, chosen such that

$c_0 \approx 1, c_1 \approx 2$  and  $c_2 \approx 0$ . The routine LSQRR solves this least-squares problem. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LSQRR_INT
USE UMACH_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables

```

```

INTEGER LDA, NRA, NCA, DESCA(9), DESCX(9), DESCR(9)
INTEGER INFO, KBASIS, MXCOL, MXLDA, MXCOLX, MXLDX, NOUT
REAL TOL
REAL, ALLOCATABLE :: A(:, :), B(:), X(:), RES(:)
REAL, ALLOCATABLE :: A0(:, :), B0(:), X0(:), RES0(:)
PARAMETER (NRA=4, NCA=3, LDA=NRA)
!
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,NCA), B(NRA), X(NCA), RES(NRA))
!
! Set values for A and B
 A(1,:) = (/ 1.0, 2.0, 4.0/)
 A(2,:) = (/ 1.0, 4.0, 16.0/)
 A(3,:) = (/ 1.0, 6.0, 36.0/)
 A(4,:) = (/ 1.0, 8.0, 64.0/)
!
 B = (/4.999, 9.001, 12.999, 17.001/)
ENDIF
!
! Set up a 2D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA, .TRUE., .FALSE.)
!
! Get the array descriptor entities MXLDA,
! MXCOL, MXLDX, and MXCOLX
CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
CALL SCALAPACK_GETDIM(NCA, 1, MP_NB, 1, MXLDX, MXCOLX)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, &
INFO)
CALL DESCINIT(DESCX, NCA, 1, MP_NB, 1, 0, 0, MP_ICTXT, MXLDX, INFO)
CALL DESCINIT(DESCR, NRA, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), B0(MXLDA), X0(MXLDX), RES0(MXLDA))
!
! Map input arrays to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
CALL SCALAPACK_MAP(B, DESCR, B0)
!
! Solve the least squares problem
TOL = 1.0E-4
CALL LSQRR (A0, B0, X0, RES0, KBASIS, TOL=TOL)
!
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
CALL SCALAPACK_UNMAP(RES0, DESCR, RES)
!
! Print results.
! Only Rank=0 has the solution.
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT,*) `KBASIS = `, KBASIS
 CALL WRRRN ('X', X, 1, NCA, 1)
 CALL WRRRN ('RES', RES, 1, NRA, 1)
ENDIF
IF (MP_RANK .EQ. 0) DEALLOCATE(A, B, RES, X)
DEALLOCATE(A0, B0, RES0, X0)
!
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)

```

```

! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
 END

```

## Output

```

KBASIS = 3

 X
 1 2 3
0.999 2.000 0.000

 RES
 1 2 3 4
-0.000400 0.001200 -0.001200 0.000400

```

## LQRRV



Computes the least-squares solution using Householder transformations applied in blocked form.

### Required Arguments

**A** — Real LDA by (NCA + NUMEXC) array containing the matrix and right-hand sides. (Input)

The right-hand sides are input in  $A(1 : NRA, NCA + j)$ ,  $j = 1, \dots, NUMEXC$ . The array A is preserved upon output. The Householder factorization of the matrix is computed and used to solve the systems.

**X** — Real LDX by NUMEXC array containing the solution. (Output)

### Optional Arguments

**NRA** — Number of rows in the matrix. (Input)

Default: NRA = size (A,1).

**NCA** — Number of columns in the matrix. (Input)

Default: NCA = size (A,2) - NUMEXC.

**NUMEXC** — Number of right-hand sides. (Input)

Default: NUMEXC = size (X,2).

**LDA** — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)

Default: LDA = size (A,1).

*LDX* — Leading dimension of the solution array  $X$  exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDX = \text{size}(X,1)$ .

### FORTRAN 90 Interface

Generic: `CALL LQRRV (A, X [, ...])`

Specific: The specific interface names are `S_LQRRV` and `D_LQRRV`.

### FORTRAN 77 Interface

Single: `CALL LQRRV (NRA, NCA, NUMEXC, A, LDA, X, LDX)`

Double: The double precision name is `DLQRRV`.

### ScaLAPACK Interface

Generic: `CALL LQRRV (A0, X0 [, ...])`

Specific: The specific interface names are `S_LQRRV` and `D_LQRRV`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### Description

The routine `LQRRV` computes the  $QR$  decomposition of a matrix  $A$  using blocked Householder transformations. The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. The standard algorithm is based on the storage-efficient  $WY$  representation for products of Householder transformations. See Schreiber and Van Loan (1989).

The routine `LQRRV` determines an orthogonal matrix  $Q$  and an upper triangular matrix  $R$  such that  $A = QR$ . The  $QR$  factorization of a matrix  $A$  having  $NRA$  rows and  $NCA$  columns is as follows:

Initialize  $A_1 \leftarrow A$

For  $k = 1, \min(NRA - 1, NCA)$

Determine a Householder transformation for column  $k$  of  $A_k$  having the form

$$H_k = I - \tau_k \mu_k \mu_k^T$$

where  $\mu_k$  has zeros in the first  $k - 1$  positions and  $\tau_k$  is a scalar.

Update

$$A_k \leftarrow H_k A_{k-1} = A_{k-1} - \tau_k \mu_k \left( A_{k-1}^T \mu_k \right)^T$$

End  $k$

Thus,

$$A_p = H_p H_{p-1} \cdots H_1 A = Q^T A = R$$

where  $p = \min(\text{NRA} - 1, \text{NCA})$ . The matrix  $Q$  is not produced directly by `LQRRV`. The information needed to construct the Householder transformations is saved instead. If the matrix  $Q$  is needed explicitly,  $Q^T$  can be determined while the matrix is factored. No pivoting among the columns is done. The primary purpose of `LQRRV` is to give the user a high-performance  $QR$  least-squares solver. It is intended for least-squares problems that are well-posed. For background, see Golub and Van Loan (1989, page 225). During the  $QR$  factorization, the most time-consuming step is computing the matrix-vector update  $A_k \leftarrow H_k A_{k-1}$ . The routine `LQRRV` constructs “block” of `NB` Householder transformations in which the update is “rich” in matrix multiplication. The product of `NB` Householder transformations are written in the form

$$H_k H_{k+1} \cdots H_{k+n_b-1} = I + YTY^T$$

where  $Y_{\text{NRA} \times \text{NB}}$  is a lower trapezoidal matrix and  $T_{\text{NB} \times \text{NB}}$  is upper triangular. The optimal choice of the block size parameter `NB` varies among computer systems. Users may want to change it from its default value of 1.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2RRV/DL2RRV`. The reference is:

```
CALL L2RRV (NRA, NCA, NUMEXC, A, LDA, X, LDX, FACT, LDFACT, WK)
```

The additional arguments are as follows:

**FACT** — `LDFACT`  $\times$  (`NCA` + `NUMEXC`) work array containing the Householder factorization of the matrix on output. If the input data is not needed, `A` and `FACT` can share the same storage locations.

**LDFACT** — Leading dimension of the array `FACT` exactly as specified in the dimension statement of the calling program. (Input)  
If `A` and `FACT` are sharing the same storage, then `LDA` = `LDFACT` is required.

**WK** — Work vector of length  $(\text{NCA} + \text{NUMEXC} + 1) * (\text{NB} + 1)$ . The default value is  
`NB` = 1. This value can be reset. See item 3 below.

2. Informational errors

Type Code

4 1 The input matrix is singular.

3. [Integer Options](#) with Chapter 11 Options Manager
- 5** This option allows the user to reset the blocking factor used in computing the factorization. On some computers, changing `IVAL(*)` to a value larger than 1 will result in greater efficiency. The value `IVAL(*)` is the maximum value to use. (The software is specialized so that `IVAL(*)` is reset to an “optimal” used value within routine `L2RRV`.) The user can control the blocking by resetting `IVAL(*)` to a smaller value than the default. Default values are `IVAL(*) = 1, IMACH(5)`.
- 6** This option is the vector dimension where a shift is made from in-line level-2 loops to the use of level-2 BLAS in forming the partial product of Householder transformations. Default value is `IVAL(*) = IMACH(5)`.
- 10** This option allows the user to control the factorization step. If the value is 1 the Householder factorization will be computed. If the value is 2, the factorization will not be computed. In this latter case the decomposition has already been computed. Default value is `IVAL(*) = 1`.
- 11** This option allows the user to control the solving steps. The rules for `IVAL(*)` are:
1. Compute  $b \leftarrow Q^T b$ , and  $x \leftarrow R^+ b$ .
  2. Compute  $b \leftarrow Q^T b$ .
  3. Compute  $b \leftarrow Qb$ .
  4. Compute  $x \leftarrow R^+ b$ .
- Default value is `IVAL(*) = 1`. Note that `IVAL(*) = 2` or `3` may only be set when calling `L2RRV/DL2RRV`.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix and right-hand sides. (Input)

The right-hand sides are input in `A(1 : NRA, NCA + j)`,  $j = 1, \dots, \text{NUMEXC}$ . The array `A` is preserved upon output. The Householder factorization of the matrix is computed and used to solve the systems.. (Input)

**X0** — `MXLDX` by `MXCOLX` local matrix containing the local portions of the distributed matrix `X`. `X` contains the solution. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA`, `MXLDX`, `MXCOL`, and `MXCOLX` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

Given a real  $m \times k$  matrix  $B$  it is often necessary to compute the  $k$  least-squares solutions of the linear system  $AX = B$ , where  $A$  is an  $m \times n$  real matrix. When  $m > n$  the system is considered *overdetermined*. A solution with a zero residual normally does not exist. Instead the minimization problem

$$\min_{x_j \in \mathbf{R}^n} \|Ax_j - b_j\|_2$$

is solved  $k$  times where  $x_j, b_j$  are the  $j$ -th columns of the matrices  $X, B$  respectively. When  $A$  is of full column rank there exists a unique solution  $X_{LS}$  that solves the above minimization problem. By using the routine LQRRV,  $X_{LS}$  is computed.

```
 USE LQRRV_INT
 USE WRRRN_INT
 USE SGEMM_INT
!
! Declare variables
 INTEGER LDA, LDX, NCA, NRA, NUMEXC
 PARAMETER (NCA=3, NRA=5, NUMEXC=2, LDA=NRA, LDX=NCA)
! SPECIFICATIONS FOR LOCAL VARIABLES
 REAL X(LDX, NUMEXC)
! SPECIFICATIONS FOR SAVE VARIABLES
 REAL A(LDA, NCA+NUMEXC)
 SAVE A
! SPECIFICATIONS FOR SUBROUTINES
!
! Set values for A and the
! righthand sides.
!
! A = (1 2 4 | 7 10)
! (1 4 16 | 21 10)
! (1 6 36 | 43 9)
! (1 8 64 | 73 10)
! (1 10 100 |111 10)
!
 DATA A/5*1.0, 2.0, 4.0, 6.0, 8.0, 10.0, 4.0, 16.0, 36.0, 64.0, &
 100.0, 7.0, 21.0, 43.0, 73.0, 111.0, 2*10., 9., 2*10./
!
!
! QR factorization and solution
 CALL LQRRV (A, X)
 CALL WRRRN ('SOLUTIONS 1-2', X)
! Compute residuals and print
 CALL SGEMM ('N', 'N', NRA, NUMEXC, NCA, 1.E0, A, LDA, X, LDX, &
 -1.E0, A(1:, (NCA+1):), LDA)
 CALL WRRRN ('RESIDUALS 1-2', A(1:, (NCA+1):))
!
 END
```

## Output

```
SOLUTIONS 1-2
 1 2
1 1.00 10.80
```

```

2 1.00 -0.43
3 1.00 0.04

RESIDUALS 1-2
 1 2
1 0.0000 0.0857
2 0.0000 -0.3429
3 0.0000 0.5143
4 0.0000 -0.3429
5 0.0000 0.0857

```

## ScaLAPACK Example

The previous example is repeated here as a distributed computing example. Given a real  $m \times k$  matrix  $B$  it is often necessary to compute the  $k$  least-squares solutions of the linear system  $AX = B$ , where  $A$  is an  $m \times n$  real matrix. When  $m > n$  the system is considered *overdetermined*. A solution with a zero residual normally does not exist. Instead the minimization problem

$$\min_{x_j \in \mathbf{R}^n} \|Ax_j - b_j\|_2$$

is solved  $k$  times where  $x_j, b_j$  are the  $j$ -th columns of the matrices  $X, B$  respectively. When  $A$  is of full column rank there exists a unique solution  $X_{LS}$  that solves the above minimization problem. By using the routine LQRRV,  $X_{LS}$  is computed. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LQRRV_INT
 USE SGEMM_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables
 INTEGER LDA, LDX, NCA, NRA, NUMEXC, DESCA(9), DESCX(9)
 INTEGER INFO, MXCOL, MXLDA, MXLDX, MXCOLX
 INTEGER K
 REAL, ALLOCATABLE :: A(:, :), X(:)
 REAL, ALLOCATABLE :: A0(:, :), X0(:)
 PARAMETER (NRA=5, NCA=3, NUMEXC=2, LDA=NRA, LDX=NCA)
!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,NCA+NUMEXC), X(LDX, NUMEXC))
!
! Set values for A and the righthand sides
 A(1,:) = (/ 1.0, 2.0, 4.0, 7.0, 10.0/)
 A(2,:) = (/ 1.0, 4.0, 16.0, 21.0, 10.0/)
 A(3,:) = (/ 1.0, 6.0, 36.0, 43.0, 9.0/)
 A(4,:) = (/ 1.0, 8.0, 64.0, 73.0, 10.0/)
 A(5,:) = (/ 1.0, 10.0, 100.0, 111.0, 10.0/)
 ENDIF

```

```

! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA+NUMEXC, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(NRA, NCA+NUMEXC, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA+NUMEXC, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDA, INFO)
K = MIN0(NRA, NCA)
! Need to get dimensions of local x
! separate since x's leading
! dimension differs from A's
! Get the array descriptor entities
! MXLDX, AND MXCOLX
CALL SCALAPACK_GETDIM(K, NUMEXC, MP_MB, MP_NB, MXLDX, MXCOLX)
CALL DESCINIT(DESCX, K, NUMEXC, MP_NB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDX, INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), X0(MXLDX,MXCOLX))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Solve the least squares problem
CALL LQRRV (A0, X0)
! Unmap the results from the distributed
! arrays back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(X0, DESCX, X)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0) THEN
 CALL WRRRN ('SOLUTIONS 1-2', X)
! Compute residuals and print
CALL SGEMM ('N', 'N', NRA, NUMEXC, NCA, 1.E0, A, LDA, X, LDX, &
 -1.E0, A(1:, (NCA+1):), LDA)
CALL WRRRN ('RESIDUALS 1-2', A(1:, (NCA+1):))
ENDIF
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

---

## LSBRR



Solves a linear least-squares problem with iterative refinement.

## Required Arguments

- A*** — Real  $NRA$  by  $NCA$  matrix containing the coefficient matrix of the least-squares system to be solved. (Input)
- B*** — Real vector of length  $NRA$  containing the right-hand side of the least-squares system. (Input)
- X*** — Real vector of length  $NCA$  containing the solution vector with components corresponding to the columns not used set to zero. (Output)

## Optional Arguments

- NRA*** — Number of rows of  $A$ . (Input)  
Default:  $NRA = \text{size}(A,1)$ .
- NCA*** — Number of columns of  $A$ . (Input)  
Default:  $NCA = \text{size}(A,2)$ .
- LDA*** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .
- TOL*** — Real scalar containing the nonnegative tolerance used to determine the subset of columns of  $A$  to be included in the solution. (Input)  
If  $TOL$  is zero, a full complement of  $\min(NRA, NCA)$  columns is used. See [Comments](#).  
Default:  $TOL = 0.0$
- RES*** — Real vector of length  $NRA$  containing the residual vector  $B - AX$ . (Output)
- KBASIS*** — Integer scalar containing the number of columns used in the solution. (Output)

## FORTRAN 90 Interface

Generic:    CALL LSBRR (A, B, X [, ...])

Specific:   The specific interface names are `S_LSBRR` and `D_LSBRR`.

## FORTRAN 77 Interface

Single:     CALL LSBRR (NRA, NCA, A, LDA, B, TOL, X, RES, KBASIS)

Double:     The double precision name is `DLSBRR`.

## Description

Routine `LSBRR` solves the linear least-squares problem using iterative refinement. The iterative refinement algorithm is due to Björck (1967, 1968). It is also described by Golub and Van Loan (1983, pages 182–183).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2BRR/DL2BRR`. The reference is:

```
CALL L2BRR (NRA, NCA, A, LDA, B, TOL, X, RES, KBASIS, QR, BRRUX,
 IPVT, WK)
```

The additional arguments are as follows:

**QR** — Work vector of length  $NRA * NCA$  representing an  $NRA$  by  $NCA$  matrix that contains information from the  $QR$  factorization of  $A$ . See [LQRRR](#) for details.

**BRRUX** — Work vector of length  $NCA$  containing information about the orthogonal factor of the  $QR$  factorization of  $A$ . See [LQRRR](#) for details.

**IPVT** — Integer work vector of length  $NCA$  containing the pivoting information for the  $QR$  factorization of  $A$ . See [LQRRR](#) for details.

**WK** — Work vector of length  $NRA + 2 * NCA - 1$ .

2. Informational error

Type Code

|   |   |                                                                                  |
|---|---|----------------------------------------------------------------------------------|
| 4 | 1 | The data matrix is too ill-conditioned for iterative refinement to be effective. |
|---|---|----------------------------------------------------------------------------------|

3. Routine `LSBRR` calculates the  $QR$  decomposition with pivoting of a matrix  $A$  and tests the diagonal elements against a user-supplied tolerance `TOL`. The first integer `KBASIS = k` is determined for which

$$|r_{k+1,k+1}| \leq \text{TOL} * |r_{11}|$$

In effect, this condition implies that a set of columns with a condition number approximately bounded by  $1.0/\text{TOL}$  is used. Then, `LQRSL` performs a truncated fit of the first `KBASIS` columns of the permuted  $A$  to an input vector  $B$ . The coefficient of this fit is unscrambled to correspond to the original columns of  $A$ , and the coefficients corresponding to unused columns are set to zero. It may be helpful to scale the rows and columns of  $A$  so that the error estimates in the elements of the scaled matrix are roughly equal to `TOL`. The iterative refinement method of Björck is then applied to this factorization.

#### 4. Integer Options with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2BRR` the leading dimension of `QR` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSBRR`. Additional memory allocation for `QR` and option value restoration are done automatically in `LSBRR`. Users directly calling `L2BRR` can allocate additional space for `QR` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSBRR` or `L2BRR`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSBRR` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CRG` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CRG` skips this computation. `LSBRR` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

### Example

This example solves the linear least-squares problem with  $A$ , an  $8 \times 4$  matrix. Note that the second and fourth columns of  $A$  are identical. Routine `LSBRR` determines that there are three columns in the basis.

```
USE LSBRR_INT
USE UMACH_INT
USE WRRRN_INT
!
! Declare variables
PARAMETER (NRA=8, NCA=4, LDA=NRA)
REAL A(LDA,NCA), B(NRA), X(NCA), RES(NRA), TOL
!
! Set values for A
!
! A = (1 5 15 5)
! (1 4 17 4)
! (1 7 14 7)
! (1 3 18 3)
! (1 1 15 1)
! (1 8 11 8)
! (1 3 9 3)
! (1 4 10 4)
!
!
! DATA A/8*1, 5., 4., 7., 3., 1., 8., 3., 4., 15., 17., 14., &
! 18., 15., 11., 9., 10., 5., 4., 7., 3., 1., 8., 3., 4. /
!
! Set values for B
!
! DATA B/ 30., 31., 35., 29., 18., 35., 20., 22. /
!
! Solve the least squares problem
```

```

TOL = 1.0E-4
CALL LSBRN (A, B, X, TOL=TOL, RES=RES, KBASIS=KBASIS)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'KBASIS = ', KBASIS
CALL WRRRN ('X', X, 1, NCA, 1)
CALL WRRRN ('RES', RES, 1, NRA, 1)
!
END

```

## Output

```

KBASIS = 3
 X
 1 2 3 4
0.636 2.845 1.058 0.000

 RES
 1 2 3 4 5 6 7 8
-0.733 0.996 -0.365 0.783 -1.353 -0.036 1.306 -0.597

```

---

## LCLSQ

Solves a linear least-squares problem with linear constraints.

### Required Arguments

**A** — Matrix of dimension *NRA* by *NCA* containing the coefficients of the *NRA* least squares equations. (Input)

**B** — Vector of length *NRA* containing the right-hand sides of the least squares equations. (Input)

**C** — Matrix of dimension *NCON* by *NCA* containing the coefficients of the *NCON* constraints. (Input)  
If *NCON* = 0, *C* is not referenced.

**BL** — Vector of length *NCON* containing the lower limit of the general constraints. (Input)  
If there is no lower limit on the *I*-th constraint, then *BL(I)* will not be referenced.

**BU** — Vector of length *NCON* containing the upper limit of the general constraints. (Input)  
If there is no upper limit on the *I*-th constraint, then *BU(I)* will not be referenced. If there is no range constraint, *BL* and *BU* can share the same storage locations.

**IRTYPE** — Vector of length *NCON* indicating the type of constraints exclusive of simple bounds, where *IRTYPE(I)* = 0, 1, 2, 3 indicates *.EQ.*, *.LE.*, *.GE.*, and range constraints respectively. (Input)

**XLB** — Vector of length *NCA* containing the lower bound on the variables. (Input)  
If there is no lower bound on the *I*-th variable, then *XLB(I)* should be set to 1.0E30.

**XUB** — Vector of length *NCA* containing the upper bound on the variables. (Input)  
If there is no upper bound on the *I*-th variable, then *XUB(I)* should be set  $-1.0E30$ .

**X** — Vector of length *NCA* containing the approximate solution. (Output)

### Optional Arguments

**NRA** — Number of least-squares equations. (Input)  
Default: *NRA* = size (*A*,1).

**NCA** — Number of variables. (Input)  
Default: *NCA* = size (*A*,2).

**NCON** — Number of constraints. (Input)  
Default: *NCON* = size (*C*,1).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
*LDA* must be at least *NRA*.  
Default: *LDA* = size (*A*,1).

**LDC** — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)  
*LDC* must be at least *NCON*.  
Default: *LDC* = size (*C*,1).

**RES** — Vector of length *NRA* containing the residuals  $B - AX$  of the least-squares equations at the approximate solution. (Output)

### FORTRAN 90 Interface

Generic:     CALL LCLSQ (A, B, C, BL, BU, IRTYPE, XLB, XUB, X [, ...])

Specific:    The specific interface names are *S\_LCLSQ* and *D\_LCLSQ*.

### FORTRAN 77 Interface

Single:     CALL LCLSQ (NRA, NCA, NCON, A, LDA, B, C, LDC, BL, BU, IRTYPE, XLB, XUB, X, RES)

Double:     The double precision name is *DLCLSQ*.

### Description

The routine *LCLSQ* solves linear least-squares problems with linear constraints. These are systems of least-squares equations of the form  $Ax \cong b$  subject to

$$b_l \leq Cx \leq b_u$$

$$x_l \leq x \leq x_u$$

Here,  $A$  is the coefficient matrix of the least-squares equations,  $b$  is the right-hand side, and  $C$  is the coefficient matrix of the constraints. The vectors  $b_l$ ,  $b_u$ ,  $x_l$  and  $x_u$  are the lower and upper bounds on the constraints and the variables, respectively. The system is solved by defining dependent variables  $y \equiv Cx$  and then solving the least squares system with the lower and upper bounds on  $x$  and  $y$ . The equation  $Cx - y = 0$  is a set of equality constraints. These constraints are realized by heavy weighting, i.e. a penalty method, Hanson, (1986, pages 826–834).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2LSQ/DL2LSQ`. The reference is:

```
CALL L2LSQ (NRA, NCA, NCON, A, LDA, B, C, LDC, BL, BU,
 IRTYPE, XLB, XUB, X, RES, WK, IWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length  $(NCON + MAXDIM) * (NCA + NCON + 1) + 10 * NCA + 9 * NCON + 3$ .

**IWK** — Integer work vector of length  $3 * (NCON + NCA)$ .

2. Informational errors

| Type | Code | Description                                                      |
|------|------|------------------------------------------------------------------|
| 3    | 1    | The rank determination tolerance is less than machine precision. |
| 4    | 2    | The bounds on the variables are inconsistent.                    |
| 4    | 3    | The constraint bounds are inconsistent.                          |
| 4    | 4    | Maximum number of iterations exceeded.                           |

3. [Integer Options](#) with Chapter 11 Options Manager

**13** Debug output flag. If more detailed output is desired, set this option to the value 1. Otherwise, set it to 0. Default value is 0.

**14** Maximum number of add/drop iterations. If the value of this option is zero, up to  $5 * \max(nra, nca)$  iterations will be allowed. Otherwise set this option to the desired iteration limit. Default value is 0.

4. [Floating Point Options](#) with Chapter 11 Options Manager

**2** The value of this option is the relative rank determination tolerance to be used. Default value is  $\text{sqrt}(\text{AMACH}(4))$ .

- 5 The value of this option is the absolute rank determination tolerance to be used. Default value is  $\text{sqrt}(\text{AMACH}(4))$ .

## Example

A linear least-squares problem with linear constraints is solved.

```

USE LCLSQ_INT
USE UMACH_INT
USE SNRM2_INT
!
! Solve the following in the least squares sense:
! 3x1 + 2x2 + x3 = 3.3
! 4x1 + 2x2 + x3 = 2.3
! 2x1 + 2x2 + x3 = 1.3
! x1 + x2 + x3 = 1.0
!
! Subject to: x1 + x2 + x3 <= 1
! 0 <= x1 <= .5
! 0 <= x2 <= .5
! 0 <= x3 <= .5
!
! -----
! Declaration of variables
!
INTEGER NRA, NCA, MCON, LDA, LDC
PARAMETER (NRA=4, NCA=3, MCON=1, LDC=MCON, LDA=NRA)
!
INTEGER IRTYPE(MCON), NOUT
REAL A(LDA,NCA), B(NRA), BC(MCON), C(LDC,NCA), RES(NRA), &
 RESNRM, XSOL(NCA), XLB(NCA), XUB(NCA)
!
 Data initialization!
DATA A/3.0E0, 4.0E0, 2.0E0, 1.0E0, 2.0E0, &
 2.0E0, 2.0E0, 1.0E0, 1.0E0, 1.0E0, 1.0E0, 1.0E0/, &
 B/3.3E0, 2.3E0, 1.3E0, 1.0E0/, &
 C/3*1.0E0/, &
 BC/1.0E0/, IRTYPE/1/, XLB/3*0.0E0/, XUB/3*.5E0/
!
! Solve the bounded, constrained
! least squares problem.
!
CALL LCLSQ (A, B, C, BC, BC, IRTYPE, XLB, XUB, XSOL, RES=RES)
! Compute the 2-norm of the residuals.
RESNRM = SNRM2 (NRA, RES, 1)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, 999) XSOL, RES, RESNRM
!
999 FORMAT (' The solution is ', 3F9.4, '//, ' The residuals ', &
 'evaluated at the solution are ', /, 18X, 4F9.4, '//, &
 ' The norm of the residual vector is ', F8.4)
!
END

```

## Output

```
The solution is 0.5000 0.3000 0.2000
The residuals evaluated at the solution are
 -1.0000 0.5000 0.5000 0.0000

The norm of the residual vector is 1.2247
```

---

## LQRRR



Computes the  $QR$  decomposition,  $AP = QR$ , using Householder transformations.

### Required Arguments

- A** — Real  $NRA$  by  $NCA$  matrix containing the matrix whose  $QR$  factorization is to be computed. (Input)
- QR** — Real  $NRA$  by  $NCA$  matrix containing information required for the  $QR$  factorization. (Output)  
The upper trapezoidal part of  $QR$  contains the upper trapezoidal part of  $R$  with its diagonal elements ordered in decreasing magnitude. The strict lower trapezoidal part of  $QR$  contains information to recover the orthogonal matrix  $Q$  of the factorization. Arguments **A** and **QR** can occupy the same storage locations. In this case, **A** will not be preserved on output.
- QRAUX** — Real vector of length  $NCA$  containing information about the orthogonal part of the decomposition in the first  $\min(NRA, NCA)$  position. (Output)

### Optional Arguments

- NRA** — Number of rows of **A**. (Input)  
Default:  $NRA = \text{size}(\mathbf{A}, 1)$ .
- NCA** — Number of columns of **A**. (Input)  
Default:  $NCA = \text{size}(\mathbf{A}, 2)$ .
- LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(\mathbf{A}, 1)$ .
- PIVOT** — Logical variable. (Input)  
**PIVOT** = `.TRUE.` means column pivoting is enforced.  
**PIVOT** = `.FALSE.` means column pivoting is not done.  
Default: **PIVOT** = `.TRUE.`

**IPVT** — Integer vector of length `NCA` containing information that controls the final order of the columns of the factored matrix `A`. (Input/Output)  
On input, if `IPVT(K) > 0`, then the `K`-th column of `A` is an initial column. If `IPVT(K) = 0`, then the `K`-th column of `A` is a free column. If `IPVT(K) < 0`, then the `K`-th column of `A` is a final column. See [Comments](#).  
On output, `IPVT(K)` contains the index of the column of `A` that has been interchanged into the `K`-th column. This defines the permutation matrix `P`. The array `IPVT` is referenced only if `PIVOT` is equal to `.TRUE`.  
Default: `IPVT = 0`.

**LDQR** — Leading dimension of `QR` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDQR = size (QR,1)`.

**CONORM** — Real vector of length `NCA` containing the norms of the columns of the input matrix. (Output)  
If this information is not needed, `CONORM` and `QRAUX` can share the same storage locations.

## FORTRAN 90 Interface

Generic: `CALL LQRRR (A, QR, QRAUX [, ...])`

Specific: The specific interface names are `S_LQRRR` and `D_LQRRR`.

## FORTRAN 77 Interface

Single: `CALL LQRRR (NRA, NCA, A, LDA, PIVOT, IPVT, QR, LDQR, QRAUX, CONORM)`

Double: The double precision name is `DLQRRR`.

## ScaLAPACK Interface

Generic: `CALL LQRRR (A0, QR0, QRAUX0 [, ...])`

Specific: The specific interface names are `S_LQRRR` and `D_LQRRR`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

The routine `LQRRR` computes the `QR` decomposition of a matrix using Householder transformations. The underlying code is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

LQRRR determines an orthogonal matrix  $Q$ , a permutation matrix  $P$ , and an upper trapezoidal matrix  $R$  with diagonal elements of nonincreasing magnitude, such that  $AP = QR$ . The Householder transformation for column  $k$  is of the form

$$I - \frac{u_k u_k^T}{p_k}$$

for  $k = 1, 2, \dots, \min(\text{NRA}, \text{NCA})$ , where  $u$  has zeros in the first  $k - 1$  positions. The matrix  $Q$  is not produced directly by LQRRR. Instead the information needed to reconstruct the Householder transformations is saved. If the matrix  $Q$  is needed explicitly, the subroutine LQERR can be called after LQRRR. This routine accumulates  $Q$  from its factored form.

Before the decomposition is computed, initial columns are moved to the beginning of the array  $A$  and the final columns to the end. Both initial and final columns are frozen in place during the computation. Only free columns are pivoted. Pivoting, when requested, is done on the free columns of largest reduced norm.

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2RRR/DL2RRR. The reference is:

```
CALL L2RRR (NRA, NCA, A, LDA, PIVOT, IPVT, QR, LDQR, QRAUX, CONORM,
 WORK)
```

The additional argument is

**WORK** — Work vector of length  $2\text{NCA} - 1$ . Only  $\text{NCA} - 1$  locations of **WORK** are referenced if **PIVOT** = **.FALSE.** .

2. LQRRR determines an orthogonal matrix  $Q$ , permutation matrix  $P$ , and an upper trapezoidal matrix  $R$  with diagonal elements of nonincreasing magnitude, such that  $AP = QR$ . The Householder transformation for column  $k$ ,  $k = 1, \dots, \min(\text{NRA}, \text{NCA})$  is of the form

$$I - u_k^{-1} u u^T$$

where  $u$  has zeros in the first  $k - 1$  positions. If the explicit matrix  $Q$  is needed, the user can call routine LQERR after calling LQRRR. This routine accumulates  $Q$  from its factored form.

3. Before the decomposition is computed, initial columns are moved to the beginning and the final columns to the end of the array  $A$ . Both initial and final columns are not moved during the computation. Only free columns are moved. Pivoting, if requested, is done on the free columns of largest reduced norm.
4. When pivoting has been selected by having entries of **IPVT** initialized to zero, an estimate of the condition number of  $A$  can be obtained from the output by computing the magnitude of the number  $\text{QR}(1, 1)/\text{QR}(K, K)$ , where  $K = \text{MIN}(\text{NRA}, \text{NCA})$ . This

estimate can be used to select the number of columns, `KBASIS`, used in the solution step computed with routine [LQRSL](#).

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix whose *QR* factorization is to be computed. (Input)

**QR0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `QR`. `QR` contains the information required for the *QR* factorization. (Output)  
The upper trapezoidal part of *QR* contains the upper trapezoidal part of *R* with its diagonal elements ordered in decreasing magnitude. The strict lower trapezoidal part of *QR* contains information to recover the orthogonal matrix *Q* of the factorization. Arguments `A` and `QR` can occupy the same storage locations. In this case, `A` will not be preserved on output.

**QRAUX0** — Real vector of length `MXCOL` containing the local portions of the distributed matrix `QRAUX`. `QRAUX` contains information about the orthogonal part of the decomposition in the first `MIN(NRA, NCA)` position. (Output)

**IPVT0** — Integer vector of length `MXLDB` containing the local portions of the distributed vector `IPVT`. `IPVT` contains the information that controls the final order of the columns of the factored matrix `A`. (Input/Output)  
On input, if `IPVT(K) > 0`, then the *K*-th column of `A` is an initial column. If `IPVT(K) = 0`, then the *K*-th column of `A` is a free column. If `IPVT(K) < 0`, then the *K*-th column of `A` is a final column. See [Comments](#).  
On output, `IPVT(K)` contains the index of the column of `A` that has been interchanged into the *K*-th column. This defines the permutation matrix *P*. The array `IPVT` is referenced only if `PIVOT` is equal to `.TRUE`.  
Default: `IPVT = 0`.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA`, `MXLDB`, and `MXCOL` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

In various statistical algorithms it is necessary to compute  $q = x^T(A^T A)^{-1}x$ , where *A* is a rectangular matrix of full column rank. By using the *QR* decomposition, *q* can be computed without forming  $A^T A$ . Note that

$$A^T A = (QRP^{-1})^T (QRP^{-1}) = P^{-T} R^T (Q^T Q) R P^{-1} = P R^T R P^T$$

since *Q* is orthogonal ( $Q^T Q = I$ ) and *P* is a permutation matrix. Let

$$Q^T AP = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

where  $R_1$  is an upper triangular nonsingular matrix. Then

$$x^T (A^T A)^{-1} x = x^T P R_1^{-1} R_1^{-T} P^{-1} x = \|R_1^{-T} P^{-1} x\|_2^2$$

In the following program, first the vector  $t = P^{-1} x$  is computed. Then

$$t := R_1^{-T} t$$

Finally,

$$q = \|t\|_2^2$$

```

USE IMSL_LIBRARIES
!
! Declare variables
INTEGER LDA, LDQR, NCA, NRA
PARAMETER (NCA=3, NRA=4, LDA=NRA, LDQR=NRA)
!
! SPECIFICATIONS FOR PARAMETERS
INTEGER LDQ
PARAMETER (LDQ=NRA)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IPVT(NCA), NOUT
REAL CONORM(NCA), Q, QR(LDQR,NCA), QRAUX(NCA), T(NCA)
LOGICAL PIVOT
REAL A(LDA,NCA), X(NCA)
!
! Set values for A
!
! A = (1 2 4)
! (1 4 16)
! (1 6 36)
! (1 8 64)
!
DATA A/4*1.0, 2.0, 4.0, 6.0, 8.0, 4.0, 16.0, 36.0, 64.0/
!
! Set values for X
!
! X = (1 2 3)
!
DATA X/1.0, 2.0, 3.0/
!
! QR factorization
PIVOT = .TRUE.
IPVT=0
CALL LQRRR (A, QR, QRAUX, PIVOT=PIVOT, IPVT=IPVT)
!
! Set t = inv(P)*x
CALL PERMU (X, IPVT, T, IPATH=1)
!
! Compute t = inv(trans(R))*t
CALL LSLRT (QR, T, T, IPATH=4)
!
! Compute 2-norm of t, squared.
Q = SDOT(NCA,T,1,T,1)
!
! Print result
CALL UMACH (2, NOUT)

```

```

 WRITE (NOUT,*) 'Q = ', Q
!
 END

```

## Output

Q = 0.840624

## ScaLAPACK Example

The previous example is repeated here as a distributed computing example. In various statistical algorithms it is necessary to compute  $q = x^T(A^T A)^{-1}x$ , where  $A$  is a rectangular matrix of full column rank. By using the  $QR$  decomposition,  $q$  can be computed without forming  $A^T A$ . Note that

$$A^T A = (QRP^{-1})^T(QRP^{-1}) = P^{-T} R^T(Q^T Q)RP^{-1} = P R^T R P^T$$

since  $Q$  is orthogonal ( $Q^T Q = I$ ) and  $P$  is a permutation matrix. Let

$$Q^T A P = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

where  $R_1$  is an upper triangular nonsingular matrix. Then

$$x^T (A^T A)^{-1} x = x^T P R_1^{-1} R_1^{-T} P^{-1} x = \|R_1^{-T} P^{-1} x\|_2^2$$

In the following program, first the vector  $t = P^{-1} x$  is computed. Then

$$t := R_1^{-T} t$$

Finally,

$$q = \|t\|_2^2$$

SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LQRRR_INT
 USE PERMU_INT
 USE LSLRT_INT
 USE UMACH_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
 DECLARE VARIABLES
 INTEGER LDA, LDQR, NCA, NRA, DESCA(9), DESCB(9), DESCL(9)
 INTEGER INFO, MXCOL, MXLDA, MXLDB, MXCOLB, NOUT
 INTEGER, ALLOCATABLE :: IPVT(:), IPVT0(:)
 LOGICAL PIVOT

```

```

REAL Q
REAL, ALLOCATABLE :: A(:, :), X(:), T(:)
REAL, ALLOCATABLE :: A0(:, :), T0(:), QR0(:, :), QRAUX0(:)
REAL, (KIND(1E0))SDOT
PARAMETER (NRA=4, NCA=3, LDA=NRA, LDQR=NRA)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,NCA), X(NCA), T(NCA), IPVT(NCA))
! Set values for A and the righthand side
 A(1,:) = (/ 1.0, 2.0, 4.0/)
 A(2,:) = (/ 1.0, 4.0, 16.0/)
 A(3,:) = (/ 1.0, 6.0, 36.0/)
 A(4,:) = (/ 1.0, 8.0, 64.0/)
!
 X = (/ 1.0, 2.0, 3.0/)
!
 IPVT = 0
ENDIF
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! MXCOL, MXLDB, MXCOLB
CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
CALL SCALAPACK_GETDIM(NCA, 1, MP_NB, 1, MXLDB, MXCOLB)
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, &
INFO)
CALL DESCINIT(DESCL, 1, NCA, 1, MP_NB, 0, 0, MP_ICTXT, 1, INFO)
CALL DESCINIT(DESCB, NCA, 1, MP_NB, 1, 0, 0, MP_ICTXT, MXLDB, &
INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), QR0(MXLDA,MXCOL), QRAUX0(MXCOL), &
IPVT0(MXCOL), T0(MXLDB))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
PIVOT = .TRUE.

CALL SCALAPACK_MAP(IPVT, DESCL, IPVT0)
! QR factorization
CALL LQRRR (A0, QR0, QRAUX0, PIVOT=PIVOT, IPVT=IPVT0)
! Unmap the results from the distributed
! array back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(IPVT0, DESCL, IPVT, NCA, .FALSE.)
IF(MP_RANK .EQ. 0) CALL PERMU (X, IPVT, T, IPATH=1)
CALL SCALAPACK_MAP(T, DESCB, T0)
CALL LSLRT (QR0, T0, T0, IPATH=4)
CALL SCALAPACK_UNMAP(T0, DESCB, T)
! Print results.
! Only Rank=0 has the solution.
IF(MP_RANK .EQ. 0) THEN
 Q = SDOT(NCA, T, 1, T, 1)
 CALL UMACH (2, NOUT)
 WRITE (NOUT, *) 'Q = ', Q

```

```

ENDIF
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

Q = 0.840624

---

# LQERR



Accumulates the orthogonal matrix  $Q$  from its factored form given the  $QR$  factorization of a rectangular matrix  $A$ .

## Required Arguments

**QR** — Real  $NR_{QR}$  by  $NC_{QR}$  matrix containing the factored form of the matrix  $Q$  in the first  $\min(NR_{QR}, NC_{QR})$  columns of the strict lower trapezoidal part of  $QR$  as output from subroutine `LQRRR/DLQRRR`. (Input)

**QRAUX** — Real vector of length  $NC_{QR}$  containing information about the orthogonal part of the decomposition in the first  $\min(NR_{QR}, NC_{QR})$  position as output from routine `LQRRR/DLQRRR`. (Input)

**Q** — Real  $NR_{QR}$  by  $NR_{QR}$  matrix containing the accumulated orthogonal matrix  $Q$ ;  $Q$  and  $QR$  can share the same storage locations if  $QR$  is not needed. (Output)

## Optional Arguments

**NRQR** — Number of rows in  $QR$ . (Input)  
Default:  $NR_{QR} = \text{size}(QR,1)$ .

**NCQR** — Number of columns in  $QR$ . (Input)  
Default:  $NC_{QR} = \text{size}(QR,2)$ .

**LDQR** — Leading dimension of  $QR$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LD_{QR} = \text{size}(QR,1)$ .

**LDQ** — Leading dimension of  $Q$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDQ = \text{size}(Q,1)$ .

### **FORTRAN 90 Interface**

Generic:     CALL LQERR (QR, QRAUX, Q [, ...])

Specific:    The specific interface names are S\_LQERR and D\_LQERR.

### **FORTRAN 77 Interface**

Single:      CALL LQERR (NRQR, NCQR, QR, LDQR, QRAUX, Q, LDQ)

Double:     The double precision name is DLQERR.

### **ScaLAPACK Interface**

Generic:     CALL LQERR (QR0, QRAUX0, Q0 [, ...])

Specific:    The specific interface names are S\_LQERR and D\_LQERR.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

### **Description**

The routine LQERR accumulates the Householder transformations computed by IMSL routine [LQRRR](#) to produce the orthogonal matrix  $Q$ .

The underlying code is based on either LINPACK, LAPACK, or ScaLAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

### **Comments**

1.     Workspace may be explicitly provided, if desired, by use of L2ERR/DL2ERR. The reference is:

```
CALL L2ERR (NRQR, NCQR, QR, LDQR, QRAUX, Q, LDQ, WK)
```

The additional argument is

**WK** — Work vector of length  $2 * NRQR$ .

### **ScaLAPACK Usage Notes**

The arguments which differ from the standard version of this routine are:

**QR0** — MXLDA by MXCOL local matrix containing the local portions of the distributed matrix QR. QR contains the factored form of the matrix Q in the first min(NRQR, NCQR) columns of the strict lower trapezoidal part of QR as output from subroutine LQRRR/DLQRRR. (Input)

**QRAUX0** — Real vector of length MXCOL containing the local portions of the distributed matrix QRAUX. QRAUX contains the information about the orthogonal part of the decomposition in the first min(NRA, NCA) positions as output from subroutine LQRRR/DLQRRR. (Input)

**Q0** — MXLDA by MXLDA local matrix containing the local portions of the distributed matrix Q. Q contains the accumulated orthogonal matrix ; Q and QR can share the same storage locations if QR is not needed. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

In this example, the orthogonal matrix Q in the QR decomposition of a matrix A is computed. The product  $X = QR$  is also computed. Note that X can be obtained from A by reordering the columns of A according to IPVT.

```

USE IMSL_LIBRARIES
!
! Declare variables
INTEGER LDA, LDQ, LDQR, NCA, NRA
PARAMETER (NCA=3, NRA=4, LDA=NRA, LDQ=NRA, LDQR=NRA)
!
INTEGER IPVT(NCA), J
REAL A(LDA,NCA), CONORM(NCA), Q(LDQ,NRA), QR(LDQR,NCA), &
 QRAUX(NCA), R(NRA,NCA), X(NRA,NCA)
LOGICAL PIVOT
!
! Set values for A
!
! A = (1 2 4)
! (1 4 16)
! (1 6 36)
! (1 8 64)
!
DATA A/4*1.0, 2.0, 4.0, 6.0, 8.0, 4.0, 16.0, 36.0, 64.0/
!
! QR factorization
! Set IPVT = 0 (all columns free)
IPVT = 0
PIVOT = .TRUE.
CALL LQRRR (A, QR, QRAUX, IPVT=IPVT, PIVOT=PIVOT)
! Accumulate Q
CALL LQERR (QR, QRAUX, Q)
! R is the upper trapezoidal part of QR

```

```

 R = 0.0E0
 DO 10 J=1, NCA
 CALL SCOPY (J, QR(:,J), 1, R(:,J), 1)
10 CONTINUE
!
! Compute X = Q*R
 CALL MRRRR (Q, R, X)
!
! Print results
 CALL WRIRN ('IPVT', IPVT, 1, NCA, 1)
 CALL WRRRN ('Q', Q)
 CALL WRRRN ('R', R)
 CALL WRRRN ('X = Q*R', X)
!
 END

```

## Output

```

IPVT
 1 2 3
 3 2 1

 Q
 1 2 3 4
 1 -0.0531 -0.5422 0.8082 -0.2236
 2 -0.2126 -0.6574 -0.2694 0.6708
 3 -0.4783 -0.3458 -0.4490 -0.6708
 4 -0.8504 0.3928 0.2694 0.2236

 R
 1 2 3
 1 -75.26 -10.63 -1.59
 2 0.00 -2.65 -1.15
 3 0.00 0.00 0.36
 4 0.00 0.00 0.00

 X = Q*R
 1 2 3
 1 4.00 2.00 1.00
 2 16.00 4.00 1.00
 3 36.00 6.00 1.00
 4 64.00 8.00 1.00

```

## ScaLAPACK Example

In this example, the orthogonal matrix  $Q$  in the  $QR$  decomposition of a matrix  $A$  is computed. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

 USE MPI_SETUP_INT
 USE LQRRR_INT
 USE LQERR_INT
 USE WRRRN_INT
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'
!
! Declare variables

```

```

INTEGER LDA, LDQR, NCA, NRA, DESCA(9), DESCL(9), DESCQ(9)
INTEGER INFO, MXCOL, MXLDA, LDQ
INTEGER, ALLOCATABLE :: IPVT(:), IPVT0(:)
LOGICAL PIVOT
REAL, ALLOCATABLE :: A(:, :), QR(:, :), Q(:, :), QRAUX(:)
REAL, ALLOCATABLE :: A0(:, :), QR0(:, :), Q0(:, :), QRAUX0(:)
PARAMETER (NRA=4, NCA=3, LDA=NRA, LDQR=NRA, LDQ=NRA)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(NRA,NCA), Q(NRA,NRA), QR(NRA,NCA), &
 QRAUX(NCA), IPVT(NCA))
!
! Set values for A and the righthand sides
 A(1,:) = (/ 1.0, 2.0, 4.0/)
 A(2,:) = (/ 1.0, 4.0, 16.0/)
 A(3,:) = (/ 1.0, 6.0, 36.0/)
 A(4,:) = (/ 1.0, 8.0, 64.0/)
!
 IPVT = 0
ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA, .FALSE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, &
INFO)
CALL DESCINIT(DESCL, 1, NCA, 1, MP_NB, 0, 0, MP_ICTXT, 1, INFO)
CALL DESCINIT(DESCQ, NRA, NRA, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, &
INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), QR0(MXLDA,MXCOL), QRAUX0(MXCOL), &
 IPVT0(MXCOL), Q0(MXLDA,MXLDA))
!
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
PIVOT = .TRUE.

CALL SCALAPACK_MAP(IPVT, DESCL, IPVT0)
!
! QR factorization
CALL LQRRR (A0, QR0, QRAUX0, PIVOT=PIVOT, IPVT=IPVT0)
CALL LQERR (QR0, QRAUX0, Q0)
!
! Unmap the results from the distributed
! array back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(Q0, DESCQ, Q)
!
! Print results.
! Only Rank=0 has the solution, Q.
IF(MP_RANK .EQ. 0) CALL WRRRN ('Q', Q)
!
! Exit Scalapack usage
CALL SCALAPACK_EXIT(MP_ICTXT)
!
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

---

# LQRSL



Computes the coordinate transformation, projection, and complete the solution of the least-squares problem  $Ax = b$ .

## Required Arguments

**KBASIS** — Number of columns of the submatrix  $A_k$  of  $A$ . (Input)

The value `KBASIS` must not exceed  $\min(\text{NRA}, \text{NCA})$ , where `NCA` is the number of columns in matrix  $A$ . The value `NCA` is an argument to routine `LQRRR`. The value of `KBASIS` is normally `NCA` unless the matrix is rank-deficient. The user must analyze the problem data and determine the value of `KBASIS`. See [Comments](#).

**QR** —  $\text{NRA}$  by  $\text{NCA}$  array containing information about the  $QR$  factorization of  $A$  as output from routine `LQRRR/DLQRRR`. (Input)

**QRAUX** — Vector of length `NCA` containing information about the  $QR$  factorization of  $A$  as output from routine `LQRRR/DLQRRR`. (Input)

**B** — Vector  $b$  of length `NRA` to be manipulated. (Input)

**IPATH** — Option parameter specifying what is to be computed. (Input)

The value `IPATH` has the decimal expansion `IJKLM`, such that:

`I`  $\neq$  0 means compute  $Qb$ ;

`J`  $\neq$  0 means compute  $Q^T b$ ;

`K`  $\neq$  0 means compute  $Q^T b$  and  $x$ ;

`L`  $\neq$  0 means compute  $Q^T b$  and  $b - Ax$ ;

`M`  $\neq$  0 means compute  $Q^T b$  and  $Ax$ .

For example, if the decimal number `IPATH = 01101`, then `I = 0`, `J = 1`, `K = 1`, `L = 0`, and `M = 1`.

## Optional Arguments

**NRA** — Number of rows of matrix  $A$ . (Input)

Default: `NRA = size (QR,1)`.

**LDQR** — Leading dimension of  $QR$  exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDQR = size (QR,1)`.

**QB** — Vector of length `NRA` containing  $Qb$  if requested in the option `IPATH`. (Output)

**QTB** — Vector of length `NRA` containing  $Q^T b$  if requested in the option `IPATH`. (Output)

**X** — Vector of length `KBASIS` containing the solution of the least-squares problem  $A_k x = b$ , if this is requested in the option `IPATH`. (Output)

If pivoting was requested in routine `LQRRR/DLQRRR`, then the `J`-th entry of `X` will be associated with column `IPVT(J)` of the original matrix  $A$ . See [Comments](#).

**RES** — Vector of length `NRA` containing the residuals  $(b - Ax)$  of the least-squares problem if requested in the option `IPATH`. (Output)

This vector is the orthogonal projection of  $b$  onto the orthogonal complement of the column space of  $A$ .

**AX** — Vector of length `NRA` containing the least-squares approximation  $Ax$  if requested in the option `IPATH`. (Output)

This vector is the orthogonal projection of  $b$  onto the column space of  $A$ .

## FORTRAN 90 Interface

Generic: `CALL LQRSL (KBASIS, QR, QRAUX, B, IPATH [, ...])`

Specific: The specific interface names are `S_LQRSL` and `D_LQRSL`.

## FORTRAN 77 Interface

Single: `CALL LQRSL (NRA, KBASIS, QR, LDQR, QRAUX, B, IPATH, QB, QTB, X, RES, AX)`

Double: The double precision name is `DLQRSL`.

## ScaLAPACK Interface

Generic: `CALL LQRSL (KBASIS, QR0, QRAUX0, B0, IPATH [, ...])`

Specific: The specific interface names are `S_LQRSL` and `D_LQRSL`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

The underlying code of routine `LQRSL` is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

The most important use of `LQRSL` is for solving the least-squares problem  $Ax = b$ , with coefficient matrix  $A$  and data vector  $b$ . This problem can be formulated, using the *normal equations* method, as  $A^T Ax = A^T b$ . Using `LQRRR` the  $QR$  decomposition of  $A$ ,  $AP = QR$ , is computed. Here  $P$  is a

permutation matrix ( $P = P$ ),  $Q$  is an orthogonal matrix ( $Q = Q^T$ ) and  $R$  is an upper trapezoidal matrix. The normal equations can then be written as

$$(PR^T)(Q^TQ)R(P^T x) = (PR^T)Q^T b$$

If  $A^T A$  is nonsingular, then  $R$  is also nonsingular and the normal equations can be written as  $R(P^T x) = Q^T b$ . `LQRSL` can be used to compute  $Q^T b$  and then solve for  $P^T x$ . Note that the *permuted* solution is returned.

The routine `LQRSL` can also be used to compute the least-squares residual,  $b - Ax$ . This is the projection of  $b$  onto the orthogonal complement of the column space of  $A$ . It can also compute  $Qb$ ,  $Q^T b$  and  $Ax$ , the orthogonal projection of  $x$  onto the column space of  $A$ .

## Comments

1. Informational error

Type Code

4 1 Computation of the least-squares solution of  $A_k * X = B$  is requested, but the upper triangular matrix  $R$  from the  $QR$  factorization is singular.

2. This routine is designed to be used together with `LQRRR`. It assumes that `LQRRR/DLQRR` has been called to get  $QR$ ,  $QRAUX$  and  $IPVT$ . The submatrix  $A_k$  mentioned above is actually equal to  $A_k = (A(IPVT(1)), A(IPVT(2)), \dots, A(IPVT(KBASIS)))$ , where  $A(IPVT(I))$  is the  $IPVT(I)$ -th column of the original matrix.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**QR0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix  $QR$ .  $QR$  contains the factored form of the matrix  $Q$  in the first  $\min(NRQR, NCQR)$  columns of the strict lower trapezoidal part of  $QR$  as output from subroutine `LQRRR/DLQRRR`. (Input)

**QRAUX0** — Real vector of length `MXCOL` containing the local portions of the distributed matrix  $QRAUX$ .  $QRAUX$  contains the information about the orthogonal part of the decomposition in the first  $\min(NRA, NCA)$  positions as output from subroutine `LQRRR/DLQRRR`. (Input)

**B0** — Real vector of length `MXLDA` containing the local portions of the distributed vector  $B$ .  $B$  contains the vector to be manipulated. (Input)

**QB0** — Real vector of length `MXLDA` containing the local portions of the distributed vector  $Qb$  if requested in the option `IPATH`. (Output)

**QTB0** — Real vector of length `MXLDA` containing the local portions of the distributed vector  $Q^T b$  if requested in the option `IPATH`. (Output)

**X0** — Real vector of length MXLDX containing the local portions of the distributed vector  $x$ .  $x$  contains the solution of the least-squares problem  $A_k x = b$ , if this is requested in the option IPATH. (Output)

If pivoting was requested in routine LQRRR/DLQRRR, then the  $J$ -th entry of  $x$  will be associated with column IPVT( $J$ ) of the original matrix  $A$ . See [Comments](#).

**RES0** — Real vector of length MXLDA containing the local portions of the distributed vector RES. RES contains the residuals  $(b - Ax)$  of the least-squares problem if requested in the option IPATH. (Output)

This vector is the orthogonal projection of  $b$  onto the orthogonal complement of the column space of  $A$ .

**AX0** — Real vector of length MXLDA containing the local portions of the distributed vector AX. AX contains the least-squares approximation  $Ax$  if requested in the option IPATH. (Output)

This vector is the orthogonal projection of  $b$  onto the column space of  $A$ .

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, MXLDA, MXLDX and MXCOL can be obtained through a call to SCALAPACK\_GETDIM (see [Utilities](#)) after a call to SCALAPACK\_SETUP (see [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

Consider the problem of finding the coefficients  $c_i$  in

$$f(x) = c_0 + c_1x + c_2x^2$$

given data at  $x_i = 2^i, i = 1, 2, 3, 4$ , using the method of least squares. The row of the matrix  $A$  contains the value of 1,  $x_i$  and

$$x_i^2$$

at the data points. The vector  $b$  contains the data. The routine LQRRR is used to compute the  $QR$  decomposition of  $A$ . Then LQRSL is then used to solve the least-squares problem and compute the residual vector.

```

 USE IMSL_LIBRARIES
!
! Declare variables
PARAMETER (NRA=4, NCA=3, KBASIS=3, LDA=NRA, LDQR=NRA)
INTEGER IPVT(NCA)
REAL A(LDA,NCA), QR(LDQR,NCA), QRAUX(NCA), CONORM(NCA), &
 X(KBASIS), QB(1), QTB(NRA), RES(NRA), &
 AX(1), B(NRA)
LOGICAL PIVOT
!
! Set values for A
!
! A = (1 2 4)
! (1 4 16)
! (1 6 36)
! (1 8 64)

```

```

!
DATA A/4*1.0, 2.0, 4.0, 6.0, 8.0, 4.0, 16.0, 36.0, 64.0/
!
! Set values for B
!
! B = (16.99 57.01 120.99 209.01)
DATA B/ 16.99, 57.01, 120.99, 209.01 /
!
! QR factorization
PIVOT = .TRUE.
IPVT = 0
CALL LQRRR (A, QR, QRAUX, PIVOT=PIVOT, IPVT=IPVT)
! Solve the least squares problem
IPATH = 00110
CALL LQRSL (KBASIS, QR, QRAUX, B, IPATH, X=X, RES=RES)
! Print results
CALL WRIRN ('IPVT', IPVT, 1, NCA, 1)
CALL WRRRN ('X', X, 1, KBASIS, 1)
CALL WRRRN ('RES', RES, 1, NRA, 1)
!
END

```

## Output

```

IPVT
 1 2 3
 3 2 1

 X
 1 2 3
3.000 2.002 0.990

 RES
 1 2 3 4
-0.00400 0.01200 -0.01200 0.00400

```

Note that since IPVT is (3, 2, 1) the array X contains the solution coefficients  $c_i$  in reverse order.

## ScaLAPACK Example

The previous example is repeated here as a distributed example. Consider the problem of finding the coefficients  $c_i$  in

$$f(x) = c_0 + c_1x + c_2x^2$$

given data at  $x_i = 2i$ ,  $i = 1, 2, 3, 4$ , using the method of least squares. The row of the matrix  $A$  contains the value of 1,  $x_i$  and

$$x_i^2$$

at the data points. The vector  $b$  contains the data. The routine LQRRR is used to compute the QR decomposition of  $A$ . Then LQRSL is then used to solve the least-squares problem and compute the residual vector. SCALAPACK\_MAP and SCALAPACK\_UNMAP are IMSL utility routines (see [Utilities](#)) used to map and unmap arrays to and from the processor grid. They are used here

for brevity. DESCINIT is a ScaLAPACK tools routine which initializes the descriptors for the local arrays.

```

USE MPI_SETUP_INT
USE LQRRR_INT
USE LQRSL_INT
USE WRIRN_INT
USE WRRRN_INT
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER KBASIS, LDA, LDQR, NCA, NRA, DESCA(9), DESCL(9), &
 DESCX(9), DESCB(9)
INTEGER INFO, MXCOL, MXCOLX, MXLDA, MXLDX, LDQ, IPATH
INTEGER, ALLOCATABLE :: IPVT(:), IPVT0(:)
REAL, ALLOCATABLE :: A(:, :), B(:), QR(:, :), QRAUX(:), X(:), &
 RES(:)
REAL, ALLOCATABLE :: A0(:, :), QR0(:, :), QRAUX0(:), X0(:), &
 RES0(:), B0(:), QTB0(:)
LOGICAL PIVOT
PARAMETER (NRA=4, NCA=3, LDA=NRA, LDQR=NRA, KBASIS=3)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,NCA), B(NRA), QR(LDQR,NCA), &
 QRAUX(NCA), IPVT(NCA), X(NCA), RES(NRA))
!
! Set values for A and the righthand sides
 A(1,:) = (/ 1.0, 2.0, 4.0/)
 A(2,:) = (/ 1.0, 4.0, 16.0/)
 A(3,:) = (/ 1.0, 6.0, 36.0/)
 A(4,:) = (/ 1.0, 8.0, 64.0/)
!
 B = (/ 16.99, 57.01, 120.99, 209.01 /)
!
 IPVT = 0
ENDIF
!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! and MXCOL
CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
CALL SCALAPACK_GETDIM(KBASIS, 1, MP_NB, 1, MXLDX, MXCOLX)
!
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDA, INFO)
CALL DESCINIT(DESCL, 1, NCA, 1, MP_NB, 0, 0, MP_ICTXT, 1, INFO)
CALL DESCINIT(DESCX, KBASIS, 1, MP_NB, 1, 0, 0, MP_ICTXT, MXLDX, INFO)
CALL DESCINIT(DESCB, NRA, 1, MP_MB, 1, 0, 0, MP_ICTXT, MXLDA, INFO)
!
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), QR0(MXLDA,MXCOL), QRAUX0(MXCOL), &
 IPVT0(MXCOL), B0(MXLDA), X0(MXLDX), RES0(MXLDA), QTB0(MXLDA))
!
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)

```

```

CALL SCALAPACK_MAP(B, DESCB, B0)
PIVOT = .TRUE.
CALL SCALAPACK_MAP(IPVT, DESCL, IPVT0)
! QR factorization
CALL LQRRR (A0, QR0, QRAUX0, PIVOT=PIVOT, IPVT=IPVT0)
IPATH = 00110
CALL LQRSL (KBASIS, QR0, QRAUX0, B0, IPATH, QTB=QTB0, X=X0, RES=RES0)
! Unmap the results from the distributed
! array back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(IPVT0, DESCL, IPVT, NCA, .FALSE.)
CALL SCALAPACK_UNMAP(X0, DESCX, X)
CALL SCALAPACK_UNMAP(RES0, DESCB, RES)
! Print results.
! Only Rank=0 has the solution, X.
IF(MP_RANK .EQ. 0) THEN
 CALL WRIRN ('IPVT', IPVT, 1, NCA, 1)
 CALL WRRRN ('X', X, 1, KBASIS, 1)
 CALL WRRRN ('RES', RES, 1, NRA, 1)
ENDIF

! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

IPVT
 1 2 3
 3 2 1

 X
 1 2 3
3.000 2.002 0.990

 RES
 1 2 3 4
-0.00400 0.01200 -0.01200 0.00400

```

Note that since `IPVT` is (3, 2, 1) the array `X` contains the solution coefficients  $c_i$  in reverse order.

---

## LUPQR

Computes an updated *QR* factorization after the rank-one matrix  $\alpha xy^T$  is added.

### Required Arguments

**ALPHA** — Scalar determining the rank-one update to be added. (Input)

**W** — Vector of length `NROW` determining the rank-one matrix to be added. (Input)  
The updated matrix is  $A + \alpha xy^T$ . If `I` = 0 then *W* contains the vector *x*. If `I` = 1 then *W* contains the vector  $Q^T x$ .

**Y** — Vector of length `NCOL` determining the rank-one matrix to be added. (Input)

**R** — Matrix of order `NROW` by `NCOL` containing the *R* matrix from the *QR* factorization. (Input)  
Only the upper trapezoidal part of *R* is referenced.

**IPATH** — Flag used to control the computation of the *QR* update. (Input)  
`IPATH` has the decimal expansion `IJ` such that: `I` = 0 means *W* contains the vector *x*.  
`I` = 1 means *W* contains the vector  $Q^T x$ .  
`J` = 0 means do not update the matrix *Q*. `J` = 1 means update the matrix *Q*. For example, if `IPATH` = 10 then, `I` = 1 and `J` = 0.

**RNEW** — Matrix of order `NROW` by `NCOL` containing the updated *R* matrix in the *QR* factorization. (Output)  
Only the upper trapezoidal part of *RNEW* is updated. *R* and *RNEW* may be the same.

## Optional Arguments

**NROW** — Number of rows in the matrix  $A = Q * R$ . (Input)  
Default: `NROW` = size (*W*,1).

**NCOL** — Number of columns in the matrix  $A = Q * R$ . (Input)  
Default: `NCOL` = size (*Y*,1).

**Q** — Matrix of order `NROW` containing the *Q* matrix from the *QR* factorization. (Input)  
Ignored if `IPATH` = 0.  
Default: *Q* is 1x1 and un-initialized.

**LDQ** — Leading dimension of *Q* exactly as specified in the dimension statement of the calling program. (Input)  
Ignored if `IPATH` = 0.  
Default: `LDQ` = size (*Q*,1).

**LDR** — Leading dimension of *R* exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDR` = size (*R*,1).

**QNEW** — Matrix of order `NROW` containing the updated *Q* matrix in the *QR* factorization. (Output)  
Ignored if `J` = 0, see `IPATH` for definition of `J`.

**LDQNEW** — Leading dimension of *QNEW* exactly as specified in the dimension statement of the calling program. (Input)

Ignored if  $J = 0$ ; see `IPATH` for definition of  $J$ .  
 Default: `LDQNEW = size(QNEW,1)`.

**LDRNEW** — Leading dimension of `RNEW` exactly as specified in the dimension statement of the calling program. (Input)  
 Default: `LDRNEW = size(RNEW,1)`.

## FORTRAN 90 Interface

Generic: `CALL LUPQR (ALPHA, W, Y, R, IPATH, RNEW [, ...])`

Specific: The specific interface names are `S_LUPQR` and `D_LUPQR`.

## FORTRAN 77 Interface

Single: `CALL LUPQR (NROW, NCOL, ALPHA, W, Y, Q, LDQ, R, LDR, IPATH, QNEW, LDQNEW, RNEW, LDRNEW)`

Double: The double precision name is `DLUPQR`.

## Description

Let  $A$  be an  $m \times n$  matrix and let  $A = QR$  be its  $QR$  decomposition. (In the program,  $m$  is called `NROW` and  $n$  is called `NCOL`.) Then

$$A + \alpha xy^T = QR + \alpha xy^T = Q(R + \alpha Q^T xy^T) = Q(R + \alpha wy^T)$$

where  $w = Q^T x$ . An orthogonal transformation  $J$  can be constructed, using a sequence of  $m - 1$  Givens rotations, such that  $Jw = \omega e_1$ , where  $\omega = \pm \|w\|_2$  and  $e_1 = (1, 0, \dots, 0)^T$ . Then

$$A + \alpha xy^T = (QJ^T)(JR + \alpha \omega e_1 y^T)$$

Since  $JR$  is an upper Hessenberg matrix,  $H = JR + \alpha \omega e_1 y^T$  is also an upper Hessenberg matrix.

Again using  $m - 1$  Givens rotations, an orthogonal transformation  $G$  can be constructed such that  $GH$  is an upper triangular matrix. Then

$$A + \alpha xy^T = \tilde{Q}\tilde{R}, \text{ where } \tilde{Q} = QJ^T G^T$$

is orthogonal and

$$\tilde{R} = GH$$

is upper triangular.

If the last  $k$  components of  $w$  are zero, then the number of Givens rotations needed to construct  $J$  or  $G$  is  $m - k - 1$  instead of  $m - 1$ .

For further information, see Dennis and Schnabel (1983, pages 55–58 and 311–313), or Golub and Van Loan (1983, pages 437–439).

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2PQR/DL2PQR. The reference is:

```
CALL L2PQR (NROW, NCOL, ALPHA, W, Y, Q, LDQ, R, LDR, IPATH, QNEW,
 LDQNEW, RNEW, LDRNEW, Z, WORK)
```

The additional arguments are as follows:

**Z** — Work vector of length NROW.

**WORK** — Work vector of length MIN(NROW - 1, NCOL).

## Example

The  $QR$  factorization of  $A$  is found. It is then used to find the  $QR$  factorization of  $A + xy^T$ . Since pivoting is used, the  $QR$  factorization routine finds  $AP = QR$ , where  $P$  is a permutation matrix determined by  $IPVT$ . We compute

$$AP + \alpha xy^T = \left( A + \alpha x(Py)^T \right) P = \tilde{Q}\tilde{R}$$

The IMSL routine `PERMU` (see [Utilities](#)) is used to compute  $Py$ . As a check

$$\tilde{Q}\tilde{R}$$

is computed and printed. It can also be obtained from  $A + xy^T$  by permuting its columns using the order given by  $IPVT$ .

```
USE IMSL_LIBRARIES
!
! Declare variables
INTEGER LDA, LDAQR, LDQ, LDQNEW, LDQR, LDR, LDRNEW, NCOL, NROW
PARAMETER (NCOL=3, NROW=4, LDA=NROW, LDAQR=NROW, LDQ=NROW, &
 LDQNEW=NROW, LDQR=NROW, LDR=NROW, LDRNEW=NROW)
!
INTEGER IPATH, IPVT(NCOL), J, MIN0
REAL A(LDA, NCOL), ALPHA, AQR(LDAQR, NCOL), CONORM(NCOL), &
 Q(LDQ, NROW), QNEW(LDQNEW, NROW), QR(LDQR, NCOL), &
 QRAUX(NCOL), R(LDR, NCOL), RNEW(LDRNEW, NCOL), W(NROW), &
 Y(NCOL)
LOGICAL PIVOT
INTRINSIC MIN0
!
! Set values for A
!
! A = (1 2 4)
! (1 4 16)
! (1 6 36)
! (1 8 64)
!
DATA A/4*1.0, 2.0, 4.0, 6.0, 8.0, 4.0, 16.0, 36.0, 64.0/
```

```

! Set values for W and Y
 DATA W/1., 2., 3., 4./
 DATA Y/3., 2., 1./
!
! QR factorization
! Set IPVT = 0 (all columns free)
 IPVT = 0
 PIVOT = .TRUE.
 CALL LQRRR (A, QR, QRAUX, IPVT=IPVT, PIVOT=PIVOT)
! Accumulate Q
 CALL LQERR (QR, QRAUX, Q)
! Permute Y
 CALL PERMU (Y, IPVT, Y)
! R is the upper trapezoidal part of QR
 R = 0.0E0
 DO 10 J=1, NCOL
 CALL SCOPY (MIN0(J,NROW), QR(:,J), 1, R(:,J), 1)
10 CONTINUE
! Update Q and R
 ALPHA = 1.0
 IPATH = 01
 CALL LUPQR (ALPHA, W, Y, R, IPATH, RNEW, Q=Q, QNEW=QNEW)
! Compute AQR = Q*R
 CALL MRRRR (QNEW, RNEW, AQR)
! Print results
 CALL WRIRN ('IPVT', IPVT, 1, NCOL,1)
 CALL WRRRN ('QNEW', QNEW)
 CALL WRRRN ('RNEW', RNEW)
 CALL WRRRN ('QNEW*RNEW', AQR)
 END

```

## Output

```

IPVT
1 2 3
3 2 1

 QNEW
 1 2 3 4
1 -0.0620 -0.5412 0.8082 -0.2236
2 -0.2234 -0.6539 -0.2694 0.6708
3 -0.4840 -0.3379 -0.4490 -0.6708
4 -0.8438 0.4067 0.2694 0.2236

 RNEW
 1 2 3
1 -80.59 -21.34 -17.62
2 0.00 -4.94 -4.83
3 0.00 0.00 0.36
4 0.00 0.00 0.00

 QNEW*RNEW
 1 2 3
1 5.00 4.00 4.00
2 18.00 8.00 7.00
3 39.00 12.00 10.00
4 68.00 16.00 13.00

```

---

# LCHRG

Computes the Cholesky decomposition of a symmetric positive definite matrix with optional column pivoting.

## Required Arguments

*A* —  $N$  by  $N$  symmetric positive definite matrix to be decomposed. (Input)  
Only the upper triangle of *A* is referenced.

*FACT* —  $N$  by  $N$  matrix containing the Cholesky factor of the permuted matrix in its upper triangle. (Output)  
If *A* is not needed, *A* and *FACT* can share the same storage locations.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default:  $N = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

*PIVOT* — Logical variable. (Input)  
`PIVOT = .TRUE.` means column pivoting is done. `PIVOT = .FALSE.` means no pivoting is done.  
Default: `PIVOT = .TRUE.`

*IPVT* — Integer vector of length  $N$  containing information that controls the selection of the pivot columns. (Input/Output)  
On input, if  $IPVT(K) > 0$ , then the  $K$ -th column of *A* is an initial column; if  $IPVT(K) = 0$ , then the  $K$ -th column of *A* is a free column; if  $IPVT(K) < 0$ , then the  $K$ -th column of *A* is a final column. See [Comments](#). On output,  $IPVT(K)$  contains the index of the diagonal element of *A* that was moved into the  $K$ -th position. *IPVT* is only referenced when *PIVOT* is equal to `.TRUE.`

*LDFACT* — Leading dimension of *FACT* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFACT = \text{size}(FACT,1)$ .

## FORTRAN 90 Interface

Generic:     `CALL LCHRG (A, FACT [, ...])`

Specific:    The specific interface names are `S_LCHRG` and `D_LCHRG`.

## FORTRAN 77 Interface

Single:      CALL LCHRG (N, A, LDA, PIVOT, IPVT, FACT, LDFACT)

Double:     The double precision name is DLCHRG.

## Description

Routine LCHRG is based on the LINPACK routine SCHDC; see Dongarra et al. (1979).

Before the decomposition is computed, initial elements are moved to the leading part of  $A$  and final elements to the trailing part of  $A$ . During the decomposition only rows and columns corresponding to the free elements are moved. The result of the decomposition is an upper triangular matrix  $R$  and a permutation matrix  $P$  that satisfy  $P^T AP = R^T R$ , where  $P$  is represented by IPVT.

## Comments

1. Informational error

Type   Code

4      1           The input matrix is not positive definite.

2. Before the decomposition is computed, initial elements are moved to the leading part of  $A$  and final elements to the trailing part of  $A$ . During the decomposition only rows and columns corresponding to the free elements are moved. The result of the decomposition is an upper triangular matrix  $R$  and a permutation matrix  $P$  that satisfy  $P^T AP = R^T R$ , where  $P$  is represented by IPVT.

3. LCHRG can be used together with subroutines PERMU and LSLDS to solve the positive definite linear system  $AX = B$  with the solution  $X$  overwriting the right-hand side  $B$  as follows:

```
CALL ISET (N, 0, IPVT, 1)
CALL LCHRG (A, FACT, N, LDA, .TRUE, IPVT, LDFACT)
CALL PERMU (B, IPVT, B, N, 1)
CALL LSLDS (FACT, B, B, N, LDFACT)
CALL PERMU (B, IPVT, B, N, 2)
```

## Example

Routine LCHRG can be used together with the IMSL routines PERMU (see [Chapter 11](#)) and LFSDS to solve a positive definite linear system  $Ax = b$ . Since  $A = PR^T RP$ , the system  $Ax = b$  is equivalent to  $R^T R(Px) = Pb$ . LFSDS is used to solve  $R^T Ry = Pb$  for  $y$ . The routine PERMU is used to compute both  $Pb$  and  $x = Py$ .

```
USE IMSL_LIBRARIES
!
! Declare variables
PARAMETER (N=3, LDA=N, LDFACT=N)
INTEGER IPVT(N)
REAL A(LDA,N), FACT(LDFACT,N), B(N), X(N)
```

```

LOGICAL PIVOT
!
! Set values for A and B
!
! A = (1 -3 2)
! (-3 10 -5)
! (2 -5 6)
!
! B = (27 -78 64)
!
! DATA A/1.,-3.,2.,-3.,10.,-5.,2.,-5.,6./
! DATA B/27.,-78.,64./
!
! Pivot using all columns
PIVOT = .TRUE.
IPVT = 0
!
! Compute Cholesky factorization
CALL LCHRG (A, FACT, PIVOT=PIVOT, IPVT=IPVT)
!
! Permute B and store in X
CALL PERMU (B, IPVT, X, IPATH=1)
!
! Solve for X
CALL LFSDS (FACT, X, X)
!
! Inverse permutation
CALL PERMU (X, IPVT, X, IPATH=2)
!
! Print X
CALL WRRRN ('X', X, 1, N, 1)
!
END

```

## Output

```

 X
 1 2 3
1.000 -4.000 7.000

```

---

## LUPCH

Updates the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.

### Required Arguments

**R** —  $N$  by  $N$  upper triangular matrix containing the upper triangular factor to be updated.  
(Input)  
Only the upper triangle of  $R$  is referenced.

**X** — Vector of length  $N$  determining the rank-one matrix to be added to the factorization  $R^T R$ . (Input)

**RNEW** —  $N$  by  $N$  upper triangular matrix containing the updated triangular factor of  $R^T R + XX^T$ . (Output)

Only the upper triangle of  $R_{NEW}$  is referenced. If  $R$  is not needed,  $R$  and  $R_{NEW}$  can share the same storage locations.

### Optional Arguments

$N$  — Order of the matrix. (Input)  
Default:  $N = \text{size}(R, 2)$ .

$LDR$  — Leading dimension of  $R$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDR = \text{size}(R, 1)$ .

$LDR_{NEW}$  — Leading dimension of  $R_{NEW}$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDR_{NEW} = \text{size}(R_{NEW}, 1)$ .

$CS$  — Vector of length  $N$  containing the cosines of the rotations. (Output)

$SN$  — Vector of length  $N$  containing the sines of the rotations. (Output)

### FORTRAN 90 Interface

Generic: `CALL LUPCH (R, X, RNEW [, ...])`

Specific: The specific interface names are `S_LUPCH` and `D_LUPCH`.

### FORTRAN 77 Interface

Single: `CALL LUPCH (N, R, LDR, X, RNEW, LDRNEW, CS, SN)`

Double: The double precision name is `DLUPCH`.

### Description

The routine `LUPCH` is based on the LINPACK routine `SCHUD`; see Dongarra et al. (1979).

The Cholesky factorization of a matrix is  $A = R^T R$ , where  $R$  is an upper triangular matrix. Given this factorization, `LUPCH` computes the factorization

$$A + xx^T = \tilde{R}^T \tilde{R}$$

In the program

$$\tilde{R}$$

is called  $R_{NEW}$ .

`LUPCH` determines an orthogonal matrix  $U$  as the product  $G_N \dots G_1$  of Givens rotations, such that

$$U \begin{bmatrix} R \\ x^T \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ 0 \end{bmatrix}$$

By multiplying this equation by its transpose, and noting that  $U^T U = I$ , the desired result

$$R^T R + x x^T = \tilde{R}^T \tilde{R}$$

is obtained.

Each Givens rotation,  $G_i$ , is chosen to zero out an element in  $x^T$ . The matrix  $G_i$  is  $(N+1) \times (N+1)$  and has the form

$$G_i = \begin{bmatrix} I_{i-1} & 0 & 0 & 0 \\ 0 & c_i & 0 & s_i \\ 0 & 0 & I_{N-i} & 0 \\ 0 & -s_i & 0 & c_i \end{bmatrix}$$

Where  $I_k$  is the identity matrix of order  $k$  and  $c_i = \cos \theta_i = \text{CS}(I)$ ,  $s_i = \sin \theta_i = \text{SN}(I)$  for some  $\theta_i$ .

### Example

A linear system  $Az = b$  is solved using the Cholesky factorization of  $A$ . This factorization is then updated and the system  $(A + xx^T)z = b$  is solved using this updated factorization.

```

 USE IMSL_LIBRARIES
!
! Declare variables
 INTEGER LDA, LDFACT, N
 PARAMETER (LDA=3, LDFACT=3, N=3)
 REAL A(LDA,LDA), FACT(LDFACT,LDFACT), FACNEW(LDFACT,LDFACT), &
 X(N), B(N), CS(N), SN(N), Z(N)
!
! Set values for A
! A = (1.0 -3.0 2.0)
! (-3.0 10.0 -5.0)
! (2.0 -5.0 6.0)
!
 DATA A/1.0, -3.0, 2.0, -3.0, 10.0, -5.0, 2.0, -5.0, 6.0/
!
! Set values for X and B
!
 DATA X/3.0, 2.0, 1.0/
 DATA B/53.0, 20.0, 31.0/
!
! Factor the matrix A
 CALL LFTDS (A, FACT)
!
! Solve the original system
 CALL LFSDS (FACT, B, Z)
!
! Print the results
 CALL WRRRN ('FACT', FACT, ITRING=1)
 CALL WRRRN ('Z', Z, 1, N, 1)
!
! Update the factorization
 CALL LUPCH (FACT, X, FACNEW)

```

```

! Solve the updated system
 CALL LFSDS (FACNEW, B, Z)
! Print the results
 CALL WRRRN ('FACNEW', FACNEW, ITRING=1)
 CALL WRRRN ('Z', Z, 1, N, 1)
!
 END

```

## Output

```

 FACT
 1 2 3
1 1.000 -3.000 2.000
2 1.000 1.000
3 1.000

```

```

 Z
 1 2 3
1860.0 433.0 -254.0

```

```

 FACNEW
 1 2 3
1 3.162 0.949 1.581
2 3.619 -1.243
3 -1.719

```

```

 Z
 1 2 3
4.000 1.000 2.000

```

---

## LDNCH



Downdates the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.

### Required Arguments

**R** —  $N$  by  $N$  upper triangular matrix containing the upper triangular factor to be downdated.  
(Input)  
Only the upper triangle of **R** is referenced.

**X** — Vector of length  $N$  determining the rank-one matrix to be subtracted from the factorization  $R^T R$ . (Input)

**RNEW** —  $N$  by  $N$  upper triangular matrix containing the downdated triangular factor of  $R^T R - X X^T$ . (Output)  
Only the upper triangle of **RNEW** is referenced. If **R** is not needed, **R** and **RNEW** can share the same storage locations.

## Optional Arguments

*N* — Order of the matrix. (Input)

Default:  $N = \text{size}(R, 2)$ .

*LDR* — Leading dimension of *R* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDR = \text{size}(R, 1)$ .

*LDRNEW* — Leading dimension of *RNEW* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDRNEW = \text{size}(RNEW, 1)$ .

*CS* — Vector of length *N* containing the cosines of the rotations. (Output)

*SN* — Vector of length *N* containing the sines of the rotations. (Output)

## FORTRAN 90 Interface

Generic:     CALL LDNCH (R, X, RNEW [, ...])

Specific:    The specific interface names are S\_LDNCH and D\_LDNCH.

## FORTRAN 77 Interface

Single:     CALL LDNCH (N, R, LDR, X, RNEW, LDRNEW, CS, SN)

Double:     The double precision name is DLDNCH.

## Description

The routine LDNCH is based on the LINPACK routine SCHDD; see Dongarra et al. (1979).

The Cholesky factorization of a matrix is  $A = R^T R$ , where *R* is an upper triangular matrix. Given this factorization, LDNCH computes the factorization

$$A - xx^T = \tilde{R}^T \tilde{R}$$

In the program

$$\tilde{R}$$

is called *RNEW*. This is not always possible, since  $A - xx^T$  may not be positive definite.

LDNCH determines an orthogonal matrix *U* as the product  $G_N \dots G_1$  of Givens rotations, such that

$$U \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ x^T \end{bmatrix}$$

By multiplying this equation by its transpose and noting that  $U^T U = I$ , the desired result

$$R^T R - xx^T = \tilde{R}^T \tilde{R}$$

is obtained.

Let  $a$  be the solution of the linear system  $R^T a = x$  and let

$$\alpha = \sqrt{1 - \|a\|_2^2}$$

The Givens rotations,  $G_i$ , are chosen such that

$$G_1 \cdots G_N \begin{bmatrix} a \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The  $G_i$  are  $(N + 1) \times (N + 1)$  matrices of the form

$$G_i = \begin{bmatrix} I_{i-1} & 0 & 0 & 0 \\ 0 & c_i & 0 & -s_i \\ 0 & 0 & I_{N-i} & 0 \\ 0 & s_i & 0 & c_i \end{bmatrix}$$

where  $I_k$  is the identity matrix of order  $k$ ; and  $c_i = \cos \theta_i = \text{CS}(\mathcal{I})$ ,  $s_i = \sin \theta_i = \text{SN}(\mathcal{I})$  for some  $\theta_i$ .

The Givens rotations are then used to form

$$\tilde{R}, G_1 \cdots G_N \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ \tilde{x}^T \end{bmatrix}$$

The matrix

$$\tilde{R}$$

is upper triangular and

$$\tilde{x} = x$$

because

$$x = (R^T 0) \begin{bmatrix} a \\ \alpha \end{bmatrix} = (R^T 0) U^T U \begin{bmatrix} a \\ \alpha \end{bmatrix} = (\tilde{R}^T \tilde{x}) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \tilde{x}$$

## Comments

Informational error

Type Code

4 1  $R^T R - X X^T$  is not positive definite. R cannot be downdated.

## Example

A linear system  $Az = b$  is solved using the Cholesky factorization of  $A$ . This factorization is then downdated, and the system  $(A - xx^T)z = b$  is solved using this downdated factorization.

```
 USE LDNCH_INT
 USE LFTDS_INT
 USE LFSDS_INT
 USE WRRRN_INT
!
! Declare variables
 INTEGER LDA, LDFACT, N
 PARAMETER (LDA=3, LDFACT=3, N=3)
 REAL A(LDA,LDA), FACT(LDFACT,LDFACT), FACNEW(LDFACT,LDFACT), &
 X(N), B(N), CS(N), SN(N), Z(N)
!
! Set values for A
! A = (10.0 3.0 5.0)
! (3.0 14.0 -3.0)
! (5.0 -3.0 7.0)
!
 DATA A/10.0, 3.0, 5.0, 3.0, 14.0, -3.0, 5.0, -3.0, 7.0/
!
! Set values for X and B
 DATA X/3.0, 2.0, 1.0/
 DATA B/53.0, 20.0, 31.0/
!
! Factor the matrix A
 CALL LFTDS (A, FACT)
!
! Solve the original system
 CALL LFSDS (FACT, B, Z)
!
! Print the results
 CALL WRRRN ('FACT', FACT, ITRING=1)
 CALL WRRRN ('Z', Z, 1, N, 1)
!
! Downdate the factorization
 CALL LDNCH (FACT, X, FACNEW)
!
! Solve the updated system
 CALL LFSDS (FACNEW, B, Z)
!
! Print the results
 CALL WRRRN ('FACNEW', FACNEW, ITRING=1)
 CALL WRRRN ('Z', Z, 1, N, 1)
!
 END
```

## Output

```
 FACT
 1 2 3
1 3.162 0.949 1.581
2 3.619 -1.243
3 1.719
 Z
 1 2 3
4.000 1.000 2.000
 FACNEW
 1 2 3
```

```

1 1.000 -3.000 2.000
2 1.000 1.000
3 1.000

```

```

 Z
 1 2 3
1859.9 433.0 -254.0

```

## LSVRR



Computes the singular value decomposition of a real matrix.

### Required Arguments

**A** —  $NRA$  by  $NCA$  matrix whose singular value decomposition is to be computed. (Input)

**IPATH** — Flag used to control the computation of the singular vectors. (Input)

**IPATH** has the decimal expansion  $IJ$  such that:

$I = 0$  means do not compute the left singular vectors;

$I = 1$  means return the  $NRA$  left singular vectors in  $U$ ;

**NOTE:** This option is not available for the ScaLAPACK interface. If this option is chosen for ScaLAPACK usage, the  $\min(NRA, NCA)$  left singular vectors will be returned.

$I = 2$  means return only the  $\min(NRA, NCA)$  left singular vectors in  $U$ ;

$J = 0$  means do not compute the right singular vectors,

$J = 1$  means return the right singular vectors in  $V$ .

For example,  $IPATH = 20$  means  $I = 2$  and  $J = 0$ .

**S** — Vector of length  $\min(NRA + 1, NCA)$  containing the singular values of  $A$  in descending order of magnitude in the first  $\min(NRA, NCA)$  positions. (Output)

### Optional Arguments

**NRA** — Number of rows in the matrix  $A$ . (Input)

Default:  $NRA = \text{size}(A,1)$ .

**NCA** — Number of columns in the matrix  $A$ . (Input)

Default:  $NCA = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

**TOL** — Scalar containing the tolerance used to determine when a singular value is negligible. (Input)

If **TOL** is positive, then a singular value  $\sigma_i$  considered negligible if  $\sigma_i \leq \text{TOL}$ . If **TOL** is negative, then a singular value  $\sigma_i$  considered negligible if  $\sigma_i \leq |\text{TOL}| * \|A\|_\infty$ . In this case,  $|\text{TOL}|$  generally contains an estimate of the level of the relative error in the data. Default: **TOL** = 1.0e-5 for single precision and 1.0d-10 for double precision.

**IRANK** — Scalar containing an estimate of the rank of **A**. (Output)

**U** — **NRA** by **NCU** matrix containing the left singular vectors of **A**. (Output)

**NCU** must be equal to **NRA** if **I** is equal to 1. **NCU** must be equal to  $\min(\text{NRA}, \text{NCA})$  if **I** is equal to 2. **U** will not be referenced if **I** is equal to zero. If **NRA** is less than or equal to **NCU**, then **U** can share the same storage locations as **A**. See [Comments](#).

**LDU** — Leading dimension of **U** exactly as specified in the dimension statement of the calling program. (Input)

Default: **LDU** = size (**U**,1).

**V** — **NCA** by **NCA** matrix containing the right singular vectors of **A**. (Output)

**V** will not be referenced if **J** is equal to zero. **V** can share the same storage location as **A**, however, **U** and **V** cannot both coincide with **A** simultaneously.

**LDV** — Leading dimension of **V** exactly as specified in the dimension statement of the calling program. (Input)

Default: **LDV** = size (**V**,1).

## **FORTRAN 90 Interface**

Generic:     CALL LSVRR (A, IPATH, S [ , ... ])

Specific:    The specific interface names are **S\_LSVRR** and **D\_LSVRR**.

## **FORTRAN 77 Interface**

Single:     CALL LSVRR (NRA, NCA, A, LDA, IPATH, TOL, IRANK, S, U, LDU, V, LDV)

Double:     The double precision name is **DLSVRR**.

## **ScaLAPACK Interface**

Generic:     CALL LSVRR (A0, IPATH, S [ , ... ])

Specific:    The specific interface names are **S\_LSVRR** and **D\_LSVRR**.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

The underlying code of routine `LSVRR` is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

Let  $n = \text{NRA}$  (the number of rows in  $A$ ) and let  $p = \text{NCA}$  (the number of columns in  $A$ ). For any  $n \times p$  matrix  $A$ , there exists an  $n \times n$  orthogonal matrix  $U$  and a  $p \times p$  orthogonal matrix  $V$  such that

$$U^T A V = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \geq p \\ \begin{bmatrix} \Sigma & 0 \end{bmatrix} & \text{if } n \leq p \end{cases}$$

where  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_m)$ , and  $m = \min(n, p)$ . The scalars  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_m \geq 0$  are called the *singular values* of  $A$ . The columns of  $U$  are called the *left singular vectors* of  $A$ . The columns of  $V$  are called the *right singular vectors* of  $A$ .

The estimated rank of  $A$  is the number of  $\sigma_k$  that is larger than a tolerance  $\eta$ . If  $\tau$  is the parameter `TOL` in the program, then

$$\eta = \begin{cases} \tau & \text{if } \tau > 0 \\ |\tau| \|A\|_\infty & \text{if } \tau < 0 \end{cases}$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2VRR/DL2VRR`. The reference is:

```
CALL L2VRR (NRA, NCA, A, LDA, IPATH, TOL, IRANK, S, U, LDU, V, LDV,
 ACOPY, WK)
```

The additional arguments are as follows:

**ACOPY** —  $\text{NRA} \times \text{NCA}$  work array for the matrix  $A$ . If  $A$  is not needed, then  $A$  and  $ACOPY$  may share the same storage locations.

**WK** — Work vector of length  $\text{NRA} + \text{NCA} + \max(\text{NRA}, \text{NCA}) - 1$ .

2. Informational error
 

| Type | Code |                                                                                                      |
|------|------|------------------------------------------------------------------------------------------------------|
| 4    | 1    | Convergence cannot be achieved for all the singular values and their corresponding singular vectors. |
3. When  $\text{NRA}$  is much greater than  $\text{NCA}$ , it might not be reasonable to store the whole matrix  $U$ . In this case, `IPATH` with `I = 2` allows a singular value factorization of  $A$  to be

computed in which only the first  $NCA$  columns of  $U$  are computed, and in many applications those are all that are needed.

4. [Integer Options](#) with Chapter 11 Options Manager

- 16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `L2VRR` the leading dimension of `ACOPY` is increased by `IVAL(3)` when  $N$  is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in `LSVRR`. Additional memory allocation for `ACOPY` and option value restoration are done automatically in `LSVRR`. Users directly calling `L2VRR` can allocate additional space for `ACOPY` and set `IVAL(3)` and `IVAL(4)` so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing applications that use `LSVRR` or `L2VRR`. Default values for the option are `IVAL(*) = 1, 16, 0, 1`.
- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine `LSVRR` temporarily replaces `IVAL(2)` by `IVAL(1)`. The routine `L2CRG` computes the condition number if `IVAL(2) = 2`. Otherwise `L2CRG` skips this computation. `LSVRR` restores the option. Default values for the option are `IVAL(*) = 1, 2`.

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

- A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix  $A$ .  $A$  contains the matrix whose singular value decomposition is to be computed. (Input)
- U0** — `MXLDU` by `MXCOLU` local matrix containing the local portions of the left singular vectors of the distributed matrix  $A$ . (Output)  
**U0** will not be referenced if  $I$  is equal to zero. If  $NRA$  is less than or equal to  $NCU$ , then **U0** can share the same storage locations as **A0**. See [Comments](#).
- V0** — `MXLDV` by `MXCOLV` local matrix containing the local portions of the right singular vectors of the distributed matrix  $A$ . (Output)  
**V0** will not be referenced if  $J$  is equal to zero. **V0** can share the same storage location as **A0**, however, **U0** and **V0** cannot both coincide with **A0** simultaneously.

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA`, `MXCOL`, `MXLDU`, `MXCOLU`, `MXLDV` and `MXCOLV` can be obtained through a call to `ScaLAPACK_GETDIM` (Chapter 11, “Utilities”) after a call to `ScaLAPACK_SETUP` (Chapter 11, “Utilities”) has been made. See the [ScaLAPACK Example](#) below.

## Example

This example computes the singular value decomposition of a  $6 \times 4$  matrix  $A$ . The matrices  $U$  and  $V$  containing the left and right singular vectors, respectively, and the diagonal of  $\Sigma$ , containing singular values, are printed. On some systems, the signs of some of the columns of  $U$  and  $V$  may be reversed.

```
 USE IMSL_LIBRARIES
!
! Declare variables
PARAMETER (NRA=6, NCA=4, LDA=NRA, LDU=NRA, LDV=NCA)
REAL A(LDA,NCA), U(LDU,NRA), V(LDV,NCA), S(NCA)
!
! Set values for A
!
! A = (1 2 1 4)
! (3 2 1 3)
! (4 3 1 4)
! (2 1 3 1)
! (1 5 2 2)
! (1 2 2 3)
!
DATA A/1., 3., 4., 2., 1., 1., 2., 2., 3., 1., 5., 2., 3*1., &
 3., 2., 2., 4., 3., 4., 1., 2., 3./
!
! Compute all singular vectors
IPATH = 11
TOL = AMACH(4)
TOL = 10.*TOL
CALL LSVRR(A, IPATH, S, TOL=TOL, IRANK=IRANK, U=U, V=V)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, *) 'IRANK = ', IRANK
CALL WRRRN ('U', U, NRA, NCA)
CALL WRRRN ('S', S, 1, NCA, 1)
CALL WRRRN ('V', V)
!
END
```

## Output

```
IRANK = 4

 U
 1 2 3 4
1 -0.3805 0.1197 0.4391 -0.5654
2 -0.4038 0.3451 -0.0566 0.2148
3 -0.5451 0.4293 0.0514 0.4321
4 -0.2648 -0.0683 -0.8839 -0.2153
5 -0.4463 -0.8168 0.1419 0.3213
6 -0.3546 -0.1021 -0.0043 -0.5458

 S
 1 2 3 4
11.49 3.27 2.65 2.09

 V
 1 2 3 4
```

```

1 -0.4443 0.5555 -0.4354 0.5518
2 -0.5581 -0.6543 0.2775 0.4283
3 -0.3244 -0.3514 -0.7321 -0.4851
4 -0.6212 0.3739 0.4444 -0.5261

```

## ScaLAPACK Example

The previous example is repeated here as a distributed example. This example computes the singular value decomposition of a  $6 \times 4$  matrix  $A$ . The matrices  $U$  and  $V$  containing the left and right singular vectors, respectively, and the diagonal of  $S$ , containing singular values, are printed. On some systems, the signs of some of the columns of  $U$  and  $V$  may be reversed..

```

 USE MPI_SETUP_INT
 USE IMSL_LIBRARIES
 USE SCALAPACK_SUPPORT
 IMPLICIT NONE
 INCLUDE 'mpif.h'

!
! Declare variables
 INTEGER KBASIS, LDA, LDQR, NCA, NRA, DESCA(9), DESCU(9), &
 DESCV(9), MXLDV, MXCOLV, NSZ, MXLDU, MXCOLU
 INTEGER INFO, MXCOL, MXLDA, LDU, LDV, IPATH, IRANK
 REAL TOL, AMACH
 REAL, ALLOCATABLE :: A(:, :), U(:, :), V(:, :), S(:)
 REAL, ALLOCATABLE :: A0(:, :), U0(:, :), V0(:, :), S0(:)
 PARAMETER (NRA=6, NCA=4, LDA=NRA, LDU=NRA, LDV=NCA)
 NSZ = MIN(NRA, NCA)

!
! Set up for MPI
 MP_NPROCS = MP_SETUP()
 IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA, NCA), U(LDU, NCA), V(LDV, NCA), S(NCA))
!
! Set values for A
 A(1, :) = (/ 1.0, 2.0, 1.0, 4.0/)
 A(2, :) = (/ 3.0, 2.0, 1.0, 3.0/)
 A(3, :) = (/ 4.0, 3.0, 1.0, 4.0/)
 A(4, :) = (/ 2.0, 1.0, 3.0, 1.0/)
 A(5, :) = (/ 1.0, 5.0, 2.0, 2.0/)
 A(6, :) = (/ 1.0, 2.0, 2.0, 3.0/)
 ENDIF

!
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
 CALL SCALAPACK_SETUP(NRA, NCA, .TRUE., .TRUE.)
!
! Get the array descriptor entities MXLDA,
! MXCOL, MXLDU, MXCOLU, MXLDV, AND MXCOLV
 CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
 CALL SCALAPACK_GETDIM(NRA, NSZ, MP_MB, MP_NB, MXLDU, MXCOLU)
 CALL SCALAPACK_GETDIM(NSZ, NCA, MP_MB, MP_NB, MXLDV, MXCOLV)
!
! Set up the array descriptors
 CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDA, INFO)
 CALL DESCINIT(DESCU, NRA, NSZ, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDU, INFO)
 CALL DESCINIT(DESCV, NSZ, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDV, INFO)
!
! Allocate space for the local arrays
 ALLOCATE (A0(MXLDA, MXCOL), U0(MXLDU, MXCOLU), V0(MXLDV, MXCOLV), S(NCA))

```

```

! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Compute all singular vectors
IPATH = 11
TOL = AMACH(4)
TOL = 10. * TOL
CALL LSVRR (A0, IPATH, S, TOL=TOL, IRANK=IRANK, U=U0, V=V0)
! Unmap the results from the distributed
! array back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(U0, DESCU, U)
CALL SCALAPACK_UNMAP(V0, DESCV, V)
! Print results.
! Only Rank=0 has the solution.
IF(MP_RANK .EQ. 0) THEN
 CALL WRRRN ('U', U, NRA, NCA)
 CALL WRRRN ('S', S, 1, NCA, 1)
 CALL WRRRN ('V', V)
ENDIF
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

```

IRANK = 4

 U
 1 2 3 4
1 -0.3805 0.1197 0.4391 -0.5654
2 -0.4038 0.3451 -0.0566 0.2148
3 -0.5451 0.4293 0.0514 0.4321
4 -0.2648 -0.0683 -0.8839 -0.2153
5 -0.4463 -0.8168 0.1419 0.3213
6 -0.3546 -0.1021 -0.0043 -0.5458

 S
 1 2 3 4
11.49 3.27 2.65 2.09

 V
 1 2 3 4
1 -0.4443 0.5555 -0.4354 0.5518
2 -0.5581 -0.6543 0.2775 0.4283
3 -0.3244 -0.3514 -0.7321 -0.4851
4 -0.6212 0.3739 0.4444 -0.5261

```

---

# LSVCR



Computes the singular value decomposition of a complex matrix.

## Required Arguments

**A** — Complex  $NRA$  by  $NCA$  matrix whose singular value decomposition is to be computed. (Input)

**IPATH** — Integer flag used to control the computation of the singular vectors. (Input)  
 $IPATH$  has the decimal expansion  $IJ$  such that:

$I=0$  means do not compute the left singular vectors;  
 $I=1$  means return the  $NCA$  left singular vectors in  $U$ ;  
 $I=2$  means return only the  $\min(NRA, NCA)$  left singular vectors in  $U$ ;  
 $J=0$  means do not compute the right singular vectors;  
 $J=1$  means return the right singular vectors in  $V$ .

For example,  $IPATH = 20$  means  $I = 2$  and  $J = 0$ .

**S** — Complex vector of length  $\min(NRA + 1, NCA)$  containing the singular values of  $A$  in descending order of magnitude in the first  $\min(NRA, NCA)$  positions. (Output)

## Optional Arguments

**NRA** — Number of rows in the matrix  $A$ . (Input)  
Default:  $NRA = \text{size}(A,1)$ .

**NCA** — Number of columns in the matrix  $A$ . (Input)  
Default:  $NCA = \text{size}(A,2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A,1)$ .

**TOL** — Real scalar containing the tolerance used to determine when a singular value is negligible. (Input)  
If  $TOL$  is positive, then a singular value  $SI$  is considered negligible if  $SI \leq TOL$ . If  $TOL$  is negative, then a singular value  $SI$  is considered negligible if  $SI \leq |TOL| * (\text{Infinity norm of } A)$ . In this case  $|TOL|$  should generally contain an estimate of the level of relative error in the data.  
Default:  $TOL = 1.0e-5$  for single precision and  $1.0d-10$  for double precision.

**IRANK** — Integer scalar containing an estimate of the rank of  $A$ . (Output)

**U** — Complex  $NRA$  by  $NRA$  if  $I = 1$  or  $NRA$  by  $\min(NRA, NCA)$  if  $I = 2$  matrix containing the left singular vectors of  $A$ . (Output)

$U$  will not be referenced if  $I$  is equal to zero. If  $NRA$  is less than or equal to  $NCA$  or  $IPATH = 2$ , then  $U$  can share the same storage locations as  $A$ .

**LDU** — Leading dimension of  $U$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDU = \text{size}(U, 1)$ .

**V** — Complex  $NCA$  by  $NCA$  matrix containing the right singular vectors of  $A$ . (Output)

$V$  will not be referenced if  $J$  is equal to zero. If  $NCA$  is less than or equal to  $NRA$ , then  $V$  can share the same storage locations as  $A$ ; however  $U$  and  $V$  cannot both coincide with  $A$  simultaneously.

**LDV** — Leading dimension of  $V$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDV = \text{size}(V, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LSVCR (A, IPATH, S [, ...])`

Specific: The specific interface names are `S_LSVCR` and `D_LSVCR`.

## FORTRAN 77 Interface

Single: `CALL LSVCR (NRA, NCA, A, LDA, IPATH, TOL, IRANK, S, U, LDU, V, LDV)`

Double: The double precision name is `DLSVCR`.

## Description

The underlying code of routine `LSVCR` is based on either `LINPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

Let  $n = NRA$  (the number of rows in  $A$ ) and let  $p = NCA$  (the number of columns in  $A$ ). For any  $n \times p$  matrix  $A$  there exists an  $n \times n$  orthogonal matrix  $U$  and a  $p \times p$  orthogonal matrix  $V$  such that

$$U^T A V = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \geq p \\ \begin{bmatrix} \Sigma & 0 \end{bmatrix} & \text{if } n \leq p \end{cases}$$

where  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_m)$ , and  $m = \min(n, p)$ . The scalars  $\sigma_1 \geq \sigma_2 \geq \dots \geq 0$  are called the *singular values* of  $A$ . The columns of  $U$  are called the *left singular vectors* of  $A$ . The columns of  $V$  are called the *right singular vectors* of  $A$ .

The estimated rank of  $A$  is the number of  $\sigma_k$  which are larger than a tolerance  $\eta$ . If  $\tau$  is the parameter TOL in the program, then

$$\eta = \begin{cases} \tau & \text{if } \tau > 0 \\ |\tau| \|A\|_\infty & \text{if } \tau < 0 \end{cases}$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of L2VCR/DL2VCR. The reference is

```
CALL L2VCR (NRA, NCA, A, LDA, IPATH, TOL, IRANK, S, U, LDU, V, LDV,
 ACOPIY, WK)
```

The additional arguments are as follows:

**ACOPY** — NRA \* NCA complex work array of length for the matrix A. If A is not needed, then A and ACOPIY can share the same storage locations.

**WK** — Complex work vector of length NRA + NCA + max(NRA, NCA) - 1.

2. Informational error

Type Code

4 1 Convergence cannot be achieved for all the singular values and their corresponding singular vectors.

3. When NRA is much greater than NCA, it might not be reasonable to store the whole matrix U. In this case IPATH with I = 2 allows a singular value factorization of A to be computed in which only the first NCA columns of U are computed, and in many applications those are all that are needed.

4. [Integer Options](#) with Chapter 11 Options Manager

**16** This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine L2VCR the leading dimension of ACOPIY is increased by IVAL(3) when N is a multiple of IVAL(4). The values IVAL(3) and IVAL(4) are temporarily replaced by IVAL(1) and IVAL(2), respectively, in LSVCR. Additional memory allocation for ACOPIY and option value restoration are done automatically in LSVCR. Users directly calling L2VCR can allocate additional space for ACOPIY and set IVAL(3) and IVAL(4) so that memory bank conflicts no longer cause inefficiencies. There is no requirement that users change existing

applications that use LSVCR or L2VCR. Default values for the option are IVAL(\*) = 1, 16, 0, 1.

- 17** This option has two values that determine if the  $L_1$  condition number is to be computed. Routine LSVCR temporarily replaces IVAL(2) by IVAL(1). The routine L2CCG computes the condition number if IVAL(2) = 2. Otherwise L2CCG skips this computation. LSVCR restores the option. Default values for the option are IVAL(\*) = 1, 2.

## Example

This example computes the singular value decomposition of a  $6 \times 3$  matrix  $A$ . The matrices  $U$  and  $V$  containing the left and right singular vectors, respectively, and the diagonal of  $\Sigma$ , containing singular values, are printed. On some systems, the signs of some of the columns of  $U$  and  $V$  may be reversed.

```

USE IMSL_LIBRARIES
!
! Declare variables
PARAMETER (NRA=6, NCA=3, LDA=NRA, LDU=NRA, LDV=NCA)
COMPLEX A(LDA,NCA), U(LDU,NRA), V(LDV,NCA), S(NCA)
!
! Set values for A
!
! A = (1+2i 3+2i 1-4i)
! (3-2i 2-4i 1+3i)
! (4+3i -2+1i 1+4i)
! (2-1i 3+0i 3-1i)
! (1-5i 2-5i 2+2i)
! (1+2i 4-2i 2-3i)
!
DATA A/(1.0,2.0), (3.0,-2.0), (4.0,3.0), (2.0,-1.0), (1.0,-5.0), &
 (1.0,2.0), (3.0,2.0), (2.0,-4.0), (-2.0,1.0), (3.0,0.0), &
 (2.0,-5.0), (4.0,-2.0), (1.0,-4.0), (1.0,3.0), (1.0,4.0), &
 (3.0,-1.0), (2.0,2.0), (2.0,-3.0)/
!
! Compute all singular vectors
IPATH = 11
TOL = AMACH(4)
TOL = 10. * TOL
CALL LSVCR(A, IPATH, S, TOL = TOL, IRANK=IRANK, U=U, V=V)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, *) 'IRANK = ', IRANK
CALL WRCRN ('U', U, NRA, NCA)
CALL WRCRN ('S', S, 1, NCA, 1)
CALL WRCRN ('V', V)
!
END

```

## Output

```
IRANK = 3
```

|   | U                  |                    |                    |
|---|--------------------|--------------------|--------------------|
|   | 1                  | 2                  | 3                  |
| 1 | ( 0.1968, 0.2186)  | ( 0.5011, 0.0217)  | (-0.2007, -0.1003) |
| 2 | ( 0.3443, -0.3542) | (-0.2933, 0.0248)  | ( 0.1155, -0.2338) |
| 3 | ( 0.1457, 0.2307)  | (-0.5424, 0.1381)  | (-0.4361, -0.4407) |
| 4 | ( 0.3016, -0.0844) | ( 0.2157, 0.2659)  | (-0.0523, -0.0894) |
| 5 | ( 0.2283, -0.6008) | (-0.1325, 0.1433)  | ( 0.3152, -0.0090) |
| 6 | ( 0.2876, -0.0350) | ( 0.4377, -0.0400) | ( 0.0458, -0.6205) |

|  | S              |               |               |
|--|----------------|---------------|---------------|
|  | 1              | 2             | 3             |
|  | ( 11.77, 0.00) | ( 9.30, 0.00) | ( 4.99, 0.00) |

|   | V                  |                    |                    |
|---|--------------------|--------------------|--------------------|
|   | 1                  | 2                  | 3                  |
| 1 | ( 0.6616, 0.0000)  | (-0.2651, 0.0000)  | (-0.7014, 0.0000)  |
| 2 | ( 0.7355, 0.0379)  | ( 0.3850, -0.0707) | ( 0.5482, 0.0624)  |
| 3 | ( 0.0507, -0.1317) | ( 0.1724, 0.8642)  | (-0.0173, -0.4509) |

## LSGRR



Computes the generalized inverse of a real matrix.

### Required Arguments

*A* —  $NRA$  by  $NCA$  matrix whose generalized inverse is to be computed. (Input)

*GINVA* —  $NCA$  by  $NRA$  matrix containing the generalized inverse of *A*. (Output)

### Optional Arguments

*NRA* — Number of rows in the matrix *A*. (Input)

Default:  $NRA = \text{size}(A,1)$ .

*NCA* — Number of columns in the matrix *A*. (Input)

Default:  $NCA = \text{size}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{size}(A,1)$ .

*TOL* — Scalar containing the tolerance used to determine when a singular value (from the singular value decomposition of *A*) is negligible. (Input)

If *TOL* is positive, then a singular value  $\sigma_i$  considered negligible if  $\sigma_i \leq TOL$ . If *TOL* is

negative, then a singular value  $\sigma_i$  considered negligible if  $\sigma_i \leq |\text{TOL}| * \|A\|_\infty$ . In this case,  $|\text{TOL}|$  generally contains an estimate of the level of the relative error in the data. Default:  $\text{TOL} = 1.0\text{e-}5$  for single precision and  $1.0\text{d-}10$  for double precision.

**IRANK** — Scalar containing an estimate of the rank of  $A$ . (Output)

**LDGINV** — Leading dimension of  $\text{GINVA}$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $\text{LDGINV} = \text{size}(\text{GINV}, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL LSGRR (A, GINVA [ , ... ])`

Specific: The specific interface names are `S_LSGRR` and `D_LSGRR`.

## FORTRAN 77 Interface

Single: `CALL LSGRR (NRA, NCA, A, LDA, TOL, IRANK, GINVA, LDGINV)`

Double: The double precision name is `DLSGRR`.

## ScaLAPACK Interface

Generic: `CALL LSGRR (A0, GINVA0 [ , ... ])`

Specific: The specific interface names are `S_LSGRR` and `D_LSGRR`.

See the [ScaLAPACK Usage Notes](#) below for a description of the arguments for distributed computing.

## Description

Let  $k = \text{IRANK}$ , the rank of  $A$ ; let  $n = \text{NRA}$ , the number of rows in  $A$ ; let  $p = \text{NCA}$ , the number of columns in  $A$ ; and let

$$A^\dagger = \text{GINV}$$

be the generalized inverse of  $A$ .

To compute the *Moore-Penrose generalized inverse*, the routine `LSVRR` is first used to compute the singular value decomposition of  $A$ . A singular value decomposition of  $A$  consists of an  $n \times n$  orthogonal matrix  $U$ , a  $p \times p$  orthogonal matrix  $V$  and a diagonal matrix  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_m)$ ,  $m = \min(n, p)$ , such that  $U^T AV = [\Sigma, 0]$  if  $n \leq p$  and  $U^T AV = [\Sigma, 0]^T$  if  $n \geq p$ . Only the first  $p$  columns of  $U$  are computed. The rank  $k$  is estimated by counting the number of nonnegligible  $\sigma_i$ .

The matrices  $U$  and  $V$  can be partitioned as  $U = (U_1, U_2)$  and  $V = (V_1, V_2)$  where both  $U_1$  and  $V_1$  are  $k \times k$  matrices. Let  $\Sigma_1 = \text{diag}(\sigma_1, \dots, \sigma_k)$ . The Moore-Penrose generalized inverse of  $A$  is

$$A^\dagger = V_1 \Sigma_1^{-1} U_1^T$$

The underlying code of routine `LSGRR` is based on either `LINPACK`, `LAPACK`, or `ScaLAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2GRR/DL2GRR`. The reference is:

```
CALL L2GRR (NRA, NCA, A, LDA, TOL, IRANK, GINVA, LDGINV,
 WKA, WK)
```

The additional arguments are as follows:

**WKA** — Work vector of length `NRA * NCA` used as workspace for the matrix `A`. If `A` is not needed, `WKA` and `A` can share the same storage locations.

**WK** — Work vector of length `LWK` where `LWK` is equal to  
 $NRA^2 + NCA^2 + \min(NRA + 1, NCA) + NRA + NCA + \max(NRA, NCA) - 2$ .

2. Informational error

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                      |
|---|---|------------------------------------------------------------------------------------------------------|
| 4 | 1 | Convergence cannot be achieved for all the singular values and their corresponding singular vectors. |
|---|---|------------------------------------------------------------------------------------------------------|

## ScaLAPACK Usage Notes

The arguments which differ from the standard version of this routine are:

**A0** — `MXLDA` by `MXCOL` local matrix containing the local portions of the distributed matrix `A`. `A` contains the matrix for which the generalized inverse is to be computed. (Input)

**GINVA0** — `MXLDG` by `MXCOLG` local matrix containing the local portions of the distributed matrix `GINVA`. `GINVA` contains the generalized inverse of matrix `A`. (Output)

All other arguments are global and are the same as described for the standard version of the routine. In the argument descriptions above, `MXLDA`, `MXCOL`, `MXLDG`, and `MXCOLG` can be obtained through a call to `SCALAPACK_GETDIM` (see [Utilities](#)) after a call to `SCALAPACK_SETUP` (see Chapter 11, [Utilities](#)) has been made. See the [ScaLAPACK Example](#) below.

## Example

This example computes the generalized inverse of a  $3 \times 2$  matrix `A`. The rank  $k = \text{IRANK}$  and the inverse

$$A^\dagger = \text{GINV}$$

are printed.

```

USE IMSL_LIBRARIES
!
! Declare variables
PARAMETER (NRA=3, NCA=2, LDA=NRA, LDGINV=NCA)
REAL A(LDA,NCA), GINV(LDGINV,NRA)
!
! Set values for A
!
! A = (1 0)
! (1 1)
! (100 -50)
!
! DATA A/1., 1., 100., 0., 1., -50./
!
! Compute generalized inverse
TOL = AMACH(4)
TOL = 10.*TOL
CALL LSGRR (A, GINV, TOL=TOL, IRANK=IRANK)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, *) 'IRANK = ', IRANK
CALL WRRRN ('GINV', GINV)
!
END

```

## Output

```

IRANK = 2
 GINV
 1 2 3
1 0.1000 0.3000 0.0060
2 0.2000 0.6000 -0.0080

```

## ScaLAPACK Example

This example computes the generalized inverse of a  $6 \times 4$  matrix  $A$  as a distributed example. The rank  $k = \text{IRANK}$  and the inverse

$$A^\dagger = \text{GINV}$$

are printed.

```

USE MPI_SETUP_INT
USE IMSL_LIBRARIES
USE SCALAPACK_SUPPORT
IMPLICIT NONE
INCLUDE 'mpif.h'
!
! Declare variables
INTEGER IRANK, LDA, NCA, NRA, DESCA(9), DESCG(9), &
 LDGINV, MXLDG, MXCOLG, NOUT
INTEGER INFO, MXCOL, MXLDA
REAL TOL, AMACH

```

```

REAL, ALLOCATABLE :: A(:, :), GINVA(:, :)
REAL, ALLOCATABLE :: A0(:, :), GINVA0(:, :)
PARAMETER (NRA=6, NCA=4, LDA=NRA, LDGINV=NCA)
!
! Set up for MPI
MP_NPROCS = MP_SETUP()
IF(MP_RANK .EQ. 0) THEN
 ALLOCATE (A(LDA,NCA), GINVA(NCA,NRA))
! Set values for A
 A(1,:) = (/ 1.0, 2.0, 1.0, 4.0/)
 A(2,:) = (/ 3.0, 2.0, 1.0, 3.0/)
 A(3,:) = (/ 4.0, 3.0, 1.0, 4.0/)
 A(4,:) = (/ 2.0, 1.0, 3.0, 1.0/)
 A(5,:) = (/ 1.0, 5.0, 2.0, 2.0/)
 A(6,:) = (/ 1.0, 2.0, 2.0, 3.0/)
ENDIF
! Set up a 1D processor grid and define
! its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(NRA, NCA, .TRUE., .TRUE.)
! Get the array descriptor entities MXLDA,
! MXCOL, MXLDG, and MXCOLG
CALL SCALAPACK_GETDIM(NRA, NCA, MP_MB, MP_NB, MXLDA, MXCOL)
CALL SCALAPACK_GETDIM(NCA, NRA, MP_NB, MP_MB, MXLDG, MXCOLG)
! Set up the array descriptors
CALL DESCINIT(DESCA, NRA, NCA, MP_MB, MP_NB, 0, 0, MP_ICTXT, MXLDA, &
INFO)
CALL DESCINIT(DESCG, NCA, NRA, MP_NB, MP_MB, 0, 0, MP_ICTXT, MXLDG, &
INFO)
! Allocate space for the local arrays
ALLOCATE (A0(MXLDA,MXCOL), GINVA0(MXLDG,MXCOLG))
! Map input array to the processor grid
CALL SCALAPACK_MAP(A, DESCA, A0)
! Compute the generalized inverse
TOL = AMACH(4)
TOL = 10. * TOL
CALL LSGRR (A0, GINVA0, TOL=TOL, IRANK=IRANK)
! Unmap the results from the distributed
! array back to a non-distributed array.
! After the unmap, only Rank=0 has the full
! array.
CALL SCALAPACK_UNMAP(GINVA0, DESCG, GINVA)
! Print results.
! Only Rank=0 has the solution, GINVA
IF(MP_RANK .EQ. 0) THEN
 CALL UMACH (2, NOUT)
 WRITE (NOUT, *) `IRANK = `, IRANK
 CALL WRRRN (`GINVA`, GINVA)
ENDIF
! Exit ScaLAPACK usage
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP(`FINAL`)
END

```





# Chapter 2: Eigensystem Analysis

---

## Routines

|                                                                                      |                                                                                                |                                  |
|--------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|----------------------------------|
| <b>2.1. Eigenvalue Decomposition</b>                                                 |                                                                                                |                                  |
| 2.1.1                                                                                | Computes the eigenvalues of a self-adjoint matrix, $A$ .....                                   | <a href="#">LIN_EIG_SELF</a> 526 |
| 2.1.2                                                                                | Computes the eigenvalues of an $n \times n$ matrix, $A$ .....                                  | <a href="#">LIN_EIG_GEN</a> 533  |
| 2.1.3                                                                                | Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av = \lambda Bv$ ..... | <a href="#">LIN_GEIG_GEN</a> 542 |
| <b>2.2. Eigenvalues and (Optionally) Eigenvectors of <math>Ax = \lambda x</math></b> |                                                                                                |                                  |
| 2.2.1                                                                                | Real General Problem $Ax = \lambda x$                                                          |                                  |
|                                                                                      | All eigenvalues.....                                                                           | <a href="#">EVLRG</a> 549        |
|                                                                                      | All eigenvalues and eigenvectors .....                                                         | <a href="#">EVCRG</a> 552        |
|                                                                                      | Performance index.....                                                                         | <a href="#">EPIRG</a> 555        |
| 2.2.2                                                                                | Complex General Problem $Ax = \lambda x$                                                       |                                  |
|                                                                                      | All eigenvalues.....                                                                           | <a href="#">EVLCG</a> 557        |
|                                                                                      | All eigenvalues and eigenvectors .....                                                         | <a href="#">EVCCG</a> 559        |
|                                                                                      | Performance index.....                                                                         | <a href="#">EPICG</a> 562        |
| 2.2.3                                                                                | Real Symmetric Problem $Ax = \lambda x$                                                        |                                  |
|                                                                                      | All eigenvalues.....                                                                           | <a href="#">EVLSF</a> 564        |
|                                                                                      | All eigenvalues and eigenvectors .....                                                         | <a href="#">EVCSF</a> 566        |
|                                                                                      | Extreme eigenvalues .....                                                                      | <a href="#">EVASF</a> 568        |
|                                                                                      | Extreme eigenvalues and their eigenvectors.....                                                | <a href="#">EVESF</a> 570        |
|                                                                                      | Eigenvalues in an interval.....                                                                | <a href="#">EVBSF</a> 573        |
|                                                                                      | Eigenvalues in an interval and their eigenvectors .....                                        | <a href="#">EVFSF</a> 575        |
|                                                                                      | Performance index.....                                                                         | <a href="#">EPISF</a> 578        |
| 2.2.4                                                                                | Real Band Symmetric Matrices in Band Storage Mode                                              |                                  |
|                                                                                      | All eigenvalues.....                                                                           | <a href="#">EVLSB</a> 580        |
|                                                                                      | All eigenvalues and eigenvectors .....                                                         | <a href="#">EVCSB</a> 582        |
|                                                                                      | Extreme eigenvalues .....                                                                      | <a href="#">EVASB</a> 585        |
|                                                                                      | Extreme eigenvalues and their eigenvectors.....                                                | <a href="#">EVESB</a> 588        |
|                                                                                      | Eigenvalues in an interval.....                                                                | <a href="#">EVBSB</a> 591        |
|                                                                                      | Eigenvalues in an interval and their eigenvectors .....                                        | <a href="#">EVFSB</a> 593        |
|                                                                                      | Performance index.....                                                                         | <a href="#">EPISB</a> 596        |

|             |                                                                                  |                                     |     |
|-------------|----------------------------------------------------------------------------------|-------------------------------------|-----|
| 2.2.5       | Complex Hermitian Matrices                                                       |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">EVLHF</a>               | 598 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">EVCHF</a>               | 601 |
|             | Extreme eigenvalues.....                                                         | <a href="#">EVAHF</a>               | 604 |
|             | Extreme eigenvalues and their eigenvectors .....                                 | <a href="#">EVEHF</a>               | 606 |
|             | Eigenvalues in an interval .....                                                 | <a href="#">EVBHF</a>               | 609 |
|             | Eigenvalues in an interval and their eigenvectors.....                           | <a href="#">EVFHF</a>               | 612 |
|             | Performance index .....                                                          | <a href="#">EPIHF</a>               | 615 |
| 2.2.6       | Real Upper Hessenberg Matrices                                                   |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">EVLRH</a>               | 617 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">EVCRH</a>               | 619 |
| 2.2.7       | Complex Upper Hessenberg Matrices                                                |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">EVLCH</a>               | 621 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">EVCCH</a>               | 623 |
| <b>2.3.</b> | <b>Eigenvalues and (Optionally) Eigenvectors of <math>Ax = \lambda Bx</math></b> |                                     |     |
| 2.3.1       | Real General Problem $Ax = \lambda Bx$                                           |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">GVLRG</a>               | 626 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">GVCRG</a>               | 629 |
|             | Performance index .....                                                          | <a href="#">GPIRG</a>               | 632 |
| 2.3.2       | Complex General Problem $Ax = \lambda Bx$                                        |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">GVLCG</a>               | 634 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">GVCCG</a>               | 637 |
|             | Performance index .....                                                          | <a href="#">GPICG</a>               | 640 |
| 2.3.3       | Real Symmetric Problem $Ax = \lambda Bx$                                         |                                     |     |
|             | All eigenvalues .....                                                            | <a href="#">GVLSP</a>               | 642 |
|             | All eigenvalues and eigenvectors.....                                            | <a href="#">GVCSP</a>               | 645 |
|             | Performance index .....                                                          | <a href="#">GPISP</a>               | 648 |
| <b>2.4.</b> | <b>Eigenvalues and Eigenvectors Computed with ARPACK</b>                         |                                     |     |
|             | <b>Fortran 2003 Usage</b> .....                                                  |                                     | 651 |
|             | The Base Class .....                                                             | <a href="#">ARPACKBASE</a>          | 653 |
|             | Real Symmetric Problem $Ax = \lambda Bx$ .....                                   | <a href="#">ARPACK_SYMMETRIC</a>    | 654 |
|             | Real singular value decomposition $AV = US$ .....                                | <a href="#">ARPACK_SVD</a>          | 668 |
|             | Real General Problem $Ax = \lambda Bx$ .....                                     | <a href="#">ARPACK_NONSYMMETRIC</a> | 676 |
|             | Complex General Problem $Ax = \lambda Bx$ .....                                  | <a href="#">ARPACK_COMPLEX</a>      | 685 |

---

## Usage Notes

This chapter includes routines for linear eigensystem analysis. Many of these are for matrices with special properties. Some routines compute just a portion of the eigensystem. Use of the appropriate routine can substantially reduce computing time and storage requirements compared to computing a full eigensystem for a general complex matrix.

An ordinary linear eigensystem problem is represented by the equation  $Ax = \lambda x$  where  $A$  denotes an  $n \times n$  matrix. The value  $\lambda$  is an *eigenvalue* and  $x \neq 0$  is the corresponding *eigenvector*. The eigenvector is determined up to a scalar factor. In all routines, we have chosen this factor so that  $x$

has Euclidean length with value one, and the component of  $x$  of smallest index and largest magnitude is positive. In case  $x$  is a complex vector, this largest component is real and positive.

Similar comments hold for the use of the remaining Level 1 routines in the following tables in those cases where the second character of the Level 2 routine name is no longer the character "2".

A generalized linear eigensystem problem is represented by  $Ax = \lambda Bx$  where  $A$  and  $B$  are  $n \times n$  matrices. The value  $\lambda$  is an eigenvalue, and  $x$  is the corresponding eigenvector. The eigenvectors are normalized in the same manner as for the ordinary eigensystem problem. The linear eigensystem routines have names that begin with the letter "E". The generalized linear eigensystem routines have names that begin with the letter "G". This prefix is followed by a two-letter code for the type of analysis that is performed. That is followed by another two-letter suffix for the form of the coefficient matrix. The following tables summarize the names of the eigensystem routines.

| <b>Symmetric and Hermitian Eigensystems</b> |                        |                        |                        |
|---------------------------------------------|------------------------|------------------------|------------------------|
|                                             | <b>Symmetric Full</b>  | <b>Symmetric Band</b>  | <b>Hermitian Full</b>  |
| All eigenvalues                             | <a href="#">EVL SF</a> | <a href="#">EVL SB</a> | <a href="#">EVL HF</a> |
| All eigenvalues and eigenvectors            | <a href="#">EVC SF</a> | <a href="#">EVC SB</a> | <a href="#">EVC HF</a> |
| Extreme eigenvalues                         | <a href="#">EVA SF</a> | <a href="#">EVA SB</a> | <a href="#">EVA HF</a> |
| Extreme eigenvalues and eigenvectors        | <a href="#">EVE SF</a> | <a href="#">EVE SB</a> | <a href="#">EVE HF</a> |
| Eigenvalues in an interval                  | <a href="#">EVB SF</a> | <a href="#">EVB SB</a> | <a href="#">EVB HF</a> |
| Eigenvalues and eigenvectors in an interval | <a href="#">EVF SF</a> | <a href="#">EVF SB</a> | <a href="#">EVF HF</a> |
| Performance index                           | <a href="#">EPI SF</a> | <a href="#">EPI SB</a> | <a href="#">EPI HF</a> |

| <b>General Eigensystems</b>      |                        |                        |                        |                           |
|----------------------------------|------------------------|------------------------|------------------------|---------------------------|
|                                  | <b>Real General</b>    | <b>Complex General</b> | <b>Real Hessenberg</b> | <b>Complex Hessenberg</b> |
| All eigenvalues                  | <a href="#">EVL RG</a> | <a href="#">EVL CG</a> | <a href="#">EVL RH</a> | <a href="#">EVL CH</a>    |
| All eigenvalues and eigenvectors | <a href="#">EVC RG</a> | <a href="#">EVC CG</a> | <a href="#">EVC RH</a> | <a href="#">EVC CH</a>    |
| Performance index                | <a href="#">EPI RG</a> | <a href="#">EPI CG</a> | <a href="#">EPI RH</a> | <a href="#">EPI CH</a>    |

| Generalized Eigensystems $Ax = \lambda Bx$ |                 |                    |                                       |
|--------------------------------------------|-----------------|--------------------|---------------------------------------|
|                                            | Real<br>General | Complex<br>General | A Symmetric<br>B Positive<br>Definite |
| All eigenvalues                            | GVLRG           | GVLCG              | GVLSP                                 |
| All eigenvalues and<br>eigenvectors        | GVCRG           | GVCCG              | GVCSG                                 |
| Performance index                          | GPIRG           | GPICG              | GPISP                                 |

### Error Analysis and Accuracy

The remarks in this section are for the ordinary eigenvalue problem. Except in special cases, routines will not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem  $Ax = \lambda x$ . The computed pair

$$\tilde{x}, \tilde{\lambda}$$

is an exact eigenvector-eigenvalue pair for a “nearby” matrix  $A + E$ . Information about  $E$  is known only in terms of bounds of the form  $\|E\|_2 \leq f(n) \|A\|_2 \varepsilon$ . The value of  $f(n)$  depends on the algorithm but is typically a small fractional power of  $n$ . The parameter  $\varepsilon$  is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan [1989, page 342]),

$$\min |\tilde{\lambda} - \lambda| \leq \kappa(X) \|E\|_2 \quad \text{for all } \lambda \text{ in } \sigma(A)$$

where  $\sigma(A)$  is the set of all eigenvalues of  $A$  (called the *spectrum* of  $A$ ),  $X$  is the matrix of eigenvectors,  $\|\cdot\|_2$  is the 2-norm, and  $\kappa(X)$  is the condition number of  $X$  defined as  $\kappa(X) = \|X\|_2 \|X^{-1}\|_2$ . If  $A$  is a real symmetric or complex Hermitian matrix, then its eigenvector matrix  $X$  is respectively orthogonal or unitary. For these matrices,  $\kappa(X) = 1$ .

The eigenvalues

$$\tilde{\lambda}_j$$

and eigenvectors

$$\tilde{x}_j$$

computed by `EVC**` can be checked by computing their performance index  $\tau$  using `EPI**`. The performance index is defined by Smith et al. (1976, pages 124–126) to be

$$\tau = \max_{1 \leq j \leq n} \frac{\|A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j\|_1}{10n\varepsilon \|A\|_1 \|\tilde{x}_j\|_1}$$

No significance should be attached to the factor of 10 used in the denominator. For a real vector  $x$ , the symbol  $\|x\|_1$  represents the usual 1-norm of  $x$ . For a complex vector  $x$ , the symbol  $\|x\|_1$  is defined by

$$\|x\|_1 = \sum_{k=1}^N (|\Re x_k| + |\Im x_k|)$$

The performance index  $\tau$  is related to the error analysis because

$$\|E\tilde{x}_j\|_2 \doteq \|A\tilde{x}_j - \tilde{\lambda}_j\tilde{x}_j\|_2$$

where  $E$  is the “nearby” matrix discussed above.

While the exact value of  $\tau$  is machine and precision dependent, the performance of an eigensystem analysis routine is defined as excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . This is an arbitrary definition, but large values of  $\tau$  can serve as a warning that there is a blunder in the calculation. There are also similar routines `GPTE**` to compute the performance index for generalized eigenvalue problems.

If the condition number  $\kappa(X)$  of the eigenvector matrix  $X$  is large, there can be large errors in the eigenvalues even if  $\tau$  is small. In particular, it is often difficult to recognize near multiple eigenvalues or unstable mathematical problems from numerical results. This facet of the eigenvalue problem is difficult to understand: A user often asks for the accuracy of an individual eigenvalue. This can be answered approximately by computing the *condition number of an individual eigenvalue*. See Golub and Van Loan (1989, pages 344-345). For matrices  $A$  such that the computed array of normalized eigenvectors  $X$  is invertible, the condition number of  $\lambda_j$  is  $\kappa_j \equiv$  the Euclidean length of row  $j$  of the inverse matrix  $X^{-1}$ . Users can choose to compute this matrix with routine `LINCG`, see [Chapter 1, Linear Systems](#). An approximate bound for the accuracy of a computed eigenvalue is then given by  $\kappa_j \varepsilon \|A\|$ . To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by  $|\lambda_j|$ .

## Reformulating Generalized Eigenvalue Problems

The generalized eigenvalue problem  $Ax = \lambda Bx$  is often difficult for users to analyze because it is frequently ill-conditioned. There are occasionally changes of variables that can be performed on the given problem to ease this ill-conditioning. Suppose that  $B$  is singular but  $A$  is nonsingular.

Define the reciprocal  $\mu = \lambda^{-1}$ . Then, the roles of  $A$  and  $B$  are interchanged so that the reformulated problem

$Bx = \mu Ax$  is solved. Those generalized eigenvalues  $\mu_j = 0$  correspond to eigenvalues  $\lambda_j = \infty$ . The remaining

$$\lambda_j = \mu_j^{-1}$$

The generalized eigenvectors for  $\lambda_j$  correspond to those for  $\mu_j$ . Other reformulations can be made:

If  $B$  is nonsingular, the user can solve the ordinary eigenvalue problem  $Cx \equiv B^{-1}Ax = \lambda x$ . This is not recommended as a computational algorithm for two reasons. First, it is generally less efficient than solving the generalized problem directly. Second, the matrix  $C$  will be subject to

perturbations due to ill-conditioning and rounding errors when computing  $B^{-1}A$ . Computing the condition numbers of the eigenvalues for  $C$  may, however, be helpful for analyzing the accuracy of results for the generalized problem.

There is another method that users can consider to reduce the generalized problem to an alternate ordinary problem. This technique is based on first computing a matrix decomposition  $B = PQ$ , where both  $P$  and  $Q$  are matrices that are “simple” to invert. Then, the given generalized problem

is equivalent to the ordinary eigenvalue problem  $Fy = \lambda y$ . The matrix  $F \equiv P^{-1}AQ^{-1}$ . The unnormalized eigenvectors of the generalized problem are given by  $x = Q^{-1}y$ . An example of this reformulation is used in the case where  $A$  and  $B$  are real and symmetric with  $B$  positive definite. The IMSL routines [GVLSP](#) and [GVCSP](#) use  $P = R^T$  and  $Q = R$  where  $R$  is an upper triangular matrix obtained from a Cholesky decomposition,  $B = R^T R$ . The matrix  $F = R^{-T}AR^{-1}$  is symmetric and real. Computation of the eigenvalue-eigenvector expansion for  $F$  is based on routine [EVCSEF](#).

## Using ARPACK for Ordinary and Generalized Eigenvalue Problems

ARPACK consists of a set of Fortran 77 subroutines which use the Arnoldi method (Sorensen, 1992) to solve eigenvalue problems. ARPACK is well suited for large structured eigenvalue problems where structured means that a matrix-vector product  $w \leftarrow Av$  requires  $O(n)$  rather than the usual  $O(n^2)$  floating point operations.

The suite of features that we have implemented from ARPACK are described in the work of Lehoucq, Sorensen and Yang, *ARPACK Users' Guide*, SIAM Publications, (1998). Users will find access to this *Guide* helpful. Due to the size of the package, we provide for the use of double precision real and complex arithmetic only.

The ARPACK computational algorithm computes a partial set of approximate eigenvalues or singular values for various classes of problems. This includes the ordinary problem,  $Ax = \lambda x$ , the generalized problem,  $Ax = \lambda Bx$ , and the singular value decomposition,  $A = USV^T$ .

The original API for ARPACK is a Reverse Communication Interface. This interface can be used as illustrated in the *Guide*. However, we provide a Fortran 2003 interface to ARPACK that will be preferred by some users. This is a forward communication interface based on user-written functions for matrix-vector products or linear equation solving steps required by the algorithms in ARPACK. It is not necessary that the linear operators be expressed as dense or sparse matrices. That is permitted, but for some problems the best approach is the ability to form a product of the operator with a vector.

The forward communication interface includes an argument of a user-extended derived type or class object. The intent of producing this argument is that an extended type provides access to threaded user data or other required information, including procedure pointers, for use in the user-written product functions. It also hides information that can often be ignored with a first use.

---

## LIN\_EIG\_SELF

Computes the eigenvalues of a self-adjoint (i.e. real symmetric or complex Hermitian) matrix,  $A$ . Optionally, the eigenvectors can be computed. This gives the decomposition  $A = VDV^T$ , where  $V$  is an  $n \times n$  orthogonal matrix and  $D$  is a real diagonal matrix.

### Required Arguments

$A$  — Array of size  $n \times n$  containing the matrix. (Input [/Output])

$D$  — Array of size  $n$  containing the eigenvalues. The values are in order of decreasing absolute value. (Output)

## Optional Arguments

`NROWS = n` (Input)

Uses array `A(1:n, 1:n)` for the input matrix.

Default: `n = size(A, 1)`

`v = v(:, :)` (Output)

Array of the same type and kind as `A(1:n, 1:n)`. It contains the  $n \times n$  orthogonal matrix `V`.

`iopt = iopt(:)` (Input)

Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>LIN_EIG_SELF</code> |                                           |              |
|------------------------------------------------|-------------------------------------------|--------------|
| Option Prefix = ?                              | Option Name                               | Option Value |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_set_small</code>       | 1            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_overwrite_input</code> | 2            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_scan_for_NaN</code>    | 3            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_use_QR</code>          | 4            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_skip_Orth</code>       | 5            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_use_Gauss_elim</code>  | 6            |
| <code>s_,d_,c_,z_</code>                       | <code>Lin_eig_self_set_perf_ratio</code>  | 7            |

`iopt(IO) = ?_options(?_lin_eig_self_set_small, Small)`

If a denominator term is smaller in magnitude than the value *Small*, it is replaced by *Small*.

Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_eig_self_overwrite_input, ?_dummy)`

Do not save the input array `A(:, :)`.

`iopt(IO) = ?_options(?_lin_eig_self_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

```
isNaN(a(i,j)) == .true.
```

See the `isNaN()` function, [Chapter 10](#).

Default: The array is not scanned for NaNs.

`iopt(IO) = ?_options(?_lin_eig_use_QR, ?_dummy)`

Uses a rational *QR* algorithm to compute eigenvalues. Accumulate the eigenvectors using this algorithm.

Default: the eigenvectors computed using inverse iteration

`iopt(IO) = ?_options(?_lin_eig_skip_Orth, ?_dummy)`

If the eigenvalues are computed using inverse iteration, skips the final

orthogonalization of the vectors. This will result in a more efficient computation but the eigenvectors, while a complete set, may be far from orthogonal.

Default: the eigenvectors are normally orthogonalized if obtained using inverse iteration.

```
iopt(IO) = ?_options(?_lin_eig_use_Gauss_elim, ?_dummy)
```

If the eigenvalues are computed using inverse iteration, uses standard elimination with partial pivoting to solve the inverse iteration problems.

Default: the eigenvectors computed using cyclic reduction

```
iopt(IO) = ?_options(?_lin_eig_self_set_perf_ratio, perf_ratio)
```

Uses residuals for approximate normalized eigenvectors if they have a performance index no larger than *perf\_ratio*. Otherwise an alternate approach is taken and the eigenvectors are computed again: Standard elimination is used instead of cyclic reduction, or the standard *QR* algorithm is used as a backup procedure to inverse iteration. Larger values of *perf\_ratio* are less likely to cause these exceptions.

Default: *perf\_ratio* = 4

## FORTRAN 90 Interface

Generic:     CALL LIN\_EIG\_SELF (A, D [, ...])

Specific:    The specific interface names are S\_LIN\_EIG\_SELF, D\_LIN\_EIG\_SELF, C\_LIN\_EIG\_SELF, and Z\_LIN\_EIG\_SELF.

## Description

Routine `LIN_EIG_SELF` is an implementation of the *QR* algorithm for self-adjoint matrices. An orthogonal similarity reduction of the input matrix to self-adjoint tridiagonal form is performed. Then, the eigenvalue-eigenvector decomposition of a real tridiagonal matrix is calculated. The expansion of the matrix as  $AV = VD$  results from a product of these matrix factors. See Golub and Van Loan (1989, Chapter 8) for details.

## Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `LIN_EIG_SELF`. These error messages are numbered 81–90; 101–110; 121–129; 141–149.

## Example 1: Computing Eigenvalues

The eigenvalues of a self-adjoint matrix are computed. The matrix  $A = C + C^T$  is used, where  $C$  is random. The magnitudes of eigenvalues of  $A$  agree with the singular values of  $A$ . Also, see `operator_ex25`, supplied with the product examples.

```
use lin_eig_self_int
use lin_sol_svd_int
use rand_gen_int

implicit none
```

```

! This is Example 1 for LIN_EIG_SELF.

integer, parameter :: n=64
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) :: A(n,n), b(n,0), D(n), S(n), x(n,0), y(n*n)

! Generate a random matrix and from it
! a self-adjoint matrix.
call rand_gen(y)
A = reshape(y, (/n,n/))
A = A + transpose(A)

! Compute the eigenvalues of the matrix.
call lin_eig_self(A, D)

! For comparison, compute the singular values.
call lin_sol_svd(A, b, x, nrhs=0, s=S)

! Check the results: Magnitude of eigenvalues should equal
! the singular values.

if (sum(abs(abs(D) - S)) <= &
 sqrt(epsilon(one))*S(1)) then
 write (*,*) 'Example 1 for LIN_EIG_SELF is correct.'
end if
end

```

## Output

Example 1 for LIN\_EIG\_SELF is correct.

## Additional Examples

### Example 2: Eigenvalue-Eigenvector Expansion of a Square Matrix

A self-adjoint matrix is generated and the eigenvalues and eigenvectors are computed. Thus,  $A = VD V^T$ , where  $V$  is orthogonal and  $D$  is a real diagonal matrix. The matrix  $V$  is obtained using an optional argument. Also, see `operator_ex26`, [Chapter 10](#).

```

use lin_eig_self_int
use rand_gen_int

implicit none
! This is Example 2 for LIN_EIG_SELF.

integer, parameter :: n=8
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) :: a(n,n), d(n), v_s(n,n), y(n*n)

! Generate a random self-adjoint matrix.
call rand_gen(y)
a = reshape(y, (/n,n/))
a = a + transpose(a)
! Compute the eigenvalues and eigenvectors.
call lin_eig_self(a, d, v=v_s)
! Check the results for small residuals.

```

```

 if (sum(abs(matmul(a,v_s)-v_s*spread(d,1,n)))/d(1) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_EIG_SELF is correct.'
 end if
end
end

```

## Output

Example 2 for LIN\_EIG\_SELF is correct.

### Example 3: Computing a few Eigenvectors with Inverse Iteration

A self-adjoint  $n \times n$  matrix is generated and the eigenvalues,  $\{d_i\}$ , are computed. The eigenvectors associated with the first  $k$  of these are computed using the self-adjoint solver, `lin_sol_self`, and inverse iteration. With random right-hand sides, these systems are as follows:

$$(A - d_i I)v_i = b_i$$

The solutions are then orthogonalized as in Hanson et al. (1991) to comprise a partial decomposition  $AV = VD$  where  $V$  is an  $n \times k$  matrix resulting from the orthogonalized  $\{v_i\}$  and  $D$  is the  $k \times k$  diagonal matrix of the distinguished eigenvalues. It is necessary to suppress the error message when the matrix is singular. Since these singularities are desirable, it is appropriate to ignore the exceptions and not print the message text. Also, see `operator_ex27`, supplied with the product examples.

```

 use lin_eig_self_int
 use lin_sol_self_int
 use rand_gen_int
 use error_option_packet

 implicit none

! This is Example 3 for LIN_EIG_SELF.

 integer i, j
 integer, parameter :: n=64, k=8
 real(kind(1d0)), parameter :: one=1d0, zero=0d0
 real(kind(1d0)) big, err
 real(kind(1d0)) :: a(n,n), b(n,1), d(n), res(n,k), temp(n,n), &
 v(n,k), y(n*n)
 type(d_options) :: iopti(2)=d_options(0,zero)

! Generate a random self-adjoint matrix.
 call rand_gen(y)
 a = reshape(y, (/n,n/))
 a = a + transpose(a)

! Compute just the eigenvalues.
 call lin_eig_self(a, d)

 do i=1, k

```

```

! Define a temporary array to hold the matrices A - eigenvalue*I.
 temp = a
 do j=1, n
 temp(j,j) = temp(j,j) - d(i)
 end do

! Use packaged option to reset the value of a small diagonal.
 iopti(1) = d_options(d_lin_sol_self_set_small,&
 epsilon(one)*abs(d(i)))

! Use packaged option to skip singularity messages.
 iopti(2) = d_options(d_lin_sol_self_no_sing_mess,&
 zero)
 call rand_gen(b(1:n,1))
 call lin_sol_self(temp, b, v(1:,i:i),&
 iopt=iopti)
end do

! Orthogonalize the eigenvectors.
do i=1, k
 big = maxval(abs(v(1:,i)))
 v(1:,i) = v(1:,i)/big
 v(1:,i) = v(1:,i)/sqrt(sum(v(1:,i)**2))
 if (i == k) cycle
 v(1:,i+1:k) = v(1:,i+1:k) + &
 spread(-matmul(v(1:,i),v(1:,i+1:k)),1,n)* &
 spread(v(1:,i),2,k-i)
end do
do i=k-1, 1, -1
 v(1:,i+1:k) = v(1:,i+1:k) + &
 spread(-matmul(v(1:,i),v(1:,i+1:k)),1,n)* &
 spread(v(1:,i),2,k-i)
end do

! Check the results for both orthogonality of vectors and small
! residuals.
res(1:k,1:k) = matmul(transpose(v),v)
do i=1,k
 res(i,i)=res(i,i)-one
end do
err = sum(abs(res))/k**2
res = matmul(a,v) - v*spread(d(1:k),1,n)
if (err <= sqrt(epsilon(one))) then
 if (sum(abs(res))/abs(d(1)) <= sqrt(epsilon(one))) then
 write (*,*) 'Example 3 for LIN_EIG_SELF is correct.'
 end if
end if
end
end

```

## Output

Example 3 for LIN\_EIG\_SELF is correct.

#### Example 4: Analysis and Reduction of a Generalized Eigensystem

A generalized eigenvalue problem is  $Ax = \lambda Bx$ , where  $A$  and  $B$  are  $n \times n$  self-adjoint matrices. The matrix  $B$  is positive definite. This problem is reduced to an ordinary self-adjoint eigenvalue problem  $Cy = \lambda y$  by changing the variables of the generalized problem to an equivalent form. The eigenvalue-eigenvector decomposition  $B = VSV^T$  is first computed, labeling an eigenvalue *too small* if it is less than `epsilon(1.d0)`. The ordinary self-adjoint eigenvalue problem is  $Cy = \lambda y$  provided that the rank of  $B$ , based on this definition of *Small*, has the value  $n$ . In that case,

$$C = DV^TAVD$$

where

$$D = S^{-1/2}$$

The relationship between  $x$  and  $y$  is summarized as  $X = VDY$ , computed after the ordinary eigenvalue problem is solved for the eigenvectors  $Y$  of  $C$ . The matrix  $X$  is normalized so that each column has Euclidean length of value one. This solution method is nonstandard for any but the most ill-conditioned matrices  $B$ . The standard approach is to compute an ordinary self-adjoint problem following computation of the Cholesky decomposition

$$B = R^T R$$

where  $R$  is upper triangular. The computation of  $C$  can also be completed efficiently by exploiting its self-adjoint property. See Golub and Van Loan (1989, Chapter 8) for more information. Also, see `operator_ex28`, [Chapter 10](#).

```
use lin_eig_self_int
use rand_gen_int
implicit none

! This is Example 4 for LIN_EIG_SELF.

integer i
integer, parameter :: n=64
real(kind(1e0)), parameter :: one=1d0
real(kind(1e0)) b_sum
real(kind(1e0)), dimension(n,n) :: A, B, C, D(n), lambda(n), &
 S(n), vb_d, X, ytemp(n*n), res

! Generate random self-adjoint matrices.
call rand_gen(ytemp)
A = reshape(ytemp, (/n,n/))
A = A + transpose(A)

call rand_gen(ytemp)
B = reshape(ytemp, (/n,n/))
B = B + transpose(B)

b_sum = sqrt(sum(abs(B**2))/n)

! Add a scalar matrix so B is positive definite.
```

```

do i=1, n
 B(i,i) = B(i,i) + b_sum
end do

! Get the eigenvalues and eigenvectors for B.

call lin_eig_self(B, S, v=vb_d)

! For full rank problems, convert to an ordinary self-adjoint
! problem. (All of these examples are full rank.)
if (S(n) > epsilon(one)) then

 D = one/sqrt(S)

 C = spread(D,2,n)*matmul(transpose(vb_d), &
 matmul(A,vb_d))*spread(D,1,n)

! Get the eigenvalues and eigenvectors for C.
call lin_eig_self(C, lambda, v=X)

! Compute the generalized eigenvectors.
X = matmul(vb_d,spread(D,2,n)*X)

! Normalize the eigenvectors for the generalized problem.
X = X * spread(one/sqrt(sum(X**2,dim=2)),1,n)

res = matmul(A,X) - &
 matmul(B,X)*spread(lambda,1,n)

! Check the results.
if (sum(abs(res))/(sum(abs(A))+sum(abs(B))) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_EIG_SELF is correct.'
end if
end if
end

```

## Output

Example 4 for LIN\_EIG\_SELF is correct.

---

## LIN\_EIG\_GEN



Computes the eigenvalues of an  $n \times n$  matrix,  $A$ . Optionally, the eigenvectors of  $A$  or  $A^T$  are computed. Using the eigenvectors of  $A$  gives the decomposition  $AV = VE$ , where  $V$  is an  $n \times n$  complex matrix of eigenvectors, and  $E$  is the complex diagonal matrix of eigenvalues. Other options include the reduction of  $A$  to upper triangular or Schur form, reduction to block upper

triangular form with  $2 \times 2$  or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

### Required Arguments

$A$  — Array of size  $n \times n$  containing the matrix. (Input [/Output])

$E$  — Array of size  $n$  containing the eigenvalues. These complex values are in order of decreasing absolute value. The signs of imaginary parts of the eigenvalues are in no predictable order. (Output)

### Optional Arguments

`NROWS = n` (Input)  
 Uses array `A(1:n, 1:n)` for the input matrix.  
 Default: `n = SIZE(A, 1)`

`v = V(:, :)` (Output)  
 Returns the complex array of eigenvectors for the matrix  $A$ .

`v_adj = U(:, :)` (Output)  
 Returns the complex array of eigenvectors for the matrix  $A^T$ . Thus the residuals

$$S = A^T U - U \bar{E}$$

are small.

`tri = T(:, :)` (Output)  
 Returns the complex upper-triangular matrix  $T$  associated with the reduction of the matrix  $A$  to Schur form. Optionally a unitary matrix  $W$  is returned in array `V(:, :)` such that the residuals  $Z = AW - WT$  are small.

`iopt = iopt(:)` (Input)  
 Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

| Packaged Options for LIN_EIG_GEN |                                          |              |
|----------------------------------|------------------------------------------|--------------|
| Option Prefix = ?                | Option Name                              | Option Value |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_set_small</code>       | 1            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_overwrite_input</code> | 2            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_scan_for_NaN</code>    | 3            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_no_balance</code>      | 4            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_set_iterations</code>  | 5            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_in_Hess_form</code>    | 6            |
| <code>s_, d_, c_, z_</code>      | <code>lin_eig_gen_out_Hess_form</code>   | 7            |

| Packaged Options for LIN_EIG_GEN |                             |    |
|----------------------------------|-----------------------------|----|
| s_d_c_z_                         | lin_eig_gen_out_block_form  | 8  |
| s_d_c_z_                         | lin_eig_gen_out_tri_form    | 9  |
| s_d_c_z_                         | lin_eig_gen_continue_with_v | 10 |
| s_d_c_z_                         | lin_eig_gen_no_sorting      | 11 |

`iopt(IO) = ?_options(?_lin_eig_gen_set_small, Small)`

This is the tolerance used to declare off-diagonal values effectively zero compared with the size of the numbers involved in the computation of a shift.

Default: *Small* = `epsilon()`, the relative accuracy of arithmetic

`iopt(IO) = ?_options(?_lin_eig_gen_overwrite_input, ?_dummy)`

Does not save the input array `A(:, :)`.

Default: The array is saved.

`iopt(IO) = ?_options(?_lin_eig_gen_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(a(i,j)) == .true..`

See the `isNaN()` function, [Chapter 10](#).

Default: The array is not scanned for NaNs.

`iopt(IO) = ?_options(?_lin_eig_no_balance, ?_dummy)`

The input matrix is not preprocessed searching for isolated eigenvalues followed by rescaling. See Golub and Van Loan (1989, Chapter 7) for references. With some optional uses of the routine, this option flag is required.

Default: The matrix is first balanced.

`iopt(IO) = ?_options(?_lin_eig_gen_set_iterations, ?_dummy)`

Resets the maximum number of iterations permitted to isolate each diagonal block matrix.

Default: The maximum number of iterations is 52.

`iopt(IO) = ?_options(?_lin_eig_gen_in_Hess_form, ?_dummy)`

The input matrix is in upper Hessenberg form. This flag is used to avoid the initial reduction phase which may not be needed for some problem classes.

Default: The matrix is first reduced to Hessenberg form.

`iopt(IO) = ?_options(?_lin_eig_gen_out_Hess_form, ?_dummy)`

The output matrix is transformed to upper Hessenberg form,  $H_1$ . If the optional argument "`v=V(:, :)`" is passed by the calling program unit, then the array `V(:, :)` contains an orthogonal matrix  $Q_1$  such that

$$AQ_1 - Q_1H_1 \cong 0$$

Requires the simultaneous use of option `?_lin_eig_no_balance`.

Default: The matrix is reduced to diagonal form.

`iopt(IO) = ?_options(?_lin_eig_gen_out_block_form, ?_dummy)`

The output matrix is transformed to upper Hessenberg form,  $H_2$ , which is block upper triangular. The dimensions of the blocks are either  $2 \times 2$  or unit sized. Nonzero subdiagonal values of  $H_2$  determine the size of the blocks. If the optional argument "`v=v(:, :)`" is passed by the calling program unit, then the array `v(:, :)` contains an orthogonal matrix  $Q_2$  such that

$$AQ_2 - Q_2H_2 \cong 0$$

Requires the simultaneous use of option `?_lin_eig_no_balance`.  
Default: The matrix is reduced to diagonal form.

`iopt(IO) = ?_options(?_lin_eig_gen_out_tri_form, ?_dummy)`

The output matrix is transformed to upper-triangular form,  $T$ . If the optional argument "`v=v(:, :)`" is passed by the calling program unit, then the array `v(:, :)` contains a unitary matrix  $W$  such that

$AW - WT \cong 0$ . The upper triangular matrix  $T$  is returned in the optional argument "`tri=T(:, :)`". The eigenvalues of  $A$  are the diagonal entries of the matrix  $T$ . They are in no particular order. The output array `E(:)` is blocked with NaNs using this option. This option requires the simultaneous use of option `?_lin_eig_no_balance`.  
Default: The matrix is reduced to diagonal form.

`iopt(IO) = ?_options(?_lin_eig_gen_continue_with_V, ?_dummy)`

As a convenience or for maintaining efficiency, the calling program unit sets the optional argument "`v=v(:, :)`" to a matrix that has transformed a problem to the similar matrix,  $\dot{A}$ . The contents of `v(:, :)` are updated by the transformations used in the algorithm. Requires the simultaneous use of option `?_lin_eig_no_balance`.  
Default: The array `v(:, :)` is initialized to the identity matrix.

`iopt(IO) = ?_options(?_lin_eig_gen_no_sorting, ?_dummy)`

Does not sort the eigenvalues as they are isolated by solving the  $2 \times 2$  or unit sized blocks. This will have the effect of guaranteeing that complex conjugate pairs of eigenvalues are adjacent in the array `E(:)`.  
Default: The entries of `E(:)` are sorted so they are non-increasing in absolute value.

## FORTRAN 90 Interface

Generic:     `CALL LIN_EIG_GEN (A, E [, ...])`

Specific:    The specific interface names are `S_LIN_EIG_GEN`, `D_LIN_EIG_GEN`,  
              `C_LIN_EIG_GEN`, and `Z_LIN_EIG_GEN`.

## Description

The input matrix  $A$  is first balanced. The resulting similar matrix is transformed to upper Hessenberg form using orthogonal transformations. The double-shifted  $QR$  algorithm transforms the Hes-

senberg matrix so that  $2 \times 2$  or unit sized blocks remain along the main diagonal. Any off-diagonal that is classified as “small” in order to achieve this block form is set to the value zero. Next the block upper triangular matrix is transformed to upper triangular form with unitary rotations. The eigenvectors of the upper triangular matrix are computed using back substitution. Care is taken to avoid overflows during this process. At the end, eigenvectors are normalized to have Euclidean length one, with the largest component real and positive. This algorithm follows that given in Golub and Van Loan, (1989, Chapter 7), with some novel organizational details for additional options, efficiency and robustness.

## Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `LIN_EIG_GEN`. These error messages are numbered 841–858; 861–878; 881–898; 901–918.

### Example 1: Computing Eigenvalues

The eigenvalues of a random real matrix are computed. These values define a complex diagonal matrix  $E$ . Their correctness is checked by obtaining the eigenvector matrix  $V$  and verifying that the residuals  $R = AV - VE$  are small. Also, see `operator_ex29`, supplied with the product examples.

```

use lin_eig_gen_int
use rand_gen_int

implicit none

! This is Example 1 for LIN_EIG_GEN.

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)) A(n,n), y(n*n), err
complex(kind(1d0)) E(n), V(n,n), E_T(n)
type(d_error) :: d_epack(16) = d_error(0,0d0)

! Generate a random matrix.
call rand_gen(y)
A = reshape(y, (/n,n/))

! Compute only the eigenvalues.
call lin_eig_gen(A, E)

! Compute the decomposition, A*V = V*values,
! obtaining eigenvectors.
call lin_eig_gen(A, E_T, v=V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
err = sum(abs(matmul(A,V) - V*spread(E,DIM=1,NCOPIES=n))) &
 / sum(abs(E))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_EIG_GEN is correct.'
end if

end

```

## Output

Example 1 for LIN\_EIG\_GEN is correct.

## Additional Examples

### Example 2: Complex Polynomial Equation Roots

The roots of a complex polynomial equation,

$$f(z) \equiv \sum_{k=1}^n b_k z^{n-k} + z^n = 0$$

are required. This algebraic equation is formulated as a matrix eigenvalue problem. The equivalent matrix eigenvalue problem is solved using the upper Hessenberg matrix which has the value zero except in row number 1 and along the first subdiagonal. The entries in the first row are given by  $a_{1j} = -b_j$ ,  $j = 1, \dots, n$ , while those on the first subdiagonal have the value one. This is a *companion matrix* for the polynomial. The results are checked by testing for small values of  $|f(e_i)|$ ,  $i = 1, \dots, n$ , at the eigenvalues of the matrix, which are the roots of  $f(z)$ . Also, see `operator_ex30`, supplied with the product examples.

```
use lin_eig_gen_int
use rand_gen_int

implicit none
! This is Example 2 for LIN_EIG_GEN.

integer i
integer, parameter :: n=12
real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
real(kind(1d0)) err, t(2*n)
type(d_options) :: iopti(1)=d_options(0,zero)
complex(kind(1d0)) a(n,n), b(n), e(n), f(n), fg(n)

call rand_gen(t)
b = cmplx(t(1:n),t(n+1:),kind(one))

! Define the companion matrix with polynomial coefficients
! in the first row.

a = zero

do i=2, n
 a(i,i-1) = one
end do

a(1,1:n) = -b

! Note that the input companion matrix is upper Hessenberg.
iopti(1) = d_options(z_lin_eig_gen_in_Hess_form,zero)

! Compute complex eigenvalues of the companion matrix.
```

```

call lin_eig_gen(a, e, iopt=iopti)

f=one; fg=one

! Use Horner's method for evaluation of the complex polynomial
! and size gauge at all roots.

do i=1, n
 f = f*e + b(i)
 fg = fg*abs(e) + abs(b(i))
end do

! Check for small errors at all roots.

err = sum(abs(f/fg))/n
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_EIG_GEN is correct.'
end if
end

```

## Output

Example 2 for LIN\_EIG\_GEN is correct.

### Example 3: Solving Parametric Linear Systems with a Scalar Change

The efficient solution of a family of linear algebraic equations is required. These systems are  $(A + hI)x = b$ . Here  $A$  is an  $n \times n$  real matrix,  $I$  is the identity matrix, and  $b$  is the right-hand side matrix. The scalar  $h$  is such that the coefficient matrix is nonsingular. The method is based on the Schur form for matrix  $A$ :  $AW = WT$ , where  $W$  is unitary and  $T$  is upper triangular. This provides an efficient solution method for several values of  $h$ , once the Schur form is computed. The solution steps solve, for  $y$ , the upper triangular linear system

$$(T + hI)y = \bar{W}^T b$$

Then,  $x = x(h) = Wy$ . This is an efficient and accurate method for such parametric systems provided the expense of computing the Schur form has a pay-off in later efficiency. Using the Schur form in this way, it is not required to compute an  $LU$  factorization of  $A + hI$  with each new value of  $h$ . Note that even if the data  $A$ ,  $h$ , and  $b$  are real, subexpressions for the solution may involve complex intermediate values, with  $x(h)$  finally a real quantity. Also, see `operator_ex31`, supplied with the product examples.

```

use lin_eig_gen_int
use lin_sol_gen_int
use rand_gen_int

implicit none

! This is Example 3 for LIN_EIG_GEN.

integer i
integer, parameter :: n=32, k=2
real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
real(kind(1e0)) a(n,n), b(n,k), x(n,k), temp(n*max(n,k)), h, err
type(s_options) :: iopti(2)

```

```

complex(kind(1e0)) w(n,n), t(n,n), e(n), z(n,k)

call rand_gen(temp)
a = reshape(temp, (/n,n/))

call rand_gen(temp)
b = reshape(temp, (/n,k/))

iopti(1) = s_options(s_lin_eig_gen_out_tri_form,zero)
iopti(2) = s_options(s_lin_eig_gen_no_balance,zero)

! Compute the Schur decomposition of the matrix.

call lin_eig_gen(a, e, v=w, tri=t, &
 iopt=iopti)

! Choose a value so that A+h*I is non-singular.
h = one

! Solve for (A+h*I)x=b using the Schur decomposition.

z = matmul(conjg(transpose(w)),b)

! Solve intermediate upper-triangular system with implicit
! additive diagonal, h*I. This is the only dependence on
! h in the solution process.
do i=n,1,-1
 z(i,1:k) = z(i,1:k)/(t(i,i)+h)
 z(1:i-1,1:k) = z(1:i-1,1:k) + &
 spread(-t(1:i-1,i),dim=2,ncopies=k)* &
 spread(z(i,1:k),dim=1,ncopies=i-1)
end do

! Compute the solution. It should be the same as x, but will not be
! exact due to rounding errors. (The quantity real(z,kind(one)) is
! the real-valued answer when the Schur decomposition method is used.)

z = matmul(w,z)

! Compute the solution by solving for x directly.
do i=1, n
 a(i,i) = a(i,i) + h
end do

call lin_sol_gen(a, b, x)

! Check that x and z agree approximately.
err = sum(abs(x-z))/sum(abs(x))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 3 for LIN_EIG_GEN is correct.'
end if

end

```

## Output

Example 3 for LIN\_EIG\_GEN is correct.

### Example 4: Accuracy Estimates of Eigenvalues Using Adjoint and Ordinary Eigenvectors

A matrix  $A$  has entries that are subject to uncertainty. This is expressed as the realization that  $A$  can be replaced by the matrix  $A + \eta B$ , where the value  $\eta$  is “small” but still significantly larger than machine precision. The matrix  $B$  satisfies  $\|B\| \leq \|A\|$ . A variation in eigenvalues is estimated using analysis found in Golub and Van Loan, (1989, Chapter 7, p. 344). Each eigenvalue and eigenvector is expanded in a power series in  $\eta$ . With

$$e_i(\eta) \approx e_i + \eta \dot{e}_i \eta$$

and normalized eigenvectors, the bound

$$|\dot{e}_i| \leq \frac{\|A\|}{|u_i^* v_i|}$$

is satisfied. The vectors  $u_i$  and  $v_i$  are the ordinary and adjoint eigenvectors associated respectively with  $e_i$  and its complex conjugate. This gives an upper bound on the size of the change to each  $|e_i|$  due to changing the matrix data. The reciprocal

$$|u_i^* v_i|^{-1}$$

is defined as the *condition number* of  $e_i$ . Also, see `operator_ex32`, [Chapter 10](#).

```
use lin_eig_gen_int
use rand_gen_int

implicit none

! This is Example 4 for LIN_EIG_GEN.

integer i
integer, parameter :: n=17
real(kind(ld0)), parameter :: one=1d0
real(kind(ld0)) a(n,n), c(n,n), variation(n), y(n*n), temp(n), &
 norm_of_a, eta
complex(kind(ld0)), dimension(n,n) :: e(n), d(n), u, v

! Generate a random matrix.
call rand_gen(y)
a = reshape(y, (/n,n/))

! Compute the eigenvalues, left- and right- eigenvectors.
call lin_eig_gen(a, e, v=v, v_adj=u)

! Compute condition numbers and variations of eigenvalues.
```

```

norm_of_a = sqrt(sum(a**2)/n)
do i=1, n
 variation(i) = norm_of_a/abs(dot_product(u(1:n,i), &
 v(1:n,i)))
end do

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again. Check the
! differences compared to the estimates. They should not exceed
! the bounds.

eta = sqrt(epsilon(one))
do i=1, n
 call rand_gen(temp)
 c(1:n,i) = a(1:n,i) + (2*temp - 1)*eta*a(1:n,i)
end do

call lin_eig_gen(c,d)

! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
if (count(abs(d)-abs(e) > eta*variation) == 0) then
 write (*,*) 'Example 4 for LIN_EIG_GEN is correct.'
end if

end

```

## Output

Example 4 for LIN\_EIG\_GEN is correct.

---

# LIN\_GEIG\_GEN



Computes the generalized eigenvalues of an  $n \times n$  matrix pencil,  $Av = \lambda Bv$ . Optionally, the generalized eigenvectors are computed. If either of  $A$  or  $B$  is nonsingular, there are diagonal matrices  $\alpha$  and  $\beta$ , and a complex matrix  $V$ , all computed such that  $AV\beta = BV\alpha$ .

## Required Arguments

**A** — Array of size  $n \times n$  containing the matrix  $A$ . (Input [/Output])

**B** — Array of size  $n \times n$  containing the matrix  $B$ . (Input [/Output])

**ALPHA** — Array of size  $n$  containing diagonal matrix factors of the generalized eigenvalues. These complex values are in order of decreasing absolute value. (Output)

**BETA $V$**  — Array of size  $n$  containing diagonal matrix factors of the generalized eigenvalues. These real values are in order of decreasing value. (Output)

## Optional Arguments

`NROWS = n` (Input)

Uses arrays `A(1:n, 1:n)` and `B(1:n, 1:n)` for the input matrix pencil.

Default: `n = SIZE(A, 1)`

`v = V(:, :)` (Output)

Returns the complex array of generalized eigenvectors for the matrix pencil.

`iopt = iopt(:)` (Input)

Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>LIN_GEIG_GEN</code> |                                            |              |
|------------------------------------------------|--------------------------------------------|--------------|
| Option Prefix = ?                              | Option Name                                | Option Value |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_set_small</code>        | 1            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_overwrite_input</code>  | 2            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_scan_for_NaN</code>     | 3            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_self_adj_pos</code>     | 4            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_for_lin_sol_self</code> | 5            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_for_lin_eig_self</code> | 6            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_for_lin_sol_lsq</code>  | 7            |
| <code>s_, d_, c_, z_</code>                    | <code>lin_geig_gen_for_lin_eig_gen</code>  | 8            |

`iopt(IO) = ?_options(?_lin_geig_gen_set_small, Small)`

This tolerance, multiplied by the sum of absolute value of the matrix  $B$ , is used to define a small diagonal term in the routines `lin_sol_lsq` and `lin_sol_self`. That value can be replaced using the option flags `lin_geig_gen_for_lin_sol_lsq`, and `lin_geig_gen_for_lin_sol_self`.

Default: *Small* = `epsilon(.)`, the relative accuracy of arithmetic

`iopt(IO) = ?_options(?_lin_geig_gen_overwrite_input, ?_dummy)`

Does not save the input arrays `A(:, :)` and `B(:, :)`.

Default: The array is saved.

`iopt(IO) = ?_options(?_lin_geig_gen_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

`isNaN(a(i,j)) .or. isNaN(b(i,j)) == .true.`

See the `isNaN()` function, [Chapter 10](#).

Default: The arrays are not scanned for NaNs.

```

iopt(IO) = ?_options(?_lin_geig_gen_self_adj_pos, ?_dummy)
 If both matrices A and B are self-adjoint and additionally B is positive-definite, then the
 Cholesky algorithm is used to reduce the matrix pencil to an ordinary self-adjoint
 eigenvalue problem.

iopt(IO) = ?_options(?_lin_geig_gen_for_lin_sol_self, ?_dummy)

iopt(IO+1) = ?_options((k=size of options for lin_sol_self), ?_dummy)
 The options for lin_sol_self follow as data in iopt().

iopt(IO) = ?_options(?_lin_geig_gen_for_lin_eig_self, ?_dummy)

iopt(IO+1) = ?_options((k=size of options for lin_eig_self), ?_dummy)
 The options for lin_eig_self follow as data in iopt().

iopt(IO) = ?_options(?_lin_geig_gen_for_lin_sol_lsq, ?_dummy)

iopt(IO+1) = ?_options((k=size of options for lin_sol_lsq), ?_dummy)
 The options for lin_sol_lsq follow as data in iopt().

iopt(IO) = ?_options(?_lin_geig_gen_for_lin_eig_gen, ?_dummy)

iopt(IO+1) = ?_options((k=size of options for lin_eig_gen), ?_dummy)
 The options for lin_eig_gen follow as data in iopt().

```

## FORTRAN 90 Interface

Generic:    CALL LIN\_GEIG\_GEN (A, B, ALPHA, BETAV [, ...])

Specific:   The specific interface names are S\_LIN\_GEIG\_GEN, D\_LIN\_GEIG\_GEN,  
C\_LIN\_GEIG\_GEN, and Z\_LIN\_GEIG\_GEN.

## Description

Routine `LIN_GEIG_GEN` implements a standard algorithm that reduces a generalized eigenvalue or matrix pencil problem to an ordinary eigenvalue problem. An orthogonal decomposition is computed

$$BP^T = HR$$

The orthogonal matrix  $H$  is the product of  $n - 1$  row permutations, each followed by a Householder transformation. Column permutations,  $P$ , are chosen at each step to maximize the Euclidian length of the pivot column. The matrix  $R$  is upper triangular. Using the default tolerance  $\tau = \varepsilon \|B\|$ , where  $\varepsilon$  is machine relative precision, each diagonal entry of  $R$  exceeds  $\tau$  in value. Otherwise,  $R$  is singular. In that case  $A$  and  $B$  are interchanged and the orthogonal decomposition is computed one more time. If both matrices are singular the problem is declared *singular* and is not solved. The interchange of  $A$  and  $B$  is accounted for in the output diagonal matrices  $\alpha$  and  $\beta$ . The ordinary eigenvalue problem is  $Cx = \lambda x$ , where

$$C = H^T A P^T R^{-1}$$

and

$$RPv = x$$

If the matrices  $A$  and  $B$  are self-adjoint and if, in addition,  $B$  is positive-definite, then a more efficient reduction than the default algorithm can be optionally used to solve the problem: A Cholesky decomposition is obtained,  $R^T R R = P B P^T$ . The matrix  $R$  is upper triangular and  $P$  is a permutation matrix. This is equivalent to the ordinary self-adjoint eigenvalue problem  $Cx = \lambda x$ , where  $RPv = x$  and

$$C = R^{-T} P A P^T R^{-1}$$

The self-adjoint eigenvalue problem is then solved.

### Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `LIN_GEIG_GEN`. These error messages are numbered 921–936; 941–956; 961–976; 981–996.

### Example 1: Computing Generalized Eigenvalues

The generalized eigenvalues of a random real matrix pencil are computed. These values are checked by obtaining the generalized eigenvectors and then showing that the residuals

$$AV - BV\alpha\beta^{-1}$$

are *small*. Note that when the matrix  $B$  is nonsingular  $\beta = I$ , the identity matrix. When  $B$  is singular and  $A$  is nonsingular, some diagonal entries of  $\beta$  are essentially zero. This corresponds to “infinite eigenvalues” of the matrix pencil. This random matrix pencil example has all finite eigenvalues. Also, see `operator_ex33`, [Chapter 10](#).

```
use lin_geig_gen_int
use rand_gen_int

implicit none

! This is Example 1 for LIN_GEIG_GEN.

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)) A(n,n), B(n,n), betav(n), beta_t(n), err, y(n*n)
complex(kind(1d0)) alpha(n), alpha_t(n), V(n,n)

! Generate random matrices for both A and B.
call rand_gen(y)
A = reshape(y, (/n,n/))
call rand_gen(y)
B = reshape(y, (/n,n/))

! Compute the generalized eigenvalues.
call lin_geig_gen(A, B, alpha, betav)

! Compute the full decomposition once again, A*V = B*V*values.
call lin_geig_gen(A, B, alpha_t, beta_t, &
 v=V)
```

```

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
 err = sum(abs(matmul(A,V) - &
 matmul(B,V)*spread(alpha/betav,DIM=1,NCOPIES=n))) / &
 sum(abs(a)+abs(b))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_GEIG_GEN is correct.'
 end if

 end

```

## Output

Example 1 for LIN\_GEIG\_GEN is correct.

## Additional Examples

### Example 2: Self-Adjoint, Positive-Definite Generalized Eigenvalue Problem

This example illustrates the use of optional flags for the special case where  $A$  and  $B$  are complex self-adjoint matrices, and  $B$  is positive-definite. For purposes of maximum efficiency an option is passed to routine `LIN_SOL_SELF` so that pivoting is not used in the computation of the Cholesky decomposition of matrix  $B$ . This example does not require that secondary option. Also, see `operator_ex34`, supplied with the product examples.

```

 use lin_geig_gen_int
 use lin_sol_self_int
 use rand_gen_int

 implicit none

! This is Example 2 for LIN_GEIG_GEN.

 integer i
 integer, parameter :: n=32
 real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
 real(kind(1d0)) betav(n), temp_c(n,n), temp_d(n,n), err
 type(d_options) :: iopti(4)=d_options(0,zero)
 complex(kind(1d0)), dimension(n,n) :: A, B, C, D, V, alpha(n)

! Generate random matrices for both A and B.
 do i=1, n
 call rand_gen(temp_c(1:n,i))
 call rand_gen(temp_d(1:n,i))
 end do
 c = temp_c; d = temp_c
 do i=1, n
 call rand_gen(temp_c(1:n,i))
 call rand_gen(temp_d(1:n,i))
 end do
 c = cmplx(real(c),temp_c,kind(one))
 d = cmplx(real(d),temp_d,kind(one))

```

```

a = conjg(transpose(c)) + c
b = matmul(conjg(transpose(d)),d)

! Set option so that the generalized eigenvalue solver uses an
! efficient method for well-posed, self-adjoint problems.
iopti(1) = d_options(z_lin_geig_gen_self_adj_pos,zero)
iopti(2) = d_options(z_lin_geig_gen_for_lin_sol_self,zero)

! Number of secondary optional data items and the options:
iopti(3) = d_options(1,zero)
iopti(4) = d_options(z_lin_sol_self_no_pivoting,zero)

call lin_geig_gen(a, b, alpha, betav, v=v, &
iopt=iopti)

! Check that residuals are small. Use the real part of alpha
! since the values are known to be real.
err = sum(abs(matmul(a,v) - matmul(b,v)* &
spread(real(alpha,kind(one))/betav,dim=1,ncopies=n))) / &
sum(abs(a)+abs(b))
if (err <= sqrt(epsilon(one))) then
write (*,*) 'Example 2 for LIN_GEIG_GEN is correct.'
end if

end

```

## Output

Example 2 for LIN\_GEIG\_GEN is correct.

### Example 3: A Test for a Regular Matrix Pencil

In the classification of Differential Algebraic Equations (DAE), a system with linear constant coefficients is given by  $A \dot{x} + Bx = f$ . Here  $A$  and  $B$  are  $n \times n$  matrices, and  $f$  is an  $n$ -vector that is not part of this example. The DAE system is defined as *solvable* if and only if the quantity  $\det(\mu A + B)$  does not vanish identically as a function of the dummy parameter  $\mu$ . A sufficient condition for solvability is that the generalized eigenvalue problem  $Av = \lambda Bv$  is nonsingular. By constructing  $A$  and  $B$  so that both are singular, the routine flags nonsolvability in the DAE by returning NaN for the generalized eigenvalues. Also, see `operator_ex35`, supplied with the product examples.

```

use lin_geig_gen_int
use rand_gen_int
use error_option_packet
use isnan_int

implicit none

! This is Example 3 for LIN_GEIG_GEN.

integer, parameter :: n=6
real(kind(ld0)), parameter :: one=1.0d0, zero=0.0d0
real(kind(ld0)) a(n,n), b(n,n), betav(n), y(n*n)
type(d_options) iopti(1)
type(d_error) epack(1)

```

```

 complex(kind(1d0)) alpha(n)

! Generate random matrices for both A and B.
 call rand_gen(y)
 a = reshape(y, (/n,n/))

 call rand_gen(y)
 b = reshape(y, (/n,n/))

! Make columns of A and B zero, so both are singular.
 a(1:n,n) = 0; b(1:n,n) = 0

! Set internal tolerance for a small diagonal term.
 iopti(1) = d_options(d_lin_geig_gen_set_small,sqrt(epsilon(one)))

! Compute the generalized eigenvalues.
 call lin_geig_gen(a, b, alpha, betav, &
 iopt=iopti,epack=epack)

! See if singular DAE system is detected.
! (The size of epack() is too small for the message, so
! output is blocked with NaNs.)
 if (isnan(alpha)) then
 write (*,*) 'Example 3 for LIN_GEIG_GEN is correct.'
 end if

end

```

## Output

Example 3 for LIN\_GEIG\_GEN is correct.

### Example 4: Larger Data Uncertainty than Working Precision

Data values in both matrices  $A$  and  $B$  are assumed to have relative errors that can be as large as  $\varepsilon^{1/2}$  where  $\varepsilon$  is the relative machine precision. This example illustrates the use of an optional flag that resets the tolerance used in routine `lin_sol_lsq` for determining a singularity of either matrix. The tolerance is reset to the new value  $\varepsilon^{1/2} \|B\|$  and the generalized eigenvalue problem is solved. We anticipate that  $B$  might be singular and detect this fact. Also, see `operator_ex36`, [Chapter 10](#).

```

 use lin_geig_gen_int
 use lin_sol_lsq_int
 use rand_gen_int
 use isNaN_int

 implicit none

! This is Example 4 for LIN_GEIG_GEN.

 integer, parameter :: n=32
 real(kind(1d0)), parameter :: one=1d0, zero=0d0
 real(kind(1d0)) a(n,n), b(n,n), betav(n), y(n*n), err

```

```

type(d_options) iopti(4)
type(d_error) epack(1)
complex(kind(1d0)) alpha(n), v(n,n)

! Generate random matrices for both A and B.

call rand_gen(y)
a = reshape(y, (/n,n/))

call rand_gen(y)
b = reshape(y, (/n,n/))

! Set the option, a larger tolerance than default for lin_sol_lsq.
iopti(1) = d_options(d_lin_geig_gen_for_lin_sol_lsq,zero)

! Number of secondary optional data items
iopti(2) = d_options(2,zero)
iopti(3) = d_options(d_lin_sol_lsq_set_small,sqrt(epsilon(one))*&
sqrt(sum(b**2)/n))
iopti(4) = d_options(d_lin_sol_lsq_no_sing_mess,zero)

! Compute the generalized eigenvalues.
call lin_geig_gen(A, B, alpha, betav, v=v, &
iopt=iopti, epack=epack)

if(.not. isNaN(alpha)) then

! Check the residuals.
err = sum(abs(matmul(A,V)*spread(betav,dim=1,ncopies=n) - &
matmul(B,V)*spread(alpha,dim=1,ncopies=n))) / &
sum(abs(a)+abs(b))
if (err <= sqrt(epsilon(one))) then
write (*,*) 'Example 4 for LIN_GEIG_GEN is correct.'

end if
end if
end

```

## Output

Example 4 for LIN\_GEIG\_GEN is correct.

---

## EVLRG



Computes all of the eigenvalues of a real matrix.

### Required Arguments

*A* — Real full matrix of order *N*. (Input)

*EVAL* — Complex vector of length  $N$  containing the eigenvalues of  $A$  in decreasing order of magnitude. (Output)

### Optional Arguments

$N$  — Order of the matrix. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

### FORTRAN 90 Interface

Generic:     `CALL EVLRG (A, EVAL [, ...])`

Specific:    The specific interface names are `S_EVLRG` and `D_EVLRG`.

### FORTRAN 77 Interface

Single:      `CALL EVLRG (N, A, LDA, EVAL)`

Double:      The double precision name is `DEVLRG`.

### Description

Routine `EVLRG` computes the eigenvalues of a real matrix. The matrix is first balanced. Elementary or Gauss similarity transformations with partial pivoting are used to reduce this balanced matrix to a real upper Hessenberg matrix. A hybrid double – shifted LR – QR algorithm is used to compute the eigenvalues of the Hessenberg matrix, Watkins and Elsner (1990).

The underlying code is based on either `EISPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. The LR – QR algorithm is based on software work of Watkins and Haag. Further details, some timing data, and credits are given in Hanson et al. (1990).

### Comments

1.    Workspace may be explicitly provided, if desired, by use of `E3LRG/DE3LRG`. The reference is:

```
CALL E3LRG (N, A, LDA, EVAL, ACOPY, WK, IWK)
```

The additional arguments are as follows:

*ACOPY* — Real work array of length  $N^2$ .  $A$  and *ACOPY* may be the same, in which case the first  $N^2$  elements of  $A$  will be destroyed.

**WK** — Floating-point work array of size  $4N$ .

**IWK** — Integer work array of size  $2N$ .

2. Informational error

Type Code

4 1 The iteration for an eigenvalue failed to converge.

3. [Integer Options](#) with Chapter 11 Options Manager

**1** This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `E3LRG`, the internal or working leading dimension of `ACOPY` is increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `EVLRG`. Additional memory allocation and option value restoration are automatically done in `EVLRG`. There is no requirement that users change existing applications that use `EVLRG` or `E3LRG`. Default values for the option are

`IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5–8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `EVLRG`.

## Example

In this example, a `DATA` statement is used to set `A` to a matrix given by Gregory and Karney (1969, page 85). The eigenvalues of this real matrix are computed and printed. The exact eigenvalues are known to be  $\{4, 3, 2, 1\}$ .

```
USE EVLRG_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (N=4, LDA=N)
!
REAL A(LDA,N)
COMPLEX EVAL(N)
!
! Set values of A
!
! 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)
DATA A/-2.0, -3.0, -2.0, -1.0, 2.0, 3.0, 0.0, 0.0, 2.0, 2.0, &
 4.0, 0.0, 2.0, 2.0, 2.0, 5.0/
!
! Find eigenvalues of A
CALL EVLRG (A, EVAL)
!
! Print results
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
END
```

## Output

EVAL

( 4.000, 0.000) <sup>1</sup> ( 3.000, 0.000) <sup>2</sup> ( 2.000, 0.000) <sup>3</sup> ( 1.000, 0.000) <sup>4</sup>

---

## EVCRG



Computes all of the eigenvalues and eigenvectors of a real matrix.

### Required Arguments

*A* — Floating-point array containing the matrix. (Input)

*EVAL* — Complex array of size *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Complex array containing the matrix of eigenvectors. (Output)  
The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:    CALL EVCRG (A, EVAL, EVEC [, ...])

Specific:   The specific interface names are S\_EVCRG and D\_EVCRG.

## FORTRAN 77 Interface

Single:      CALL EVCRG (N, A, LDA, EVAL, EVEC, LDEVEC)

Double:     The double precision name is DEVCRG.

## Description

Routine `EVCRG` computes the eigenvalues and eigenvectors of a real matrix. The matrix is first balanced. Orthogonal similarity transformations are used to reduce the balanced matrix to a real upper Hessenberg matrix. The implicit double-shifted QR algorithm is used to compute the eigenvalues and eigenvectors of this Hessenberg matrix. The eigenvectors are normalized such that each has Euclidean length of value one. The largest component is real and positive.

The underlying code is based on either `EISPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. Further details, some timing data, and credits are given in Hanson et al. (1990).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of `E8CRG/DE8CRG`. The reference is:

```
CALL E8CRG (N, A, LDA, EVAL, EVEC, LDEVEC, ACOPY, ECOPY,
 WK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Floating-point work array of size  $N$  by  $N$ . The arrays `A` and `ACOPY` may be the same, in which case the first  $N^2$  elements of `A` will be destroyed. The array `ACOPY` can have its working row dimension increased from  $N$  to a larger value. An optional usage is required. See Item 3 below for further details.

**ECOPY** — Floating-point work array of default size  $N$  by  $N + 1$ . The working, leading dimension of `ECOPY` is the same as that for `ACOPY`. To increase this value, an optional usage is required. See Item 3 below for further details.

**WK** — Floating-point work array of size  $6N$ .

**IWK** — Integer work array of size  $N$ .

2.    Informational error

Type   Code

4      1      The iteration for the eigenvalues failed to converge. No eigenvalues or eigenvectors are computed.

3.    [Integer Options](#) with Chapter 11 Options Manager

- 1 This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `E8CRG`, the internal or working leading dimensions of `ACOPY` and `ECOPY` are both increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `EVCRG`. Additional memory allocation and option value restoration are automatically done in `EVCRG`. There is no requirement that users change existing applications that use `EVCRG` or `E8CRG`. Default values for the option are `IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5–8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `EVCRG`.

## Example

In this example, a `DATA` statement is used to set `A` to a matrix given by Gregory and Karney (1969, page 82). The eigenvalues and eigenvectors of this real matrix are computed and printed. The performance index is also computed and printed. This serves as a check on the computations. For more details, see IMSL routine `EPIRG`.

```

USE EVCRG_INT
USE EPIRG_INT
USE UMACH_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=3, LDA=N, LDEVEC=N)
INTEGER NOUT
REAL PI
COMPLEX EVAL(N), EVEC(LDEVEC,N)
REAL A(LDA,N)

! Define values of A:
!
! A = (8.0 -1.0 -5.0)
! (-4.0 4.0 -2.0)
! (18.0 -5.0 -7.0)
!
DATA A/8.0, -4.0, 18.0, -1.0, 4.0, -5.0, -5.0, -2.0, -7.0/
!
! Find eigenvalues and vectors of A
CALL EVCRG (A, EVAL, EVEC)
!
! Compute performance index
PI = EPIRG(N,A,EVAL,EVEC)
!
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3
(2.000, 4.000) (2.000,-4.000) (1.000, 0.000)

 EVEC
 1 2 3
1 (0.3162, 0.3162) (0.3162,-0.3162) (0.4082, 0.0000)
2 (-0.0000, 0.6325) (-0.0000,-0.6325) (0.8165, 0.0000)
3 (0.6325, 0.0000) (0.6325, 0.0000) (0.4082, 0.0000)

Performance index = 0.026
```

---

## EPIRG

This function computes the performance index for a real eigensystem.

### Function Return Value

*EPIRG* — Performance index. (Output)

### Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs on which the performance index computation is based. (Input)

*A* — Matrix of order *N*. (Input)

*EVAL* — Complex vector of length *NEVAL* containing eigenvalues of *A*. (Input)

*EVEC* — Complex *N* by *NEVAL* array containing eigenvectors of *A*. (Input)  
The eigenvector corresponding to the eigenvalue *EVAL*(*J*) must be in the *J*-th column of *EVEC*.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic: EPIRG (NEVAL, A, EVAL, EVEC [, ...])

Specific: The specific interface names are S\_EPIRG and D\_EPIRG.

## FORTRAN 77 Interface

Single: EPIRG (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC)

Double: The double precision function name is DEPIRG.

## Description

Let  $M = \text{NEVAL}$ ,  $\lambda = \text{EVAL}$ ,  $x_j = \text{EVEC}(*, J)$ , the  $j$ -th column of  $\text{EVEC}$ . Also, let  $\varepsilon$  be the machine precision given by  $\text{AMACH}(4)$ . The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\varepsilon \|A\|_1 \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{|\Re v_i| + |\Im v_i|\}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of  $\text{EVCSF}$  is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ .

The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

## Comments

1. Workspace may be explicitly provided, if desired, by use of  $\text{E2IRG}/\text{DE2IRG}$ . The reference is:

$\text{E2IRG} (N, \text{NEVAL}, A, \text{LDA}, \text{EVAL}, \text{EVEC}, \text{LDEVEC}, \text{CWK})$

The additional argument is:

**CWK** — Complex work array of length  $N$ .

2. Informational errors

Type Code

3 1 The performance index is greater than 100.

3 2 An eigenvector is zero.

3 3 The matrix is zero.

### Example

For an example of `EPIRG`, see IMSL routine [EVCRG](#).

---

## EVLCG



Computes all of the eigenvalues of a complex matrix.

### Required Arguments

*A* — Complex matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = `SIZE (A,2)`.

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = `SIZE (A,1)`.

### FORTRAN 90 Interface

Generic: `CALL EVLCG (A, EVAL [, ...])`

Specific: The specific interface names are `S_EVLCG` and `D_EVLCG`.

### FORTRAN 77 Interface

Single: `CALL EVLCG (N, A, LDA, EVAL)`

Double: The double precision name is `EVLCG`.

### Description

Routine `EVLCG` computes the eigenvalues of a complex matrix. The matrix is first balanced. Unitary similarity transformations are used to reduce this balanced matrix to a complex upper

Hessenberg matrix. The shifted QR algorithm is used to compute the eigenvalues of this Hessenberg matrix.

The underlying code is based on either EISPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of E3LCG/DE3LCG. The reference is:

```
CALL E3LCG (N, A, LDA, EVAL, ACOFY, RWK, CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . A and ACOFY may be the same, in which case the first  $N^2$  elements of A will be destroyed.

**RWK** — Work array of length N.

**CWK** — Complex work array of length 2N.

**IWK** — Integer work array of length N.

2. Informational error

| Type | Code |
|------|------|
|------|------|

|   |   |                                                     |
|---|---|-----------------------------------------------------|
| 4 | 1 | The iteration for an eigenvalue failed to converge. |
|---|---|-----------------------------------------------------|

3. [Integer Options](#) with Chapter 11 Options Manager

**1** This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine E3LCG, the internal or working, leading dimension of ACOFY is increased by IVAL(3) when N is a multiple of IVAL(4). The values IVAL(3) and IVAL(4) are temporarily replaced by IVAL(1) and IVAL(2), respectively, in routine EVLCG. Additional memory allocation and option value restoration are automatically done in EVLCG. There is no requirement that users change existing applications that use EVLCG or E3LCG. Default values for the option are IVAL(\*) = 1, 16, 0, 1, 1, 16, 0, 1. Items 5–8 in IVAL(\*) are for the generalized eigenvalue problem and are not used in EVLCG.

## Example

In this example, a DATA statement is used to set A to a matrix given by Gregory and Karney (1969, page 115). The program computes the eigenvalues of this matrix.

```
USE EVLCG_INT
```

```

 USE WRCRN_INT
! Declare variables
 INTEGER LDA, N
 PARAMETER (N=3, LDA=N)
!
 COMPLEX A(LDA,N), EVAL(N)
! Set values of A
! A = (1+2i 3+4i 21+22i)
! (43+44i 13+14i 15+16i)
! (5+6i 7+8i 25+26i)
!
 DATA A/(1.0,2.0), (43.0,44.0), (5.0,6.0), (3.0,4.0), &
 (13.0,14.0), (7.0,8.0), (21.0,22.0), (15.0,16.0), &
 (25.0,26.0)/
!
! Find eigenvalues of A
 CALL EVLCG (A, EVAL)
! Print results
 CALL WRCRN ('EVAL', EVAL, 1, N, 1)
 END

```

## Output

```

 EVAL
 1 2 3
(39.78, 43.00) (6.70, -7.88) (-7.48, 6.88)

```

---

## EVCCG



Computes all of the eigenvalues and eigenvectors of a complex matrix.

### Required Arguments

*A* — Complex matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Complex matrix of order *N*. (Output)

The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = *SIZE* (*A*,2).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

**LDEVEC** — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDEVEC = SIZE (EVEC,1)`.

### **FORTRAN 90 Interface**

Generic:     `CALL EVCCG (A, EVAL, EVEC [, ...])`

Specific:    The specific interface names are `S_EVCCG` and `D_EVCCG`.

### **FORTRAN 77 Interface**

Single:      `CALL EVCCG (N, A, LDA, EVAL, EVEC, LDEVEC)`

Double:     The double precision name is `DEVCCG`.

### **Description**

Routine `EVCCG` computes the eigenvalues and eigenvectors of a complex matrix. The matrix is first balanced. Unitary similarity transformations are used to reduce this balanced matrix to a complex upper Hessenberg matrix. The QR algorithm is used to compute the eigenvalues and eigenvectors of this Hessenberg matrix. The eigenvectors of the original matrix are computed by transforming the eigenvectors of the complex upper Hessenberg matrix.

The underlying code is based on either `EISPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `E6CCG/DE6CCG`. The reference is:

```
CALL E6CCG (N, A, LDA, EVAL, EVEC, LDEVEC, ACOPY, RWK,
 CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . The arrays *A* and *ACOPY* may be the same, in which case the first  $N^2$  elements of *A* will be destroyed.

**RWK** — Work array of length *N*.

**CWK** — Complex work array of length  $2N$ .

**IWK** — Integer work array of length N.

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge. No eigenvalues or eigenvectors are computed.

3. [Integer Options](#) with Chapter 11 Options Manager

**1** This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `E6CCG`, the internal or working leading dimensions of `ACOPY` and `ECOPY` are both increased by `IVAL(3)` when N is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `EVCCG`. Additional memory allocation and option value restoration are automatically done in `EVCCG`. There is no requirement that users change existing applications that use `EVCCG` or `E6CCG`. Default values for the option are `IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5–8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `EVCCG`.

## Example

In this example, a `DATA` statement is used to set  $A$  to a matrix given by Gregory and Karney (1969, page 116). Its eigenvalues are known to be  $\{1 + 5i, 2 + 6i, 3 + 7i, 4 + 8i\}$ . The program computes the eigenvalues and eigenvectors of this matrix. The performance index is also computed and printed. This serves as a check on the computations, for more details, see IMSL routine [EPICG](#).

```
USE EVCCG_INT
USE EPICG_INT
USE WRCRN_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=4, LDA=N, LDEVEC=N)
!
INTEGER NOUT
REAL PI
COMPLEX A(LDA,N), EVAL(N), EVEC(LDEVEC,N)
!
! Set values of A
!
! A = (5+9i 5+5i -6-6i -7-7i)
! (3+3i 6+10i -5-5i -6-6i)
! (2+2i 3+3i -1+3i -5-5i)
! (1+i 2+2i -3-3i 4i)
!
DATA A/(5.0,9.0), (3.0,3.0), (2.0,2.0), (1.0,1.0), (5.0,5.0), &
 (6.0,10.0), (3.0,3.0), (2.0,2.0), (-6.0,-6.0), (-5.0,-5.0), &
 (-1.0,3.0), (-3.0,-3.0), (-7.0,-7.0), (-6.0,-6.0), &
 (-5.0,-5.0), (0.0,4.0)/
!
```

```

! Find eigenvalues and vectors of A
 CALL EVCCG (A, EVAL, EVEC)
! Compute performance index
 PI = EPICG(N,A,EVAL,EVEC)
! Print results
 CALL UMACH (2, NOUT)
 CALL WRCRN ('EVAL', EVAL, 1, N, 1)
 CALL WRCRN ('EVEC', EVEC)

 WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
 END

```

## Output

```

 EVAL
 1 2 3 4
(4.000, 8.000) (3.000, 7.000) (2.000, 6.000) (1.000, 5.000)

 EVEC
 1 2 3
4
1 (0.5774, 0.0000) (0.5774, 0.0000) (0.3780, 0.0000) (0.7559,
0.0000)
2 (0.5774,-0.0000) (0.5773,-0.0000) (0.7559, 0.0000) (0.3780,
0.0000)
3 (0.5774,-0.0000) (-0.0000,-0.0000) (0.3780, 0.0000) (0.3780,
0.0000)
4 (0.0000, 0.0000) (0.5774, 0.0000) (0.3780, 0.0000) (0.3780,
0.0000)

Performance index = 0.016

```

---

## EPICG

This function computes the performance index for a complex eigensystem.

### Function Return Value

*EPICG* — Performance index. (Output)

### Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs on which the performance index computation is based. (Input)

*A* — Complex matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues of *A*. (Input)

*EVEC* — Complex matrix of order *N* containing the eigenvectors of *A*. (Input)  
The *J*-th eigenvalue/eigenvector pair should be in *EVAL*(*J*) and in the *J*-th column of *EVEC*.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default:  $N = \text{SIZE}(A, 2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A, 1)$ .

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDEVEC = \text{SIZE}(EVEC, 1)$ .

## FORTRAN 90 Interface

Generic:    `EPICG (NEVAL, A, EVAL, EVEC [, ...])`

Specific:    The specific interface names are `S_EPICG` and `D_EPICG`.

## FORTRAN 77 Interface

Single:    `EPICG (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC)`

Double:    The double precision function name is `DEPICG`.

## Description

Let  $M = \text{NEVAL}$ ,  $\lambda = \text{EVAL}$ ,  $x_j = \text{EVEC}(*, J)$ , the  $j$ -th column of *EVEC*. Also, let  $\varepsilon$  be the machine precision given by `AMACH(4)`. The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\varepsilon \|A\|_1 \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{|\Re v_i| + |\Im v_i|\}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of `EVCSE` is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of `E2ICG/DE2ICG`. The reference is:

E2ICG (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC, WK)

The additional argument is:

**WK** — Complex work array of length N.

2. Informational errors

Type Code

|   |   |                                        |
|---|---|----------------------------------------|
| 3 | 1 | Performance index is greater than 100. |
| 3 | 2 | An eigenvector is zero.                |
| 3 | 3 | The matrix is zero.                    |

### Example

For an example of `EPICG`, see IMSL routine [EVCCG](#).

---

## EVLSF

Computes all of the eigenvalues of a real symmetric matrix.

### Required Arguments

**A** — Real symmetric matrix of order N. (Input)

**EVAL** — Real vector of length N containing the eigenvalues of A in decreasing order of magnitude. (Output)

### Optional Arguments

**N** — Order of the matrix A. (Input)  
Default: `N = SIZE (A,2)`.

**LDA** — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

### FORTRAN 90 Interface

Generic: `CALL EVLSF (A, EVAL [, ...])`

Specific: The specific interface names are `S_EVLSF` and `D_EVLSF`.

### FORTRAN 77 Interface

Single: `CALL EVLSF (N, A, LDA, EVAL)`

Double: The double precision name is DEVLSEF.

## Description

Routine EVLSF computes the eigenvalues of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. Then, an implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix.

The underlying code is based on either EISPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1. Workspace may be explicitly provided, if desired, by use of E4LSF/DE4LSF. The reference is:

```
CALL E4LSF (N, A, LDA, EVAL, WORK, IWORK)
```

The additional arguments are as follows:

**WORK** — Work array of length 2N.

**IWORK** — Integer array of length N.

2. Informational error

Type Code

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 1 | The iteration for the eigenvalue failed to converge in 100 iterations before deflating. |
|---|---|-----------------------------------------------------------------------------------------|

## Example

In this example, the eigenvalues of a real symmetric matrix are computed and printed. This matrix is given by Gregory and Karney (1969, page 56).

```
USE EVLSF_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (N=4, LDA=N)
!
REAL A(LDA,N), EVAL(N)
! Set values of A
!
! A = (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)
!
DATA A /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 /
!
! Find eigenvalues of A
CALL EVLSF (A, EVAL)
! Print results
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
END

```

## Output

|       | EVAL |      |       |  |
|-------|------|------|-------|--|
| 1     | 2    | 3    | 4     |  |
| 15.00 | 5.00 | 5.00 | -1.00 |  |

---

## EVCSF

Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.

### Required Arguments

*A* — Real symmetric matrix of order *N*. (Input)

*EVAL* — Real vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Real matrix of order *N*. (Output)

The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:    CALL EVCSF (A, EVAL, EVEC [, ...])

Specific:   The specific interface names are *S\_EVCSF* and *D\_EVCSF*.

## FORTRAN 77 Interface

Single:      CALL EVCSF (N, A, LDA, EVAL, EVEC, LDEVEC)

Double:     The double precision name is DEVCSF.

## Description

Routine `EVCSF` computes the eigenvalues and eigenvectors of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. These transformations are accumulated. An implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix. The eigenvectors are computed using the eigenvalues as perfect shifts, Parlett (1980, pages 169, 172). The underlying code is based on either `EISPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual. Further details, some timing data, and credits are given in Hanson et al. (1990).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of `E5CSF/DE5CSF`. The reference is:

```
CALL E5CSF (N, A, LDA, EVAL, EVEC, LDEVEC, WORK, IWK)
```

The additional argument is:

**WORK** — Work array of length  $3N$ .

**IWK** — Integer array of length  $N$ .

2.    Informational error

Type   Code

|   |   |                                                                                         |
|---|---|-----------------------------------------------------------------------------------------|
| 3 | 1 | The iteration for the eigenvalue failed to converge in 100 iterations before deflating. |
|---|---|-----------------------------------------------------------------------------------------|

## Example

The eigenvalues and eigenvectors of this real symmetric matrix are computed and printed. The performance index is also computed and printed. This serves as a check on the computations. For more details, see [EPISF](#).

```
USE EVCSF_INT
USE EPISF_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=3, LDA=N, LDEVEC=N)
```

```

!
INTEGER NOUT
REAL A(LDA,N), EVAL(N), EVEC(LDEVEC,N), PI
!
! Set values of A
!
! A = (7.0 -8.0 -8.0)
! (-8.0 -16.0 -18.0)
! (-8.0 -18.0 13.0)
!
DATA A/7.0, -8.0, -8.0, -8.0, -16.0, -18.0, -8.0, -18.0, 13.0/
!
! Find eigenvalues and vectors of A
CALL EVCSF (A, EVAL, EVEC)
!
! Compute performance index
PI = EPISF (N, A, EVAL, EVEC)
!
! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
CALL WRRRN ('EVEC', EVEC)

WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3
-27.90 22.68 9.22

 EVEC
 1 2 3
1 0.2945 -0.2722 0.9161
2 0.8521 -0.3591 -0.3806
3 0.4326 0.8927 0.1262

Performance index = 0.019

```

---

## EVASF

Computes the largest or smallest eigenvalues of a real symmetric matrix.

### Required Arguments

*NEVAL* — Number of eigenvalues to be computed. (Input)

*A* — Real symmetric matrix of order *N*. (Input)

*SMALL* — Logical variable. (Input)

If *.TRUE.*, the smallest *NEVAL* eigenvalues are computed. If *.FALSE.*, the largest *NEVAL* eigenvalues are computed.

*EVAL* — Real vector of length *NEVAL* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

### FORTRAN 90 Interface

Generic:     `CALL EVASF (NEVAL, A, SMALL, EVAL [, ...])`

Specific:    The specific interface names are *S\_EVASF* and *D\_EVASF*.

### FORTRAN 77 Interface

Single:      `CALL EVASF (N, NEVAL, A, LDA, SMALL, EVAL)`

Double:     The double precision name is *DEVASF*.

### Description

Routine *EVASF* computes the largest or smallest eigenvalues of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. Then, an implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix.

The reduction routine is based on the EISPACK routine *TRED2*. See Smith et al. (1976). The rational QR algorithm is called the PWK algorithm. It is given in Parlett (1980, page 169).

### Comments

1.    Workspace may be explicitly provided, if desired, by use of *E4ASF/DE4ASF*. The reference is:

`CALL E4ASF (N, NEVAL, A, LDA, SMALL, EVAL, WORK, IWK)`

***WORK*** — Work array of length *4N*.

***IWK*** — Integer work array of length *N*.

2.    Informational error

Type   Code

3      1      The iteration for an eigenvalue failed to converge. The best estimate will be returned.

## Example

In this example, the three largest eigenvalues of the computed Hilbert matrix  $a_{ij} = 1/(i + j - 1)$  of order  $N = 10$  are computed and printed.

```
 USE EVASF_INT
 USE WRRRN_INT

 IMPLICIT NONE
! Declare variables
 INTEGER LDA, N, NEVAL
 PARAMETER (N=10, NEVAL=3, LDA=N)
!
 INTEGER I, J
 REAL A(LDA,N), EVAL(NEVAL), REAL
 LOGICAL SMALL
 INTRINSIC REAL
! Set up Hilbert matrix
 DO 20 J=1, N
 DO 10 I=1, N
 A(I,J) = 1.0/REAL(I+J-1)
10 CONTINUE
20 CONTINUE
! Find the 3 largest eigenvalues
 SMALL = .FALSE.
 CALL EVASF (NEVAL, A, SMALL, EVAL)
! Print results
 CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)

 END
```

## Output

```
 EVAL
 1 2 3
1.752 0.343 0.036
```

---

## EVESF

Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.

### Required Arguments

*NEVEC* — Number of eigenvalues to be computed. (Input)

*A* — Real symmetric matrix of order  $N$ . (Input)

*SMALL* — Logical variable. (Input)

If *.TRUE.*, the smallest *NEVEC* eigenvalues are computed. If *.FALSE.*, the largest *NEVEC* eigenvalues are computed.

*EVAL* — Real vector of length *NEVEC* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Real matrix of dimension *N* by *NEVEC*. (Output)  
The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:     CALL *EVESF* (*NEVEC*, *A*, *SMALL*, *EVAL*, *EVEC* [, ...])

Specific:    The specific interface names are *S\_EVESF* and *D\_EVESF*.

### FORTRAN 77 Interface

Single:     CALL *EVESF* (*N*, *NEVEC*, *A*, *LDA*, *SMALL*, *EVAL*, *EVEC*, *LDEVEC*)

Double:     The double precision name is *DEVESF*.

### Description

Routine *EVESF* computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. Then, an implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix. Inverse iteration is used to compute the eigenvectors of the tridiagonal matrix. This is followed by orthogonalization of these vectors. The eigenvectors of the original matrix are computed by back transforming those of the tridiagonal matrix.

The reduction routine is based on the EISPACK routine *TRED2*. See Smith et al. (1976). The rational QR algorithm is called the PWK algorithm. It is given in Parlett (1980, page 169). The inverse iteration and orthogonalization computation is discussed in Hanson et al. (1990). The back transformation routine is based on the EISPACK routine *TRBAK1*.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E5ESF/DE5ESF`. The reference is:

```
CALL E5ESF (N, NEVEC, A, LDA, SMALL, EVAL, EVEC, LDEVEC,
 WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $9N$ .

**IWK** — Integer array of length  $N$ .

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                              |
|---|---|----------------------------------------------------------------------------------------------|
| 3 | 1 | The iteration for an eigenvalue failed to converge. The best estimate will be returned.      |
| 3 | 2 | Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue. |
| 3 | 3 | The eigenvectors have lost orthogonality.                                                    |

## Example

In this example, a `DATA` statement is used to set  $A$  to a matrix given by Gregory and Karney (1969, page 55). The largest two eigenvalues and their eigenvectors are computed and printed. The performance index is also computed and printed. This serves as a check on the computations. For more details, see IMSL routine [EPISF](#).

```
USE EVESF_INT
USE EPISF_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=4, LDA=N, LDEVEC=N)
!
INTEGER NEVEC, NOUT
REAL A(LDA,N), EVAL(N), EVEC(LDEVEC,N), PI
LOGICAL SMALL

!
! Set values of A
!
! 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)
!
```

```

DATA 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/
!
! Find eigenvalues and vectors of A
NEVEC = 2
SMALL = .FALSE.
CALL EVESF (NEVEC, A, SMALL, EVAL, EVEC)
! Compute performance index
PI = EPISF (NEVEC, A, EVAL, EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, NEVEC, 1)
CALL WRRRN ('EVEC', EVEC, N, NEVEC, LDEVEC)

WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2
10.00 5.00

 EVEC
 1 2
1 0.6325 -0.3162
2 0.6325 -0.3162
3 0.3162 0.6325
4 0.3162 0.6325

Performance index = 0.031

```

---

## EVBSF

Computes selected eigenvalues of a real symmetric matrix.

### Required Arguments

***MXEVAL*** — Maximum number of eigenvalues to be computed. (Input)

***A*** — Real symmetric matrix of order *N*. (Input)

***ELOW*** — Lower limit of the interval in which the eigenvalues are sought. (Input)

***EHIGH*** — Upper limit of the interval in which the eigenvalues are sought. (Input)

***NEVAL*** — Number of eigenvalues found. (Output)

***EVAL*** — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are significant.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

## FORTRAN 90 Interface

Generic: CALL EVBSF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL [, ...])

Specific: The specific interface names are S\_EVBSF and D\_EVBSF.

## FORTRAN 77 Interface

Single: CALL EVBSF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL)

Double: The double precision name is DEVBSF.

## Description

Routine EVBSF computes the eigenvalues in a given interval for a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. Then, an implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix. The reduction step is based on the EISPACK routine TRED1. See Smith et al. (1976). The rational QR algorithm is called the PWK algorithm. It is given in Parlett (1980, page 169).

## Comments

1. Workspace may be explicitly provided, if desired, by use of E5BSF/DE5BSF. The reference is

```
CALL E5BSF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL,
 WK, IWK)
```

The additional arguments are as follows:

*WK* — Work array of length 5*N*.

*IWK* — Integer work array of length 1*N*.

2. Informational error

Type Code

3 1 The number of eigenvalues in the specified interval exceeds MXEVAL. NEVAL contains the number of eigenvalues in the interval. No eigenvalues will be returned.

### Example

In this example, a DATA statement is used to set A to a matrix given by Gregory and Karney (1969, page 56). The eigenvalues of A are known to be -1, 5, 5 and 15. The eigenvalues in the interval [1.5, 5.5] are computed and printed. As a test, this example uses MXEVAL = 4. The routine EVBSF computes NEVAL, the number of eigenvalues in the given interval. The value of NEVAL is 2.

```

USE EVBSF_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, MXEVAL, N
PARAMETER (MXEVAL=4, N=4, LDA=N)
!
INTEGER NEVAL, NOUT
REAL A(LDA,N), EHIGH, ELOW, EVAL(MXEVAL)
!
! Set values of A
!
! A = (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)
!
DATA A/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/
!
! Find eigenvalues of A
ELOW = 1.5
EHIGH = 5.5
CALL EVBSF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, '(/,A,I2)') ' NEVAL = ', NEVAL
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
END

```

### Output

```

NEVAL = 2

EVAL
 1 2
5.000 5.000

```

---

## EVFSF

Computes selected eigenvalues and eigenvectors of a real symmetric matrix.

## Required Arguments

***MXEVAL*** — Maximum number of eigenvalues to be computed. (Input)

***A*** — Real symmetric matrix of order *N*. (Input)

***ELOW*** — Lower limit of the interval in which the eigenvalues are sought. (Input)

***EHIGH*** — Upper limit of the interval in which the eigenvalues are sought. (Input)

***NEVAL*** — Number of eigenvalues found. (Output)

***EVAL*** — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are significant.

***EVEC*** — Real matrix of dimension *N* by *MXEVAL*. (Output)  
The *J*-th eigenvector corresponding to *EVAL*(*J*), is stored in the *J*-th column. Only the first *NEVAL* columns of *EVEC* are significant. Each vector is normalized to have Euclidean length equal to the value one.

## Optional Arguments

***N*** — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

***LDA*** — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

***LDEVEC*** — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic:     CALL EVFSF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL, EVEC [, ...] )

Specific:    The specific interface names are *S\_EVFSF* and *D\_EVFSF*.

## FORTRAN 77 Interface

Single:     CALL EVFSF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL, EVEC,  
                  LDEVEC)

Double:     The double precision name is *DEVFSF*.

## Description

Routine `EVFSF` computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. Then, an implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix. Inverse iteration is used to compute the eigenvectors of the tridiagonal matrix. This is followed by orthogonalization of these vectors. The eigenvectors of the original matrix are computed by back transforming those of the tridiagonal matrix.

The reduction step is based on the EISPACK routine `TRED1`. The rational QR algorithm is called the PWK algorithm. It is given in Parlett (1980, page 169). The inverse iteration and orthogonalization processes are discussed in Hanson et al. (1990). The transformation back to the users's input matrix is based on the EISPACK routine `TRBAK1`. See Smith et al. (1976) for the EISPACK routines.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E3FSF/DE3FSF`. The reference is:

```
CALL E3FSF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, VAL,
 EVEC, LDEVEC, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $9N$ .

**IWK** — Integer work array of length  $N$ .

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                                                                     |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The number of eigenvalues in the specified range exceeds <code>MXEVAL</code> . <code>NEVAL</code> contains the number of eigenvalues in the range. No eigenvalues will be computed. |
| 3 | 2 | Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue.                                                                                        |
| 3 | 3 | The eigenvectors have lost orthogonality.                                                                                                                                           |

## Example

In this example, `A` is set to the computed Hilbert matrix. The eigenvalues in the interval  $[0.001, 1]$  and their corresponding eigenvectors are computed and printed. This example uses `MXEVAL = 3`. The routine `EVFSF` computes the number of eigenvalues `NEVAL` in the given interval. The value of `NEVAL` is 2. The performance index is also computed and printed. For more details, see IMSL routine [EPISF](#).

```
USE EVFSF_INT
USE EPISF_INT
USE WRRRN_INT
```

```

 USE UMACH_INT

 IMPLICIT NONE

! Declare variables
 INTEGER LDA, LDEVEC, MXEVAL, N, J, I
 PARAMETER (MXEVAL=3, N=3, LDA=N, LDEVEC=N)

!
 INTEGER NEVAL, NOUT
 REAL A(LDA,N), EHIGH, ELOW, EVAL(MXEVAL), &
 EVEC(LDEVEC,MXEVAL), PI

! Compute Hilbert matrix
 DO 20 J=1,N
 DO 10 I=1,N
 A(I,J) = 1.0/FLOAT(I+J-1)
10 CONTINUE
20 CONTINUE

! Find eigenvalues and vectors
 ELOW = 0.001
 EHIGH = 1.0
 CALL EVFSF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL, EVEC, LDEVEC)

! Compute performance index
 PI = EPISF(NEVAL,A,EVAL,EVEC)

! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,'(/,A,I2)') ' NEVAL = ', NEVAL
 CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
 CALL WRRRN ('EVEC', EVEC, N, NEVAL, LDEVEC)
 WRITE (NOUT,'(/,A,F6.3)') ' Performance index = ', PI
 END

```

## Output

```

NEVAL = 2

 EVAL
 1 2
0.1223 0.0027

 EVEC
 1 2
1 -0.5474 -0.1277
2 0.5283 0.7137
3 0.6490 -0.6887

Performance index = 0.008

```

---

## EPISF

This function computes the performance index for a real symmetric eigensystem.

### Function Return Value

*EPISF* — Performance index. (Output)

## Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs on which the performance index computation is based on. (Input)

*A* — Symmetric matrix of order *N*. (Input)

*EVAL* — Vector of length *NEVAL* containing eigenvalues of *A*. (Input)

*EVEC* — *N* by *NEVAL* array containing eigenvectors of *A*. (Input)  
The eigenvector corresponding to the eigenvalue *EVAL*(*J*) must be in the *J*-th column of *EVEC*.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)

Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic: `EPISF (NEVAL, A, EVAL, EVEC [, ...])`

Specific: The specific interface names are `S_EPISF` and `D_EPISF`.

## FORTRAN 77 Interface

Single: `EPISF (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC)`

Double: The double precision function name is `DEPISF`.

## Description

Let  $M = \text{NEVAL}$ ,  $\lambda = \text{EVAL}$ ,  $x_j = \text{EVEC}(*,J)$ , the *j*-th column of *EVEC*. Also, let  $\varepsilon$  be the machine precision, given by `AMACH(4)`, see the [Reference](#) chapter of this manual. The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\varepsilon \|A\|_1 \|x_j\|_1}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of [EVCSF](#) is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E2ISF/DE2ISF`. The reference is:

```
E2ISF (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC, WORK)
```

The additional argument is:

**WORK** — Work array of length `N`.

**E2ISF** — Performance Index.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                        |
|---|---|----------------------------------------|
| 3 | 1 | Performance index is greater than 100. |
|---|---|----------------------------------------|

|   |   |                         |
|---|---|-------------------------|
| 3 | 2 | An eigenvector is zero. |
|---|---|-------------------------|

|   |   |                     |
|---|---|---------------------|
| 3 | 3 | The matrix is zero. |
|---|---|---------------------|

## Example

For an example of `EPISF`, see routine [EVCSF](#).

---

# EVLSB

Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode.

## Required Arguments

**A** — Band symmetric matrix of order `N`. (Input)

**NCODA** — Number of codiagonals in `A`. (Input)

**EVAL** — Vector of length `N` containing the eigenvalues of `A` in decreasing order of magnitude. (Output)

## Optional Arguments

**N** — Order of the matrix `A`. (Input)  
Default: `N = SIZE(A,2)`.

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

### **FORTRAN 90 Interface**

Generic: `CALL EVLSB (A, NCODA, EVAL [, ...])`

Specific: The specific interface names are `S_EVLSB` and `D_EVLSB`.

### **FORTRAN 77 Interface**

Single: `CALL EVLSB (N, A, LDA, NCODA, EVAL)`

Double: The double precision name is `DEVLSB`.

### **Description**

Routine `EVLSB` computes the eigenvalues of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. The implicit QL algorithm is used to compute the eigenvalues of the resulting tridiagonal matrix.

The reduction routine is based on the EISPACK routine `BANDR`; see Garbow et al. (1977). The QL routine is based on the EISPACK routine `IMTQL1`; see Smith et al. (1976).

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `E3LSB/DE3LSB`. The reference is:

```
CALL E3LSB (N, A, LDA, NCODA, EVAL, ACOPY, WK)
```

The additional arguments are as follows:

**ACOPY** — Work array of length  $N(NCODA + 1)$ . The arrays *A* and *ACOPY* may be the same, in which case the first  $N(NCODA + 1)$  elements of *A* will be destroyed.

**WK** — Work array of length *N*.

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

### **Example**

In this example, a `DATA` statement is used to set *A* to a matrix given by Gregory and Karney (1969, page 77). The eigenvalues of this matrix are given by

$$\lambda_k = \left(1 - 2 \cos \frac{k\pi}{N+1}\right)^2 - 3$$

Since the eigenvalues returned by EVLSB are in decreasing magnitude, the above formula for  $k = 1, \dots, N$  gives the values in a different order. The eigenvalues of this real band symmetric matrix are computed and printed.

```

 USE EVLSB_INT
 USE WRRRN_INT

 IMPLICIT NONE
!
! Declare variables
 INTEGER LDA, LDEVEC, N, NCODA
 PARAMETER (N=5, NCODA=2, LDA=NCODA+1, LDEVEC=N)
!
 REAL A(LDA,N), EVAL(N)
!
! Define values of A:
! A = (-1 2 1)
! (2 0 2 1)
! (1 2 0 2 1)
! (1 2 0 2)
! (1 2 -1)
! Represented in band symmetric
! form this is:
! A = (0 0 1 1 1)
! (0 2 2 2 2)
! (-1 0 0 0 -1)
!
 DATA A/0.0, 0.0, -1.0, 0.0, 2.0, 0.0, 1.0, 2.0, 0.0, 1.0, 2.0, &
 0.0, 1.0, 2.0, -1.0/
!
 CALL EVLSB (A, NCODA, EVAL)
!
! Print results
 CALL WRRRN ('EVAL', EVAL, 1, N, 1)
 END

```

## Output

|  | EVAL  |        |        |        |       |
|--|-------|--------|--------|--------|-------|
|  | 1     | 2      | 3      | 4      | 5     |
|  | 4.464 | -3.000 | -2.464 | -2.000 | 1.000 |

---

## EVCSB

Computes all of the eigenvalues and eigenvectors of a real symmetric matrix in band symmetric storage mode.

### Required Arguments

*A* — Band symmetric matrix of order *N*. (Input)

*NCODA* — Number of codiagonals in *A*. (Input)

*EVAL* — Vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Matrix of order *N* containing the eigenvectors. (Output)  
The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:     CALL *EVCSB* (*A*, *NCODA*, *EVAL*, *EVEC* [, ...])

Specific:    The specific interface names are *S\_EVCSB* and *D\_EVCSB*.

### FORTRAN 77 Interface

Single:      CALL *EVCSB* (*N*, *A*, *LDA*, *NCODA*, *EVAL*, *EVEC*, *LDEVEC*)

Double:     The double precision name is *DEVCSB*.

### Description

Routine *EVCSB* computes the eigenvalues and eigenvectors of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. These transformations are accumulated. The implicit QL algorithm is used to compute the eigenvalues and eigenvectors of the resulting tridiagonal matrix.

The reduction routine is based on the EISPACK routine *BANDR*; see Garbow et al. (1977). The QL routine is based on the EISPACK routine *IMTQL2*; see Smith et al. (1976).

### Comments

1.     Workspace may be explicitly provided, if desired, by use of *E4CSB/DE4CSB*. The reference is:

```
CALL E4CSB (N, A, LDA, NCODA, EVAL, EVEC, LDEVEC, COPY,
 WK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Work array of length  $N(NCODA + 1)$ . **A** and **ACOPY** may be the same, in which case the first  $N * NCODA$  elements of **A** will be destroyed.

**WK** — Work array of length  $N$ .

**IWK** — Integer work array of length  $N$ .

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

3. The success of this routine can be checked using [EPISB](#).

### Example

In this example, a **DATA** statement is used to set **A** to a band matrix given by Gregory and Karney (1969, page 75). The eigenvalues,  $\lambda_k$ , of this matrix are given by

$$\lambda_k = 16 \sin^4 \left( \frac{k\pi}{2N+2} \right)$$

The eigenvalues and eigenvectors of this real band symmetric matrix are computed and printed. The performance index is also computed and printed. This serves as a check on the computations; for more details, see IMSL routine [EPISB](#).

```
USE EVCSB_INT
USE EPISB_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDEVEC, N, NCODA
PARAMETER (N=6, NCODA=2, LDA=NCODA+1, LDEVEC=N)
!
INTEGER NOUT
REAL A(LDA,N), EVAL(N), EVEC(LDEVEC,N), PI
!
! Define values of A:
! A = (5 -4 1
! -4 6 -4 1
! 1 -4 6 -4 1
! 1 -4 6 -4 1
! 1 -4 6 -4
! 1 -4 5)
```

```

! Represented in band symmetric
! form this is:
! A = (0 0 1 1 1 1)
! (0 -4 -4 -4 -4 -4)
! (5 6 6 6 6 5)
!
DATA A/0.0, 0.0, 5.0, 0.0, -4.0, 6.0, 1.0, -4.0, 6.0, 1.0, -4.0, &
 6.0, 1.0, -4.0, 6.0, 1.0, -4.0, 5.0/
!
! Find eigenvalues and vectors
CALL EVCSB (A, NCODA, EVAL, EVEC)
! Compute performance index
PI = EPISB(N,A,NCODA,EVAL,EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
CALL WRRRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3 4 5 6
14.45 10.54 5.98 2.42 0.57 0.04

 EVEC
 1 2 3 4 5 6
1 -0.2319 -0.4179 -0.5211 0.5211 -0.4179 0.2319
2 0.4179 0.5211 0.2319 0.2319 -0.5211 0.4179
3 -0.5211 -0.2319 0.4179 -0.4179 -0.2319 0.5211
4 0.5211 -0.2319 -0.4179 -0.4179 0.2319 0.5211
5 -0.4179 0.5211 -0.2319 0.2319 0.5211 0.4179
6 0.2319 -0.4179 0.5211 0.5211 0.4179 0.2319

```

Performance index = 0.029

---

## EVASB

Computes the largest or smallest eigenvalues of a real symmetric matrix in band symmetric storage mode.

### Required Arguments

*NEVAL* — Number of eigenvalues to be computed. (Input)

*A* — Band symmetric matrix of order N. (Input)

*NCODA* — Number of codiagonals in A. (Input)

*SMALL* — Logical variable. (Input)

If *.TRUE.*, the smallest *NEVAL* eigenvalues are computed. If *.FALSE.*, the largest *NEVAL* eigenvalues are computed.

*EVAL* — Vector of length *NEVAL* containing the computed eigenvalues in decreasing order of magnitude. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

### FORTRAN 90 Interface

Generic:     `CALL EVASB (NEVAL, A, NCODA, SMALL, EVAL [, ...])`

Specific:    The specific interface names are `S_EVASB` and `D_EVASB`.

### FORTRAN 77 Interface

Single:      `CALL EVASB (N, NEVAL, A, LDA, NCODA, SMALL, EVAL)`

Double:     The double precision name is `DEVASB`.

### Description

Routine `EVASB` computes the largest or smallest eigenvalues of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. The rational QR algorithm with Newton corrections is used to compute the extreme eigenvalues of this tridiagonal matrix.

The reduction routine is based on the EISPACK routine `BANDR`; see Garbow et al. (1978). The QR routine is based on the EISPACK routine `RATQR`; see Smith et al. (1976).

### Comments

1. Workspace may be explicitly provided, if desired, by use of `E3ASB/DE3ASB`. The reference is:

```
CALL E3ASB (N, NEVAL, A, LDA, NCODA, SMALL, EVAL, ACOPY,
 WK)
```

The additional arguments are as follows:

*ACOPY* — Work array of length  $N(NCODA + 1)$ . *A* and *ACOPY* may be the same, in which case the first  $N(NCODA + 1)$  elements of *A* will be destroyed.



```

A(4,N) = 5.0
A(3,2) = 2.0
A(3,N) = 2.0
!
! Find the 4 smallest eigenvalues
SMALL = .TRUE.
CALL EVASB (NEVAL, A, NCODA, SMALL, EVAL)
!
! Print results
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
END

```

## Output

|  | EVAL  |       |       |       |
|--|-------|-------|-------|-------|
|  | 1     | 2     | 3     | 4     |
|  | 4.000 | 3.172 | 1.804 | 0.522 |

---

## EVESB

Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix in band symmetric storage mode.

### Required Arguments

*NEVEC* — Number of eigenvectors to be calculated. (Input)

*A* — Band symmetric matrix of order *N*. (Input)

*NCODA* — Number of codiagonals in *A*. (Input)

*SMALL* — Logical variable. (Input)

If *.TRUE.*, the smallest *NEVEC* eigenvectors are computed. If *.FALSE.*, the largest *NEVEC* eigenvectors are computed.

*EVAL* — Vector of length *NEVEC* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Real matrix of dimension *N* by *NEVEC*. (Output)

The *J*-th eigenvector, corresponding to *EVAL(J)*, is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

**LDEVEC** — Leading dimension of **EVEC** exactly as specified in the dimension statement in the calling program. (Input)  
 Default: `LDEVEC = SIZE (EVEC,1)`.

### FORTRAN 90 Interface

Generic: `CALL EVESB (NEVEC, A, NCODA, SMALL, EVAL, EVEC [, ...])`  
 Specific: The specific interface names are `S_EVESB` and `D_EVESB`.

### FORTRAN 77 Interface

Single: `CALL EVESB (N, NEVEC, A, LDA, NCODA, SMALL, EVAL, EVEC, LDEVEC)`  
 Double: The double precision name is `DEVESB`.

### Description

Routine `EVESB` computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. The rational QR algorithm with Newton corrections is used to compute the extreme eigenvalues of this tridiagonal matrix. Inverse iteration and orthogonalization are used to compute the eigenvectors of the given band matrix. The reduction routine is based on the EISPACK routine `BANDR`; see Garbow et al. (1977). The QR routine is based on the EISPACK routine `RATQR`; see Smith et al. (1976). The inverse iteration and orthogonalization steps are based on EISPACK routine `BANDV` using the additional steps given in Hanson et al. (1990).

### Comments

1. Workspace may be explicitly provided, if desired, by use of `E4ESB/DE4ESB`. The reference is:

```
CALL E4ESB (N, NEVEC, A, LDA, NCODA, SMALL, EVAL, EVEC,
 LDEVEC, ACOPY, WK, IWK)
```

The additional argument is:

**ACOPY** — Work array of length  $N(NCODA + 1)$ .

**WK** — Work array of length  $N(2NCODA + 5)$ .

**IWK** — Integer work array of length  $N$ .

2. Informational errors

| Type | Code | Description                                                                                  |
|------|------|----------------------------------------------------------------------------------------------|
| 3    | 1    | Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue. |

3     2           The eigenvectors have lost orthogonality.

3.    The success of this routine can be checked using EPISB.

### Example

The following example is given in Gregory and Karney (1969, page 75). The largest three eigenvalues and the corresponding eigenvectors of the matrix are computed and printed.

```
USE EVESB_INT
USE EPISB_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDEVEC, N, NCODA, NEVEC
PARAMETER (N=6, NCODA=2, NEVEC=3, LDA=NCODA+1, LDEVEC=N)
!
INTEGER NOUT
REAL A(LDA,N), EVAL(NEVEC), EVEC(LDEVEC,NEVEC), PI
LOGICAL SMALL

! Define values of A:
! A = (5 -4 1
! (-4 6 -4 1
! (1 -4 6 -4 1
! (1 -4 6 -4 1
! (1 -4 6 -4 1
! (1 -4 6 -4 5
! Represented in band symmetric
! form this is:
! A = (0 0 1 1 1 1
! (0 -4 -4 -4 -4 -4
! (5 6 6 6 6 5
!
DATA A/0.0, 0.0, 5.0, 0.0, -4.0, 6.0, 1.0, -4.0, 6.0, 1.0, -4.0, &
6.0, 1.0, -4.0, 6.0, 1.0, -4.0, 5.0/
!
! Find the 3 largest eigenvalues
! and their eigenvectors.
SMALL = .FALSE.
CALL EVESB (NEVEC, A, NCODA, SMALL, EVAL, EVEC)
! Compute performance index
PI = EPISB(NEVEC,A,NCODA,EVAL,EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, NEVEC, 1)
CALL WRRRN ('EVEC', EVEC)
WRITE (NOUT,'(/,A,F6.3)') ' Performance index = ', PI
END
```

### Output

EVAL

|  |       |       |      |
|--|-------|-------|------|
|  | 1     | 2     | 3    |
|  | 14.45 | 10.54 | 5.98 |

|   |         |         |         |
|---|---------|---------|---------|
|   | EVEC    |         |         |
|   | 1       | 2       | 3       |
| 1 | 0.2319  | -0.4179 | 0.5211  |
| 2 | -0.4179 | 0.5211  | -0.2319 |
| 3 | 0.5211  | -0.2319 | -0.4179 |
| 4 | -0.5211 | -0.2319 | 0.4179  |
| 5 | 0.4179  | 0.5211  | 0.2319  |
| 6 | -0.2319 | -0.4179 | -0.5211 |

Performance index = 0.175

---

## EVBSB

Computes the eigenvalues in a given interval of a real symmetric matrix stored in band symmetric storage mode.

### Required Arguments

*MXEVAL* — Maximum number of eigenvalues to be computed. (Input)

*A* — Band symmetric matrix of order *N*. (Input)

*NCODA* — Number of codiagonals in *A*. (Input)

*ELOW* — Lower limit of the interval in which the eigenvalues are sought. (Input)

*EHIGH* — Upper limit of the interval in which the eigenvalues are sought. (Input)

*NEVAL* — Number of eigenvalues found. (Output)

*EVAL* — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are set.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

### FORTRAN 90 Interface

Generic: CALL EVBSB (MXEVAL, A, NCODA, ELOW, EHIGH, NEVAL, EVAL [, ...])

Specific: The specific interface names are S\_EVBSB and D\_EVBSB.

## FORTRAN 77 Interface

Single:      CALL EVBSB (N, MXEVAL, A, LDA, NCODA, ELOW, EHIGH, NEVAL, EVAL)

Double:      The double precision name is DEVBSB.

## Description

Routine EVBSB computes the eigenvalues in a given range of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. A bisection algorithm is used to compute the eigenvalues of the tridiagonal matrix in a given range.

The reduction routine is based on the EISPACK routine BANDR; see Garbow et al. (1977). The bisection routine is based on the EISPACK routine BISECT; see Smith et al. (1976).

## Comments

1.      Workspace may be explicitly provided, if desired, by use of E3BSB/DE3BSB. The reference is:

```
CALL E3BSB (N, MXEVAL, A, LDA, NCODA, ELOW, EHIGH, NEVAL, EVAL,
ACOPY, WK)
```

The additional arguments are as follows:

**ACOPY** — Work matrix of size  $NCODA + 1$  by  $N$ . **A** and **ACOPY** may be the same, in which case the first  $N(NCODA + 1)$  elements of **A** will be destroyed.

**WK** — Work array of length  $5N$ .

2.      Informational error

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                                                |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The number of eigenvalues in the specified interval exceeds MXEVAL. NEVAL contains the number of eigenvalues in the interval. No eigenvalues will be returned. |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Example

In this example, a DATA statement is used to set **A** to a matrix given by Gregory and Karney (1969, page 77). The eigenvalues in the range (-2.5, 1.5) are computed and printed. As a test, this example uses  $MXEVAL = 5$ . The routine EVBSB computes NEVAL, the number of eigenvalues in the given range, has the value 3.

```
USE EVBSB_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
```

```

!
! Declare variables
INTEGER LDA, MXEVAL, N, NCODA
PARAMETER (MXEVAL=5, N=5, NCODA=2, LDA=NCODA+1)
!
INTEGER NEVAL, NOUT
REAL A(LDA,N), EHIGH, ELOW, EVAL(MXEVAL)
!
! Define values of A:
! A = (-1 2 1)
! (2 0 2 1)
! (1 2 0 2 1)
! (1 2 0 2)
! (1 2 -1)
! Represented in band symmetric
! form this is:
! A = (0 0 1 1 1)
! (0 2 2 2 2)
! (-1 0 0 0 -1)
!
DATA A/0.0, 0.0, -1.0, 0.0, 2.0, 0.0, 1.0, 2.0, 0.0, 1.0, 2.0, &
 0.0, 1.0, 2.0, -1.0/
!
ELOW = -2.5
EHIGH = 1.5
CALL EVBSB (MXEVAL, A, NCODA, ELOW, EHIGH, NEVAL, EVAL)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, '(/,A,I1)') ' NEVAL = ', NEVAL
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
END

```

## Output

```

NEVAL = 3

 EVAL
 1 2 3
-2.464 -2.000 1.000

```

---

## EVFSB

Computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix stored in band symmetric storage mode.

### Required Arguments

*MXEVAL* — Maximum number of eigenvalues to be computed. (Input)

*A* — Band symmetric matrix of order N. (Input)

*NCODA* — Number of codiagonals in A. (Input)

*ELOW* — Lower limit of the interval in which the eigenvalues are sought. (Input)

*EHIGH* — Upper limit of the interval in which the eigenvalues are sought. (Input)

*NEVAL* — Number of eigenvalues found. (Output)

*EVAL* — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are significant.

*EVEC* — Real matrix containing in its first *NEVAL* columns the eigenvectors associated with the eigenvalues found and stored in *EVAL*. Eigenvector *J* corresponds to eigenvalue *J* for *J* = 1 to *NEVAL*. Each vector is normalized to have Euclidean length equal to the value one. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:     CALL *EVFSB* (*MXEVAL*, *A*, *NCODA*, *ELOW*, *EHIGH*, *NEVAL*, *EVAL*, *EVEC* [, ...])

Specific:    The specific interface names are *S\_EVFSB* and *D\_EVFSB*.

### FORTRAN 77 Interface

Single:     CALL *EVFSB* (*N*, *MXEVAL*, *A*, *LDA*, *NCODA*, *ELOW*, *EHIGH*, *NEVAL*, *EVAL*, *EVEC*,  
              *LDEVEC*)

Double:     The double precision name is *DEVFSB*.

### Description

Routine *EVFSB* computes the eigenvalues in a given range and the corresponding eigenvectors of a real band symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent tridiagonal matrix. A bisection algorithm is used to compute the eigenvalues of the tridiagonal matrix in the required range. Inverse iteration and orthogonalization are used to compute the eigenvectors of the given band symmetric matrix.

The reduction routine is based on the EISPACK routine *BANDR*; see Garbow et al. (1977). The bisection routine is based on the EISPACK routine *BISECT*; see Smith et al. (1976). The inverse

iteration and orthogonalization steps are based on the EISPACK routine `BANDV` using remarks from Hanson et al. (1990).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E3FSB/DE3FSB`. The reference is:

```
CALL E3FSB (N, MXEVAL, A, LDA, NCODA, ELOW, EHIGH, NEVAL,
 EVAL, EVEC, LDEVEC, ACOPY, WK1, WK2, IWK)
```

The additional arguments are as follows:

**ACOPY** — Work matrix of size  $NCODA + 1$  by  $N$ .

**WK1** — Work array of length  $6N$ .

**WK2** — Work array of length  $2N * NCODA + N$

**IWK** — Integer work array of length  $N$ .

2. Informational errors

| Type | Code |                                                                                                                                                                                           |
|------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | The number of eigenvalues in the specified interval exceeds <code>MXEVAL</code> . <code>NEVAL</code> contains the number of eigenvalues in the interval. No eigenvalues will be returned. |
| 3    | 2    | Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue.                                                                                              |
| 3    | 3    | The eigenvectors have lost orthogonality.                                                                                                                                                 |

## Example

In this example, a `DATA` statement is used to set `A` to a matrix given by Gregory and Karney (1969, page 75). The eigenvalues in the range  $[1, 6]$  and their corresponding eigenvectors are computed and printed. As a test, this example uses `MXEVAL = 4`. The routine `EVFSB` computes `NEVAL`, the number of eigenvalues in the given range has the value 2. As a check on the computations, the performance index is also computed and printed. For more details, see IMSL routine [EPISB](#).

```
USE EVFSB_INT
USE EPISB_INT
USE WRRRN_INT
USE UMACH_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, LDEVEC, MXEVAL, N, NCODA
PARAMETER (MXEVAL=4, N=6, NCODA=2, LDA=NCODA+1, LDEVEC=N)

!
INTEGER NEVAL, NOUT
REAL A(LDA,N), EHIGH, ELOW, EVAL(MXEVAL), &
```

```

 EVEC(LDEVEC,MXEVAL), PI
!
! Define values of A:
! A = (5 -4 1)
! (-4 6 -4 1)
! (1 -4 6 -4 1)
! (1 -4 6 -4 1)
! (1 -4 6 -4)
! (1 -4 5)
!
! Represented in band symmetric
! form this is:
! A = (0 0 1 1 1 1)
! (0 -4 -4 -4 -4 -4)
! (5 6 6 6 6 5)
DATA A/0.0, 0.0, 5.0, 0.0, -4.0, 6.0, 1.0, -4.0, 6.0, 1.0, -4.0, &
 6.0, 1.0, -4.0, 6.0, 1.0, -4.0, 5.0/
!
! Find eigenvalues and vectors
ELOW = 1.0
EHIGH = 6.0
CALL EVFSB (MXEVAL, A, NCODA, ELOW, EHIGH, NEVAL, EVAL, EVEC)
! Compute performance index
PI = EPISB (NEVAL,A,NCODA,EVAL,EVEC)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,'(/,A,I1)') ' NEVAL = ', NEVAL
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
CALL WRRRN ('EVEC', EVEC, N, NEVAL, LDEVEC)
WRITE (NOUT,'(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

NEVAL = 2

```

 EVAL
 1 2
5.978 2.418

```

```

 EVEC
 1 2
1 0.5211 0.5211
2 -0.2319 0.2319
3 -0.4179 -0.4179
4 0.4179 -0.4179
5 0.2319 0.2319
6 -0.5211 0.5211

```

Performance index = 0.083

---

## EPISB

This function computes the performance index for a real symmetric eigensystem in band symmetric storage mode.

## Required Arguments

*EPISB* — Performance index. (Output)

## Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs on which the performance is based. (Input)

*A* — Band symmetric matrix of order *N*. (Input)

*NCODA* — Number of codiagonals in *A*. (Input)

*EVAL* — Vector of length *NEVAL* containing eigenvalues of *A*. (Input)

*EVEC* — *N* by *NEVAL* array containing eigenvectors of *A*. (Input)  
The eigenvector corresponding to the eigenvalue *EVAL*(*J*) must be in the *J*-th column of *EVEC*.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic: *EPISB* (*NEVAL*, *A*, *NCODA*, *EVAL*, *EVEC* [, ...])

Specific: The specific interface names are *S\_EPISB* and *D\_EPISB*.

## FORTRAN 77 Interface

Single: *EPISB* (*N*, *NEVAL*, *A*, *LDA*, *NCODA*, *EVAL*, *EVEC*, *LDEVEC*)

Double: The double precision function name is *DEPISB*.

## Description

Let  $M = NEVAL$ ,  $\lambda = EVAL$ ,  $x_j = EVEC(*,J)$ , the *j*-th column of *EVEC*. Also, let  $\epsilon$  be the machine precision, given by *AMACH*(4), see the [Reference](#) chapter of the manual. The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\epsilon \|A\|_1 \|x_j\|_1}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of [EVCSF](#) is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

### Comments

1. Workspace may be explicitly provided, if desired, by use of `E2ISB/DE2ISB`. The reference is:

`E2ISB (N, NEVAL, A, LDA, NCODA, EVAL, EVEC, LDEVEC, WK)`

The additional argument is:

**WK** — Work array of length `N`.

2. Informational errors

Type Code

|   |   |                                        |
|---|---|----------------------------------------|
| 3 | 1 | Performance index is greater than 100. |
| 3 | 2 | An eigenvector is zero.                |
| 3 | 3 | The matrix is zero.                    |

### Example

For an example of `EPISB`, see IMSL routine [EVCSB](#).

## EVLHF



Computes all of the eigenvalues of a complex Hermitian matrix.

### Required Arguments

- A** — Complex Hermitian matrix of order `N`. (Input)  
Only the upper triangle is used.

*EVAL* — Real vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

### FORTRAN 90 Interface

Generic: CALL EVLHF (*A*, *EVAL* [, ...])

Specific: The specific interface names are *S\_EVLHF* and *D\_EVLHF*.

### FORTRAN 77 Interface

Single: CALL EVLHF (*N*, *A*, *LDA*, *EVAL*)

Double: The double precision name is *DEVLFH*.

### Description

Routine *EVLHF* computes the eigenvalues of a complex Hermitian matrix. Unitary similarity transformations are used to reduce the matrix to an equivalent real symmetric tridiagonal matrix. The implicit QL algorithm is used to compute the eigenvalues of this tridiagonal matrix.

The underlying code is based on either *EISPACK* or *LAPACK* code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

### Comments

1. Workspace may be explicitly provided, if desired, by use of *E3LHF/DE3LHF*. The reference is:

```
CALL E3LHF (N, A, LDA, EVAL, ACOPY, RWK, CWK, IWK)
```

The additional arguments are as follows:

*ACOPY* — Complex work array of length  $N^2$ . *A* and *ACOPY* may be the same in which case *A* will be destroyed.

*RWK* — Work array of length *N*.

*CWK* — Complex work array of length  $2N$ .

**IWK** — Integer work array of length N.

2. Informational errors

Type Code

- |   |   |                                                                                   |
|---|---|-----------------------------------------------------------------------------------|
| 3 | 1 | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4 | 1 | The iteration for an eigenvalue failed to converge.                               |
| 4 | 2 | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

3. [Integer Options](#) with Chapter 11 Options Manager

- 1** This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `E3LHF`, the internal or working leading dimensions of `ACOPY` and `ECOPY` are both increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `EVLHF`. Additional memory allocation and option value restoration are automatically done in `EVLHF`. There is no requirement that users change existing applications that use `EVLHF` or `E3LHF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5 – 8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `EVLHF`.

## Example

In this example, a `DATA` statement is used to set `A` to a matrix given by Gregory and Karney (1969, page 114). The eigenvalues of this complex Hermitian matrix are computed and printed.

```
USE EVLHF_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (N=2, LDA=N)
!
REAL EVAL(N)
COMPLEX A(LDA,N)
!
! Set values of A
!
! A = (1 -i)
! (i 1)
!
DATA A/(1.0,0.0), (0.0,1.0), (0.0,-1.0), (1.0,0.0)/
!
! Find eigenvalues of A
CALL EVLHF (A, EVAL)
!
! Print results
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
```

END

## Output

```
 EVAL
 1 2
2.000 0.000
```

---

## EVCHF



Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.

### Required Arguments

*A* — Complex Hermitian matrix of order *N*. (Input)  
Only the upper triangle is used.

*EVAL* — Real vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Complex matrix of order *N*. (Output)  
The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic:    CALL EVCHF (*A*, *EVAL*, *EVEC* [, ...])

Specific:   The specific interface names are *S\_EVCHF* and *D\_EVCHF*.

## FORTRAN 77 Interface

Single:      CALL EVCHF (N, A, LDA, EVAL, EVEC, LDEVEC)

Double:     The double precision name is DEVCHF.

## Description

Routine `EVCHF` computes the eigenvalues and eigenvectors of a complex Hermitian matrix. Unitary similarity transformations are used to reduce the matrix to an equivalent real symmetric tridiagonal matrix. The implicit QL algorithm is used to compute the eigenvalues and eigenvectors of this tridiagonal matrix. These eigenvectors and the transformations used to reduce the matrix to tridiagonal form are combined to obtain the eigenvectors for the user's problem. The underlying code is based on either `EISPACK` or `LAPACK` code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.   Workspace may be explicitly provided, if desired, by use of `E5CHF/DE5CHF`. The reference is:

```
CALL E5CHF (N, A, LDA, EVAL, EVEC, LDEVEC, ACOPY, RWK,
 CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . `A` and `ACOPY` may be the same, in which case `A` will be destroyed.

**RWK** — Work array of length  $N^2 + N$ .

**CWK** — Complex work array of length  $2N$ .

**IWK** — Integer work array of length  $N$ .

2.   Informational error

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                   |
|---|---|-----------------------------------------------------------------------------------|
| 3 | 1 | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
|---|---|-----------------------------------------------------------------------------------|

|   |   |                                                     |
|---|---|-----------------------------------------------------|
| 4 | 1 | The iteration for an eigenvalue failed to converge. |
|---|---|-----------------------------------------------------|

|   |   |                                                                              |
|---|---|------------------------------------------------------------------------------|
| 4 | 2 | The matrix is not Hermitian. It has a diagonal entry with an imaginary part. |
|---|---|------------------------------------------------------------------------------|

3.   The success of this routine can be checked using [EPIHF](#).
4.   [Integer Options](#) with Chapter 11 Options Manager

- 1 This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `E5CHF`, the internal or working leading dimensions of `ACOPY` and `ECOPY` are both increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `EVCHF`. Additional memory allocation and option value restoration are automatically done in `EVCHF`. There is no requirement that users change existing applications that use `EVCHF` or `E5CHF`. Default values for the option are `IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5–8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `EVCHF`.

## Example

In this example, a `DATA` statement is used to set `A` to a complex Hermitian matrix. The eigenvalues and eigenvectors of this matrix are computed and printed. The performance index is also computed and printed. This serves as a check on the computations, for more details, see routine `EPIHF`.

```

USE IMSL_libraries

IMPLICIT NONE

! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=3, LDA=N, LDEVEC=N)

!
INTEGER NOUT
REAL EVAL(N), PI
COMPLEX A(LDA,N), EVEC(LDEVEC,N)

! Set values of A
!
! A = ((1, 0) (1,-7i) (0,- i))
! ((1,7i) (5, 0) (10,-3i))
! ((0, i) (10, 3i) (-2, 0))
!
DATA A/(1.0,0.0), (1.0,7.0), (0.0,1.0), (1.0,-7.0), (5.0,0.0), &
 (10.0, 3.0), (0.0,-1.0), (10.0,-3.0), (-2.0,0.0)/

!
! Find eigenvalues and vectors of A
CALL EVCHF (A, EVAL, EVEC)

! Compute performance index
PI = EPIHF(N,A,EVAL,EVEC)

! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3
15.38 -10.63 -0.75

```

```

 EVEC
 1 2 3
1 (0.0631,-0.4075) (-0.0598,-0.3117) (0.8539, 0.0000)
2 (0.7703, 0.0000) (-0.5939, 0.1841) (-0.0313,-0.1380)
3 (0.4668, 0.1366) (0.7160, 0.0000) (0.0808,-0.4942)

Performance index = 0.093

```

---

## EVAHF

Computes the largest or smallest eigenvalues of a complex Hermitian matrix.

### Required Arguments

*NEVAL* — Number of eigenvalues to be calculated. (Input)

*A* — Complex Hermitian matrix of order *N*. (Input)  
Only the upper triangle is used.

*SMALL* — Logical variable. (Input)  
If *.TRUE.*, the smallest *NEVAL* eigenvalues are computed. If *.FALSE.*, the largest *NEVAL* eigenvalues are computed.

*EVAL* — Real vector of length *N* containing the extreme eigenvalues of *A* in decreasing order of magnitude in the first *NEVAL* elements. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

### FORTRAN 90 Interface

Generic:     CALL EVAHF (NEVAL, A, SMALL, EVAL [, ...])

Specific:    The specific interface names are *S\_EVAHF* and *D\_EVAHF*.

### FORTRAN 77 Interface

Single:       CALL EVAHF (N, NEVAL, A, LDA, SMALL, EVAL)

Double:      The double precision name is *DEVAHF*.

## Description

Routine `EVAHF` computes the largest or smallest eigenvalues of a complex Hermitian matrix. Unitary transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. The rational QR algorithm with Newton corrections is used to compute the extreme eigenvalues of this tridiagonal matrix.

The reduction routine is based on the EISPACK routine `HTRIDI`. The QR routine is based on the EISPACK routine `RATQR`. See Smith et al. (1976) for the EISPACK routines.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E3AHF/DE3AHF`. The reference is

```
CALL E3AHF (N, NEVAL, A, LDA, SMALL, EVAL, ACOPY, RWK,
 CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . `A` and `ACOPY` may be the same in which case `A` will be destroyed.

**RWK** — Work array of length  $2N$ .

**CWK** — Complex work array of length  $2N$ .

**IWK** — Work array of length  $N$ .

2. Informational errors

| Type | Code |                                                                                         |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 1    | The iteration for an eigenvalue failed to converge. The best estimate will be returned. |
| 3    | 2    | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part.       |
| 4    | 2    | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.            |

## Example

In this example, a `DATA` statement is used to set `A` to a matrix given by Gregory and Karney (1969, page 114). Its largest eigenvalue is computed and printed.

```
USE EVAHF_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, N
PARAMETER (N=2, LDA=N)
```

```

!
INTEGER NEVAL
REAL EVAL(N)
COMPLEX A(LDA,N)
LOGICAL SMALL

! Set values of A
!
! A = (1 -i)
! (i 1)
!
DATA A/(1.0,0.0), (0.0,1.0), (0.0,-1.0), (1.0,0.0)/

!
! Find the largest eigenvalue of A
NEVAL = 1
SMALL = .FALSE.
CALL EVAHF (NEVAL, A, SMALL, EVAL)
! Print results
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
END

```

## Output

```

EVAL
2.000

```

---

## EVEHF

Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.

### Required Arguments

*NEVEC* — Number of eigenvectors to be computed. (Input)

*A* — Complex Hermitian matrix of order *N*. (Input)  
Only the upper triangle is used.

*SMALL* — Logical variable. (Input)  
If *.TRUE.*, the smallest *NEVEC* eigenvectors are computed. If *.FALSE.*, the largest *NEVEC* eigenvectors are computed.

*EVAL* — Real vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Complex matrix of dimension *N* by *NEVEC*. (Output)  
The *J*-th eigenvector corresponding to *EVAL(J)*, is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDEVEC = \text{SIZE}(EVEC,1)$ .

## FORTRAN 90 Interface

Generic:     CALL EVEHF (NEVEC, A, SMALL, EVAL, EVEC [, ...])

Specific:    The specific interface names are *S\_EVEHF* and *D\_EVEHF*.

## FORTRAN 77 Interface

Single:     CALL EVEHF (N, NEVEC, A, LDA, SMALL, EVAL, EVEC, LDEVEC)

Double:     The double precision name is *DEVEHF*.

## Description

Routine *EVEHF* computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix. Unitary transformations are used to reduce the matrix to an equivalent real symmetric tridiagonal matrix. The rational QR algorithm with Newton corrections is used to compute the extreme eigenvalues of the tridiagonal matrix. Inverse iteration is used to compute the eigenvectors of the tridiagonal matrix. Eigenvectors of the original matrix are found by back transforming the eigenvectors of the tridiagonal matrix.

The reduction routine is based on the EISPACK routine *HTRIDI*. The QR routine used is based on the EISPACK routine *RATQR*. The inverse iteration routine is based on the EISPACK routine *TINVT*. The back transformation routine is based on the EISPACK routine *HTRIBK*. See Smith et al. (1976) for the EISPACK routines.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of *E3EHF/DE3EHF*. The reference is:

```
CALL E3EHF (N, NEVEC, A, LDA, SMALL, EVAL, EVEC, LDEVEC,
 ACOPY, RW1, RW2, CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . **A** and **ACOPY** may be the same, in which case **A** will be destroyed.

**RW1** — Work array of length  $N * NEVEC$ . Used to store the real eigenvectors of a symmetric tridiagonal matrix.

**RW2** — Work array of length  $8N$ .

**CWK** — Complex work array of length  $2N$ .

**IWK** — Work array of length  $N$ .

2. Informational errors

| Type | Code |                                                                                         |
|------|------|-----------------------------------------------------------------------------------------|
| 3    | 1    | The iteration for an eigenvalue failed to converge. The best estimate will be returned. |
| 3    | 2    | The iteration for an eigenvector failed to converge. The eigenvector will be set to 0.  |
| 3    | 3    | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part.       |
| 4    | 2    | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.            |

3. The success of this routine can be checked using [EPIHF](#).

## Example

In this example, a `DATA` statement is used to set **A** to a matrix given by Gregory and Karney (1969, page 115). The smallest eigenvalue and its corresponding eigenvector is computed and printed. The performance index is also computed and printed. This serves as a check on the computations. For more details, see IMSL routine [EPIHF](#).

```
USE IMSL_LIBRARIES

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDEVEC, N, NEVEC
PARAMETER (N=3, NEVEC=1, LDA=N, LDEVEC=N)
!
INTEGER NOUT
REAL EVAL(N), PI
COMPLEX A(LDA,N), EVEC(LDEVEC,NEVEC)
LOGICAL SMALL
! Set values of A
!
! A = (2 -i 0)
! (i 2 0)
! (0 0 3)
```

```

!
DATA A/(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)/
!
! Find smallest eigenvalue and its
! eigenvectors
SMALL = .TRUE.
CALL EVEHF (NEVEC, A, SMALL, EVAL, EVEC)
! Compute performance index
PI = EPIHF(NEVEC,A,EVAL,EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL, 1, NEVEC, 1)
CALL WRNCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

EVAL
1.000

 EVEC
1 (0.0000, 0.7071)
2 (0.7071, 0.0000)
3 (0.0000, 0.0000)

Performance index = 0.031

```

---

## EVBHF

Computes the eigenvalues in a given range of a complex Hermitian matrix.

### Required Arguments

***MXEVAL*** — Maximum number of eigenvalues to be computed. (Input)

***A*** — Complex Hermitian matrix of order *N*. (Input)  
Only the upper triangle is used.

***ELOW*** — Lower limit of the interval in which the eigenvalues are sought. (Input)

***EHIGH*** — Upper limit of the interval in which the eigenvalues are sought. (Input)

***NEVAL*** — Number of eigenvalues found. (Output)

***EVAL*** — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are significant.

## Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default:  $LDA = \text{SIZE}(A,1)$ .

## FORTRAN 90 Interface

Generic: `CALL EVBHF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL [, ...])`

Specific: The specific interface names are `S_EVBHF` and `D_EVBHF`.

## FORTRAN 77 Interface

Single: `CALL EVBHF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL)`

Double: The double precision name is `DEVBHF`.

## Description

Routine `EVBHF` computes the eigenvalues in a given range of a complex Hermitian matrix. Unitary transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. A bisection algorithm is used to compute the eigenvalues in the given range of this tridiagonal matrix.

The reduction routine is based on the EISPACK routine `HTRIDI`. The bisection routine used is based on the EISPACK routine `BISECT`. See Smith et al. (1976) for the EISPACK routines.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E3BHF/DE3BHF`. The reference is:

```
CALL E3BHF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL,
 ACOPY, RWK, CWK, IWK)
```

The additional arguments are as follows:

*ACOPY* — Complex work matrix of size *N* by *N*. *A* and *ACOPY* may be the same, in which case the first  $N^2$  elements of *A* will be destroyed.

*RWK* — Work array of length  $5N$ .

*CWK* — Complex work array of length  $2N$ .

*IWK* — Work array of length *MXEVAL*.

## 2. Informational errors

| Type | Code |                                                                                                                                                          |
|------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | The number of eigenvalues in the specified range exceeds MXEVAL. NEVAL contains the number of eigenvalues in the range. No eigenvalues will be computed. |
| 3    | 2    | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part.                                                                        |
| 4    | 2    | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.                                                                             |

### Example

In this example, a DATA statement is used to set A to a matrix given by Gregory and Karney (1969, page 114). The eigenvalues in the range [1.5, 2.5] are computed and printed. This example allows a maximum number of eigenvalues MXEVAL = 2. The routine computes that there is one eigenvalue in the given range. This value is returned in NEVAL.

```
USE EVBHF_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, MXEVAL, N
PARAMETER (MXEVAL=2, N=2, LDA=N)
!
INTEGER NEVAL, NOUT
REAL EHIGH, ELOW, EVAL(MXEVAL)
COMPLEX A(LDA,N)
!
! Set values of A
!
! A = (1 -i)
! (i 1)
!
DATA A/(1.0,0.0), (0.0,1.0), (0.0,-1.0), (1.0,0.0)/
!
! Find eigenvalue
ELOW = 1.5
EHIGH = 2.5
CALL EVBHF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, '(/,A,I3)') ' NEVAL = ', NEVAL
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
END
```

## Output

NEVAL = 1

EVAL  
2.000

---

## EVFHF

Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.

### Required Arguments

*MXEVAL* — Maximum number of eigenvalues to be computed. (Input)

*A* — Complex Hermitian matrix of order *N*. (Input)  
Only the upper triangle is used.

*ELOW* — Lower limit of the interval in which the eigenvalues are sought. (Input)

*EHIGH* — Upper limit of the interval in which the eigenvalues are sought. (Input)

*NEVAL* — Number of eigenvalues found. (Output)

*EVAL* — Real vector of length *MXEVAL* containing the eigenvalues of *A* in the interval (*ELOW*, *EHIGH*) in decreasing order of magnitude. (Output)  
Only the first *NEVAL* elements of *EVAL* are significant.

*EVEC* — Complex matrix containing in its first *NEVAL* columns the eigenvectors associated with the eigenvalues found stored in *EVAL*. Each vector is normalized to have Euclidean length equal to the value one. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic:     CALL EVFHF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL, EVEC [, ...])

Specific:    The specific interface names are S\_EVFHF and D\_EVFHF.

## FORTRAN 77 Interface

Single:     CALL EVFHF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL, EVEC,  
                  LDEVEC)

Double:     The double precision name is DEVHFH.

## Description

Routine `EVFHF` computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix. Unitary transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. A bisection algorithm is used to compute the eigenvalues in the given range of this tridiagonal matrix. Inverse iteration is used to compute the eigenvectors of the tridiagonal matrix. The eigenvectors of the original matrix are computed by back transforming the eigenvectors of the tridiagonal matrix.

The reduction routine is based on the EISPACK routine `HTRIDI`. The bisection routine is based on the EISPACK routine `BISECT`. The inverse iteration routine is based on the EISPACK routine `TINVIT`. The back transformation routine is based on the EISPACK routine `HTRIBK`. See Smith et al. (1976) for the EISPACK routines.

## Comments

1.     Workspace may be explicitly provided, if desired, by use of `E3FHF/DE3FHF`. The reference is:

```
CALL E3FHF (N, MXEVAL, A, LDA, ELOW, EHIGH, NEVAL, EVAL,
 EVEC, LDEVEC, ACOPY, ECOPY, RWK, CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work matrix of size  $N$  by  $N$ . `A` and `ACOPY` may be the same, in which case the first  $N^2$  elements of `A` will be destroyed.

**ECOPY** — Work matrix of size  $N$  by `MXEVAL`. Used to store eigenvectors of a real tridiagonal matrix.

**RWK** — Work array of length  $8N$ .

**CWK** — Complex work array of length  $2N$ .

**IWK** — Work array of length `MXEVAL`.

2.     Informational errors

| Type | Code |                                                                                                                                                          |
|------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | The number of eigenvalues in the specified range exceeds MXEVAL. NEVAL contains the number of eigenvalues in the range. No eigenvalues will be computed. |
| 3    | 2    | The iteration for an eigenvector failed to converge. The eigenvector will be set to 0.                                                                   |
| 3    | 3    | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part.                                                                        |
| 4    | 2    | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.                                                                             |

## Example

In this example, a DATA statement is used to set  $A$  to a complex Hermitian matrix. The eigenvalues in the range  $[-15, 0]$  and their corresponding eigenvectors are computed and printed. As a test, this example uses  $\text{MXEVAL} = 3$ . The routine `EVFHF` computes the number of eigenvalues in the given range. That value,  $\text{NEVAL}$ , is two. As a check on the computations, the performance index is also computed and printed. For more details, see routine [EPIHF](#).

```

USE IMSL_LIBRARIES

IMPLICIT NONE

! Declare variables
INTEGER LDA, LDEVEC, MXEVAL, N
PARAMETER (MXEVAL=3, N=3, LDA=N, LDEVEC=N)

!
INTEGER NEVAL, NOUT
REAL EHIGH, ELOW, EVAL(MXEVAL), PI
COMPLEX A(LDA,N), EVEC(LDEVEC,MXEVAL)

! Set values of A
!
! A = ((1, 0) (1,-7i) (0,- i))
! ((1,7i) (5, 0) (10,-3i))
! ((0, i) (10, 3i) (-2, 0))
!
DATA A/(1.0,0.0), (1.0,7.0), (0.0,1.0), (1.0,-7.0), (5.0,0.0), &
 (10.0,3.0), (0.0,-1.0), (10.0,-3.0), (-2.0,0.0)/

!
! Find eigenvalues and vectors
ELOW = -15.0
EHIGH = 0.0
CALL EVFHF (MXEVAL, A, ELOW, EHIGH, NEVAL, EVAL, EVEC)

! Compute performance index
PI = EPIHF(NEVAL,A,EVAL,EVEC)

! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,'(/,A,I3)') ' NEVAL = ', NEVAL
CALL WRRRN ('EVAL', EVAL, 1, NEVAL, 1)
CALL WRRCRN ('EVEC', EVEC, N, NEVAL, LDEVEC)
WRITE (NOUT,'(/,A,F6.3)') ' Performance index = ', PI

```

END

## Output

NEVAL = 2

```
 EVAL
 1 2
-10.63 -0.75
```

```
 EVEC
 1 2
1 (-0.0598,-0.3117) (0.8539, 0.0000)
2 (-0.5939, 0.1841) (-0.0313,-0.1380)
3 (0.7160, 0.0000) (0.0808,-0.4942)
```

Performance index = 0.057

---

## EPIHF

This function computes the performance index for a complex Hermitian eigensystem.

### Function Return Value

*EPIHF* — Performance index. (Output)

### Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs on which the performance index computation is based. (Input)

*A* — Complex Hermitian matrix of order *N*. (Input)

*EVAL* — Vector of length *NEVAL* containing eigenvalues of *A*. (Input)

*EVEC* — Complex *N* by *NEVAL* array containing eigenvectors of *A*. (Input)  
The eigenvector corresponding to the eigenvalue *EVAL*(*J*) must be in the *J*-th column of *EVEC*.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic: EPIHF (NEVAL, A, EVAL, EVEC [, ...])

Specific: The specific interface names are S\_EPIHF and D\_EPIHF.

## FORTRAN 77 Interface

Single: EPIHF (N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC)

Double: The double precision function name is DEPIHF.

## Description

Let  $M = \text{NEVAL}$ ,  $\hat{\lambda} = \text{EVAL}$ ,  $x_j = \text{EVEC}(*, J)$ , the  $j$ -th column of  $\text{EVEC}$ . Also, let  $\varepsilon$  be the machine precision, given by  $\text{AMACH}(4)$ , see the [Reference](#) chapter of this manual. The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\varepsilon \|A\|_1 \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{|\Re v_i| + |\Im v_i|\}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of [EVCSE](#) is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E2IHF/DE2IHF`. The reference is:

```
E2IHF(N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC, WK)
```

The additional argument is

**WK** — Complex work array of length  $N$ .

2. Informational errors

Type Code

3 1 Performance index is greater than 100.

|   |   |                         |
|---|---|-------------------------|
| 3 | 2 | An eigenvector is zero. |
| 3 | 3 | The matrix is zero.     |

### Example

For an example of `EPIHF`, see IMSL routine `EVCHF`.

---

## EVLRH

Computes all of the eigenvalues of a real upper Hessenberg matrix.

### Required Arguments

*A* — Real upper Hessenberg matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues in decreasing order of magnitude. (Output)

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: `N = SIZE (A,2)`.

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

### FORTRAN 90 Interface

Generic: `CALL EVLRH (A, EVAL [, ...])`

Specific: The specific interface names are `S_EVLRH` and `D_EVLRH`.

### FORTRAN 77 Interface

Single: `CALL EVLRH (N, A, LDA, EVAL)`

Double: The double precision name is `DEVLRH`.

### Description

Routine `EVLRH` computes the eigenvalues of a real upper Hessenberg matrix by using the QR algorithm. The QR Algorithm routine is based on the `EISPACK` routine `HQR`, Smith et al. (1976).

### Comments

1. Workspace may be explicitly provided, if desired, by use of `E3LRH/DE3LRH`. The reference is:

```
CALL E3LRH (N, A, LDA, EVAL, ACOFY, WK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Real  $N$  by  $N$  work matrix.

**WK** — Real vector of length  $3n$ .

**IWK** — Integer vector of length  $n$ .

## 2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

## Example

In this example, a `DATA` statement is used to set `A` to an upper Hessenberg matrix of integers. The eigenvalues of this matrix are computed and printed.

```
USE EVLRH_INT
USE UMACH_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (N=4, LDA=N)
!
INTEGER NOUT
REAL A(LDA,N)
COMPLEX EVAL(N)
!
! Set values of A
!
! A = (2.0 1.0 3.0 4.0)
! (1.0 0.0 0.0 0.0)
! (1.0 0.0 0.0)
! (1.0 0.0 0.0)
!
DATA A/2.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 3.0, 0.0, 0.0, &
1.0, 4.0, 0.0, 0.0, 0.0/
!
! Find eigenvalues of A
CALL EVLRH (A, EVAL)
!
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
END
```

## Output

EVAL

|                 |                 |                  |                 |   |
|-----------------|-----------------|------------------|-----------------|---|
|                 | 1               | 2                | 3               | 4 |
| ( 2.878, 0.000) | ( 0.011, 1.243) | ( 0.011, -1.243) | (-0.900, 0.000) |   |

---

## EVCRH

Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.

### Required Arguments

*A* — Real upper Hessenberg matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues in decreasing order of magnitude. (Output)

*EVEC* — Complex matrix of order *N*. (Output)  
The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = SIZE (*EVEC*,1).

### FORTRAN 90 Interface

Generic: CALL EVCRH (*A*, *EVAL*, *EVEC* [, ...])

Specific: The specific interface names are *S\_EVCRH* and *D\_EVCRH*.

### FORTRAN 77 Interface

Single: CALL EVCRH (*N*, *A*, *LDA*, *EVAL*, *EVEC*, *LDEVEC*)

Double: The double precision name is *DEVCRH*.

## Description

Routine `E6CRH` computes the eigenvalues and eigenvectors of a real upper Hessenberg matrix by using the QR algorithm. The QR algorithm routine is based on the EISPACK routine `HQR2`; see Smith et al. (1976).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `E6CRH/DE6CRH`. The reference is:

```
CALL E6CRH (N, A, LDA, EVAL, EVEC, LDEVEC, ACOPY, ECOPY,
 RWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Real  $N$  by  $N$  work matrix.

**ECOPY** — Real  $N$  by  $N$  work matrix.

**RWK** — Real array of length  $3N$ .

**IWK** — Integer array of length  $N$ .

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

## Example

In this example, a `DATA` statement is used to set `A` to a Hessenberg matrix with integer entries. The values are returned in decreasing order of magnitude. The eigenvalues, eigenvectors and performance index of this matrix are computed and printed. See routine `EPIRG` for details.

```
USE EVCRH_INT
USE EPIRG_INT
USE UMACH_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=4, LDA=N, LDEVEC=N)
!
INTEGER NOUT
REAL A(LDA,N), PI
COMPLEX EVAL(N), EVEC(LDEVEC,N)
!
! Define values of A:
!
! A = (-1.0 -1.0 -1.0 -1.0)
```

```

! (1.0 0.0 0.0 0.0)
! (1.0 0.0 0.0)
! (1.0 0.0)
!
DATA A/-1.0, 1.0, 0.0, 0.0, -1.0, 0.0, 1.0, 0.0, -1.0, 0.0, 0.0, &
 1.0, -1.0, 0.0, 0.0, 0.0/
!
! Find eigenvalues and vectors of A
CALL EVCRH (A, EVAL, EVEC)
!
! Compute performance index
PI = EPIRG(N,A,EVAL,EVEC)
!
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3 4
(-0.8090, 0.5878) (-0.8090,-0.5878) (0.3090, 0.9511) (0.3090,-0.9511)

 EVEC
 1 2 3
4
1 (-0.4045, 0.2939) (-0.4045,-0.2939) (-0.4045,-0.2939) (-0.4045,
0.2939)
2 (0.5000, 0.0000) (0.5000, 0.0000) (-0.4045, 0.2939) (-0.4045,-
0.2939)
3 (-0.4045,-0.2939) (-0.4045, 0.2939) (0.1545, 0.4755) (0.1545,-
0.4755)
4 (0.1545, 0.4755) (0.1545,-0.4755) (0.5000, 0.0000) (0.5000,
0.0000)

Performance index = 0.098

```

---

## EVLCH

Computes all of the eigenvalues of a complex upper Hessenberg matrix.

### Required Arguments

*A* — Complex upper Hessenberg matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

### Required Arguments

*N* — Order of the matrix *A*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

### **FORTRAN 90 Interface**

Generic:     CALL *EVLCH* (*A*, *EVAL* [, ...])

Specific:    The specific interface names are *S\_EVLCH* and *D\_EVLCH*.

### **FORTRAN 77 Interface**

Single:      CALL *EVLCH* (*N*, *A*, *LDA*, *EVAL*)

Double:     The double precision name is *DEVLCH*.

### **Description**

Routine *EVLCH* computes the eigenvalues of a complex upper Hessenberg matrix using the QR algorithm. This routine is based on the *EISPACK* routine *COMQR2*; see Smith et al. (1976).

### **Comments**

1.    Workspace may be explicitly provided, if desired, by use of *E3LCH/DE3LCH*. The reference is:

```
CALL E3LCH (N, A, LDA, EVAL, ACOPY, RWK, IWK)
```

The additional arguments are as follows:

*ACOPY* — Complex *N* by *N* work array. *A* and *ACOPY* may be the same, in which case *A* is destroyed.

*RWK* — Real work array of length *N*.

*IWK* — Integer work array of length *N*.

2.    Informational error

| Type | Code |
|------|------|
|------|------|

|   |   |                                                       |
|---|---|-------------------------------------------------------|
| 4 | 1 | The iteration for the eigenvalues failed to converge. |
|---|---|-------------------------------------------------------|

### **Example**

In this example, a *DATA* statement is used to set the matrix *A*. The program computes and prints the eigenvalues of this matrix.

```
USE EVLCH_INT
```

```

USE WRRCRN_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, N
PARAMETER (N=4, LDA=N)
COMPLEX A(LDA,N), EVAL(N)

! Set values of A
!
! A = (5+9i 5+5i -6-6i -7-7i)
! (3+3i 6+10i -5-5i -6-6i)
! (0 3+3i -1+3i -5-5i)
! (0 0 -3-3i 4i)
!
DATA A / (5.0,9.0), (3.0,3.0), (0.0,0.0), (0.0,0.0), &
(5.0,5.0), (6.0,10.0), (3.0,3.0), (0.0,0.0), &
(-6.0,-6.0), (-5.0,-5.0), (-1.0,3.0), (-3.0,-3.0), &
(-7.0,-7.0), (-6.0,-6.0), (-5.0,-5.0), (0.0,4.0)/

!
! Find the eigenvalues of A
CALL EVLCH (A, EVAL)

! Print results
CALL WRRCRN ('EVAL', EVAL, 1, N, 1)
END

```

## Output

```

 EVAL
 2
1 (8.22, 12.22) (3.40, 7.40) (1.60, 5.60) (-3.22, 0.78)

```

---

## EVCCH

Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.

### Required Arguments

*A* — Complex upper Hessenberg matrix of order *N*. (Input)

*EVAL* — Complex vector of length *N* containing the eigenvalues of *A* in decreasing order of magnitude. (Output)

*EVEC* — Complex matrix of order *N*. (Output)

The *J*-th eigenvector, corresponding to *EVAL*(*J*), is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

### Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = SIZE (*A*,2).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

**LDEVEC** — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDEVEC = SIZE (EVEC,1)`.

### **FORTRAN 90 Interface**

Generic: `CALL EVCCH (A, EVAL, EVEC [, ...])`

Specific: The specific interface names are `S_EVCCH` and `D_EVCCH`.

### **FORTRAN 77 Interface**

Single: `CALL EVCCH (N, A, LDA, EVAL, EVEC, LDEVEC)`

Double: The double precision name is `DEVCCCH`.

### **Description**

Routine `EVCCH` computes the eigenvalues and eigenvectors of a complex upper Hessenberg matrix using the QR algorithm. This routine is based on the EISPACK routine `COMQR2`; see Smith et al. (1976).

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `E4CCH/DE4CCH`. The reference is:

```
CALL E4CCH (N, A, LDA, EVAL, EVEC, LDEVEC, ACOPY, CWORK,
 RWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex *N* by *N* work array. *A* and *ACOPY* may be the same, in which case *A* is destroyed.

**CWORK** — Complex work array of length  $2N$ .

**RWK** — Real work array of length *N*.

**IWK** — Integer work array of length *N*.

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

- The results of `EVCCH` can be checked using `EPICG`. This requires that the matrix `A` explicitly contains the zeros in `A(I, J)` for  $(I - 1) > J$  which are assumed by `EVCCH`.

## Example

In this example, a `DATA` statement is used to set the matrix `A`. The program computes the eigenvalues and eigenvectors of this matrix. The performance index is also computed and printed. This serves as a check on the computations; for more details, see IMSL routine `EPICG`. The zeros in the lower part of the matrix are not referenced by `EVCCH`, but they are required by `EPICG`.

```

USE EVCCH_INT
USE EPICG_INT
USE UMACH_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDEVEC, N
PARAMETER (N=4, LDA=N, LDEVEC=N)
!
INTEGER NOUT
REAL PI
COMPLEX A(LDA,N), EVAL(N), EVEC(LDEVEC,N)
!
! Set values of A
!
! A = (5+9i 5+5i -6-6i -7-7i)
! (3+3i 6+10i -5-5i -6-6i)
! (0 3+3i -1+3i -5-5i)
! (0 0 -3-3i 4i)
!
DATA A/(5.0,9.0), (3.0,3.0), (0.0,0.0), (0.0,0.0), (5.0,5.0), &
 (6.0,10.0), (3.0,3.0), (0.0,0.0), (-6.0,-6.0), (-5.0,-5.0), &
 (-1.0,3.0), (-3.0,-3.0), (-7.0,-7.0), (-6.0,-6.0), &
 (-5.0,-5.0), (0.0,4.0)/
!
! Find eigenvalues and vectors of A
CALL EVCCH (A, EVAL, EVEC)
!
! Compute performance index
PI = EPICG(N,A,EVAL,EVEC)
!
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3 4
(8.22, 12.22) (3.40, 7.40) (1.60, 5.60) (-3.22, 0.78)

 EVEC

```

|   | 1                  | 2                  | 3                 |
|---|--------------------|--------------------|-------------------|
| 4 |                    |                    |                   |
| 1 | ( 0.7167, 0.0000)  | (-0.0704, 0.0000)  | (-0.3678, 0.0000) |
| 2 | ( 0.6402, -0.0000) | (-0.0046, -0.0000) | ( 0.6767, 0.0000) |
| 3 | ( 0.2598, 0.0000)  | ( 0.7477, 0.0000)  | (-0.3005, 0.0000) |
| 4 | (-0.0948, -0.0000) | (-0.6603, -0.0000) | ( 0.5625, 0.0000) |

Performance index = 0.020

## GVLRG

Computes all of the eigenvalues of a generalized real eigensystem  $Az = \lambda Bz$ .

### Required Arguments

**A** — Real matrix of order N. (Input)

**B** — Real matrix of order N. (Input)

**ALPHA** — Complex vector of size N containing scalars  $\alpha_i$ ,  $i = 1, \dots, n$ . If  $\beta_i \neq 0$ ,  $\lambda_i = \alpha_i / \beta_i$  the eigenvalues of the system in decreasing order of magnitude. (Output)

**BETAV** — Vector of size N containing scalars  $\beta_i$ . (Output)

### Optional Arguments

**N** — Order of the matrices A and B. (Input)  
Default:  $N = \text{SIZE}(A, 2)$ .

**LDA** — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A, 1)$ .

**LDB** — Leading dimension of B exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B, 1)$ .

### FORTRAN 90 Interface

Generic:    CALL GVLRG (A, B, ALPHA, BETAV [, ...])

Specific:    The specific interface names are S\_GVLRG and D\_GVLRG.

## FORTRAN 77 Interface

Single:      CALL GVLGR (N, A, LDA, B, LDB, ALPHA, BETAV)

Double:      The double precision name is DGVLGR.

## Description

Routine GVLGR computes the eigenvalues of the generalized eigensystem  $Ax = \lambda Bx$  where  $A$  and  $B$  are real matrices of order  $N$ . The eigenvalues for this problem can be infinite; so instead of returning  $\lambda$ , GVLGR returns  $\alpha$  and  $\beta$ . If  $\beta$  is nonzero, then  $\lambda = \alpha/\beta$ .

The first step of the QZ algorithm is to simultaneously reduce  $A$  to upper Hessenberg form and  $B$  to upper triangular form. Then, orthogonal transformations are used to reduce  $A$  to quasi-upper-triangular form while keeping  $B$  upper triangular. The generalized eigenvalues are then computed.

The underlying code is based on either EISPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation, see “Using ScaLAPACK, LAPACK, LINPACK, and EISPACK” in the Introduction section of this manual.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of G3LRG/DG3LRG. The reference is:

```
CALL G3LRG (N, A, LDA, B, LDB, ALPHA, BETAV, ACOPI, BCOPI,
 RWK, CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Work array of size  $N^2$ . The arrays  $A$  and  $ACOPY$  may be the same, in which case the first  $N^2$  elements of  $A$  will be destroyed.

**BCOPY** — Work array of size  $N^2$ . The arrays  $B$  and  $BCOPY$  may be the same, in which case the first  $N^2$  elements of  $B$  will be destroyed.

**RWK** — Real work array of size  $N$ .

**CWK** — Complex work array of size  $N$ .

**IWK** — Integer work array of size  $N$ .

2.    [Integer Options](#) with Chapter 11 Options Manager

- 1    This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine G3LRG, the internal or working leading dimension of  $ACOPY$  is increased by  $IVAL(3)$  when  $N$  is a multiple of  $IVAL(4)$ . The values  $IVAL(3)$  and  $IVAL(4)$  are temporarily replaced by  $IVAL(1)$  and  $IVAL(2)$ , respectively, in routine GVLGR. Analogous comments hold for  $BCOPY$  and the values  $IVAL(5) - IVAL(8)$ . Additional memory allocation and option

value restoration are automatically done in GVLGR. There is no requirement that users change existing applications that use GVLGR or G3LRG. Default values for the option are IVAL(\*) = 1, 16, 0, 1, 1, 16, 0, 1.

## Example

In this example, DATA statements are used to set *A* and *B*. The eigenvalues are computed and printed.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDA, LDB, N
PARAMETER (N=3, LDA=N, LDB=N)

!
INTEGER I
REAL A(LDA,N), B(LDB,N), BETAV(N)
COMPLEX ALPHA(N), EVAL(N)

!
! Set values of A and B
! A = (1.0 0.5 0.0)
! (-10.0 2.0 0.0)
! (5.0 1.0 0.5)
!
! B = (0.5 0.0 0.0)
! (3.0 3.0 0.0)
! (4.0 0.5 1.0)
!
! Declare variables
DATA A/1.0, -10.0, 5.0, 0.5, 2.0, 1.0, 0.0, 0.0, 0.5/
DATA B/0.5, 3.0, 4.0, 0.0, 3.0, 0.5, 0.0, 0.0, 1.0/

!
CALL GVLGR (A, B, ALPHA, BETAV)

! Compute eigenvalues
DO 10 I=1, N
 EVAL(I) = ALPHA(I)/BETAV(I)
10 CONTINUE

! Print results
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
END

```

## Output

```

 EVAL
 1 2 3
(0.833, 1.993) (0.833,-1.993) (0.500, 0.000)

```

---

# GVCRG



Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem  $Az = \lambda Bz$ .

## Required Arguments

*A* — Real matrix of order *N*. (Input)

*B* — Real matrix of order *N*. (Input)

*ALPHA* — Complex vector of size *N* containing scalars  $\alpha_i$ . If  $\beta_i \neq 0$ ,  $\lambda_i = \alpha_i / \beta_i$ ,  $i = 1, \dots, n$  are the eigenvalues of the system.

*BETAV* — Vector of size *N* containing scalars  $\beta_i$ . (Output)

*EVEC* — Complex matrix of order *N*. (Output)  
The *J*-th eigenvector, corresponding to  $\lambda_j$ , is stored in the *J*-th column. Each vector is normalized to have Euclidean length equal to the value one.

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDB* = SIZE (*B*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = SIZE (*EVEC*,1).

## FORTRAN 90 Interface

Generic:    CALL GVCRG (*A*, *B*, *ALPHA*, *BETAV*, *EVEC* [, ...])

Specific:   The specific interface names are *S\_GVCRG* and *D\_GVCRG*.

## FORTRAN 77 Interface

Single:      CALL GVCRG (N, A, LDA, B, LDB, ALPHA, BETAV, EVEC, LDEVEC)

Double:      The double precision name is DGVCRG.

## Description

Routine GVCRG computes the complex eigenvalues and eigenvectors of the generalized eigensystem  $Ax = \lambda Bx$  where  $A$  and  $B$  are real matrices of order  $N$ . The eigenvalues for this problem can be infinite; so instead of returning  $\lambda$ , GVCRG returns complex numbers  $\alpha$  and real numbers  $\beta$ . If  $\beta$  is nonzero, then  $\lambda = \alpha / \beta$ . For problems with small  $|\beta|$  users can choose to solve the mathematically equivalent problem  $Bx = \mu Ax$  where  $\mu = \lambda^{-1}$ .

The first step of the QZ algorithm is to simultaneously reduce  $A$  to upper Hessenberg form and  $B$  to upper triangular form. Then, orthogonal transformations are used to reduce  $A$  to quasi-upper-triangular form while keeping  $B$  upper triangular. The generalized eigenvalues and eigenvectors for the reduced problem are then computed.

The underlying code is based on either EISPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation, see “[Using ScaLAPACK, LAPACK, LINPACK, and EISPACK](#)” in the Introduction section of this manual.

## Comments

1.      Workspace may be explicitly provided, if desired, by use of G8CRG/DG8CRG. The reference is:

```
CALL G8CRG (N, A, LDA, B, LDB, ALPHA, BETAV, EVEC, LDEVEC,
 ACOPY, BCOPY, ECOPY, RWK, CWK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Work array of size  $N^2$ . The arrays  $A$  and  $ACOPY$  may be the same, in which case the first  $N^2$  elements of  $A$  will be destroyed.

**BCOPY** — Work array of size  $N^2$ . The arrays  $B$  and  $BCOPY$  may be the same, in which case the first  $N^2$  elements of  $B$  will be destroyed.

**ECOPY** — Work array of size  $N^2$ .

**RWK** — Work array of size  $N$ .

**CWK** — Complex work array of size  $N$ .

**IWK** — Integer work array of size  $N$ .

2.      [Integer Options](#) with Chapter 11 Options Manager

- 1 This option uses eight values to solve memory bank conflict (access inefficiency) problems. In routine `G8CRG`, the internal or working leading dimensions of `ACOPY` and `ECOPY` are both increased by `IVAL(3)` when `N` is a multiple of `IVAL(4)`. The values `IVAL(3)` and `IVAL(4)` are temporarily replaced by `IVAL(1)` and `IVAL(2)`, respectively, in routine `GVCRG`. Analogous comments hold for the array `BCOPY` and the option values `IVAL(5) – IVAL(8)`. Additional memory allocation and option value restoration are automatically done in `GVCRG`. There is no requirement that users change existing applications that use `GVCRG` or `G8CRG`. Default values for the option are `IVAL(*) = 1, 16, 0, 1, 1, 16, 0, 1`. Items 5–8 in `IVAL(*)` are for the generalized eigenvalue problem and are not used in `GVCRG`.

## Example

In this example, `DATA` statements are used to set  $A$  and  $B$ . The eigenvalues, eigenvectors and performance index are computed and printed for the systems  $Ax = \lambda Bx$  and  $Bx = \mu Ax$  where  $\mu = \lambda^{-1}$ . For more details about the performance index, see routine `GPIRG`.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDA, LDB, LDEVEC, N
PARAMETER (N=3, LDA=N, LDB=N, LDEVEC=N)
!
INTEGER I, NOUT
REAL A(LDA,N), B(LDB,N), BETAV(N), PI
COMPLEX ALPHA(N), EVAL(N), EVEC(LDEVEC,N)
!
!
! Define values of A and B:
! A = (1.0 0.5 0.0)
! (-10.0 2.0 0.0)
! (5.0 1.0 0.5)
!
! B = (0.5 0.0 0.0)
! (3.0 3.0 0.0)
! (4.0 0.5 1.0)
!
! Declare variables
DATA A/1.0, -10.0, 5.0, 0.5, 2.0, 1.0, 0.0, 0.0, 0.5/
DATA B/0.5, 3.0, 4.0, 0.0, 3.0, 0.5, 0.0, 0.0, 1.0/
!
CALL GVCRG (A, B, ALPHA, BETAV, EVEC)
! Compute eigenvalues
DO 10 I=1, N
EVAL(I) = ALPHA(I)/BETAV(I)
10 CONTINUE
!
! Compute performance index
PI = GPIRG(N,A,B,ALPHA,BETAV,EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
! Solve for reciprocals of values

```

```

CALL GVCRG (B, A, ALPHA, BETAV, EVEC)

! Compute reciprocals
DO 20 I=1, N
 EVAL(I) = ALPHA(I)/BETAV(I)
20 CONTINUE

! Compute performance index
PI = GPIRG(N,B,A,ALPHA,BETAV,EVEC)

! Print results
CALL WRCRN ('EVAL reciprocals', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3
(0.833, 1.993) (0.833,-1.993) (0.500, 0.000)

 EVEC
 1 2 3
1 (-0.197, 0.150) (-0.197,-0.150) (-0.000, 0.000)
2 (-0.069,-0.568) (-0.069, 0.568) (-0.000, 0.000)
3 (0.782, 0.000) (0.782, 0.000) (1.000, 0.000)

Performance index = 0.384

 EVAL reciprocals
 1 2 3
(2.000, 0.000) (0.179, 0.427) (0.179,-0.427)

 EVEC
 1 2 3
1 (0.000, 0.000) (-0.197,-0.150) (-0.197, 0.150)
2 (0.000, 0.000) (-0.069, 0.568) (-0.069,-0.568)
3 (1.000, 0.000) (0.782, 0.000) (0.782, 0.000)

Performance index = 0.283

```

---

## GPIRG

This function computes the performance index for a generalized real eigensystem  $Az = \lambda Bz$ .

### Function Return Value

*GPIRG* — Performance index. (Output)

### Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs performance index computation is based on. (Input)

*A* — Real matrix of order *N*. (Input)

*B* — Real matrix of order *N*. (Input)

*ALPHA* — Complex vector of length *NEVAL* containing the numerators of eigenvalues.  
(Input)

*BETAV* — Real vector of length *NEVAL* containing the denominators of eigenvalues. (Input)

*EVEC* — Complex *N* by *NEVAL* array containing the eigenvectors. (Input)

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDB* = *SIZE* (*B*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

### FORTRAN 90 Interface

Generic: `GPIRG (NEVAL, A, B, ALPHA, BETAV, EVEC, GPIRG [, ...])`

Specific: The specific interface names are `S_GPIRG` and `D_GPIRG`.

### FORTRAN 77 Interface

Single: `GPIRG (N, NEVAL, A, LDA, B, LDB, ALPHA, BETAV, EVEC, LDEVEC)`

Double: The double precision function name is `DGPIRG`.

Let  $M = NEVAL$ ,  $x_j = EVEC(*,J)$ , the *j*-th column of *EVEC*. Also, let  $\epsilon$  be the machine precision given by `AMACH(4)`, see the [Reference](#) chapter of this manual. The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|\beta_j Ax_j - \alpha_j Bx_j\|_1}{\varepsilon \left( |\beta_j| \|A\|_1 + |\alpha_j| \|B\|_1 \right) \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{ |\Re v_i| + |\Im v_i| \}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of [EVCSF](#) is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

## Comments

1. Workspace may be explicitly provided, if desired, by use of [G2IRG](#)/[DG2IRG](#). The reference is:

```
G2IRG (N, NEVAL, A, LDA, B, LDB, ALPHA, BETAV, EVEC,
 LDEVEC, WK)
```

The additional argument is:

**WK** — Complex work array of length  $2N$ .

2. Informational errors

| Type | Code | Description                            |
|------|------|----------------------------------------|
| 3    | 1    | Performance index is greater than 100. |
| 3    | 2    | An eigenvector is zero.                |
| 3    | 3    | The matrix A is zero.                  |
| 3    | 4    | The matrix B is zero.                  |

3. The  $J$ -th eigenvalue should be  $\text{ALPHA}(J)/\text{BETAV}(J)$ , its eigenvector should be in the  $J$ -th column of [EVEC](#).

## Example

For an example of [GPIRG](#), see routine [GVCRG](#).

---

# GVLCG

Computes all of the eigenvalues of a generalized complex eigensystem  $Az = \lambda Bz$ .

## Required Arguments

*A* — Complex matrix of order *N*. (Input)

*B* — Complex matrix of order *N*. (Input)

*ALPHA* — Complex vector of length *N*. Ultimately,  $\alpha(i)/\beta(i)$  (for  $i = 1, n$ ), will be the eigenvalues of the system in decreasing order of magnitude. (Output)

*BETAV* — Complex vector of length *N*. (Output)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default:  $LDA = \text{SIZE}(A,1)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)

Default:  $LDB = \text{SIZE}(B,1)$ .

## FORTRAN 90 Interface

Generic:     CALL GVLGC (A, B, ALPHA, BETAV [, ...])

Specific:    The specific interface names are S\_GVLGC and D\_GVLGC.

## FORTRAN 77 Interface

Single:     CALL GVLGC (N, A, LDA, B, LDB, ALPHA, BETAV)

Double:     The double precision name is DGVLCG.

## Description

Routine GVLGC computes the eigenvalues of the generalized eigensystem  $Ax = \lambda Bx$ , where *A* and *B* are complex matrices of order *n*. The eigenvalues for this problem can be infinite; so instead of returning  $\lambda$ , GVLGC returns  $\alpha$  and  $\beta$ . If  $\beta$  is nonzero, then  $\lambda = \alpha/\beta$ . If the eigenvectors are needed, then use GVCCG.

The underlying code is based on either EISPACK or LAPACK code depending upon which supporting libraries are used during linking. For a detailed explanation, see “Using ScaLAPACK, LAPACK, LINPACK, and EISPACK” in the Introduction section of this manual. Some timing information is given in Hanson et al. (1990).

## Comments

1. Workspace may be explicitly provided, if desired, by use of G3LCG/DG3LCG. The reference is:

```
CALL G3LCG (N, A, LDA, B, LDB, ALPHA, BETAV, ACOPY, BCOPY,
 CWK, WK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . A and ACOPY may be the same, in which case A will be destroyed.

**BCOPY** — Complex work array of length  $N^2$ . B and BCOPY may be the same, in which case B will be destroyed.

**CWK** — Complex work array of length N.

**WK** — Real work array of length N.

**IWK** — Integer work array of length N.

2. Informational error

Type Code

4 1 The iteration for the eigenvalues failed to converge.

## Example

In this example, DATA statements are used to set A and B. Then, the eigenvalues are computed and printed.

```
USE GVLCG_INT
USE WRCRN_INT

IMPLICIT NONE
! Declaration of variables
INTEGER LDA, LDB, N
PARAMETER (N=5, LDA=N, LDB=N)
!
INTEGER I
COMPLEX A(LDA,N), ALPHA(N), B(LDB,N), BETAV(N), EVAL(N)
!
! Define values of A and B
!
DATA A/(-238.0,-344.0), (76.0,152.0), (118.0,284.0), &
(-314.0,-160.0), (-54.0,-24.0), (86.0,178.0), &
(-96.0,-128.0), (55.0,-182.0), (132.0,78.0), &
(-205.0,-400.0), (164.0,240.0), (40.0,-32.0), &
(-13.0,460.0), (114.0,296.0), (109.0,148.0), &
(-166.0,-308.0), (60.0,184.0), (34.0,-192.0), &
```

```

(-90.0,-164.0), (158.0,312.0), (56.0,158.0), &
(-60.0,-136.0), (-176.0,-214.0), (-424.0,-374.0), &
(-38.0,-96.0)/
DATA B/(388.0,94.0), (-304.0,-76.0), (-658.0,-136.0), &
(-640.0,-10.0), (-162.0,-72.0), (-386.0,-122.0), &
(384.0,64.0), (-73.0,100.0), (204.0,-42.0), (631.0,158.0), &
(-250.0,-14.0), (-160.0,16.0), (-109.0,-250.0), &
(-692.0,-90.0), (131.0,52.0), (556.0,130.0), &
(-240.0,-92.0), (-118.0,100.0), (288.0,66.0), &
(-758.0,-184.0), (-396.0,-62.0), (240.0,68.0), &
(406.0,96.0), (-192.0,154.0), (278.0,76.0)/
!
CALL GVLGC (A, B, ALPHA, BETAV)
! Compute eigenvalues
 EVAL = ALPHA/BETAV
!
! Print results
CALL WRCRN ('EVAL', EVAL, 1, N, 1)

STOP
END

```

## Output

```

 EVAL
 2
1 (-1.000,-1.333) (0.765, 0.941) (-0.353, 0.412) (-0.353,-0.412)
 3
 4
5 (-0.353,-0.412)

```

---

## GVCCG

Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem  $Az = \lambda Bz$ .

### Required Arguments

**A** — Complex matrix of order N. (Input)

**B** — Complex matrix of order N. (Input)

**ALPHA** — Complex vector of length N. Ultimately,  $\text{alpha}(i)/\text{betav}(i)$  (for  $i = 1, \dots, n$ ), will be the eigenvalues of the system in decreasing order of magnitude. (Output)

**BETAV** — Complex vector of length N. (Output)

**EVEC** — Complex matrix of order N. (Output)

The J-th eigenvector, corresponding to  $\text{ALPHA}(J)/\text{BETAV}(J)$ , is stored in the J-th column. Each vector is normalized to have Euclidean length equal to the value one.

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDB* = SIZE (*B*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDEVEC* = SIZE (*EVEC*,1).

## FORTRAN 90 Interface

Generic:     CALL GVCCG (*A*, *B*, ALPHA, BETAV, EVEC [, ...])

Specific:    The specific interface names are S\_GVCCG and D\_GVCCG.

## FORTRAN 77 Interface

Single:     CALL GVCCG (*N*, *A*, *LDA*, *B*, *LDB*, ALPHA, BETAV, EVEC, LDEVEC)

Double:     The double precision name is DGVCCG.

## Description

Routine *GVCCG* computes the eigenvalues and eigenvectors of the generalized eigensystem  $Ax = \lambda Bx$ . Here, *A* and *B*, are complex matrices of order *n*. The eigenvalues for this problem can be infinite; so instead of returning  $\lambda$ , *GVCCG* returns  $\alpha$  and  $\beta$ . If  $\beta$  is nonzero, then  $\lambda = \alpha/\beta$ .

The routine *GVCCG* uses the QZ algorithm described by Moler and Stewart (1973). The implementation is based on routines of Garbow (1978). Some timing results are given in Hanson et al. (1990).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of *G6CCG*/*DG6CCG*. The reference is:

```
CALL G6CCG (N, A, LDA, B, LDB, ALPHA, BETAV, EVEC, LDEVEC,
 ACOPY, BCOPY, CWK, WK, IWK)
```

The additional arguments are as follows:

**ACOPY** — Complex work array of length  $N^2$ . **A** and **ACOPY** may be the same in which case the first  $N^2$  elements of **A** will be destroyed.

**BCOPY** — Complex work array of length  $N^2$ . **B** and **BCOPY** may be the same in which case the first  $N^2$  elements of **B** will be destroyed.

**CWK** — Complex work array of length  $N$ .

**WK** — Real work array of length  $N$ .

**IWK** — Integer work array of length  $N$ .

2. Informational error

Type Code

4 1 The iteration for an eigenvalue failed to converge.

3. The success of this routine can be checked using [GPICG](#).

## Example

In this example, **DATA** statements are used to set **A** and **B**. The eigenvalues and eigenvectors are computed and printed. The performance index is also computed and printed. This serves as a check on the computations. For more details, see routine [GPICG](#).

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDA, LDB, LDEVEC, N
PARAMETER (N=3, LDA=N, LDB=N, LDEVEC=N)
!
INTEGER I, NOUT
REAL PI
COMPLEX A(LDA,N), ALPHA(N), B(LDB,N), BETAV(N), EVAL(N), &
EVEC(LDEVEC,N)
!
! Define values of A and B
! A = (1+0i 0.5+i 0+5i)
! (-10+0i 2+i 0+0i)
! (5+i 1+0i 0.5+3i)
!
! B = (0.5+0i 0+0i 0+0i)
! (3+3i 3+3i 0+i)
! (4+2i 0.5+i 1+i)
!
! Declare variables
DATA A/(1.0,0.0), (-10.0,0.0), (5.0,1.0), (0.5,1.0), (2.0,1.0), &
(1.0,0.0), (0.0,5.0), (0.0,0.0), (0.5,3.0)/
DATA B/(0.5,0.0), (3.0,3.0), (4.0,2.0), (0.0,0.0), (3.0,3.0), &
(0.5,1.0), (0.0,0.0), (0.0,1.0), (1.0,1.0)/
! Compute eigenvalues
CALL GVCCG (A, B, ALPHA, BETAV, EVEC)
```

```

 EVAL = ALPHA/BETAV
! Compute performance index
PI = GPICG(N,A,B,ALPHA,BETAV,EVEC)
! Print results
CALL UMACH (2, NOUT)
CALL WRCRN ('EVAL', EVAL, 1, N, 1)
CALL WRCRN ('EVEC', EVEC)
WRITE (NOUT, '(/,A,F6.3)') ' Performance index = ', PI
END

```

## Output

```

 EVAL
 1 2 3
(-8.18,-25.38) (2.18, 0.61) (0.12, -0.39)

 EVEC
 1 2 3
1 (-0.3267,-0.1245) (-0.3007,-0.2444) (0.0371, 0.1518)
2 (0.1767, 0.0054) (0.8959, 0.0000) (0.9577, 0.0000)
3 (0.9201, 0.0000) (-0.2019, 0.0801) (-0.2215, 0.0968)

Performance index = 0.709

```

---

## GPICG

This function computes the performance index for a generalized complex eigensystem  $Az = \lambda Bz$ .

### Function Return Value

*GPICG* — Performance index. (Output)

### Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs performance index computation is based on. (Input)

*A* — Complex matrix of order N. (Input)

*B* — Complex matrix of order N. (Input)

*ALPHA* — Complex vector of length *NEVAL* containing the numerators of eigenvalues. (Input)

*BETAV* — Complex vector of length *NEVAL* containing the denominators of eigenvalues. (Input)

*EVEC* — Complex N by *NEVAL* array containing the eigenvectors. (Input)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDB* = SIZE (*B*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = SIZE (*EVEC*,1).

## FORTRAN 90 Interface

Generic: GPICG (NEVAL, A, B, ALPHA, BETAV, EVEC [, ...])

Specific: The specific interface names are S\_GPICG and D\_GPICG.

## FORTRAN 77 Interface

Single: GPICG (N, NEVAL, A, LDA, B, LDB, ALPHA, BETAV, EVEC, LDEVEC)

Double: The double precision name is DGPICG.

## Description

Let  $M = \text{NEVAL}$ ,  $x_j = \text{EVEC}(*, J)$ , the  $j$ -th column of *EVEC*. Also, let  $\epsilon$  be the machine precision given by *AMACH*(4). The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|\beta_j A x_j - \alpha_j B x_j\|_1}{\epsilon \left( |\beta_j| \|A\|_1 + |\alpha_j| \|B\|_1 \right) \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{ |\Re v_i| + |\Im v_i| \}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of *EVCSF* is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ .

The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `G2ICG/DG2ICG`. The reference is:

```
G2ICG (N, NEVAL, A, LDA, B, LDB, ALPHA, BETAV, EVEC,
 LDEVEC, WK)
```

The additional argument is:

**WK** — Complex work array of length  $2N$ .

2. Informational errors

Type Code

|   |   |                                        |
|---|---|----------------------------------------|
| 3 | 1 | Performance index is greater than 100. |
| 3 | 2 | An eigenvector is zero.                |
| 3 | 3 | The matrix A is zero.                  |
| 3 | 4 | The matrix B is zero.                  |

3. The  $J$ -th eigenvalue should be  $\text{ALPHA}(J)/\text{BETAV}(J)$ , its eigenvector should be in the  $J$ -th column of `EVEC`.

## Example

For an example of `GPICG`, see routine [GVCCG](#).

---

# GVLSP



Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem  $Az = \lambda Bz$ , with  $B$  symmetric positive definite.

## Required Arguments

**A** — Real symmetric matrix of order  $N$ . (Input)

**B** — Positive definite symmetric matrix of order  $N$ . (Input)

**EVAL** — Vector of length  $N$  containing the eigenvalues in decreasing order of magnitude. (Output)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

## FORTRAN 90 Interface

Generic:    CALL GVLSP (A, B, EVAL [, ...])

Specific:   The specific interface names are S\_GVLSP and D\_GVLSP.

## FORTRAN 77 Interface

Single:     CALL GVLSP (N, A, LDA, B, LDB, EVAL)

Double:    The double precision name is DGVLSP.

## Description

Routine GVLSP computes the eigenvalues of  $Ax = \lambda Bx$  with *A* symmetric and *B* symmetric positive definite. The Cholesky factorization  $B = R^T R$ , with *R* a triangular matrix, is used to transform the equation  $Ax = \lambda Bx$  to

$$(R^{-T} A R^{-1})(Rx) = \lambda (Rx)$$

The eigenvalues of  $C = R^{-T} A R^{-1}$  are then computed. This development is found in Martin and Wilkinson (1968). The Cholesky factorization of *B* is computed based on IMSL routine LFTDS, (see [Chapter 1, Linear Systems](#)). The eigenvalues of *C* are computed based on routine EVLSP. Further discussion and some timing results are given Hanson et al. (1990).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of G3LSP/DG3LSP. The reference is:

```
CALL G3LSP (N, A, LDA, B, LDB, EVAL, IWK, WK1, WK2)
```

The additional arguments are as follows:

*IWK* — Integer work array of length *N*.

*WKI* — Work array of length  $2N$ .

**WK2** — Work array of length  $N^2 + N$ .

2. Informational errors

Type Code

4 1 The iteration for an eigenvalue failed to converge.

4 2 Matrix B is not positive definite.

### Example

In this example, a DATA statement is used to set the matrices *A* and *B*. The eigenvalues of the system are computed and printed.

```
USE GVLSP_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N
PARAMETER (N=3, LDA=N, LDB=N)
!
REAL A(LDA,N), B(LDB,N), EVAL(N)
! Define values of A:
! A = (2 3 5)
! (3 2 4)
! (5 4 2)
DATA A/2.0, 3.0, 5.0, 3.0, 2.0, 4.0, 5.0, 4.0, 2.0/
!
! Define values of B:
! B = (3 1 0)
! (1 2 1)
! (0 1 1)
DATA B/3.0, 1.0, 0.0, 1.0, 2.0, 1.0, 0.0, 1.0, 1.0/
!
! Find eigenvalues
CALL GVLSP (A, B, EVAL)
! Print results
CALL WRRRN ('EVAL', EVAL, 1, N, 1)
END
```

### Output

```
 EVAL
 1 2 3
-4.717 4.393 -0.676
```

---

# GVCSP



Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Az = \lambda Bz$ , with  $B$  symmetric positive definite.

## Required Arguments

$A$  — Real symmetric matrix of order  $N$ . (Input)

$B$  — Positive definite symmetric matrix of order  $N$ . (Input)

$EVAL$  — Vector of length  $N$  containing the eigenvalues in decreasing order of magnitude. (Output)

$EVEC$  — Matrix of order  $N$ . (Output)  
The  $J$ -th eigenvector, corresponding to  $EVAL(J)$ , is stored in the  $J$ -th column. Each vector is normalized to have Euclidean length equal to the value one.

## Optional Arguments

$N$  — Order of the matrices  $A$  and  $B$ . (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

$LDB$  — Leading dimension of  $B$  exactly as specified in the dimension statement in the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

$LDEVEC$  — Leading dimension of  $EVEC$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDEVEC = \text{SIZE}(EVEC,1)$ .

## FORTRAN 90 Interface

Generic:    `CALL GVCSP (A, B, EVAL, EVEC [, ...])`

Specific:    The specific interface names are `S_GVCSP` and `D_GVCSP`.

## FORTRAN 77 Interface

Single:      CALL GVCSP (N, A, LDA, B, LDB, EVAL, EVEC, LDEVEC)

Double:     The double precision name is DGVCSF.

## Description

Routine [GVLSP](#) computes the eigenvalues and eigenvectors of  $Az = \lambda Bz$ , with  $A$  symmetric and  $B$  symmetric positive definite. The Cholesky factorization  $B = R^T R$ , with  $R$  a triangular matrix, is used to transform the equation  $Az = \lambda Bz$ , to

$$(R^{-T} A R^{-1})(Rz) = \lambda (Rz)$$

The eigenvalues and eigenvectors of  $C = R^{-T} A R^{-1}$  are then computed. The generalized eigenvectors of  $A$  are given by  $z = R^{-1} x$ , where  $x$  is an eigenvector of  $C$ . This development is found in Martin and Wilkinson (1968). The Cholesky factorization is computed based on IMSL routine [LFTDS](#), see [Chapter 1, Linear Systems](#). The eigenvalues and eigenvectors of  $C$  are computed based on routine [EVCSE](#). Further discussion and some timing results are given Hanson et al. (1990).

## Comments

1.   Workspace may be explicitly provided, if desired, by use of [G3CSP](#)/[DG3CSP](#). The reference is:

```
CALL G3CSP (N, A, LDA, B, LDB, EVAL, EVEC, LDEVEC, IWK,
 WK1, WK2)
```

The additional arguments are as follows:

**IWK** — Integer work array of length  $N$ .

**WK1** — Work array of length  $3N$ .

**WK2** — Work array of length  $N^2 + N$ .

2.   Informational errors

Type   Code

4      1           The iteration for an eigenvalue failed to converge.

4      2           Matrix B is not positive definite.

3.   The success of this routine can be checked using [GPISP](#).

## Example

In this example, a `DATA` statement is used to set the matrices  $A$  and  $B$ . The eigenvalues, eigenvectors and performance index are computed and printed. For details on the performance index, see IMSL routine [GPISP](#).

```
USE GVCSP_INT
USE GPISP_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, LDB, LDEVEC, N
PARAMETER (N=3, LDA=N, LDB=N, LDEVEC=N)

!
INTEGER NOUT
REAL A(LDA,N), B(LDB,N), EVAL(N), EVEC(LDEVEC,N), PI

! Define values of A:
! A = (1.1 1.2 1.4)
! (1.2 1.3 1.5)
! (1.4 1.5 1.6)
DATA A/1.1, 1.2, 1.4, 1.2, 1.3, 1.5, 1.4, 1.5, 1.6/

! Define values of B:
! B = (2.0 1.0 0.0)
! (1.0 2.0 1.0)
! (0.0 1.0 2.0)
DATA B/2.0, 1.0, 0.0, 1.0, 2.0, 1.0, 0.0, 1.0, 2.0/

! Find eigenvalues and vectors
CALL GVCSP (A, B, EVAL, EVEC)

! Compute performance index
PI = GPISP(N,A,B,EVAL,EVEC)

! Print results
CALL UMACH (2, NOUT)
CALL WRRRN ('EVAL', EVAL)
CALL WRRRN ('EVEC', EVEC)
WRITE (NOUT,'(/,A,F6.3)') ' Performance index = ', PI
END
```

## Output

```
EVAL
1 1.386
2 -0.058
3 -0.003

 EVEC
 1 2 3
1 0.6431 -0.1147 -0.6817
2 -0.0224 -0.6872 0.7266
3 0.7655 0.7174 -0.0858

Performance index = 0.417
```

---

# GPISP

This function computes the performance index for a generalized real symmetric eigensystem problem.

## Function Return Value

*GPISP* — Performance index. (Output)

## Required Arguments

*NEVAL* — Number of eigenvalue/eigenvector pairs that the performance index computation is based on. (Input)

*A* — Symmetric matrix of order *N*. (Input)

*B* — Symmetric matrix of order *N*. (Input)

*EVAL* — Vector of length *NEVAL* containing eigenvalues. (Input)

*EVEC* — *N* by *NEVAL* array containing the eigenvectors. (Input)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDB* = *SIZE* (*B*,1).

*LDEVEC* — Leading dimension of *EVEC* exactly as specified in the dimension statement in the calling program. (Input)  
Default: *LDEVEC* = *SIZE* (*EVEC*,1).

## FORTRAN 90 Interface

Generic:     GPISP (NEVAL, A, B, EVAL, EVEC [, ...])

Specific:    The specific interface names are S\_GPISP and D\_GPISP.

## FORTRAN 77 Interface

Single: GPISP (N, NEVAL, A, LDA, B, LDB, EVAL, EVEC, LDEVEC)

Double: The double precision name is DGPISP.

## Description

Let  $M = \text{NEVAL}$ ,  $\lambda = \text{EVAL}$ ,  $x_j = \text{EVEC}(*, J)$ , the  $j$ -th column of  $\text{EVEC}$ . Also, let  $\varepsilon$  be the machine precision given by  $\text{AMACH}(4)$ . The performance index,  $\tau$ , is defined to be

$$\tau = \max_{1 \leq j \leq M} \frac{\|Ax_j - \lambda_j Bx_j\|_1}{\varepsilon (\|A\|_1 + |\lambda_j| \|B\|_1) \|x_j\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector  $v$  is

$$\|v\|_1 = \sum_{i=1}^N \{|\Re v_i| + |\Im v_i|\}$$

While the exact value of  $\tau$  is highly machine dependent, the performance of **EVCSE** is considered excellent if  $\tau < 1$ , good if  $1 \leq \tau \leq 100$ , and poor if  $\tau > 100$ . The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

## Comments

1. Workspace may be explicitly provided, if desired, by use of **G2ISP/DG2ISP**. The reference is:

G2ISP (N, NEVAL, A, LDA, B, LDB, EVAL, EVEC, LDEVEC, WORK)

The additional argument is:

**WORK** — Work array of length  $2 * N$ .

2. Informational errors

Type Code

3 1 Performance index is greater than 100.

3 2 An eigenvector is zero.

3 3 The matrix A is zero.

3 4 The matrix B is zero.

3. The  $J$ -th eigenvalue should be  $\text{ALPHA}(J)/\text{BETAV}(J)$ , its eigenvector should be in the  $J$ -th column of  $\text{EVEC}$ .

### Example

For an example of GPISP, see routine [GVCSP](#).

---

# Eigenvalues and Eigenvectors Computed with ARPACK

First see [Using ARPACK for Ordinary and Generalized Eigenvalue Problems](#) in the Usage Notes section of this chapter. We describe here the Fortran 2003 usage of four basic problem types. There must be compiler support for the object-oriented features of Fortran 2003 to use these routines.

The generalized eigenvalue problem  $Ax = \lambda Bx$  requires that some eigenvalues and eigenvectors be computed. The organization of the user-written function for matrix-vector products depends on the part of the eigenvalue spectrum that is desired.

For an ordinary problem with  $A$  symmetric and  $B = I$ , the eigenvalues of largest or smallest magnitude can be computed by providing the operator products  $w = Ax$ . Here  $x$  is an input vector and  $w$  is the result of applying the linear operator  $A$  to  $x$ . This process is repeated several times within the Arnoldi algorithm, and the net result is a few eigenvalues of  $A$  and the corresponding eigenvectors.

For a generalized problem, it is useful and efficient to consider a shift value  $\sigma$  and the ordinary eigenvalue problem  $Cx \equiv (A - \sigma B)^{-1} Bx = \nu x$ . The matrix pencil  $A - \sigma B$  is non-singular. The purpose of the user-written function is to provide results for the individual operator products  $w = Bx$ ,  $w = (A - \sigma B)^{-1} x$ , and  $w = Ax$ . Usually the inverse matrix product will be computed by solving linear systems, where the matrix pencil is the coefficient matrix. The desired eigenvalues of this ordinary problem satisfy  $\lambda_j = \sigma + \nu_j^{-1}$ .

In the special case that  $B$  is positive definite, well-conditioned, and symmetric, one may compute the Cholesky decomposition  $B = R^T R$  and then solve the ordinary eigenvalue problem  $Cy \equiv R^{-T} A R^{-1} y = \lambda y$ . The product operation required by the Arnoldi algorithm,  $w = Cx$ , is performed in steps: Solve  $Rz = x$  for  $z$ , compute  $y = Az$ , and solve  $R^T x = y$  for  $w$ . The eigenvectors,  $Y$ , of  $C$  are transformed to those of the generalized problem,  $X$ , by solving  $RX = Y$  for  $X$ .

The operations required by ARPACK codes are returned as array functions. An array of input values,  $x$ , will yield an output array,  $y$ . These functions are written by the user. They must be written according to an [abstract interface](#), given below. There are two user functions, double precision real and complex, that we support for the eigenvalue problem, and a third for the singular value decomposition, using double precision real data only. This interface, the named or enumerated constants that describe what is needed, and the eigenvalue codes are in the module `ARPACK_INT`. We use the notation: `DKIND=kind(1.D0)` to specify two double precision data types: `REAL(DKIND)` and `COMPLEX(DKIND)`. The interface `SVDMV(...)` is for the singular value decomposition products only. For that problem the components `EXTYPE%MROWS` and `EXTYPE%NCOLS` are switched between the operator sizes  $M$  and  $N$  to account for computing  $y = A_{M \times N} x$  or  $y = A^T x$ .

## The Abstract Interfaces for User-Written Array Functions

```
Abstract Interface

 FUNCTION DMV(X, TASK, EXTYPE)RESULT(Y)
 IMPORT DKIND, ARPACKBASE
 REAL(DKIND), INTENT(INOUT) :: X(:)
 INTEGER, INTENT(IN) :: TASK
 CLASS (ARPACKBASE), INTENT(INOUT) :: EXTYPE
 REAL(DKIND) Y(SIZE(X))
 END FUNCTION

 FUNCTION ZMV (X, TASK, EXTYPE) RESULT(Y)
 IMPORT DKIND, ARPACKBASE
 CLASS (ARPACKBASE), INTENT(INOUT) :: EXTYPE
 COMPLEX (DKIND), INTENT(INOUT) :: X(:)
 INTEGER, INTENT(IN) :: TASK
 COMPLEX (DKIND) Y(SIZE(X))
 END FUNCTION

 FUNCTION SVDMV (X, TASK, EXTYPE) RESULT(Y)
 IMPORT DKIND, ARPACKBASE
 CLASS (ARPACKBASE), INTENT(INOUT) :: EXTYPE
 REAL (DKIND), INTENT(INOUT) :: X(EXTYPE%NCOLS)
 INTEGER, INTENT(IN) :: TASK
 REAL (DKIND) Y(EXTYPE%MROWS)
 END FUNCTION

End Interface
```

*Figure 1 Abstract Interface for User-Written Array Functions*

---

## The Base Class ARPACKBASE

The components of the derived type `ARPACKBASE` contain data used by the ARPACK routines. These will have initial or default values assigned. The default values can usually be left unchanged with a first use of our codes. They are used as arguments to the original routines of the ARPACK package. The more experienced user may wish to change the components marked with ‘=>’ to new values, depending on their application. These can be changed prior to calling the ARPACK interface codes we provide. This base class can be extended to pass user data or procedure pointers for use within the array function.

Note that the derived type argument `EXTYPE`, is *optional* in all the `ARPACK_` eigenvalue routines, but it is not optional for the user-written array functions. If `EXTYPE` is not included in the argument list of the `ARPACK_` eigenvalue routine, an internally declared type is passed to the array functions as the argument, `EXTYPE`. Although the user may choose not to use this optional argument when calling our interface routines, they must include this argument in their user-supplied array function code. In this case, the array function code does not need to reference this argument.

```
TYPE, PUBLIC :: ARPACKBASE
 INTEGER :: TASK = 0 ! Local store in Class for compute
 ! tasks to follow. Used in
ARPACK_SVD.
 INTEGER :: MROWS=0 ! Defines output vector size
 INTEGER :: NCOLS=0 ! Defines input/output vector size
=> COMPLEX(DKIND) :: SHIFT=&
 (0. _DKIND, 0. _DKIND) ! Shift factor
 REAL(_DKIND) :: TOL=EPSILON(0. _DKIND) ! Error tolerance
 INTEGER :: ISHFTS = 1 ! Number of shifts - don't change
=> INTEGER :: MAXITR = HUGE(1) ! Max number of iterations (many!)
=> INTEGER :: MAXMV = HUGE(1) ! Max number of matrix ops (many!)
=> INTEGER :: INFO = 0 ! ARPACK error flag, = 0 is OK
=> INTEGER :: NACC = 0 ! Number of accurate eigenvalues or
 ! singular values computed.
 INTEGER :: IPARAM(11)=0 ! ARPACK array of direction and
 ! result flags
=> REAL(DKIND) :: FACTOR_MAXNCV = 2.5 _DKIND ! Factor for the Number of Ritz
vectors ! > the number of requested
eigenvalues
 LOGICAL :: RALEIGH_QUOTIENT = .TRUE. ! Compute eigenvalues using the
 ! computed eigenvectors and Raleigh
 ! quotients.
 REAL(DKIND), ALLOCATABLE :: RESID(:) ! REAL Starting vector for Arnoldi
 ! iteration, if allocated; else
random
Arnoldi COMPLEX(DKIND), ALLOCATABLE :: ZRESID(:) ! COMPLEX Starting vector for
 ! iteration, if allocated; else
random
END TYPE
```

Figure 2. The Base Class Derived Type – Users may extend this to provide problem data or procedures.

---

# ARPACK\_SYMMETRIC

Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Ax = \lambda Bx$ . This can be used for the case  $B = I$ .

## Required Arguments

$N$  — The dimension of the problem. (Input)

$F$  — User-supplied FUNCTION to return matrix-vector operations or linear solutions. This user function is written corresponding to the [abstract interface](#) for the function DMV(...). The usage is `F (X, TASK, EXTYPE)`, where

## Function Return Value

$F$  — An array of length  $N$  containing matrix-vector operations or linear equations solutions. Operations provided as code in the function  $F$  will be made depending upon the value of argument  $TASK$ .

## Required Arguments

$X$  — An array of length  $N$  containing the vector to which the operator will be applied. (Input)

$TASK$  — An enumerated type which specifies the operation to be performed. (Input)

$TASK$  is an enumerated integer value, use-associated from the module ARPACK\_INT. It will be one of the following:

| Value                        | Description                                                                                                                                                                                                                  |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARPACK_Prepere               | Take initial steps to prepare for the operations that follow. These steps can include defining the data for the matrices, factorizations for upcoming linear system solves, or recording the vectors used in the operations. |
| ARPACK_A_x                   | $y = Ax$                                                                                                                                                                                                                     |
| ARPACK_B_x                   | $y = Bx$                                                                                                                                                                                                                     |
| ARPACK_inv_A_minus_Shift_x   | $y = (A - \sigma I)^{-1} x$                                                                                                                                                                                                  |
| ARPACK_inv_B_x               | $y = B^{-1}x$                                                                                                                                                                                                                |
| ARPACK_inv_A_minus_Shift_B_x | $y = (A - \sigma B)^{-1} x$                                                                                                                                                                                                  |

**EXTYPE** — A derived type of the extensible class `ARPACKBASE`, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a `USE ARPACK_INT` statement in the calling program to define this derived type. If `EXTYPE` is not included as an argument to `ARPACK_SYMMETRIC` it should be ignored in the user-function, `F`.

The function `F` must be written according to the abstract interface for `DMV`. If `F` is not use-associated nor contained in the calling program, declare it with `PROCEDURE(DMV) F`.

**VALUES** — An array of eigenvalues. (Output)

The value `NEV=size(VALUES)` defines the number of eigenvalues to be computed. The calling program declares or allocates the array `VALUES(1:NEV)`. The number of eigenvalues computed accurately is optionally available as the component `EXTYPE%NACC` of the base class `EXTYPE`.

## Optional Arguments

**PLACE** — Defines the output content of `VALUES`. (Input)

`PLACE` specifies the placement within the spectrum for the required eigenvalues. `PLACE` can be one of the following enumerated integers as defined in `ARPACK_INT`:

| Value                                   |
|-----------------------------------------|
| <code>ARPACK_Largest_Algebraic</code>   |
| <code>ARPACK_Smallest_Algebraic</code>  |
| <code>ARPACK_Largest_Magnitude</code>   |
| <code>ARPACK_inv_A_minus_Shift_x</code> |
| <code>ARPACK_Smallest_Magnitude</code>  |
| <code>ARPACK_Both_Ends</code>           |

Default: `PLACE = ARPACK_Largest_Algebraic`.

**TYPE** — Defines the eigenvalue problem as either a standard or generalized eigenvalue problem. (Input)

`TYPE` can be one of the following enumerated integers as defined in `ARPACK_INT`:

| Value                           | Description       |
|---------------------------------|-------------------|
| <code>ARPACK_Standard</code>    | $Ax = \lambda x$  |
| <code>ARPACK_Generalized</code> | $Ax = \lambda Bx$ |

Default: `TYPE = ARPACK_Standard`.

**CATEGORY** — CATEGORY and TYPE define the operation sequence provided in the user-written function. (Input)

CATEGORY can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                  | Description                                |
|------------------------|--------------------------------------------|
| ARPACK_Regular         | $y = Ax$                                   |
| ARPACK_Regular_Inverse | $y = Ax, y = Bx, y = B^{-1}x$              |
| ARPACK_Shift_Invert    | $y = Ax, y = (A - \sigma I)^{-1}x$         |
| ARPACK_Buckling        | $y = Ax, y = Bx, y = (A - \sigma B)^{-1}x$ |
| ARPACK_Cayley          | $y = Ax, y = Bx, y = (A - \sigma B)^{-1}x$ |

Default: CATEGORY = ARPACK\_Regular.

**EXTYPE** — A derived type of the extensible class ARPACKBASE, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a USE ARPACK\_INT statement in the calling program to define this derived type. If EXTYPE is not included as an argument to ARPACK\_SYMMETRIC it must still be supplied as an argument to user-function, F, but is not used.

**VECTORS** — An allocatable array of approximate eigenvectors. (Output)

It is not necessary to allocate VECTORS(:, :). If this argument is used the allocation occurs within the routine ARPACK\_SYMMETRIC. The output sizes are VECTORS(1:N, 1:NCV). The second dimension value is  $NCV = \min(N, \max(\text{FACTOR\_MAXNCV} * NEV, NEV + 1))$ , where the value FACTOR\_MAXNCV is a component of the base class, ARPACKBASE. The first NEV columns of VECTORS(:, :) are the eigenvectors.

### FORTRAN 2003 Interface

Generic: ARPACK\_SYMMETRIC (N, F, VALUES [, ...])

Specific: The specific interface name is D\_ARPACK\_SYMMETRIC.

### FORTRAN 90 Interface

A Fortran 90 compiler version is not available for this routine.

## Description

Routine `ARPACK_SYMMETRIC` calls ARPACK subroutines to compute partial eigenvalue-eigenvector decompositions for symmetric real matrices. The ARPACK routines are `dsaupd` and `dseupd` (see *ARPACK Users' Guide*, SIAM Publications, (1998)), which use “reverse communication” to obtain the required matrix-vector operations for this approximation. Responses to these requests are made by calling the user-written function `F`. By including the class object `EXTYPE` as an argument to this function, user data and procedure pointers are available for the evaluations. A user code must extend the base class `EXTYPE` to include the extra data and procedure pointers.

## Comments

The user function `F` is written to supply requests for the matrix operations. The following pseudo-code outlines the required responses of `F` depending on the circumstances. Only those cases that follow from the settings of `PLACE`, `TYPE` and `CATEGORY` need to be provided in the user code. The enumerated constants, `ARPACK_A_x`, etc., are available by use-association from the module `ARPACK_INT`.

```
FUNCTION F (X, TASK, EXTYPE) RESULT(Y)
USE ARPACK_INT
IMPLICIT NONE

CLASS(ARPACKBASE), INTENT(INOUT) :: EXTYPE
REAL(DKIND), INTENT(INOUT) :: X(:)
INTEGER, INTENT(IN) :: TASK
REAL(DKIND) Y(SIZE(X))

SELECT CASE (TASK)

 CASE (ARPACK_Prepair)
 ...{Take initial steps to prepare for the operations that follow.}
 CASE (ARPACK_A_x)
 ... $y = Ax$
 CASE (ARPACK_B_x)
 ... $y = Bx$
 CASE (ARPACK_inv_A_minus_Shift_x)
 ... $y = (A - \sigma I)^{-1} x$
 CASE (ARPACK_inv_B_x)
 ... $y = B^{-1}x$
 CASE (ARPACK_inv_A_minus_Shift_B_x)
 ... $y = (A - \sigma B)^{-1} x$
 CASE DEFAULT
 ...{This is an error condition. }
END SELECT
END FUNCTION
```

## Example 1

We approximate eigenvalues and eigenfunctions of the Laplacian operator

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \equiv \Delta u, -\Delta u = \lambda u$$

on the unit square,  $[0,1] \times [0,1]$ , with zero Dirichlet boundary values. The full set of eigenvalues and their eigenfunctions are given by the sequence  $\lambda_{m,n} = (m^2 + n^2)\pi^2$ ,  $u_{m,n}(x, y) = \sin(m\pi)\sin(n\pi)$ , where  $m, n$  are positive integers.

This provides a check on the accuracy of the numerical results. Equally spaced divided differences on the unit square are used to approximate  $-\Delta u$ . A few eigenvalues of *smallest magnitude*, and their eigenvectors, are requested. This application requires the optional argument `PLACE=ARPACK_Smallest_Magnitude`. The user function code provides the second order divided difference operator applied to an input vector. The problem is a symmetric matrix eigenvalue computation. It involves only the single `TASK, ARPACK_A_x`, in the user functions.

The function `FCN` defines a grid of values and provides the operation that derives from this eigenvalue problem. The class argument `EXTYPE` must be declared but need not be used. Within the main program, the function interface for the external function `FCN` is specified with the declaration `PROCEDURE (DMV) FCN`.

[Link to example source \(arpack\\_symmetric\\_ex1.f90\)](#)

```

PROGRAM ARPACK_SYMMETRIC_EX1
USE ARPACK_INT
USE UMACH_INT
USE WRRRN_INT
IMPLICIT NONE
! Compute the smallest eigenvalues of a discrete Laplacian,
! based on second order divided differences.
! The matrix used is the 2 dimensional discrete Laplacian on
! the unit square with zero Dirichlet boundary condition.
INTEGER :: J, NOUT
INTEGER, parameter :: NEV=5 !number of Eigenvalues required
INTEGER, parameter :: NV=0, NX=10
INTEGER, parameter :: N=NX**2 !size of matrix problem
REAL(DKIND) :: VALUES(NEV), RES(N), EF(NX, NX)
REAL(DKIND), ALLOCATABLE :: VECTORS(:, :)
REAL(DKIND) :: NORM
LOGICAL :: SMALL, SOLVED
TYPE(ARPACKBASE) :: Q
PROCEDURE (DMV) :: FCN

CALL UMACH(2, NOUT)
! Note that VECTORS(:, :) does not need to be allocated
! in the calling program. That happens within the
! routine ARPACK_SYMMETRIC(). It is OK to do this but
! the sizes (N,NCV) are determined in ARPACK_SYMMETRIC.
CALL ARPACK_SYMMETRIC(N, FCN, VALUES, &
PLACE=ARPACK_Smallest_Magnitude, VECTORS=VECTORS)
WRITE(NOUT, *) 'Number of eigenvalues requested, and declared
accurate'
```

```

WRITE(NOUT, *) '-----'
,
WRITE(NOUT, '(5X, I4, 5X, I4)') NEV, Q%NACC
WRITE(NOUT, *) 'Number of Matrix-Vector Products Recorded, EX-11'
WRITE(NOUT, *) '-----'
WRITE(NOUT, '(5X, I4)') NV
CALL WRRRN('Smallest Laplacian Eigenvalues', VALUES)

! Check residuals, A*vectors = values*vectors:
DO J=1,NEV
! Necessary to have an unused TYPE(ARPACKBASE) :: Q as an argument:
RES=FCN(VECTORS(:,J), ARPACK_A_x, Q)-VALUES(J)*VECTORS(:,J)
NORM=maxval(abs(RES))
SMALL=(NORM <= ABS(VALUES(J))*SQRT(EPSILON(NORM)))
IF(J==1) SOLVED=SMALL
SOLVED=SOLVED .and. SMALL
END DO
IF(SOLVED) THEN
WRITE(nout,'(A///)') &
'All Ritz Values and Vectors have small residuals.'
ELSE
WRITE(nout,'(A///)') &
'Some Ritz Values and Vectors have large residuals.'
ENDIF

! The first eigenvector is scaled to be positive.
! It defines the eigenfunction for the smallest
! eigenvalue at the grid defined by the differencing.
EF=reshape(VECTORS(:,1), (/NX,NX/))
CALL WRRRN('First 2D Laplacian Eigenfunction at Grid Points', EF)

END

FUNCTION FCN(X, TASK, EX)RESULT(Y)
USE ARPACK_INT

CLASS(ARPACKBASE), INTENT(INOUT) :: EX
REAL(DKIND), INTENT(INOUT) :: X(:)
INTEGER, INTENT(IN) :: TASK
REAL(DKIND) Y(SIZE(X))

! Local variables:
INTEGER J
INTEGER, SAVE :: NX
REAL(DKIND), SAVE :: HSQ

SELECT CASE(TASK)
CASE(ARPACK_A_x)
! Computes y <-- A*x, where A is the N**2 by N**2 block
! tridiagonal matrix
!
! | T -I |
! |-I T -I |
! A = | -I T |
! | ... -I|
! | -I T|
! Y(1:NX)=T(NX,X(1:NX)) - X(NX+1:2*NX)

```

```

DO J=NX+1,NX**2-NX,NX
 Y(J:J+NX-1)=T(NX,X(J:J+NX-1)) &
 - X(J-NX:J-1)-X(J+NX:J+2*NX-1)
END DO
Y((NX-1)*NX+1:NX**2) = - X((NX-1)*NX-NX+1:(NX-1)*NX) &
 + T(NX,X((NX-1)*NX+1:NX**2))
! Note that HSQ is passed as a component of the extended type.
Y=(1._DKIND/HSQ)*Y

CASE(ARPACK_Prepare)
! Define NX, 1/H**2 so they are later available in the evaluator.
NX=10 ! This value is fixed in the evaluator.
HSQ = 1._DKIND/REAL(NX+1,DKIND)**2
Y=0._DKIND
CASE DEFAULT
 WRITE(NOUT,*) TASK, ': INVALID TASK REQUESTED'
 STOP 'IMSL_ERROR_WRONG_OPERATION'
END SELECT
CONTAINS
FUNCTION T(NX, X) RESULT(V)
 INTEGER, INTENT(IN) :: NX
 REAL(DKIND), INTENT(IN) :: X(:)
 REAL(DKIND) :: V(NX)
 REAL(DKIND) :: MONE=-1._DKIND, FOUR=4._DKIND
 INTEGER J

 V(1)=FOUR*X(1)+MONE*X(2)
 DO J=2,NX-1
 V(J)=MONE*X(J-1)+FOUR*X(J)+MONE*X(J+1)
 END DO
 V(NX)=MONE*X(NX-1)+FOUR*X(NX)
END FUNCTION
END FUNCTION

```

## Output

Number of eigenvalues requested, and declared accurate

```

5 0
```

Number of Matrix-Vector Products Recorded, EX-11

```

0
```

Smallest Laplacian Eigenvalues

```
1 19.61
2 48.22
3 48.22
4 76.83
5 93.33
```

All Ritz Values and Vectors have small residuals.

First 2D Laplacian Eigenfunction at Grid Points

|    | 1      | 2      | 3      | 4      | 5      | 6      | 7      | 8      |
|----|--------|--------|--------|--------|--------|--------|--------|--------|
| 1  | 0.0144 | 0.0277 | 0.0387 | 0.0466 | 0.0507 | 0.0507 | 0.0466 | 0.0387 |
| 2  | 0.0277 | 0.0531 | 0.0743 | 0.0894 | 0.0973 | 0.0973 | 0.0894 | 0.0743 |
| 3  | 0.0387 | 0.0743 | 0.1038 | 0.1250 | 0.1360 | 0.1360 | 0.1250 | 0.1038 |
| 4  | 0.0466 | 0.0894 | 0.1250 | 0.1504 | 0.1637 | 0.1637 | 0.1504 | 0.1250 |
| 5  | 0.0507 | 0.0973 | 0.1360 | 0.1637 | 0.1781 | 0.1781 | 0.1637 | 0.1360 |
| 6  | 0.0507 | 0.0973 | 0.1360 | 0.1637 | 0.1781 | 0.1781 | 0.1637 | 0.1360 |
| 7  | 0.0466 | 0.0894 | 0.1250 | 0.1504 | 0.1637 | 0.1637 | 0.1504 | 0.1250 |
| 8  | 0.0387 | 0.0743 | 0.1038 | 0.1250 | 0.1360 | 0.1360 | 0.1250 | 0.1038 |
| 9  | 0.0277 | 0.0531 | 0.0743 | 0.0894 | 0.0973 | 0.0973 | 0.0894 | 0.0743 |
| 10 | 0.0144 | 0.0277 | 0.0387 | 0.0466 | 0.0507 | 0.0507 | 0.0466 | 0.0387 |

|    | 9      | 10     |
|----|--------|--------|
| 1  | 0.0277 | 0.0144 |
| 2  | 0.0531 | 0.0277 |
| 3  | 0.0743 | 0.0387 |
| 4  | 0.0894 | 0.0466 |
| 5  | 0.0973 | 0.0507 |
| 6  | 0.0973 | 0.0507 |
| 7  | 0.0894 | 0.0466 |
| 8  | 0.0743 | 0.0387 |
| 9  | 0.0531 | 0.0277 |
| 10 | 0.0277 | 0.0144 |

## Example 2

We approximate eigenvalues and eigenfunctions of the 1D Laplacian operator  $-\frac{d^2u}{dx^2} = \lambda u$  on the unit interval,  $[0, 1]$ . Equally spaced divided differences are used for the operator, which yields a tri-diagonal matrix. The eigenvalues and eigenfunctions are

$\lambda_n = n^2 \pi^2$ ,  $u_n(x) = \sin(n\pi x)$ ,  $n = 1, 2, \dots$ . This example shows that using inverse iteration for approximating the largest *reciprocals* of eigenvalues is more efficient than directly computing the smallest *magnitude* eigenvalues by products of the operator. This requires the optional argument `CATEGORY=ARPACK_Shift_Invert`. The user function, `FCN`, requires the solution of a tri-diagonal system of linear equations applied to an input vector. The base class `ARPACKBASE` is extended to the user's type, `TYPE(ARPACKBASE_EXT)`, defined in the user module `ARPACK_SYMMETRIC_EX2_INT`. This extension includes the number of intervals, a total kept in `FCN` for noting the number of operations, and allocatable arrays used to store the *LU* factorization of the tri-diagonal matrix. When `FCN` is entered with `TASK=ARPACK_Prepare`, these arrays are allocated, defined, and the *LU* factorization of the shifted matrix  $(A - \sigma I)$  is computed, here with  $\sigma = 0$ . When `FCN` is later entered with `TASK=ARPACK_inv_A_minus_Shift_x`, the *LU* factorization is available to efficiently compute  $y = (A - \sigma I)^{-1} x = A^{-1} x$ . The function `FCN` is also entered with `TASK=ARPACK_A_x`, to compute  $Ax$ .

[Link to example source](#) (arpack\_symmetric\_ex2.f90)

```

MODULE ARPACK_SYMMETRIC_EX2_INT
 USE ARPACK_INT
 USE LSLCR_INT
 USE N1RTY_INT

```

```

IMPLICIT NONE

TYPE, EXTENDS(ARPACKBASE) :: ARPACKBASE_EXT
 REAL(DKIND) :: HSQ=0._DKIND
 INTEGER :: NX=0, NV=0
! This example extends the base type to
! information for solving tridiagonal systems.
 REAL(DKIND), ALLOCATABLE :: A(:), B(:), C(:)
 REAL(DKIND), ALLOCATABLE :: Y1(:), U(:)
 INTEGER, ALLOCATABLE :: IR(:), IS(:)
END TYPE ARPACKBASE_EXT

CONTAINS

FUNCTION FCN(X, TASK, EX) RESULT(Y)
 CLASS (ARPACKBASE), INTENT(INOUT) :: EX
 REAL (DKIND), INTENT(INOUT) :: X(:)
 INTEGER, INTENT(IN) :: TASK
 REAL (DKIND) Y(size(X))
 INTEGER J, IERR, IJOB, NSIZE

 SELECT TYPE(EX)
 TYPE IS(ARPACKBASE_EXT)
 ASSOCIATE(N => EX%NCOLS, &
 NV => EX%NV, &
 HSQ => EX%HSQ, &
 SHIFT => EX%SHIFT)
 SELECT CASE(TASK)
 CASE(ARPACK_A_x)
 Y(1) = 2._DKIND*X(1) - X(2)

 DO J = 2,N-1
 Y(J) = - X(J-1) + 2._DKIND*X(J) - X(J+1)
 END DO

 Y(N) = - X(N-1) + 2._DKIND*X(N)
 Y=Y/HSQ
 CASE(ARPACK_inv_A_minus_Shift_x)
! Compute Y=inv(A-*I)*x. This is done with a solve
! step, using the LU factorization. Note that the data
! for the factorization is stored in the user's extended
! data type.

 EX%Y1(1:N) = X
 IJOB = 2
 CALL LSLCR (EX%C, EX%A, EX%B, EX%Y1, EX%U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
 Y(1:N) = EX%Y1(1:N)
 IERR= N1RTY(1)
 IF (IERR==4 .OR. IERR==5) STOP
 END SELECT
 END ASSOCIATE
 END SELECT
 END FUNCTION

! Total number of solve steps.
 NV=NV+1
 CASE(ARPACK_Prepare)
! Set up storage areas for factored tridiagonal matrix.
 IF (ALLOCATED(EX%A)) DEALLOCATE(EX%A)

```

```

 IF (ALLOCATED(EX%B)) DEALLOCATE(EX%B)
 IF (ALLOCATED(EX%C)) DEALLOCATE(EX%C)
 IF (ALLOCATED(EX%Y1)) DEALLOCATE(EX%Y1)
 IF (ALLOCATED(EX%U)) DEALLOCATE(EX%U)
 IF (ALLOCATED(EX%IR)) DEALLOCATE(EX%IR)
 IF (ALLOCATED(EX%IS)) DEALLOCATE(EX%IS)
 NSIZE = (log(dble(N))/log(2.0)) + 5
 ALLOCATE(EX%A(2*N), EX%B(2*N), EX%C(2*N), EX%Y1(2*N), &
 EX%U(2*N), EX%IR(NSIZE), &
 EX%IS(NSIZE), STAT=IERR)
 IF (IERR /= 0) STOP 'IMSL_ERROR_ALLOCATING_WORKSPACE'
! Define matrix values.
 HSQ=1._DKIND/REAL((N+1)**2,DKIND)
 EX%B(1:N) = -1._DKIND/HSQ
 EX%A(1:N) = 2._DKIND/HSQ - SHIFT
 EX%C(1:N) = EX%B(1:N)
 EX%Y1(:) = 0.0_DKIND
! Factor the matrix with LU and partial pivoting.
 IJOB = 3
 CALL LSLCR (EX%C, EX%A, EX%B, EX%Y1, EX%U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
 IERR = N1RTY(1)
 IF(IERR == 4 .or. IERR == 5) STOP 'IMSL FATAL ERROR'
! Give output some values to satisfy compiler.
 Y=0._DKIND
 NV=0
 CASE DEFAULT
 STOP 'IMSL_ERROR_WRONG_OPERATION'
 END SELECT
 END ASSOCIATE
END SELECT
END FUNCTION
END MODULE

USE ARPACK_SYMMETRIC_EX2_INT
USE UMACH_INT
USE WRRRN_INT
IMPLICIT NONE

! Compute the smallest eigenvalues of a discrete Laplacian,
! based on second order divided differences.

! The matrix is the 1 dimensional discrete Laplacian on
! the interval 0,1 with zero Dirichlet boundary condition.

INTEGER, PARAMETER :: NEV=4, N=100
REAL(DKIND) :: VALUES(NEV), RES(N)
REAL(DKIND), ALLOCATABLE :: VECTORS(:, :)
REAL(DKIND) NORM
LOGICAL SMALL, SOLVED
INTEGER J, NOUT
TYPE(ARPACKBASE_EXT) EX

ASSOCIATE(NX => EX%NX, &
 NV => EX%NV, &
 SIGMA => EX%SHIFT)

CALL UMACH(2, NOUT)

```

```

! Note that VECTORS(:, :) does not need to be allocated
! in the calling program. That happens within the
! routine ARPACK_SYMMETRIC(). It is OK to do this but
! the sizes (N,NCV) are determined in ARPACK_SYMMETRIC.
 SIGMA=0._DKIND
 CALL ARPACK_SYMMETRIC(N, FCN, VALUES,&
 CATEGORY=ARPACK_Shift_Invert, EXTYPE=EX, VECTORS=VECTORS)

 WRITE(NOUT,*) 'Number of Matrix-Vector Products Required, EX-2'
 WRITE(NOUT,*) '-----'
 WRITE(NOUT, '(5X, I4)') NV
 CALL WRRRN('Largest Laplacian Eigenvalues Near Zero Shift', &
 VALUES)
! Check residuals, A*vectors = values*vectors:
 DO J=1,NEV
 RES=FCN(VECTORS(:,J),ARPACK_A_x,EX)-VALUES(J)*VECTORS(:,J)
 NORM=maxval(abs(RES))
 SMALL=(NORM <= ABS(VALUES(J))*SQRT(EPSILON(NORM)))
 IF(J==1) SOLVED=SMALL
 SOLVED=SOLVED .and. SMALL
 END DO

 IF(SOLVED) THEN
 WRITE(nout,'(A//)') &
 'All Ritz Values and Vectors have small residuals.'
 ELSE
 WRITE(nout,'(A//)') &
 'Some Ritz Values and Vectors have large residuals.'
 ENDIF
END ASSOCIATE
END

```

## Output

```

Number of Matrix-Vector Products Required, EX-2

 24

Largest Laplacian Eigenvalues Near Zero Shift
 1 9.9
 2 39.5
 3 88.8
 4 157.7

All Ritz Values and Vectors have small residuals.

```

## Example 3

We compute the solution of a generalized problem. This comes from using equally spaced linear finite element test functions to solve eigenvalues and eigenfunctions of the 1D Laplacian

operator  $-\frac{d^2u}{dx^2} = \lambda u$  on the unit interval,  $[0,1]$ . This is [Example 2](#) but solved using finite

elements. With matrix notation, we have the matrix problem  $Ax = \lambda Bx$ . Both  $A$  and  $B$  are tri-

diagonal and symmetric. The matrix  $B$  is non-singular. We compute the smallest *magnitude* eigenvalues. This requires the optional arguments `TYPE = ARPACK_Generalized`, `CATEGORY = ARPACK_Regular_Inverse`, and `PLACE = ARPACK_Smallest_Magnitude`. The user function, `FCN`, requires the solution of a tri-diagonal system of linear equations applied to an input vector,  $y = B^{-1}x$ . The base class `ARPACKBASE` is extended to the user's type, `TYPE(ARPACKBASE_EXT)`, defined in the user module `ARPACK_SYMMETRIC_EX3_INT`. This extension includes the number of intervals, a total kept in `FCN` for noting the number of operations, and allocatable arrays used to store the  $LU$  factorization of  $B$ . When `FCN` is entered with `TASK=ARPACK_Prepare`, these arrays are allocated, defined, and the  $LU$  factorization of the matrix  $B$  is computed. The function `FCN` is entered with the three values `TASK=ARPACK_A_x`, for  $y = Ax$ ; `TASK=ARPACK_B_x`, for  $y = Bx$ ; and `TASK=ARPACK_inv_B_x`, for  $y = B^{-1}x$ .

Within the main program, the function interface for the external function `FCN` is specified with the declaration `PROCEDURE (DMV) FCN`.

[Link to example source \(arpack\\_symmetric\\_ex3.f90\)](#)

```

MODULE ARPACK_SYMMETRIC_EX3_INT
 USE ARPACK_INT
 USE LSLCR_INT
 USE N1RTY_INT
 IMPLICIT NONE

 TYPE, EXTENDS(ARPACKBASE) :: ARPACKBASE_EXT
 REAL(DKIND) :: H=0._DKIND
 INTEGER :: NX=0, NV=0
! This example extends the base type to
! information for solving tridiagonal systems.
 REAL(DKIND), ALLOCATABLE :: A(:), B(:), C(:)
 REAL(DKIND), ALLOCATABLE :: Y1(:), U(:)
 INTEGER, ALLOCATABLE :: IR(:), IS(:)

 END TYPE ARPACKBASE_EXT
END MODULE

PROGRAM ARPACK_SYMMETRIC_EX3
 USE ARPACK_SYMMETRIC_EX3_INT
 USE UMACH_INT
 USE WRRRN_INT
 IMPLICIT NONE

! We want to solve A*x = lambda*M*x in inverse mode,
! where A and M are obtained by the finite element method
! of the 1-dimensional discrete Laplacian
! d^2u / dx^2
! on the interval 0,1, with zero Dirichlet boundary conditions,
! using piecewise linear elements.

 INTEGER, PARAMETER :: NEV=4, N=100
 REAL(DKIND) :: VALUES(NEV), RES(N)
 REAL(DKIND), ALLOCATABLE :: VECTORS(:, :)
 REAL(DKIND) NORM

```

```

LOGICAL :: PRINTRESULTS = .FALSE.
LOGICAL SMALL, SOLVED
INTEGER J, NOUT
PROCEDURE(DMV) FCN
TYPE(ARPACKBASE_EXT) EX

ASSOCIATE(NX => EX%NX, &
 NV => EX%NV)
 EX%FACTOR_MAXNCV=5._DKIND
CALL UMACH(2, NOUT)
! Note that VECTORS(:, :) does not need to be allocated
! in the calling program. That happens within the
! routine ARPACK_SYMMETRIC(). It is OK to do this but
! the sizes (N,NCV) are determined in ARPACK_SYMMETRIC.
CALL ARPACK_SYMMETRIC(N, FCN, VALUES, &
 TYPE=ARPACK_Generalized, &
 CATEGORY=ARPACK_Regular_Inverse, &
 PLACE=ARPACK_Smallest_Magnitude, EXTYPE=EX, VECTORS=VECTORS)

WRITE(NOUT,*) 'Number of Matrix-Vector Products Required, EX-3'
WRITE(NOUT,*) '-----'
WRITE(NOUT, '(5X, I4)') NV
CALL WRRRN('Largest Laplacian Eigenvalues', VALUES)

! Check residuals, A*vectors = values*B*vectors:
DO J=1,NEV
 RES=FCN(VECTORS(:,J), ARPACK_A_x, EX) -&
 VALUES(J)*FCN(VECTORS(:,J), ARPACK_B_x, EX)
 NORM=maxval(abs(RES))
 SMALL=(NORM <= ABS(VALUES(J))*SQRT(EPSILON(NORM)))
 IF(J==1) SOLVED=SMALL
 SOLVED=SOLVED .and. SMALL
END DO

IF(SOLVED) THEN
 WRITE(nout, 'A///') &
 'All Ritz Values and Vectors have small residuals.'
ELSE
 WRITE(nout, 'A///') &
 'Some Ritz Values and Vectors have large residuals.'
ENDIF
END ASSOCIATE
END PROGRAM

FUNCTION FCN(X, TASK, EX) RESULT(Y)
USE ARPACK_SYMMETRIC_EX3_INT
CLASS(ARPACKBASE), INTENT(INOUT) :: EX
REAL(DKIND), INTENT(INOUT) :: X(:)
INTEGER, INTENT(IN) :: TASK
REAL(DKIND) Y(SIZE(X)), PI
INTEGER J, IERR, IJOB, NSIZE

SELECT TYPE(EX)
TYPE IS(ARPACKBASE_EXT)
ASSOCIATE(N => EX%NCOLS, &
 NV => EX%NV, &

```

```

 H => EX%H)

SELECT CASE (TASK)
CASE (ARPACK_A_x)
 Y(1) = 2._DKIND*X(1) - X(2)

 DO J = 2,N-1
 Y(J) = - X(J-1) + 2._DKIND*X(J) - X(J+1)
 END DO

 Y(N) = - X(N-1) + 2._DKIND*X(N)
 Y=Y/H
CASE (ARPACK_B_x)
 Y(1) = 4._DKIND*X(1) + X(2)

 DO J = 2,N-1
 Y(J) = X(J-1) + 4._DKIND*X(J) + X(J+1)
 END DO

 Y(N) = X(N-1) + 4._DKIND*X(N)
 Y=Y*(H/6._DKIND)
CASE (ARPACK_inv_B_x)
! Compute Y=inv(A-*I)*x. This is done with a solve
! step, using the LU factorization. Note that the data
! for the factorization is stored in the user's extended
! data type.
 EX%Y1(1:N) = X
 IJOB = 2
 CALL LSLCR (EX%C, EX%A, EX%B, EX%Y1, EX%U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
 Y(1:N) = EX%Y1(1:N)
 IERR= N1RTY(1)
 IF (IERR==4 .OR. IERR==5) STOP 'IMSL_FATAL_ERROR_SOLVING'
! Total number of solve steps.
 NV=NV+1
CASE (ARPACK_Prep)
! Set up storage areas for factored tridiagonal matrix.

 IF (ALLOCATED(EX%A)) DEALLOCATE (EX%A)
 IF (ALLOCATED(EX%B)) DEALLOCATE (EX%B)
 IF (ALLOCATED(EX%C)) DEALLOCATE (EX%C)
 IF (ALLOCATED(EX%Y1)) DEALLOCATE (EX%Y1)
 IF (ALLOCATED(EX%U)) DEALLOCATE (EX%U)
 IF (ALLOCATED(EX%IR)) DEALLOCATE (EX%IR)
 IF (ALLOCATED(EX%IS)) DEALLOCATE (EX%IS)
 NSIZE = (log(dble(N))/log(2.0d0)) + 5
 ALLOCATE (EX%A(2*N), EX%B(2*N), EX%C(2*N), EX%Y1(2*N), &
 EX%U(2*N), EX%IR(NSIZE), &
 EX%IS(NSIZE), STAT=IERR)
 IF (IERR /= 0) STOP 'IMSL_ERROR_ALLOCATING_WORKSPACE'

! Define matrix values.
 PI=ATAN(1._DKIND)*4._DKIND
 H=PI/REAL(N+1,DKIND)
 EX%B(1:N) = (1._DKIND/6._DKIND)*H
 EX%A(1:N) = (2._DKIND/3._DKIND)*H
 EX%C(1:N) = EX%B(1:N)

```

```

EX%Y1(:) = 0.0_DKIND

! Factor the matrix with LU and partial pivoting.

 IJOB = 3
 CALL LSLCR (EX%C, EX%A, EX%B, EX%Y1, EX%U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
 IERR = N1RTY(1)
 IF(IERR == 4 .or. IERR == 5) STOP 'IMSL FATAL ERROR'
! Give output some values to satisfy compiler.
 Y=0._DKIND
 NV=0
 CASE DEFAULT
 write(*,*)TASK
 STOP 'IMSL_ERROR_WRONG_OPERATION'
 END SELECT
 END ASSOCIATE
 END SELECT
 END FUNCTION

```

## Output

```

Number of Matrix-Vector Products Required, EX-3

 1126

Largest Laplacian Eigenvalues
 1 1.00
 2 4.00
 3 9.01
 4 16.02

All Ritz Values and Vectors have small residuals.

```

---

## ARPACK\_SVD

Computes some singular values and left and right singular vectors of a real rectangular matrix  $A_{M \times N} = USV^T$ . There is no restriction on the relative sizes,  $M$  and  $N$ . This routine calls [ARPACK\\_SYMMETRIC](#).

### Required Arguments

$M$  — The number of matrix rows. (Input)

$N$  — The number of matrix columns. (Input)

$F$  — User-supplied FUNCTION to return matrix-vector operations. This user function is written corresponding to the [abstract interface](#) for the function `SVDMV(...)`. The operations provided as code in the function `F` will be made based on the two matrix operations  $y = Ax$  and  $y = A^T x \equiv x^T A$ . The usage is `F(X, TASK, EXTYPE)`, where

### Function Return Value

*F* — An array of length *N* containing matrix-vector operations or linear equations solutions. Operations provided as code in the function *F* will be made depending upon the value of argument *TASK*.

### Required Arguments

*X* — An array of length *N* containing the vector to which the operator will be applied. (Input)

*TASK* — An enumerated type which specifies the operation to be performed. (Input)

*TASK* is an enumerated integer value, use-associated from the module `ARPACK_INT`. It will be one of the following:

| Value                       | Description                                                                                                                                                                                                                  |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ARPACK_Prepare</code> | Take initial steps to prepare for the operations that follow. These steps can include defining the data for the matrices, factorizations for upcoming linear system solves, or recording the vectors used in the operations. |
| <code>ARPACK_A_x</code>     | $y = Ax$                                                                                                                                                                                                                     |
| <code>ARPACK_xt_A</code>    | $y = Bx$                                                                                                                                                                                                                     |

*EXTYPE* — A derived type of the extensible class `ARPACKBASE`, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a `USE ARPACK_INT` statement in the calling program to define this derived type. If *EXTYPE* is not included as an argument to `ARPACK_SVD` it should be ignored in the user-function, *F*.

The function *F* must be written according to the abstract interface for `SVDMV`. If *F* is not use-associated nor contained in the calling program, declare it with `PROCEDURE(SVDMV) F`.

*SVALUES* — A rank-1 array of singular values. (Output)

The value `NEV = size(SVALUES)` defines the number of singular values to be computed. The calling program declares or allocates the array `SVALUES(1:NEV)`.

### Optional Arguments

*PLACE* — Indicates the placement in the spectrum for required singular values. (Input)

*PLACE* can be one of the following enumerated integers as defined in `ARPACK_INT`:

| Value                     |
|---------------------------|
| ARPACK_Largest_Algebraic  |
| ARPACK_Smallest_Magnitude |

Default: PLACE = ARPACK\_Largest\_Algebraic.

**ITERATION\_TYPE** — Indicates the method for obtaining the required singular values.

(Input)

ITERATION\_TYPE can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value           |
|-----------------|
| ARPACK_Normal   |
| ARPACK_Expanded |

For values  $M \geq N$ , ARPACK\_Normal specifies computing singular values based on eigenvalues and eigenvectors of the *normal* symmetric matrix  $A^T A$ ; for values  $M < N$  this will be the alternate symmetric matrix  $AA^T$ .

For all values of  $M, N$ , ARPACK\_Expanded specifies computing singular values based on the symmetric matrix eigenvalue problem for the matrices

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

Default: ITERATION\_TYPE = ARPACK\_Normal.

**CATEGORY** — An integer from a packaged enumeration with values that are passed to

ARPACK\_SYMMETRIC. (Input)

CATEGORY can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                  |
|------------------------|
| ARPACK_Regular         |
| ARPACK_Regular_Inverse |

Default: CATEGORY = Regular.

**EXTYPE** — A derived type of the extensible class ARPACKBASE, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a `USE ARPACK_INT` statement in the calling program to define this derived type. If `EXTYPE` is not included as an argument to `ARPACK_SVD` it must still be supplied as an argument to user-function, `F`, but is not used.

***U\_VECTORS*** — An allocatable array of orthogonal left singular vectors. (Output)  
It is not necessary to allocate `U_VECTORS(:, :)`. If this argument is present, the allocation occurs within the routine `ARPACK_SVD`. The output sizes are `UVECTORS(1:M, 1:NCV)`. The second dimension value is `NCV=min(M, max(FACTOR_MAXNCV*NEV, NEV+1))`, where the value `FACTOR_MAXNCV` is a component of the base class, `ARPACKBASE`. The first `NEV` columns of `U_VECTORS(:, :)` are the left singular vectors.

***V\_VECTORS*** — An allocatable array of orthogonal right singular vectors. (Output)  
It is not necessary to allocate `V_VECTORS(:, :)`. If this argument is present, the allocation occurs within the routine `ARPACK_SVD`. The output sizes are `V_VECTORS(1:N, 1:NCV)`. The second dimension value is `NCV=min(M, max(FACTOR_MAXNCV*NEV, NEV+1))`, where the value `FACTOR_MAXNCV` is a component of the base class, `ARPACKBASE`. The first `NEV` columns of `V_VECTORS(:, :)` are the right singular vectors.

## **FORTRAN 2003 Interface**

Generic: `ARPACK_SVD (M, N, F, SVALUES [, ...])`

Specific: The specific interface name is `D_ARPACK_SVD`.

## **FORTRAN 90 Interface**

A Fortran 90 compiler version is not available for this routine.

## **Description**

Routine `ARPACK_SVD` calls `ARPACK_SYMMETRIC` to compute partial singular value decompositions for rectangular real matrices. There is no restriction on the relative sizes of the number of rows and columns. A function internal to `ARPACK_SVD` is used in the call to `ARPACK_SYMMETRIC`. The internal function calls the user function, `F`, which provides matrix-vector products of the matrix and an internally generated vector. By including the class object `EXTYPE` as an argument to this function, user data and procedure pointers are available for the evaluations. A user code must extend the base class `EXTYPE` to include the extra data and procedure pointers.

## **Comments**

The user function supplies requests for the matrix operations. Those cases that follow from the settings of `PLACE`, `ITERATION_TYPE` and `CATEGORY` need to be provided in the user code. The enumerated `TASK` constants, `ARPACK_A_x` and `ARPACK_xt_A` are available by use-association from the module `ARPACK_INT`. The sizes of the inputs and outputs, `x`, `y`, switch between the values `n, m`. The values `n, m` are alternated in the base class components `EXTYPE%NCOLS` and `EXTYPE%MROWS`.

The value of `Iteration_Type` may impact the number of iterations required. Generally one expects `Iteration_Type=ARPACK_Normal` (the default) to result in the fewest iterations, and `Iteration_Type=ARPACK_Expanded` to result in singular values with the greatest accuracy.

The output arrays `U_VECTORS(:, :)`, `SVALUES(:)`, and `V_VECTORS(:, :)` allow for reconstruction of an approximation to the matrix  $A$ . This approximation is  $B = USV^T$ . The matrices  $U$ ,  $S$  and  $V$  are available in these respective routine arguments. The terms  $U_{M \times NSV}$  and  $V_{N \times NSV}$  have orthogonal columns,  $U^T U = I_{NSV} = V^T V$ . The diagonal matrix  $S_{NSV \times NSV}$  has its entries in `SVALUES(:)`, ordered from largest to smallest. Use the value  $NSV = \min(\text{size}(\text{SVALUES}), NACC)$ , where  $NACC$  is the number of accurate singular values computed by `ARPACK_SYMMETRIC`. This is the component `EXTYPE%NACC` of the base class `EXTYPE`.

After computing the singular values and right singular vectors by iteration with the normal matrix  $A^T A$ ,  $U$  is computed from the relation  $AV = US$ . The result is then processed with the modified Gram-Schmidt algorithm to assure that  $U$  is orthogonal. When iterating with  $AA^T$  we first compute the left singular vectors  $U$  and then obtain  $V$  by the Gram-Schmidt algorithm. If we use `Iteration_Type=ARPACK_Expanded`,  $U$  and  $V$  are computed simultaneously, and both are orthogonal.

## Example 1

We define the  $M \times N$  matrix  $A = \{a_{i,j}\}$  with entries  $a_{i,j} = i + j$ . This matrix has two non-zero singular values. With the pair of values  $(M, N) = (512, 265)$  and  $(M, N) = (265, 512)$  we obtain the singular decomposition for these rectangular matrices. With each pair we compute the decomposition using the input `Iteration_Type=ARPACK_Normal` and `Iteration_Type=ARPACK_Expanded`. The latter value requires the larger number of iterations. The matrix  $A$  has its storage requirements changed from  $MN$  to the value  $2(M + N + 1)$ . The resulting product  $B = USV^T$ , when rounded to the nearest integer, satisfies  $B = A$ .

The base class `ARPACKBASE` is extended to include an allocatable array, `EXTYPE%A(:, :)`. This is allocated and defined and stores the matrix  $A$ . The matrix operations  $y = Ax$  and

$y = A^T x \equiv x^T A$  are computed with `DGEMV`.

[Link to example source](#) (`arpack_svd_ex1.f90`)

```
MODULE ARPACK_SVD_EX1_INT
USE ARPACK_INT
IMPLICIT NONE

TYPE, EXTENDS(ARPACKBASE) :: ARPACKBASE_EXT
REAL(DKIND), ALLOCATABLE :: A(:, :)
INTEGER :: NV = 0
```

```

 INTEGER :: MM = 0
 INTEGER :: NN = 0
END TYPE ARPACKBASE_EXT
CONTAINS

FUNCTION FCN(X, TASK, EXTYPE) RESULT(Y)
 USE UMACH_INT
 CLASS (ARPACKBASE), INTENT(INOUT) :: EXTYPE
 REAL (DKIND), INTENT(INOUT) :: X(EXTYPE % NCOLS)
 INTEGER, INTENT(IN) :: TASK
 REAL (DKIND) Y(EXTYPE % MROWS)
 INTEGER I, J, NOUT

 CALL UMACH(2, NOUT)
 SELECT TYPE(EXTYPE)
 TYPE IS (ARPACKBASE_EXT)
 ASSOCIATE(M => EXTYPE % MM, &
 N => EXTYPE % NN, &
 A => EXTYPE % A)
 SELECT CASE(TASK)
 CASE(ARPACK_A_x)
! Computes y <-- A*x
 CALL DGEMV('N',M,N,1._DKIND,A,M,X,1,0._DKIND,Y,1)
 EXTYPE % NV = EXTYPE % NV + 1
 CASE(ARPACK_xt_A)

! Computes y <-- A^T*x = x^T * A
 CALL DGEMV('T',M,N,1._DKIND,A,M,X,1,0._DKIND,Y,1)
 EXTYPE % NV = EXTYPE % NV + 1

 CASE(ARPACK_Prepare)
 EXTYPE % NV = 0
 IF(ALLOCATED(EXTYPE % A)) DEALLOCATE(EXTYPE % A)
 ALLOCATE(EXTYPE % A(M,N))

 DO J=1,N
 DO I=1,M
 EXTYPE % A (I,J) = I + J
 END DO
 END DO

 CASE DEFAULT
 WRITE(NOUT,*) TASK, ': INVALID TASK REQUESTED'
 STOP 'IMSL_ERROR_WRONG_OPERATION'
 END SELECT

 END ASSOCIATE
 END SELECT
 END FUNCTION
END MODULE

USE ARPACK_SVD_EX1_INT
USE UMACH_INT
USE WRRRN_INT
! Compute the largest and smallest singular values of a
! patterned matrix.
INTEGER, PARAMETER :: NSV=2
INTEGER :: COUNT, I, J, N, M, nout

```

```

REAL(DKIND) :: SVALUESMax(NSV)
REAL(DKIND), ALLOCATABLE :: SVALUESMin(:)
REAL(DKIND), ALLOCATABLE :: VECTORS(:, :), B(:, :)
REAL(DKIND), ALLOCATABLE :: U_VECTORS(:, :), V_VECTORS(:, :)
REAL(DKIND) NORM
LOGICAL SMALL, SOLVED
TYPE(ARPACKBASE_EXT) EX

ASSOCIATE (M=>EX % MM, &
 N=>EX % NN, &
 NACC=>EX % NACC, &
 TOL =>EX % TOL, &
 MAXMV => EX % MAXMV)

SOLVED = .true.
CALL UMACH(2, NOUT)
! Define size of matrix problem.
N=800
M=600
DO COUNT =1,2
! Some values will not be accurately determined for rank
! deficient problems. This next value drops the number
! requested after every sequence of iterations of this size.
MAXMV=500
CALL ARPACK_SVD(M, N, FCN, SVALUESMax, &
 PLACE=ARPACK_Largest_Algebraic, & !Default
 Iteration_TYPE=ARPACK_Normal, & !Default
 CATEGORY=ARPACK_Regular, & !Default
 EXTYPE=EX, U_VECTORS=U_VECTORS, &
 V_VECTORS=V_VECTORS)
CALL WRRRN('Largest Singular values, Normal Method', &
 SVALUESMax)
WRITE(NOUT, *) 'Number of matrix-vector products'
WRITE(NOUT, *) '-----'
WRITE(NOUT, '(5X, I4)') EX % NV
IF(ALLOCATED(B)) DEALLOCATE(B)
ALLOCATE (B(M,N))
! Reconstruct an approximation to A, B = U * S * V ^T.
! Use only the singular values accurately determined.

DO I=1,NACC
 U_VECTORS(:,I)=U_VECTORS(:,I)*SVALUESMax(I)
END DO
B=matmul(U_VECTORS(:,1:NACC),transpose(V_VECTORS(:,1:NACC)))
! Truncate the approximation to nearest integers.
! Subtract known integer matrix and check agreement with
! the approximation.
DO I=1,M
 DO J=1,N
 B(I,J)=REAL(NINT(B(I,J)),DKIND)
 B(I,J)=B(I,J)-EX % A(I,J)
 END DO
END DO
WRITE(NOUT, '(/A, I6)') &
'Number of singular values, S and columns of U,V =', NACC
WRITE(NOUT, '(/A, F6.2)') &
'Integer units of error with U,V and S =', maxval(B)

```

```

 if (maxval(B) > 0.0d0) then
 solved = .false.
 else
 solved = solved .and. .true.
 end if
 SVALUESMax=0._DKIND
! Do same SVD with the Expanded form of the symmetric matrix.
 CALL ARPACK_SVD(M, N, FCN, SVALUESMax,&
 PLACE=ARPACK_Largest_Algebraic, & !Default
 Iteration_TYPE=ARPACK_Expanded, &
 CATEGORY=ARPACK_Regular, & !Default
 EXTYPE=EX, U_VECTORS=U_VECTORS, &
 V_VECTORS=V_VECTORS)

 CALL WRRRN('Largest Singular values, Expanded Method', SVALUESMax)
 WRITE(NOUT, *) 'Number of matrix-vector products'
 WRITE(NOUT, *) '-----'
 WRITE(NOUT, '(5X,I4)') EX % NV
! Reconstruct an approximation to A, B = U * S * V ^T.
! Use only the singular values accurately determined.
 DO I=1,NACC
 U_VECTORS(:,I)=U_VECTORS(:,I)*SVALUESMax(I)
 END DO
 B=matmul(U_VECTORS(:,1:NACC),transpose(V_VECTORS(:,1:NACC)))
! Truncate the approximation to nearest integers.
! Subtract known integer matrix and check agreement with
! the approximation.
 DO I=1,M
 DO J=1,N
 B(I,J)=REAL(NINT(B(I,J)),DKIND)
 B(I,J)=B(I,J)-EX % A(I,J)
 END DO
 END DO
 WRITE(NOUT,'(A,I6)')&
 'Number of singular values, S and columns of U,V =', NACC
 WRITE(NOUT,'(A,F6.2)')&
 'Integer units of error with U,V and S =', maxval(B)
 if (maxval(B) > 0.0d0) then
 solved = .false.
 else
 solved = solved .and. .true.
 end if

 M=800
 N=600
 DEALLOCATE(U_VECTORS, V_VECTORS)
 END DO

 END ASSOCIATE
END

```

## Output

```

Largest Singular values, Normal Method
 1 523955.7
 2 36644.2

```

```

Number of matrix-vector products

 12

Number of singular values, S and columns of U,V = 2

Integer units of error with U,V and S = 0.00

Largest Singular values, Expanded Method
 1 523955.7
 2 36644.2
Number of matrix-vector products

 22

Number of singular values, S and columns of U,V = 2
Integer units of error with U,V and S = 0.00

Largest Singular values, Normal Method
 1 523955.7
 2 36644.2
Number of matrix-vector products

 12

Number of singular values, S and columns of U,V = 2

Integer units of error with U,V and S = 0.00

Largest Singular values, Expanded Method
 1 523955.7
 2 36644.2
Number of matrix-vector products

 18

Number of singular values, S and columns of U,V = 2
Integer units of error with U,V and S = 0.00

```

---

## ARPACK\_NONSYMMETRIC

Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = \lambda Bx$ . This can be used for the case  $B = I$ . The values for  $A, B$  are real, but eigenvalues may be complex and occur in conjugate pairs.

### Required Arguments

$N$  — The dimension of the problem. (Input)

$F$  — User-supplied FUNCTION to return matrix-vector operations or linear solutions. This user function is written corresponding to the abstract interface for the function DMV(...). The usage is F (X, TASK, EXTYPE), where

### Function Return Value

**F** — An array of length  $N$  containing matrix-vector operations or linear equations solutions. Operations provided as code in the function **F** will be made depending upon the value of argument **TASK**.

### Required Arguments

**X** — An array of length  $N$  containing the vector to which the operator will be applied. (Input)

**TASK** — An enumerated type which specifies the operation to be performed. (Input)

**TASK** is an enumerated integer value, use-associated from the module **ARPACK\_INT**. It will be one of the following:

| Value                        | Description                                                                                                                                                                                                                  |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARPACK_Prepare               | Take initial steps to prepare for the operations that follow. These steps can include defining the data for the matrices, factorizations for upcoming linear system solves, or recording the vectors used in the operations. |
| ARPACK_A_x                   | $y = Ax$                                                                                                                                                                                                                     |
| ARPACK_B_x                   | $y = Bx$                                                                                                                                                                                                                     |
| ARPACK_inv_A_minus_Shift_x   | $y = (A - \sigma I)^{-1} x$                                                                                                                                                                                                  |
| ARPACK_inv_B_x               | $y = B^{-1} x$                                                                                                                                                                                                               |
| ARPACK_inv_A_minus_Shift_B_x | $y = (A - \sigma B)^{-1} x$                                                                                                                                                                                                  |

**EXTYPE** — A derived type of the extensible class **ARPACKBASE**, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a `USE ARPACK_INT` statement in the calling program to define this derived type. If **EXTYPE** is not included as an argument to **ARPACK\_NONSYMMETRIC** it should be ignored in the user-function, **F**.

The function **F** must be written according to the abstract interface for **DMV**. If **F** is not use-associated nor contained in the calling program, declare it with `PROCEDURE(DMV) F`.

**ZVALUES** — A complex array of eigenvalues. (Output)

The value `NEV=size(ZVALUES)` defines the number of eigenvalues to be computed.

The calling program declares or allocates the array `ZVALUES(1:NEV)`. The size value

NEV should account for pairs of complex conjugates. The number of eigenvalues computed accurately is optionally available as the component EXTTYPE%NACC of the base class EXTYPE.

## Optional Arguments

**PLACE** — Defines the placement in the spectrum for required eigenvalues. (Input)

PLACE can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                      |
|----------------------------|
| ARPACK_Largest_Magnitude   |
| ARPACK_Smallest_Magnitude  |
| ARPACK_Largest_Real_Parts  |
| ARPACK_Smallest_Real_Parts |
| ARPACK_Largest_Imag_Parts  |
| ARPACK_Smallest_Imag_Parts |

Default: ARPACK\_Largest\_Magnitude.

**TYPE** — Defines the eigenvalue problem as either a standard or generalized eigenvalue problem. (Input)

TYPE can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value              | Description       |
|--------------------|-------------------|
| ARPACK_Standard    | $Ax = \lambda x$  |
| ARPACK_Generalized | $Ax = \lambda Bx$ |

Default: TYPE = ARPACK\_Standard.

**CATEGORY** — CATEGORY and TYPE define the operation sequence provided in the user-written function. (Input)

CATEGORY can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                  | Description                                                  |
|------------------------|--------------------------------------------------------------|
| ARPACK_Regular         | $Ax = \lambda x$                                             |
| ARPACK_Regular_Inverse | $y = Ax, y = Bx, y = B^{-1}x$                                |
| ARPACK_Shift_Invert    | $y = Ax, y = (A - \sigma I)^{-1}x, y = (A - \sigma B)^{-1}x$ |

| Value                            | Description                                                                                                                                                                            |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARPACK_Regular                   | $Ax = \lambda x$                                                                                                                                                                       |
| ARPACK_Complex_Part_Shift_Invert | $y = Ax, y = (A - \sigma I)^{-1} x, y = (A - \sigma B)^{-1} x$<br>$y = Ax, y = Bx, y = \text{Im}\{(A - \sigma B)^{-1} x\}$<br>$y = Ax, y = Bx, y = \text{Re}\{(A - \sigma B)^{-1} x\}$ |

Default: CATEGORY = ARPACK\_Regular.

**EXTYPE** — A derived type of the extensible class ARPACKBASE, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a USE ARPACK\_INT statement in the calling program to define this derived type. If EXTYPE is not included as an argument to ARPACK\_NONSYMMETRIC it must still be supplied as an argument to user-function, F, but is not used.

**VECTORS** — An allocatable array of approximate eigenvectors. (Output)

It is not necessary to allocate VECTORS(:, :). If this argument is used the allocation occurs within the routine ARPACK\_NONSYMMETRIC. The output sizes are VECTORS(1:N, 1:NCV). The second dimension value is  $NCV = \min(N, \max(\text{FACTOR\_MAXNCV} * NEV, NEV + 1))$ , where the value FACTOR\_MAXNCV is a component of the base class, ARPACKBASE. The first NEV columns of VECTORS(:, :) represent the eigenvectors (see [Comments](#)).

## FORTRAN 2003 Interface

Generic: ARPACK\_NONSYMMETRIC (N, F, ZVALUES [, ...])

Specific: The specific interface name is D\_ARPACK\_NONSYMMETRIC.

## FORTRAN 90 Interface

A Fortran 90 compiler version is not available for this routine.

## Description

Routine ARPACK\_NONSYMMETRIC calls ARPACK subroutines to compute partial eigenvalue-eigenvector decompositions for real matrices. The ARPACK routines are dnaupd and dneupd (see *ARPACK Users' Guide*, SIAM Publications, (1998)), which use “reverse communication” to obtain the required matrix-vector operations for this approximation. Responses to these requests are made by calling the user-written function F. By including the class object EXTYPE as an argument to this function, user data and procedure pointers are available for the evaluations. A user code must extend the base class EXTYPE to include the extra data and procedure pointers.

## Comments

The non-symmetric problem may have complex eigenvalues that occur in conjugate pairs, and the eigenvectors are returned in the `REAL (DKIND)` array `VECTORS(:, :)` but with a compact representation: If the eigenvalue  $\lambda_j$  has an imaginary part with a negative value, construct the complex eigenvector from the relation  $w_j = v_j + iv_{j+1}$ . The real vectors  $v_j, v_{j+1}$  are consecutive columns of the array `VECTORS(:, :)`. The eigenvalue-eigenvector relationship is  $Aw_j = \lambda_j w_j$ . Since  $A$  is real,  $\bar{\lambda}_j$  is also an eigenvalue; thus the conjugate relationship  $A\bar{w}_j = \bar{\lambda}_j \bar{w}_j$  will hold. For purposes of checking results the complex residual  $r_j = Aw_j - \lambda_j w_j$  should be small in norm relative to the norm of  $A$ . If that is true, there is no need to check the alternate relationship. This compact representation of the eigenvectors can be expanded to require twice the storage requirements, but that is not done here in the interest of saving large blocks of storage.

For the generalized eigenvalue problem  $Ax = \lambda Bx$  the eigenvalues are optionally computed based on the Raleigh Quotient. Because of the shifts used, only the eigenvectors may be computed. The eigenvalues are returned by solving  $Aw_j = \lambda Bw_j$  for  $\lambda$ :

$$\lambda_j = (w_j^H Aw_j) / (w_j^H Bw_j). \quad \lambda_j \text{ is valid if the denominator is non-zero. If } \lambda_j \text{ has a non-}$$

zero imaginary part, then the complex conjugate  $\bar{\lambda}_j$  is also an eigenvalue. The Raleigh Quotient for eigenvalues of generalized problems is used when vectors are requested and the user has requested it be used with the base class component `EXTYPE%RALEIGH_QUOTIENT == .TRUE.`. This is the component's default value.

## Example 1

We solve the generalized eigenvalue problem  $Ax = \lambda Bx$  using the shift-invert category. The matrix  $A$  is tri-diagonal with the values 2 on the diagonal, -2 on the sub-diagonal, and 3 on the super-diagonal. The matrix  $B$  is tri-diagonal with the values 4 on the diagonal and 1 on the off-diagonals. We use the complex shift  $\sigma = 0.4 + 0.6i$  and increase the factor for the number of Ritz vectors from 2.5 to 5. Two strategies of shift-invert are illustrated,

$y = \text{Re}(A - \sigma B)^{-1} Bx$  and  $y = \text{Im}(A - \sigma B)^{-1} Bx$ . In each case `NEV=6` eigenvalues are obtained, each with 3 pairs of complex conjugate values.

[Link to example source](#) (arpack\_nonsymmetric\_ex1.f90)

```
MODULE ARPACK_NONSYMMETRIC_EX1_INT
USE ARPACK_INT
USE LSLCQ_INT
USE N1RTY_INT
IMPLICIT NONE

TYPE, EXTENDS(ARPACKBASE) :: ARPACKBASE_EXT
```

```

 INTEGER :: NX=0
 INTEGER :: NV=0
! This example extends the base type to
! information for solving complex tridiagonal systems.
 COMPLEX(DKIND), ALLOCATABLE :: A(:), B(:), C(:)
 INTEGER, ALLOCATABLE :: IR(:), IS(:)
! This controls the type of shifting. When
! the value is 1, use real part of inv(A-*M)*x.
! If value is 2, use imaginary part of same.
 INTEGER :: SHIFT_STRATEGY=1
 END TYPE ARPACKBASE_EXT

CONTAINS
FUNCTION FCN(X, TASK, EX) RESULT(Y)
 USE UMACH_INT

 CLASS (ARPACKBASE), INTENT(INOUT) :: EX
 REAL (DKIND), INTENT(INOUT) :: X(:)
 INTEGER, INTENT(IN) :: TASK
 INTEGER, PARAMETER :: NSIZE=12
 REAL(DKIND) Y(size(X))
 REAL(DKIND) DL, DD, DU
 COMPLEX(DKIND) Z(2*size(X))
 REAL(DKIND) U(2*size(X))
 INTEGER J, IERR, NOUT, IJOB
 CALL UMACH(2, NOUT)
 SELECT TYPE (EX)
 TYPE IS (ARPACKBASE_EXT)
 ASSOCIATE (N => EX % NCOLS, &
 NV => EX % NV, &
 SHIFT => EX % SHIFT)

 SELECT CASE (TASK)

 CASE (ARPACK_A_x)
 DL = -2._DKIND
 DD = 2._DKIND
 DU = 3._DKIND
 Y(1) = DD*X(1) + DU*X(2)
 DO J = 2,N-1
 Y(J) = DL*X(J-1) + DD*X(J) + DU*X(J+1)
 END DO
 Y(N) = DL*X(N-1) + DD*X(N)
 NV=NV+1

 CASE (ARPACK_B_x)
 Y(1) = 4._DKIND*X(1) + X(2)
 DO J = 2,N-1
 Y(J) = X(J-1) + 4._DKIND*X(J) + X(J+1)
 END DO
 Y(N) = X(N-1) + 4._DKIND*X(N)
 NV=NV+1

 CASE (ARPACK_inv_A_minus_Shift_B_x)
! Compute Y=REAL/AIMAG(inv(A-*B)*x). This is done with a solve
! step, using the LU factorization. Note that the data
! for the factorization is stored in the user's extended

```

```

! data type.
Z=CMPLX(X,0._DKIND,DKIND)
IJOB = 2
CALL LSLCQ (EX%C, EX%A, EX%B, Z, U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
IERR= N1RTY(1)
IF (IERR==4 .OR. IERR==5) &
 STOP 'IMSL_FATAL_ERROR_SOLVING'

IF(EX % SHIFT_STRATEGY == 1) THEN
 Y(1:N)=REAL(Z(1:N),DKIND)
ELSE IF (EX % SHIFT_STRATEGY == 2) THEN
 Y(1:N)=AIMAG(Z(1:N))
END IF

! Total number of solve steps.
NV=NV+1

CASE(ARPACK_Prepare)
! Set up storage areas for factored tridiagonal matrix.

IF (ALLOCATED(EX%A)) DEALLOCATE(EX%A)
IF (ALLOCATED(EX%B)) DEALLOCATE(EX%B)
IF (ALLOCATED(EX%C)) DEALLOCATE(EX%C)
IF (ALLOCATED(EX%IR)) DEALLOCATE(EX%IR)
IF (ALLOCATED(EX%IS)) DEALLOCATE(EX%IS)
ALLOCATE(EX%A(2*N), EX%B(2*N), EX%C(2*N), &
 EX%IR(NSIZE), EX%IS(NSIZE), STAT=IERR)
IF (IERR /= 0) STOP 'IMSL_ERROR_ALLOCATING_WORKSPACE'

! Define matrix, A-SHIFT*B.
EX % B(1:N) = -2._DKIND-SHIFT
EX % A(1:N) = 2._DKIND-4._DKIND*SHIFT
EX % C(1:N) = 3._DKIND-SHIFT
! Factor the matrix with LU and partial pivoting.
IJOB = 3
CALL LSLCQ (EX%C, EX%A, EX%B, Z, U, &
 EX%IR, EX%IS, N=N, IJOB=IJOB)
IERR = N1RTY(1)
IF(IERR == 4 .or. IERR == 5) STOP 'IMSL_FATAL_ERROR'

! Give output some ZVALUES to satisfy compiler.
Y=0._DKIND
NV=0

CASE DEFAULT
WRITE(NOUT,*) TASK, ': INVALID OPERATION REQUESTED'
STOP 'IMSL_ERROR_WRONG_OPERATION'
END SELECT
END ASSOCIATE
END SELECT
END FUNCTION
END MODULE

```

```

! Suppose we want to solve $A*x = \lambda*B*x$ in -invert mode
! The matrix A is the tridiagonal matrix with 2 on the diagonal,
! -2 on the subdiagonal and 3 on the superdiagonal. The matrix
! is the tridiagonal matrix with 4 on the diagonal and 1 on the
! off-diagonals.
! The sigma is a complex number (sigmar, sigmai).
! OP = Real_Part{invA-(SIGMAR,SIGMAI)*B*B.

USE ARPACK_NONSYMMETRIC_EX1_INT
USE UMACH_INT
USE WRCRN_INT

INTEGER, PARAMETER :: NEV=6, N=100
COMPLEX(DKIND) :: ZVALUES(NEV), RES(N), U(N), V(N), W(N)
REAL(DKIND), ALLOCATABLE :: VECTORS(:, :)
REAL(DKIND) NORM
LOGICAL SKIP, SMALL, SOLVED
INTEGER J, STRATEGY, NOUT
CHARACTER(LEN=12) TAG
CHARACTER(LEN=60) TITLE
TYPE(ARPACKBASE_EXT) EX

ASSOCIATE(NX => EX % NX, &
 NV => EX % NV, &
 SHIFT => EX % SHIFT, &
 FACTOR => EX % FACTOR_MAXNCV, &
 NACC => EX % NACC)

! Note that VECTORS(:, :) does not need to be allocated
! in the calling program. That happens within the
! routine ARPACK_NONSYMMETRIC(). It is OK to do this but
! the sizes (N,NCV) are determined in ARPACK_NONSYMMETRIC.

CALL UMACH(2, NOUT)
SOLVED=.TRUE.
DO STRATEGY=1,2
 SHIFT=CMPLX(0.4_DKIND,0.6_DKIND,DKIND)
 FACTOR=5._DKIND
 EX % SHIFT_STRATEGY=STRATEGY

 CALL ARPACK_NONSYMMETRIC(N, FCN, ZVALUES, &
 TYPE=ARPACK_Generalized, &
 CATEGORY=ARPACK_Complex_Part_Shift_Invert, &
 EXTYPE=EX, VECTORS=VECTORS)

 WRITE(NOUT, *) &
 'Number of Matrix-Vector Products Required, NS Ex-1'
 WRITE(NOUT, *) &
 '-----'
 WRITE(NOUT, '(5X,I4)') NV
 WRITE(NOUT, *) 'Number of accurate values determined'
 WRITE(NOUT, *) '-----'
 WRITE(NOUT, '(5X, I4)') NACC
! Check residuals, A*vectors = ZVALUES*M*vectors:

 SKIP=.FALSE.
 DO J=1,NACC
 IF(SKIP) THEN

```

```

 SKIP=.FALSE.
 CYCLE
 END IF
! The eigenvalue is complex and the pair of vectors
! for the complex eigenvector is returned.
 IF(AIMAG(ZVALUES(J)) /= 0._DKIND) THEN
! Make calls for real and imaginary parts of eigenvectors
! applied to the operators A, M
 U=CMPLX(FCN(VECTORS(:,J),ARPACK_A_x,EX), &
 FCN(VECTORS(:,J+1),ARPACK_A_x,EX),DKIND)
 V=CMPLX(FCN(VECTORS(:,J),ARPACK_B_x,EX), &
 FCN(VECTORS(:,J+1),ARPACK_B_x,EX),DKIND)
! Since the matrix is real, there is an additional conjugate:
 RES=U-ZVALUES(J)*V
 SKIP=.TRUE.
 ELSE
! The eigenvalue is real and the real eigenvector is returned.

 RES=FCN(VECTORS(:,J),ARPACK_A_x,EX)-ZVALUES(J)*&
 FCN(VECTORS(:,J),ARPACK_B_x,EX)
 END IF
 NORM=maxval(abs(RES))
 SMALL=(NORM <= ABS(ZVALUES(J))*SQRT(EPSILON(NORM)))
 SOLVED=SOLVED .and. SMALL
 END DO
 IF(STRATEGY==1) TAG='REAL SHIFT'
 IF(STRATEGY==2) TAG='IMAG SHIFT'

 TITLE = 'Largest Raleigh Quotient Eigenvalues,'//TAG
 CALL WRCRN(TITLE, ZVALUES)

 IF(SOLVED) THEN
 WRITE(NOUT,'(A///)') &
 'All Ritz Values and Vectors have small residuals.'
 ELSE
 WRITE(NOUT,'(A///)') &
 'Some Ritz Values and Vectors have large residuals.'
 END IF
 END DO ! Shift strategy
 END ASSOCIATE

END

```

## Output

Number of Matrix-Vector Products Required, NS Ex-1

```

 280
Number of accurate values determined

 6

```

```

Largest Raleigh Quotient Eigenvalues,REAL SHIFT
 1 (0.5000,-0.5958)
 2 (0.5000, 0.5958)

```

```

3 (0.5000,-0.6331)
4 (0.5000, 0.6331)
5 (0.5000, 0.5583)
6 (0.5000,-0.5583)

```

All Ritz Values and Vectors have small residuals.

Number of Matrix-Vector Products Required, NS Ex-1

```

248
Number of accurate values determined

6

```

Largest Raleigh Quotient Eigenvalues, IMAG SHIFT

```

1 (0.5000,-0.5958)
2 (0.5000, 0.5958)
3 (0.5000,-0.5583)
4 (0.5000, 0.5583)
5 (0.5000,-0.6331)
6 (0.5000, 0.6331)

```

All Ritz Values and Vectors have small residuals.

---

## ARPACK\_COMPLEX

Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = \lambda Bx$ . This can be used for the case  $B = I$ . The values for  $A, B$  are real or complex. When the values are complex the eigenvalues may be complex and are not expected to occur in complex conjugate pairs.

### Required Arguments

$N$  — The dimension of the problem. (Input)

$F$  — User-supplied FUNCTION to return matrix-vector operations or linear solutions. This user function is written corresponding to the [abstract interface](#) for the function ZMV (...). The usage is F (X, TASK, EXTYPE), where

### Function Return Value

$F$  — An array of length  $N$  containing matrix-vector operations or linear equations solutions. Operations provided as code in the function  $F$  will be made depending upon the value of argument  $TASK$ .

### Required Arguments

$X$  — An array of length  $N$  containing the vector to which the operator will be applied. (Input)

$TASK$  — An enumerated type which specifies the operation to be performed. (Input)

TASK is an enumerated integer value, use-associated from the module ARPACK\_INT. It will be one of the following:

| Value          | Description                                                                                                                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARPACK_Prepare | Take initial steps to prepare for the operations that follow. These steps can include defining the data for the matrices, factorizations for upcoming linear system solves, or recording the vectors used in the operations. |
| ARPACK_A_x     | $y = Ax$                                                                                                                                                                                                                     |
| ARPACK_xt_A    | $y = Bx$                                                                                                                                                                                                                     |

**EXTYPE** — A derived type of the extensible class ARPACKBASE, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a USE ARPACK\_INT statement in the calling program to define this derived type. If EXTYPE is not included as an argument to ARPACK\_COMPLEX it should be ignored in the user-function, F.

The function F must be written according to the abstract interface for ZMV. If F is not use-associated nor contained in the calling program, declare it with PROCEDURE(ZMV) F.

**ZVALUES** — A complex array of eigenvalues. (Output)

The value NEV=size(ZVALUES) defines the number of eigenvalues to be computed. The calling program declares or allocates the array ZVALUES(1:NEV). The number of eigenvalues computed accurately is optionally available as the component EXTYPE%NACC of the base class EXTYPE.

## Optional Arguments

**PLACE** — Defines the output content of VALUES. (Input)

PLACE specifies the placement within the spectrum for the required eigenvalues. PLACE can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                      |
|----------------------------|
| ARPACK_Largest_Magnitude   |
| ARPACK_Smallest_Magnitude  |
| ARPACK_Largest_Real_Parts  |
| ARPACK_Smallest_Real_Parts |
| ARPACK_Largest_Imag_Parts  |

| Value                      |
|----------------------------|
| ARPACK_Largest_Magnitude   |
| ARPACK_Smallest_Imag_Parts |

Default: PLACE = ARPACK\_Largest\_Magnitude.

**TYPE** — Defines the eigenvalue problem as either a standard or generalized eigenvalue problem. (Input)

TYPE can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value              | Description       |
|--------------------|-------------------|
| ARPACK_Standard    | $Ax = \lambda x$  |
| ARPACK_Generalized | $Ax = \lambda Bx$ |

Default: TYPE = ARPACK\_Standard.

**CATEGORY** — CATEGORY and TYPE define the operation sequence provided in the user-written function. (Input)

CATEGORY can be one of the following enumerated integers as defined in ARPACK\_INT:

| Value                            | Description                                                                                                        |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------|
| ARPACK_Regular                   | $y = Ax$                                                                                                           |
| ARPACK_Regular_Inverse           | $y = Ax, y = Bx, y = B^{-1}x$                                                                                      |
| ARPACK_Shift_Invert              | $y = Ax, y = (A - \sigma I)^{-1}x, y = (A - \sigma B)^{-1}x$                                                       |
| ARPACK_Complex_Part_Shift_Invert | $y = Ax, y = Bx, y = \text{Re}\{(A - \sigma B)^{-1}x\}$<br>$y = Ax, y = Bx, y = \text{Im}\{(A - \sigma B)^{-1}x\}$ |

Default: CATEGORY = ARPACK\_Regular.

**EXTYPE** — A derived type of the extensible class ARPACKBASE, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The user must include a USE ARPACK\_INT statement in the calling program to define this derived type. If EXTYPE is not included as an argument to ARPACK\_COMPLEX it must still be supplied as an argument to user-function, F, but is not used.

**VECTORS** — An allocatable array of approximate eigenvectors. (Output)

It is not necessary to allocate `VECTORS(:, :)`. If this argument is used the allocation occurs within the routine `ARPACK_NONSYMMETRIC`. The output sizes are `VECTORS(1:N, 1:NCV)`. The second dimension value is `NCV=min(N, max(FACTOR_MAXNCV*NEV, NEV+1))`, where the value `FACTOR_MAXNCV` is a component of the base class, `ARPACKBASE`. The first `NEV` columns of `VECTORS(:, :)` represent the eigenvectors  $w_j$  (see [Comments](#)).

## FORTRAN 2003 Interface

Generic: `ARPACK_COMPLEX (N, F, ZVALUES [, ...])`

Specific: The specific interface name is `Z_ARPACK_COMPLEX`.

## FORTRAN 90 Interface

A Fortran 90 compiler version is not available for this routine.

## Description

Routine `ARPACK_COMPLEX` calls `ARPACK` subroutines to compute partial eigenvalue-eigenvector decompositions for complex matrices. The `ARPACK` routines are `dzaupd` and `dzeupd` (see *ARPACK Users' Guide*, SIAM Publications, (1998)), which use “reverse communication” to obtain the required matrix-vector operations for this approximation. Responses to these requests are made by calling the user-written function `F`. By including the class object `EXTYPE` as an argument to this function, user data and procedure pointers are available for the evaluations. A user code must extend the base class `EXTYPE` to include the extra data and procedure pointers.

## Comments

For purposes of checking results the complex residual  $r_j = Aw_j - \lambda_j w_j$  should be small in norm relative to the norm of  $A$ . For the generalized eigenvalue problem  $Ax = \lambda Bx$  the eigenvalues are optionally computed based on the Raleigh Quotient. Because of the shifts used, only the eigenvectors may be computed. The eigenvalues are returned based on solving  $Aw_j = \lambda Bw_j$  for  $\lambda_j$ ,

where

$$\lambda_j = \left( w_j^H Aw_j \right) / \left( w_j^H Bw_j \right)$$

The eigenvalue  $\lambda_j$  is finite and valid if the denominator is non-zero. The Raleigh Quotient for eigenvalues of generalized problems is used only when vectors are requested and the user has requested it be used with the base class component `EXTYPE%RALEIGH_QUOTIENT = .TRUE.`. This is the component's default value.

## Example 1

This example is a variation of the first [example](#) for ARPACK\_SYMMETRIC. We approximate eigenvalues and eigenfunctions of the Laplacian operator

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \equiv \Delta u, \quad -\Delta u + \rho \frac{\partial u}{\partial x} = \lambda u,$$

on the unit square,  $[0,1] \times [0,1]$ . But now the parameter  $\rho$  is complex. Thus the eigenvalues and eigenfunctions are complex.

[Link to example source](#)

(arpack\_complex\_ex1.f90)

```
MODULE ARPACK_COMPLEX_EX1_INT
USE ARPACK_INT
IMPLICIT NONE

TYPE, EXTENDS(ARPACKBASE), PUBLIC :: ARPACKBASE_EXT
REAL(DKIND) :: H =0._DKIND
REAL(DKIND) :: HSQ=0._DKIND
COMPLEX(DKIND) :: RHO=(0._DKIND,0._DKIND)
COMPLEX(DKIND) :: DL=(0._DKIND,0._DKIND)
COMPLEX(DKIND) :: DD=(0._DKIND,0._DKIND)
COMPLEX(DKIND) :: DU=(0._DKIND,0._DKIND)
INTEGER :: NX=0
INTEGER :: NV=0
END TYPE ARPACKBASE_EXT
CONTAINS
FUNCTION FZ1(X, TASK, EXTYPE) RESULT(Y)
USE UMACH_INT

CLASS(ARPACKBASE), INTENT(INOUT) :: EXTYPE
COMPLEX(DKIND), INTENT(INOUT) :: X(:)
INTEGER, INTENT(IN) :: TASK
COMPLEX(DKIND) Y(size(X))
COMPLEX(DKIND) DT(3)
REAL(DKIND) :: ONE=1._DKIND
INTEGER J, NOUT

CALL UMACH(2, NOUT)
SELECT TYPE(EXTYPE)
TYPE IS(ARPACKBASE_EXT)
ASSOCIATE(NX => EXTYPE % NX, &
 H => EXTYPE % H, &
 HSQ => EXTYPE % HSQ, &
 RHO => EXTYPE % RHO, &
 DL => EXTYPE % DL, &
 DD => EXTYPE % DD, &
 DU => EXTYPE % DU, &
 NV => EXTYPE % NV)

SELECT CASE(TASK)
CASE(ARPACK_A_x)
! Computes y <-- A*x, where A is the N**2 by N**2 block
```

```

! tridiagonal matrix deriving from (Laplacian u) + rho*(du/dx).

 DT=(/DL,DD,DU/)
 Y(1:NX)=T(NX,X(1:NX),DT) - X(NX+1:2*NX)/HSQ
 DO J=NX+1,NX**2-NX,NX
 Y(J:J+NX-1)=T(NX,X(J:J+NX-1),DT) &
 - (X(J-NX:J-1)+ X(J+NX:J+2*NX-1))/HSQ
 END DO
 Y((NX-1)*NX+1:NX**2)= - X((NX-1)*NX-NX+1:(NX-1)*NX) &
 / HSQ + T(NX,X((NX-1)*NX+1:NX**2),DT)
! Total the number of matrix-vector products.
 NV=NV+1

 CASE(ARPACK_Prepare)
! Define 1/H**2, etc. so they are available in the evaluator.
 H = ONE/REAL(NX+1,DKIND)
 HSQ = H**2
 DD = (4.0D+0, 0.0D+0) / HSQ
 DL = -ONE/HSQ - (5.0D-1, 0.0D+0) *RHO/H
 DU = -ONE/HSQ + (5.0D-1, 0.0D+0) *RHO/H
 NV = 0
 CASE DEFAULT
 WRITE(nout,*) TASK, ': INVALID TASK REQUESTED'
 STOP 'IMSL_ERROR_WRONG_OPERATION'
 END SELECT
 END ASSOCIATE
 END SELECT
END FUNCTION

FUNCTION T(NX, X, DT)RESULT(V)
 INTEGER, INTENT(IN) :: NX
 COMPLEX(DKIND), INTENT(IN) :: X(:), DT(3)
 COMPLEX(DKIND) :: V(NX)
 INTEGER J
 ASSOCIATE(DL => DT(1), &
 DD => DT(2), &
 DU => DT(3))
 V(1) = DD*X(1) + DU*X(2)
 DO J = 2, NX-1
 V(J) = DL*X(J-1) + DD*X(J) + DU*X(J+1)
 END DO
 V(NX) = DL*X(NX-1) + DD*X(NX)
 END ASSOCIATE
END FUNCTION
END MODULE

! Compute the largest magnitude eigenvalues of a discrete Laplacian,
! based on second order divided differences.

! The matrix used is obtained from the standard central difference
! discretization of the convection-diffusion operator
! (Laplacian u) + rho*(du / dx)
! on the unit square 0,1x0,1 with zero Dirichlet boundary
! conditions.
 USE ARPACK_COMPLEX_EX1_INT
 USE UMACH_INT

```

```

USE WRCRN_INT

INTEGER, PARAMETER :: NEV=6
INTEGER :: J, N, NOUT
COMPLEX(DKIND) :: VALUES(NEV)
COMPLEX(DKIND), ALLOCATABLE :: RES(:), EF(:, :)
COMPLEX(DKIND), ALLOCATABLE :: VECTORS(:, :)
REAL(DKIND) NORM
LOGICAL SMALL, SOLVED
TYPE(ARPACKBASE_EXT) EX

ASSOCIATE(NX => EX % NX, &
 NV => EX % NV, &
 RHO => EX % RHO, &
 NACC => EX % NACC)

CALL UMACH(2, NOUT)
NX=10
RHO=(100._DKIND,1._DKIND)

! Define size of matrix problem.
N=NX**2
! Note that VECTORS(:, :) does not need to be allocated
! in the calling program. That happens within the
! routine ARPACK_COMPLEX(). It is OK to do this but
! the sizes (N,NCV) are determined in ARPACK_COMPLEX.
CALL ARPACK_COMPLEX(N, FZ1, VALUES, EXTYPE=EX, VECTORS=VECTORS)

WRITE(NOUT, *) 'Number of eigenvalues requested, and accurate'
WRITE(NOUT, *) '-----'
WRITE(NOUT, '(5X, I4, 5X, I4)') NEV, NACC
WRITE(NOUT, *) 'Number of Matrix-Vector Products Required, ZEX-1'
WRITE(NOUT, *) '-----'
WRITE(NOUT, '(5X, I4)') NV
CALL WRCRN ('Largest Magnitude Operator Eigenvalues', VALUES)
! Check residuals, A*vectors = values*vectors:
ALLOCATE(RES(N))
DO J=1,NACC
 RES=FZ1(VECTORS(:,J),ARPACK_A_x,EX)-VALUES(J)*VECTORS(:,J)
 NORM=maxval(abs(RES))
 SMALL=(NORM <= ABS(VALUES(J))*SQRT(EPSILON(NORM)))
 IF(J==1) SOLVED=SMALL
 SOLVED=SOLVED .and. SMALL
END DO

IF(SOLVED) THEN
 WRITE(NOUT,'(A///)') &
 'All Ritz Values and Vectors have small residuals.'
ELSE
 WRITE(NOUT,'(A///)') &
 'Some Ritz Values and Vectors have large residuals.'
END IF

END ASSOCIATE
END

```

## Output

Number of eigenvalues requested, and accurate

-----  
6           6

Number of Matrix-Vector Products Required, ZEX-1

-----  
475

Largest Magnitude Operator Eigenvalues

1 ( 727.0, -1029.6)

2 ( 705.4, 1029.6)

3 ( 698.4, -1029.6)

4 ( 676.8, 1029.6)

5 ( 653.3, -1029.6)

6 ( 631.7, 1029.6)

All Ritz Values and Vectors have small residuals.

# Chapter 3: Interpolation and Approximation

---

## Routines

|             |                                                                                                                      |                                     |     |
|-------------|----------------------------------------------------------------------------------------------------------------------|-------------------------------------|-----|
| <b>3.1</b>  | <b>Curve and Surface Fitting with Splines</b>                                                                        |                                     |     |
|             | Returns the derived type array result .....                                                                          | <a href="#">SPLINE_CONSTRAINTS</a>  | 702 |
|             | Returns an array result, given an array<br>of input .....                                                            | <a href="#">SPLINE_VALUES</a>       | 703 |
|             | Weighted least-squares fitting by B-splines to discrete<br>One-Dimensional data is performed .....                   | <a href="#">SPLINE_FITTING</a>      | 704 |
|             | Returns the derived type array result given<br>optional input.....                                                   | <a href="#">SURFACE_CONSTRAINTS</a> | 714 |
|             | Returns a tensor product array result, given two arrays of<br>independent variable values .....                      | <a href="#">SURFACE_VALUES</a>      | 715 |
|             | Weighted least-squares fitting by tensor product<br>B-splines to discrete two-dimensional data<br>is performed ..... | <a href="#">SURFACE_FITTING</a>     | 716 |
| <b>3.2.</b> | <b>Cubic Spline Interpolation</b>                                                                                    |                                     |     |
|             | Easy to use cubic spline routine .....                                                                               | <a href="#">CSIEZ</a>               | 727 |
|             | Not-a-knot .....                                                                                                     | <a href="#">CSINT</a>               | 729 |
|             | Derivative end conditions.....                                                                                       | <a href="#">CSDEC</a>               | 732 |
|             | Hermite .....                                                                                                        | <a href="#">CSHER</a>               | 737 |
|             | Akima .....                                                                                                          | <a href="#">CSAKM</a>               | 740 |
|             | Shape preserving.....                                                                                                | <a href="#">CSCON</a>               | 742 |
|             | Periodic .....                                                                                                       | <a href="#">CSPER</a>               | 746 |
| <b>3.3.</b> | <b>Cubic Spline Evaluation and Integration</b>                                                                       |                                     |     |
|             | Evaluation .....                                                                                                     | <a href="#">CSVAL</a>               | 749 |
|             | Evaluation of the derivative.....                                                                                    | <a href="#">CSDER</a>               | 750 |
|             | Evaluation on a grid .....                                                                                           | <a href="#">CS1GD</a>               | 753 |
|             | Integration .....                                                                                                    | <a href="#">CSITG</a>               | 756 |
| <b>3.4.</b> | <b>B-spline Interpolation</b>                                                                                        |                                     |     |
|             | Easy to use spline routine.....                                                                                      | <a href="#">SPLEZ</a>               | 758 |
|             | One-dimensional interpolation .....                                                                                  | <a href="#">BSINT</a>               | 761 |
|             | Knot sequence given interpolation data .....                                                                         | <a href="#">BSNAK</a>               | 765 |
|             | Optimal knot sequence given interpolation data.....                                                                  | <a href="#">BSOPK</a>               | 768 |

|              |                                                                                                                 |        |     |
|--------------|-----------------------------------------------------------------------------------------------------------------|--------|-----|
|              | Two-dimensional tensor product interpolation .....                                                              | BS2IN  | 771 |
|              | Three-dimensional tensor product interpolation .....                                                            | BS3IN  | 776 |
| <b>3.5.</b>  | <b>Spline Evaluation, Integration, and Conversion to Piecewise Polynomial Given the B-spline Representation</b> |        |     |
|              | Evaluation .....                                                                                                | BSVAL  | 782 |
|              | Evaluation of the derivative .....                                                                              | BSDER  | 783 |
|              | Evaluation on a grid .....                                                                                      | BS1GD  | 786 |
|              | One-dimensional integration .....                                                                               | BSITG  | 789 |
|              | Two-dimensional evaluation .....                                                                                | BS2VL  | 792 |
|              | Two-dimensional evaluation of the derivative .....                                                              | BS2DR  | 794 |
|              | Two-dimensional evaluation on a grid .....                                                                      | BS2GD  | 797 |
|              | Two-dimensional integration .....                                                                               | BS2IG  | 801 |
|              | Three-dimensional evaluation .....                                                                              | BS3VL  | 805 |
|              | Three-dimensional evaluation of the derivative .....                                                            | BS3DR  | 807 |
|              | Three-dimensional evaluation on a grid .....                                                                    | BS3GD  | 811 |
|              | Three-dimensional integration .....                                                                             | BS3IG  | 817 |
|              | Convert B-spline representation to piecewise polynomial ..                                                      | BSCPP  | 821 |
| <b>3.6.</b>  | <b>Piecewise Polynomial</b>                                                                                     |        |     |
|              | Evaluation .....                                                                                                | PPVAL  | 823 |
|              | Evaluation of the derivative .....                                                                              | PPDER  | 825 |
|              | Evaluation on a grid .....                                                                                      | PP1GD  | 828 |
|              | Integration .....                                                                                               | PPITG  | 831 |
| <b>3.7.</b>  | <b>Quadratic Polynomial Interpolation Routines for Gridded Data</b>                                             |        |     |
|              | One-dimensional evaluation .....                                                                                | QDVAL  | 833 |
|              | One-dimensional evaluation of the derivative .....                                                              | QDDER  | 835 |
|              | Two-dimensional evaluation .....                                                                                | QD2VL  | 838 |
|              | Two-dimensional evaluation of the derivative .....                                                              | QD2DR  | 840 |
|              | Three-dimensional evaluation .....                                                                              | QD3VL  | 843 |
|              | Three-dimensional evaluation of the derivative .....                                                            | QD3DR  | 847 |
| <b>3.8.</b>  | <b>Multi-dimensional Interpolation</b>                                                                          |        |     |
|              | Akima's surface fitting method .....                                                                            | SURF   | 851 |
|              | Multidimensional interpolation and differentiation .....                                                        | SURFND | 855 |
| <b>3.9.</b>  | <b>Least-Squares Approximation</b>                                                                              |        |     |
|              | Linear polynomial .....                                                                                         | RLINE  | 858 |
|              | General polynomial .....                                                                                        | RCURV  | 861 |
|              | General functions .....                                                                                         | FNLSQ  | 865 |
|              | Splines with fixed knots .....                                                                                  | BLSLQ  | 870 |
|              | Splines with variable knot .....                                                                                | BSVLS  | 874 |
|              | Splines with linear constraints .....                                                                           | CONFT  | 879 |
|              | Two-dimensional tensor-product splines with fixed knots ....                                                    | BLSL2  | 889 |
|              | Three-dimensional tensor-product splines with fixed knots .                                                     | BLSL3  | 894 |
| <b>3.10.</b> | <b>Cubic Spline Smoothing</b>                                                                                   |        |     |
|              | Smoothing by error detection .....                                                                              | CSSD   | 900 |
|              | Smoothing spline .....                                                                                          | CSSMH  | 904 |
|              | Smoothing spline using cross-validation .....                                                                   | CSSCV  | 907 |

## Usage Notes

The majority of the routines in this chapter produce piecewise polynomial or spline functions that either interpolate or approximate given data, or are support routines for the evaluation, integration, and conversion from one representation to another. Two major subdivisions of routines are provided. The cubic spline routines begin with the letters “CS” and utilize the piecewise polynomial representation described below. The B-spline routines begin with the letters “BS” and utilize the B-spline representation described below. Most of the spline routines are based on routines in the book by de Boor (1978).

### Piecewise Polynomials

A univariate piecewise polynomial (function)  $p$  is specified by giving its breakpoint sequence  $\xi \in \mathbf{R}^n$ , the order  $k$  (degree  $k - 1$ ) of its polynomial pieces, and the  $k \times (n - 1)$  matrix  $c$  of its local polynomial coefficients. In terms of this information, the piecewise polynomial (pp) function is given by

$$p(x) = \sum_{j=1}^k c_{ji} \frac{(x - \xi_i)^{j-1}}{(j-1)!} \quad \text{for } \xi_i \leq x < \xi_{i+1}$$

The breakpoint sequence  $\xi$  is assumed to be strictly increasing, and we extend the pp function to the entire real axis by extrapolation from the first and last intervals. The subroutines in this chapter will consistently make the following identifications for FORTRAN variables:

$c = \text{PPCOEF}$

$\xi = \text{BREAK}$

$k = \text{KORDER}$

$N = \text{NBREAK}$

This representation is redundant when the pp function is known to be smooth. For example, if  $p$  is known to be continuous, then we can compute  $c_{1,i+1}$  from the  $c_{ji}$  as follows

$$c_{1,i+1} = p(\xi_{i+1}) = c_{1i} + c_{2i} \Delta \xi_i + \dots + c_{ki} \frac{(\Delta \xi_i)^{k-1}}{(k-1)!}$$

where  $\Delta \xi_i := \xi_{i+1} - \xi_i$ . For smooth pp, we prefer to use the irredundant representation in terms of the B-(for ‘basis’)-splines, at least when such a function is first to be determined. The above pp representation is employed for evaluation of the pp function at many points since it is more efficient.

### Splines and B-splines

B-splines provide a particularly convenient and suitable basis for a given class of smooth pp functions. Such a class is specified by giving its breakpoint sequence, its order, and the required

smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence  $\mathbf{t} \in \mathbf{R}^M$ . The specification rule is the following: If the class is to have all derivatives up to and including the  $j$ -th derivative continuous across the interior breakpoint  $\xi_i$ , then the number  $\xi_i$  should occur  $k - j - 1$  times in the knot sequence. Assuming that  $\xi_1$  and  $\xi_n$  are the endpoints of the interval of interest, one chooses the first  $k$  knots equal to  $\xi_1$  and the last  $k$  knots equal to  $\xi_n$ . This can be done since the B-splines are defined to be right continuous near  $\xi_1$  and left continuous near  $\xi_n$ .

When the above construction is completed, we will have generated a knot sequence  $\mathbf{t}$  of length  $M$ ; and there will be  $m := M - k$  B-splines of order  $k$ , say  $B_1, \dots, B_m$  that span the pp functions on the interval with the indicated smoothness. That is, each pp function in this class has a unique representation

$$p = a_1 B_1 + a_2 B_2 + \dots + a_m B_m$$

as a linear combination of B-splines. The B-spline routines will consistently make use of the following identifiers for FORTRAN variables:

$a = \text{BSCOEF}$

$\mathbf{t} = \text{XKNOT}$

$m = \text{NCOEF}$

$M = \text{NKNOT}$

A B-spline is a particularly compact pp function.  $B_i$  is a nonnegative function that is nonzero only on the interval  $[t_i, t_{i+k}]$ . More precisely, the support of the  $i$ -th B-spline is  $[t_i, t_{i+k}]$ . No pp function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals. When it is necessary to emphasize the dependence of the B-spline on its parameters, we will use the notation

$$B_{i,k,t}$$

to denote the  $i$ -th B-spline of order  $k$  for the knot sequence  $\mathbf{t}$ .

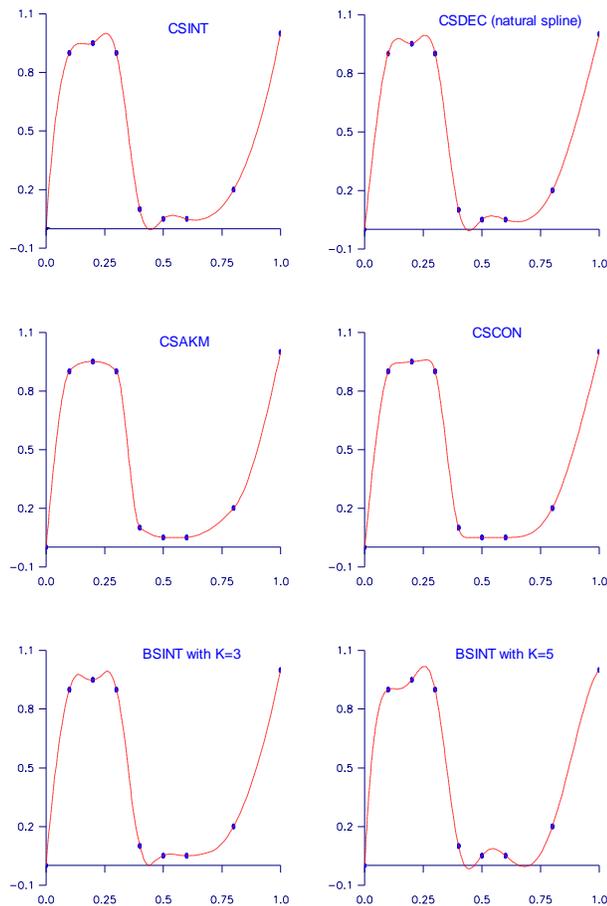


Figure 3-1 Spline Interpolants of the Same Data

## Cubic Splines

Cubic splines are smooth (i.e.,  $C^1$  or  $C^2$ ) fourth-order pp functions. For historical and other reasons, cubic splines are the most heavily used pp functions. Therefore, we provide special routines for their construction and evaluation. The routines for their determination use yet another representation (in terms of value and slope at all the breakpoints) but output the pp representation as described above for general pp functions.

We provide seven cubic spline interpolation routines: `CSIEZ`, `CSINT`, `CSDEC`, `CSHER`, `CSAKM`, `CSCON`, and `CSPER`. The first routine, `CSIEZ`, is an easy-to-use version of `CSINT` coupled with `CSVAL`. The routine `CSIEZ` will compute the value of the cubic spline interpolant (to given data using the ‘not-a-knot’ criterion) on a grid. The routine `CSDEC` allows the user to specify various endpoint conditions (such as the value of the first or second derivative at the right and left points). This means that the natural cubic spline can be obtained using this routine by setting the second derivative to zero at both endpoints. If function values and derivatives are available, then the Hermite cubic interpolant can be computed using `CSHER`. The two routines `CSAKM` and `CSCON` are

designed so that the shape of the curve matches the shape of the data. In particular, `CSCON` preserves the convexity of the data while `CSAKM` attempts to minimize oscillations. If the data is periodic, then `CSPER` will produce a periodic interpolant. The routine `CONFIT` allows the user wide latitude in enforcing shapes. This routine returns the B-spline representation.

It is possible that the cubic spline interpolation routines will produce unsatisfactory results. The adventurous user should consider using the B-spline interpolation routine `BSINT` that allows one to choose the knots and order of the spline interpolant.

In Figure 3-1, we display six spline interpolants to the same data. This data can be found in Example 1 of the IMSL routine `CSCON`. Notice the different characteristics of the interpolants. The interpolation routines `CSAKM` and `CSCON` are the only two that attempt to preserve the shape of the data. The other routines tend to have extraneous inflection points, with the piecewise quartic ( $k = 5$ ) exhibiting the most oscillation.

## Tensor Product Splines

The simplest method of obtaining multivariate interpolation and approximation routines is to take univariate methods and form a multivariate method via tensor products. In the case of two-dimensional spline interpolation, the development proceeds as follows: Let  $\mathbf{t}_x$  be a knot sequence for splines of order  $k_x$ , and  $\mathbf{t}_y$  be a knot sequence for splines of order  $k_y$ . Let  $N_x + k_x$  be the length of  $\mathbf{t}_x$ , and  $N_y + k_y$  be the length of  $\mathbf{t}_y$ . Then, the tensor product spline has the form

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y)$$

Given two sets of points

$$\{x_i\}_{i=1}^{N_x} \text{ and } \{y_i\}_{i=1}^{N_y}$$

for which the corresponding univariate interpolation problem could be solved, the tensor product interpolation problem becomes: Find the coefficients  $c_{nm}$  so that

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_i) = f_{ij}$$

This problem can be solved efficiently by repeatedly solving univariate interpolation problems as described in de Boor (1978, page 347). Three-dimensional interpolation has analogous behavior. In this chapter, we provide routines that compute the two-dimensional tensorproduct spline coefficients given two-dimensional interpolation data (`BS2IN`), compute the three-dimensional tensor-product spline coefficients given three-dimensional interpolation data (`BS3IN`) compute the two-dimensional tensor-product spline coefficients for a tensor-product least squares problem (`BSLS2`), and compute the three-dimensional tensor-product spline coefficients for a tensor-product least squares problem (`BSLS3`). In addition, we provide evaluation, differentiation, and integration routines for the two and three-dimensional tensor-product spline functions. The relevant routines are `BS2VL`, `BS3VL`, `BS2DR`, `BS3DR`, `BS2GD`, `BS3GD`, `BS2IG`, and `BS3IG`.

## Quadratic Interpolation

The routines that begin with the letters “QD” in this chapter are designed to interpolate a one-, two-, or three-dimensional (tensor product) table of values and return an approximation to the value of the underlying function or one of its derivatives at a given point. These routines are all based on quadratic polynomial interpolation.

## Multi-dimensional Interpolation

We have one routine, `SURF`, that will return values of an interpolant to scattered data in the plane. This routine is based on work by Akima (1978), which utilizes  $C^1$  piecewise quintics on a triangular mesh. `SURFND` computes a piecewise polynomial interpolant, of up to 15-th degree, to a function of up to 7 variables, defined on a multi-dimensional grid.

## Least Squares

Routines are provided to smooth noisy data: regression using linear regression using arbitrary polynomials (`RCURV`), and regression using user-supplied functions (`FNLSQ`). Additional routines compute the least-squares fit using splines with fixed knots (`BSLSQ`) or free knots (`BSVLS`). These routines can produce cubic-spline least-squares fit simply by setting the order to 4. The routine `CONFIT` computes a fixed-knot spline weighted least-squares fit subject to linear constraints. This routine is very general and is recommended if issues of shape are important. The two- and three-dimensional tensor-product spline regression routines are (`BSLS2`) and (`BSLS3`).

## Smoothing by Cubic Splines

Two “smoothing spline” routines are provided. The routine `CSSMH` returns the cubic spline that smooths the data, given a smoothing parameter chosen by the user. Whereas, `CSSCV` estimates the smoothing parameter by cross-validation and then returns the cubic spline that smooths the data. In this sense, `CSSCV` is the easier of the two routines to use. The routine `CSSD` returns a smoothed data vector approximating the values of the underlying function when the data are contaminated by a few random spikes.

## Rational Chebyshev Approximation

The routine `RATCH` computes a rational Chebyshev approximation to a user-supplied function. Since polynomials are rational functions, this routine can be used to compute best polynomial approximations.

## Using the Univariate Spline Routines

An easy to use spline interpolation routine `CSIEZ` is provided. This routine computes an interpolant and returns the values of the interpolant on a user-supplied grid. A slightly more advanced routine `SPLEZ` computes (at the users discretion) one of several interpolants or least-squares fits and returns function values or derivatives on a user-supplied grid.

For more advanced uses of the interpolation (or least squares) spline routines, one first forms an interpolant from interpolation (or least-squares) data. Then it must be evaluated, differentiated, or integrated once the interpolant has been formed. One way to perform these tasks, using cubic splines with the ‘not-a-knot’ end condition, is to call `CSINT` to obtain the local coefficients of the

piecewise cubic interpolant and then call `CSVAL` to evaluate the interpolant. A more complicated situation arises if one wants to compute a quadratic spline interpolant and then evaluate it (efficiently) many times. Typically, the sequence of routines called might be `BSNAK` (get the knots), `BSINT` (returns the B-spline coefficients of the interpolant), `BSCPP` (convert to pp form), and `PPVAL` (evaluate). The last two calls could be replaced by a call to the B-spline grid evaluator `BS1GD`, or the last call could be replaced with pp grid evaluator `PP1GD`. The interconnection of the spline routines is summarized in Figure 3-2.

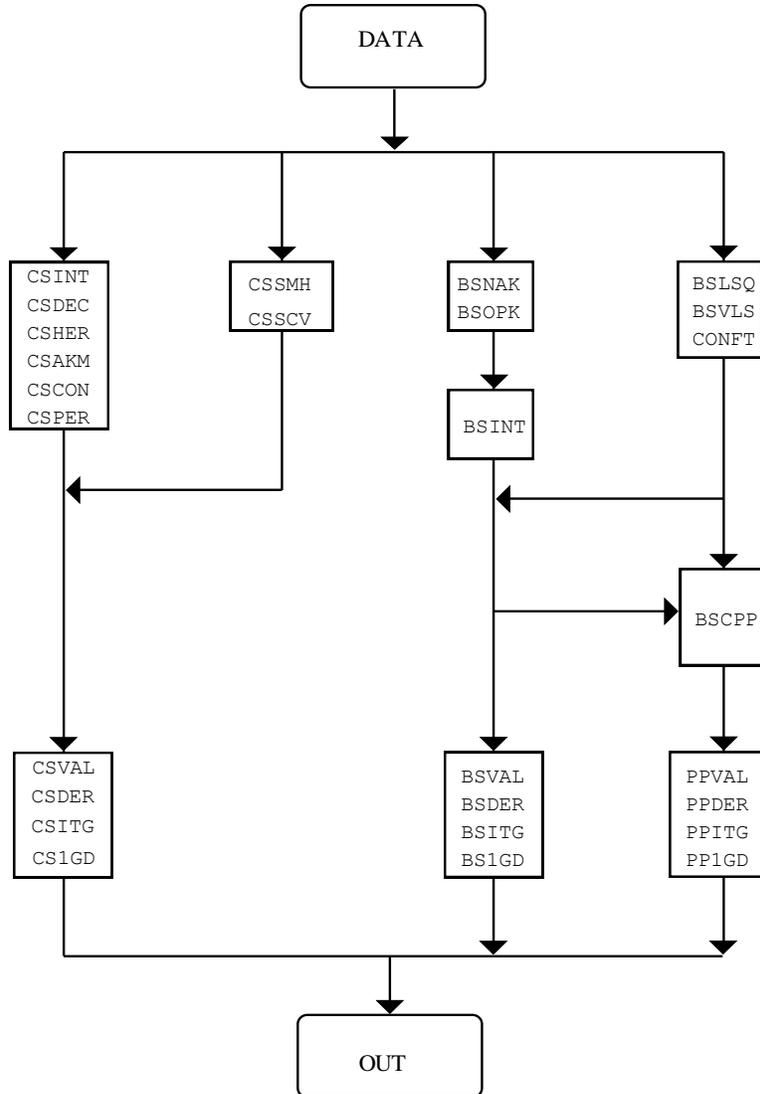


Figure 3-2 Interrelation of the Spline Routines

## Choosing an Interpolation Routine

The choice of an interpolation routine depends both on the type of data and on the use of the interpolant. We provide 19 interpolation routines. These routines are depicted in a decision tree in Figure 3-3. This figure provides a guide for selecting an appropriate interpolation routine. For example, if periodic one-dimensional (univariate) data is available, then the path through *univariate* to *periodic* leads to the IMSL routine `CSPER`, which is the proper routine for this setting. The general-purpose univariate interpolation routines can be found in the box beginning with `CSINT`. Multidimensional tensor-product interpolation routines are also provided. For two-dimensional scattered data, the appropriate routine is `SURF`.

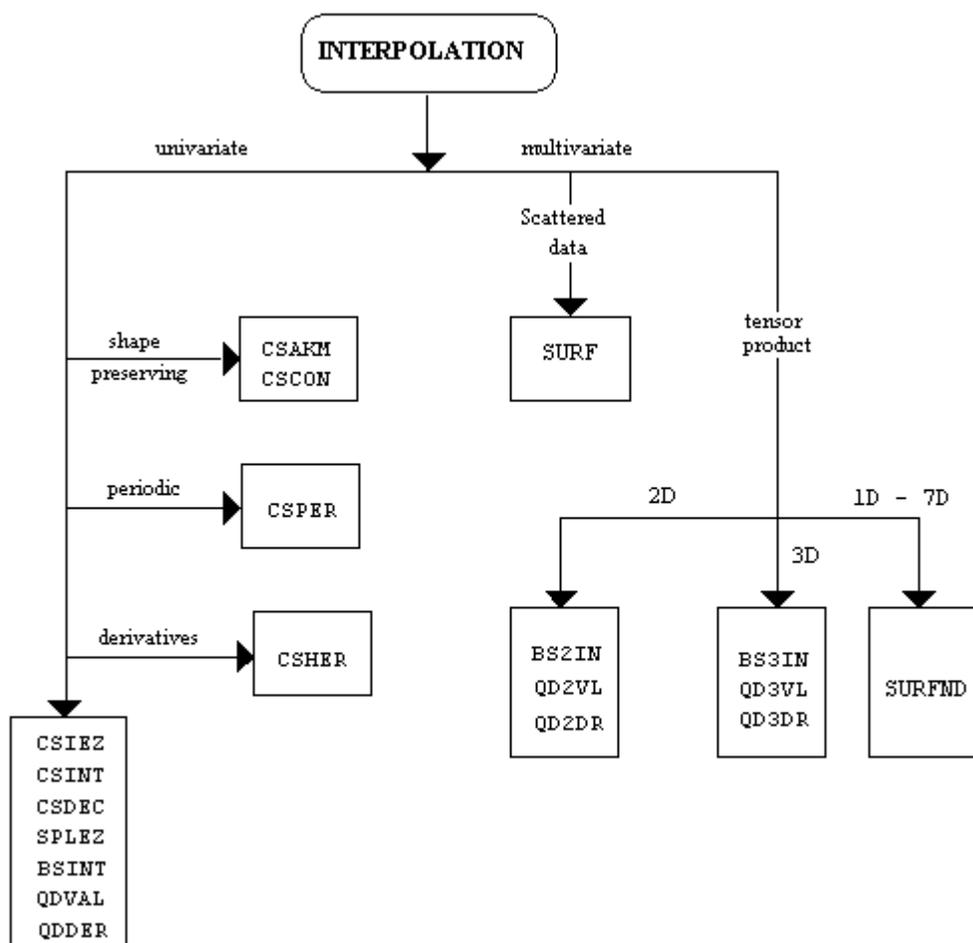


Figure 3-3 Choosing an Interpolation Routine

---

## SPLINE\_CONSTRAINTS

This function returns the derived type array result, `?_SPLINE_CONSTRAINTS`, given optional input. There are optional arguments for the derivative index, the value applied to the spline, and the periodic point for any periodic constraint.

The function is used, for entry number `j`,

```
?_SPLINE_CONSTRAINTS(J) = &
 SPLINE_CONSTRAINTS([DERIVATIVE=DERIVATIVE_INDEX,] &
 POINT = WHERE_APPLIED, [VALUE=VALUE_APPLIED,], &
 TYPE = CONSTRAINT_INDICATOR, &
 [PERIODIC_POINT = VALUE_APPLIED])
```

The square brackets enclose optional arguments. For each constraint either (but not both) the `'VALUE ='` or the `'PERIODIC_POINT ='` optional arguments must be present.

### Required Arguments

`POINT = WHERE_APPLIED (Input)`

The point in the data interval where a constraint is to be applied.

`TYPE = CONSTRAINT_INDICATOR (Input)`

The indicator for the type of constraint the spline function or its derivatives is to satisfy at the point: `where_applied`. The choices are the character strings `'=='`, `'<='`, `'>='`, `'=..'`, and `'.-.'`. They respectively indicate that the spline value or its derivatives will be equal to, not greater than, not less than, equal to the value of the spline at another point, or equal to the negative of the spline value at another point. These last two constraints are called *periodic* and *negative-periodic*, respectively. The alternate independent variable point is `value_applied` for either periodic constraint. There is a use of periodic constraints in .

### Optional Arguments

`DERIVATIVE = DERIVATIVE_INDEX (Input)`

This is the number of the derivative for the spline to apply the constraint. The value 0 corresponds to the function, the value 1 to the first derivative, etc. If this argument is not present in the list, the value 0 is substituted automatically. Thus a constraint without the derivative listed applies to the spline function.

`PERIODIC_POINT = VALUE_APPLIED`

This optional argument improves readability by automatically identifying the second independent variable value for periodic constraints.

### FORTRAN 90 Interface

Generic: `CALL SPLINE_CONSTRAINTS (POINT, TYPE [, ...])`

Specific: The specific interface names are `S_SPLINE_CONSTRAINTS` and `D_SPLINE_CONSTRAINTS`.

---

## SPLINE\_VALUES

This rank-1 array function returns an array result, given an array of input. Use the optional argument for the covariance matrix when the square root of the variance function is required. The result will be a scalar value when the input variable is scalar.

### Required Arguments

DERIVATIVE = DERIVATIVE (Input)

The index of the derivative evaluated. Use non-negative integer values. For the function itself use the value 0.

VARIABLES = VARIABLES (Input)

The independent variable values where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

KNOTS = KNOTS (Input)

The derived type `?_spline_knots`, defined as the array `COEFFS` was obtained with the function `SPLINE_FITTING`. This contains the polynomial spline degree and the number of knots and the knots themselves for this spline function.

COEFFS = C (Input)

The coefficients in the representation for the spline function,

$$f(x) = \sum_{j=1}^N c_j B_j(x)$$

These result from the fitting process or array assignment

`C=SPLINE_FITTING(...)`, defined below. The value

$N = \text{size}(C)$  satisfies the identity

$N - 1 + \text{spline\_degree} = \text{size}(\text{?}_k\text{nots})$ , where the two right-most quantities refer to components of the argument `knots`.

### Optional Arguments

COVARIANCE = G (Input)

This argument, when present, results in the evaluation of the square root of the variance function

$$e(x) = \left( b(x)^T G b(x) \right)^{1/2}$$

where

$$b(x) = \left[ B_1(x), \dots, B_N(x) \right]^T$$

and  $G$  is the covariance matrix associated with the coefficients of the spline

$$c = \left[ c_1, \dots, c_N \right]^T$$

The argument `G` is an optional output parameter from the function `SPLINE_FITTING`, described below. When the square root of the variance function is computed, the arguments `DERIVATIVE` and `C` are not used.

`IOPT = IOPT (Input)`

This optional argument, of derived type `?_options`, is not used in this release.

### **FORTRAN 90 Interface**

Generic: `CALL SPLINE_VALUES (DERIVATIVE, VARIABLES, KNOTS, COEFFS [, ...])`

Specific: The specific interface names are `S_SPLINE_VALUES` and `D_SPLINE_VALUES`.

---

## **SPLINE\_FITTING**

Weighted least-squares fitting by B-splines to discrete One-Dimensional data is performed. Constraints on the spline or its derivatives are optional. The spline function

$$f(x) = \sum_{j=1}^N c_j B_j(x)$$

its derivatives, or the square root of its variance function are evaluated after the fitting.

### **Required Arguments**

`DATA = DATA(1:3, :) (Input/Output)`

An assumed-shape array with `size(data, 1) = 3`. The data are placed in the array:

`data(1, i) = xi`, `data(2, i) = yi`, and `data(3, i) = σi`,  $i = 1, \dots, ndata$ . If the variances are not known but are proportional to an unknown value, users may set `data(3, i) = 1`,  $i = 1, \dots, ndata$ .

`KNOTS = KNOTS (Input)`

A derived type, `?_spline_knots`, that defines the degree of the spline and the breakpoints for the data fitting interval.

### **Optional Arguments**

`CONSTRAINTS = SPLINE_CONSTRAINTS (Input)`

A rank-1 array of derived type `?_spline_constraints` that give constraints the output spline is to satisfy.

`COVARIANCE = G (Output)`

An assumed-shape rank-2 array of the same precision as the data. This output is the covariance matrix of the coefficients. It is optionally used to evaluate the square root of the variance function.

IOPT = IOPT(:) (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `SPLINE_FITTING`. The options are as follows:

| Packaged Options for <code>SPLINE_FITTING</code> |                                       |              |
|--------------------------------------------------|---------------------------------------|--------------|
| Prefix = None                                    | Option Name                           | Option Value |
|                                                  | <code>SPLINE_FITTING_TOL_EQUAL</code> | 1            |
|                                                  | <code>SPLINE_FITTING_TOL_LEAST</code> | 2            |

IOPT(IO) = ?\_OPTIONS(SPLINE\_FITTING\_TOL\_EQUAL, ?\_VALUE)

This resets the value for determining that equality constraint equations are rank-deficient. The default is `?_value = 10-4`.

IOPT(IO) = ?\_OPTIONS(SPLINE\_FITTING\_TOL\_LEAST, ?\_VALUE)

This resets the value for determining that least-squares equations are rank-deficient. The default is `?_value = 10-4`.

## FORTRAN 90 Interface

Generic: `CALL SPLINE_FITTING (DATA, KNOTS [, ...])`

Specific: The specific interface names are `S_SPLINE_FITTING` and `D_SPLINE_FITTING`.

## Description

This routine has similar scope to `CONFIT` found in IMSL (2003, pp 734-743). We provide the square root of the variance function, but we do not provide for constraints on the integral of the spline. The least-squares matrix problem for the coefficients is banded, with band-width equal to the spline order. This fact is used to obtain an efficient solution algorithm when there are no constraints. When constraints are present the routine solves a linear-least squares problem with equality and inequality constraints. The processed least-squares equations result in a banded and upper triangular matrix, following accumulation of the spline fitting equations. The algorithm used for solving the constrained least-squares system will handle rank-deficient problems. A set of reference are available in Hanson (1995) and Lawson and Hanson (1995). The `CONFIT` routine uses `QPROG` (*loc cit.*, p. 959), which requires that the least-squares equations be of full rank.

## Fatal and Terminal Error Messages

See the `messages.gls` file for error messages for `SPLINE_FITTING`. These error messages are numbered 1340–1367.

## Example 1: Natural Cubic Spline Interpolation to Data

The function

$$g(x) = \exp(-x^2/2)$$

is interpolated by cubic splines on the grid of points

$$x_i = (i-1)\Delta x, i = 1, \dots, n_{\text{data}}$$

Those natural conditions are

$$f(x_i) = g(x_i), i = 0, \dots, n_{\text{data}}; \frac{d^2 f}{dx^2}(x_i) = \frac{d^2 g}{dx^2}(x_i), i = 0 \text{ and } n_{\text{data}}$$

Our program checks the term *const.* appearing in the maximum truncation error term

$$\text{error} \approx \text{const.} \times \Delta x^4$$

at a finer grid.

```

USE spline_fitting_int
USE show_int
USE norm_int

implicit none

! This is Example 1 for SPLINE_FITTING, Natural Spline
! Interpolation using cubic splines. Use the function
! exp(-x**2/2) to generate samples.

integer :: i
integer, parameter :: ndata=24, nord=4, ndegree=nord-1, &
 nbkpt=ndata+2*ndegree, ncoeff=nbkpt-nord, nvalues=2*ndata
real(kind(1e0)), parameter :: zero=0e0, one=1e0, half=5e-1
real(kind(1e0)), parameter :: delta_x=0.15, delta_xv=0.4*delta_x
real(kind(1e0)), target :: xdata(ndata), ydata(ndata), &
 spline_data(3, ndata), bkpt(nbkpt), &
 ycheck(nvalues), coeff(ncoeff), &
 xvalues(nvalues), yvalues(nvalues), diffs

real(kind(1e0)), pointer :: pointer_bkpt(:)
type (s_spline_knots) break_points
type (s_spline_constraints) constraints(2)

xdata = (/((i-1)*delta_x, i=1,ndata)/)
ydata = exp(-half*xdata**2)
xvalues = /(0.03+(i-1)*delta_xv, i=1, nvalues)/)
ycheck= exp(-half*xvalues**2)
spline_data(1,:)=xdata
spline_data(2,:)=ydata
spline_data(3,:)=one

! Define the knots for the interpolation problem.
bkpt(1:ndegree) = /(i*delta_x, i=-ndegree,-1)/)
bkpt(nord:nbkpt-ndegree) = xdata
bkpt(nbkpt-ndegree+1:nbkpt) = &
 /(xdata(ndata)+i*delta_x, i=1, ndegree)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
break_points=s_spline_knots(ndegree, pointer_bkpt)

```

```

! These are the natural conditions for interpolating cubic
! splines. The derivatives match those of the interpolating
! function at the ends.
 constraints(1)=spline_constraints &
 (derivative=2, point=bkpt(nord), type=='=', value=-one)
 constraints(2)=spline_constraints &
 (derivative=2, point=bkpt(nbkpt-ndegree), type='=', &
 value=(-one+xdata(ndata)**2)*ydata(ndata))

 coeff = spline_fitting(data=spline_data, knots=break_points, &
 constraints=constraints)
 yvalues=spline_values(0, xvalues, break_points, coeff)

 diffs=norm(yvalues-ycheck, huge(1))/delta_x**nord

 if (diffs <= one) then
 write(*,*) 'Example 1 for SPLINE_FITTING is correct.'
 end if
end

```

## Output

Example 1 for SPLINE\_FITTING is correct.

## Additional Examples

### Example 2: Shaping a Curve and its Derivatives

The function

$$g(x) = \exp(-x^2/2)(1 + noise)$$

is fit by cubic splines on the grid of equally spaced points

$$x_i = (i-1)\Delta x, i = 1, \dots, ndata$$

The term *noise* is uniform random numbers from the normalized interval

$[-\tau, \tau]$ , where  $\tau = 0.01$ . The spline curve is constrained to be convex down for  $0 \leq x \leq 1$  convex upward for  $1 < x \leq 4$ , and have the second derivative exactly equal to the value zero at  $x = 1$ . The first derivative is constrained with the value zero at  $x = 0$  and is non-negative at the right end of the interval,  $x = 4$ . A sample table of independent variables, second derivatives and square root of variance function values is printed.

```

 use spline_fitting_int
 use show_int
 use rand_int
 use norm_int

 implicit none

! This is Example 2 for SPLINE_FITTING. Use 1st and 2nd derivative
! constraints to shape the splines.

```

```

integer :: i, icurv
integer, parameter :: nbkptin=13, nord=4, ndegree=nord-1, &
 nbkpt=nbkptin+2*ndegree, ndata=21, ncoeff=nbkpt-nord
real(kind(1e0)), parameter :: zero=0e0, one=1e0, half=5e-1
real(kind(1e0)), parameter :: range=4.0, ratio=0.02, tol=ratio*half
real(kind(1e0)), parameter :: delta_x=range/(ndata-1), &
 delta_b=range/(nbkptin-1)
real(kind(1e0)), target :: xdata(ndata), ydata(ndata), ynoise(ndata), &
 sddata(ndata), spline_data(3, ndata), bkpt(nbkpt), &
 values(ndata), derivat1(ndata), derivat2(ndata), &
 coeff(ncoeff), root_variance(ndata), diffs
real(kind(1e0)), dimension(ncoeff,ncoeff) :: sigma_squared

real(kind(1e0)), pointer :: pointer_bkpt(:)
type (s_spline_knots) break_points
type (s_spline_constraints) constraints(nbkptin+2)

xdata = (/((i-1)*delta_x, i=1,ndata)/)
ydata = exp(-half*xdata**2)
ynoise = ratio*ydata*(rand(ynoise)-half)
ydata = ydata+ynoise
sddata = ynoise
spline_data(1,:)=xdata
spline_data(2,:)=ydata
spline_data(3,:)=sddata

bkpt=(/((i-nord)*delta_b, i=1,nbkpt)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
break_points=s_spline_knots(ndegree, pointer_bkpt)

icurv=int(one/delta_b)+1

! At first shape the curve to be convex down.
do i=1,icurv-1
 constraints(i)=spline_constraints &
 (derivative=2, point=bkpt(i+ndegree), type='<=', value=zero)
end do

! Force a curvature change.
constraints(icurv)=spline_constraints &
 (derivative=2, point=bkpt(icurv+ndegree), type='==', value=zero)

! Finally, shape the curve to be convex up.
do i=icurv+1,nbkptin
 constraints(i)=spline_constraints &
 (derivative=2, point=bkpt(i+ndegree), type='>=', value=zero)
end do

! Make the slope zero and value non-negative at right.
constraints(nbkptin+1)=spline_constraints &
 (derivative=1, point=bkpt(nord), type='==', value=zero)
constraints(nbkptin+2)=spline_constraints &
 (derivative=0, point=bkpt(nbkptin+ndegree), type='>=', value=zero)

```

```

coeff = spline_fitting(data=spline_data, knots=break_points, &
 constraints=constraints, covariance=sigma_squared)

! Compute value, first two derivatives and the variance.
values=spline_values(0, xdata, break_points, coeff)
root_variance=spline_values(0, xdata, break_points, coeff, &
 covariance=sigma_squared)
derivat1=spline_values(1, xdata, break_points, coeff)
derivat2=spline_values(2, xdata, break_points, coeff)

call show(reshape((/xdata, derivat2, root_variance/), (/ndata,3/)), &
"The x values, 2-nd derivatives, and square root of variance.")

! See that differences are relatively small and the curve has
! the right shape and signs.
diffs=norm(values-ydata)/norm(ydata)
if (all(values > zero) .and. all(derivat1 < epsilon(zero)) &
 .and. diffs <= tol) then
 write(*,*) 'Example 2 for SPLINE_FITTING is correct.'
end if

end

```

## Output

Example 2 for SPLINE\_FITTING is correct.

### Example 3: Splines Model a Random Number Generator

The function

$$\begin{aligned}
 g(x) &= \exp(-x^2/2), -1 < x < 1 \\
 &= 0, |x| \geq 1
 \end{aligned}$$

is an unnormalized probability distribution. This function is similar to the standard Normal distribution, with specific choices for the mean and variance, except that it is truncated. Our algorithm interpolates  $g(x)$  with a natural cubic spline,  $f(x)$ . The cumulative distribution is approximated by precise evaluation of the function

$$q(x) = \int_{-1}^x f(t) dt$$

Gauss-Legendre quadrature formulas, IMSL (1994, pp. 621-626), of order two are used on each polynomial piece of  $f(t)$  to evaluate  $q(x)$  cheaply. After normalizing the cubic spline so that  $q(1) = 1$ , we may then generate random numbers according to the distribution  $f(x) \cong g(x)$ .

The values of  $x$  are evaluated by solving  $q(x) = u$ ,  $-1 < x < 1$ . Here  $u$  is a *uniform* random sample. Newton's method, for a vector of unknowns, is used for the solution algorithm.

Recalling the relation

$$\frac{d}{dx}(q(x) - u) = f(x), -1 < x < 1$$

we believe this illustrates a method for generating a vector of random numbers according to a continuous distribution function having finite support.

```

use spline_fitting_int
use linear_operators
use Numerical_Libraries

implicit none

! This is Example 3 for SPLINE_FITTING. Use splines to
! generate random (almost normal) numbers. The normal distribution
! function has support (-1,+1), and is zero outside this interval.
! The variance is 0.5.

integer i, niterat
integer, parameter :: iweight=1, nfix=0, nord=4, ndata=50
integer, parameter :: nquad=(nord+1)/2, ndegree=nord-1
integer, parameter :: nbkpt=ndata+2*ndegree, ncoeff=nbkpt-nord
integer, parameter :: last=nbkpt-ndegree, n_samples=1000
integer, parameter :: limit=10
real(kind(1e0)), dimension(n_samples) :: fn, rn, x, alpha_x, beta_x
INTEGER LEFT_OF(n_samples)
real(kind(1e0)), parameter :: one=1e0, half=5e-1, zero=0e0, two=2e0
real(kind(1e0)), parameter :: delta_x=two/(ndata-1)
real(kind(1e0)), parameter :: qalpha=zero, qbeta=zero, domain=two
real(kind(1e0)) qx(nquad), qxi(nquad), qw(nquad), qxfix(nquad)
real(kind(1e0)) alpha_, beta_, quad(0:ndata-1)
real(kind(1e0)), target :: xdata(ndata), ydata(ndata), &
coeff(ncoeff), spline_data(3, ndata), bkpt(nbkpt)

real(kind(1e0)), pointer :: pointer_bkpt(:)
type (s_spline_knots) break_points
type (s_spline_constraints) constraints(2)

! Approximate the probability density function by splines.
xdata = (/(-one+(i-1)*delta_x, i=1,ndata)/)
ydata = exp(-half*xdata**2)

spline_data(1,:)=xdata
spline_data(2,:)=ydata
spline_data(3,:)=one

bkpt= (/(-one+(i-nord)*delta_x, i=1,nbkpt)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
break_points=s_spline_knots(ndegree, pointer_bkpt)

! Define the natural derivatives constraints:
constraints(1)=spline_constraints &
(derivative=2, point=bkpt(nord), type='==', &
value=(-one+xdata(1)**2)*ydata(1))
constraints(2)=spline_constraints &
(derivative=2, point=bkpt(last), type='==', &
value=(-one+xdata(ndata)**2)*ydata(ndata))

```

```

! Obtain the spline coefficients.
 coeff=spline_fitting(data=spline_data, knots=break_points,&
 constraints=constraints)

! Compute the evaluation points 'qx(*)' and weights 'qw(*)' for
! the Gauss-Legendre quadrature. This will give a precise
! quadrature for polynomials of degree <= nquad*2.
 call gqrul(nquad, iweight, qalpha, qbeta, nfix, qxfix, qx, qw)

! Compute pieces of the accumulated distribution function:
 quad(0)=zero
 do i=1, ndata-1
 alpha_ = (bkpt(nord+i)-bkpt(ndegree+i))*half
 beta_ = (bkpt(nord+i)+bkpt(ndegree+i))*half

! Normalized abscissas are stretched to each spline interval.
! Each polynomial piece is integrated and accumulated.
 qxi = alpha_*qx+beta_
 quad(i) = sum(qw*spline_values(0, qxi, break_points,&
 coeff))*alpha_ &
 + quad(i-1)
 end do

! Normalize the coefficients and partial integrals so that the
! total integral has the value one.
 coeff=coeff/quad(ndata-1); quad=quad/quad(ndata-1)
 rn=rand(rn)
 x=zero; niterat=0

 solve_equation: do

! Find the intervals where the x values are located.
 LEFT_OF=NDEGREE; I=NDEGREE
 do
 I=I+1; if(I >= LAST) EXIT
 WHERE(x >= BKPT(I))LEFT_OF = LEFT_OF+1
 end do

! Use Newton's method to solve the nonlinear equation:
! accumulated_distribution_function - random_number = 0.
 alpha_x = (x-bkpt(LEFT_OF))*half
 beta_x = (x+bkpt(LEFT_OF))*half
 FN=QUAD(LEFT_OF-NORD)-RN
 DO I=1,NQUAD
 FN=FN+QW(I)*spline_values(0, alpha_x*QX(I)+beta_x,&
 break_points, coeff)*alpha_x
 END DO

! This is the Newton method update step:
 x=x-fn/spline_values(0, x, break_points, coeff)
 niterat=niterat+1

! Constrain the values so they fall back into the interval.
! Newton's method may give approximates outside the interval.
 where(x <= -one .or. x >= one) x=zero

```

```

 if(norm(fn,1) <= sqrt(epsilon(one))*norm(x,1)) &
 exit solve_equation
 end do solve_equation

! Check that Newton's method converges.

 if (niterat <= limit) then
 write (*,*) 'Example 3 for SPLINE_FITTING is correct.'
 end if

end

```

## Output

Example 3 for SPLINE\_FITTING is correct.

### Example 4: Represent a Periodic Curve

The curve tracing the edge of a rectangular box, traversed in a counter-clockwise direction, is parameterized with a spline representation for each coordinate function,  $(x(t), y(t))$ . The functions are constrained to be periodic at the ends of the parameter interval. Since the perimeter arcs are piece-wise linear functions, the degree of the splines is the value one. Some breakpoints are chosen so they correspond to corners of the box, where the derivatives of the coordinate functions are discontinuous. The value of this representation is that for each  $t$  the splines representing  $(x(t), y(t))$  are points on the perimeter of the box. This “eases” the complexity of evaluating the edge of the box. This example illustrates a method for representing the edge of a domain in two dimensions, bounded by a periodic curve.

```

 use spline_fitting_int
 use norm_int

 implicit none

! This is Example 4 for SPLINE_FITTING. Use piecewise-linear
! splines to represent the perimeter of a rectangular box.

 integer i, j
 integer, parameter :: nbkpt=9, nord=2, ndegree=nord-1, &
 ncoeff=nbkpt-nord, ndata=7, ngrid=100, &
 nvalues=(ndata-1)*ngrid
 real(kind(1e0)), parameter :: zero=0e0, one=1e0
 real(kind(1e0)), parameter :: delta_t=one, delta_b=one, delta_v=0.01
 real(kind(1e0)) delta_x, delta_y
 real(kind(1e0)), dimension(ndata) :: sddata=one, &
! These are redundant coordinates on the edge of the box.
 xdata=(/0.0, 1.0, 2.0, 2.0, 1.0, 0.0, 0.0/), &
 ydata=(/0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0/)
 real(kind(1e0)) tdata(ndata), xspline_data(3, ndata), &
 yspline_data(3, ndata), tvalues(nvalues), &
 yvalues(nvalues), xcoeff(ncoeff), &
 ycoeff(ncoeff), xcheck(nvalues), ycheck(nvalues), diffs
 real(kind(1e0)), target :: bkpt(nbkpt)
 real(kind(1e0)), pointer :: pointer_bkpt(:)
 type (s_spline_knots) break_points

```

```

type (s_spline_constraints) constraints(1)

tdata = (/((i-1)*delta_t, i=1,ndata)/)
xspline_data(1,:)=tdata; yspline_data(1,:)=tdata
xspline_data(2,:)=xdata; yspline_data(2,:)=ydata
xspline_data(3,:)=sddata; yspline_data(3,:)=sddata

bkpt(nord:nbkpt-ndegree)=(/((i-nord)*delta_b, &
 i=nord, nbkpt-ndegree)/)
! Collapse the outside knots.
bkpt(1:ndegree)=bkpt(nord)
bkpt(nbkpt-ndegree+1:nbkpt)=bkpt(nbkpt-ndegree)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
break_points=s_spline_knots(ndegree, pointer_bkpt)

! Make the two parametric curves also periodic.
constraints(1)=spline_constraints &
 (derivative=0, point=bkpt(nord), type='.', &
 value=bkpt(nbkpt-ndegree))

xcoeff = spline_fitting(data=xspline_data, knots=break_points, &
 constraints=constraints)
ycoeff = spline_fitting(data=yspline_data, knots=break_points, &
 constraints=constraints)

! Use the splines to compute the coordinates of points along the perimeter.
! Compare them with the coordinates of the edge points.
tvalues= (/((i-1)*delta_v, i=1,nvalues)/)
xvalues=spline_values(0, tvalues, break_points, xcoeff)
yvalues=spline_values(0, tvalues, break_points, ycoeff)
do i=1, nvalues
 j=(i-1)/ngrid+1
 delta_x=(xdata(j+1)-xdata(j))/ngrid
 delta_y=(ydata(j+1)-ydata(j))/ngrid
 xcheck(i)=xdata(j)+mod(i+ngrid-1,ngrid)*delta_x
 ycheck(i)=ydata(j)+mod(i+ngrid-1,ngrid)*delta_y
end do

diffs=norm(xvalues-xcheck,1)/norm(xcheck,1)+&
 norm(yvalues-ycheck,1)/norm(ycheck,1)
if (diffs <= sqrt(epsilon(one))) then
 write(*,*) 'Example 4 for SPLINE_FITTING is correct.'
end if

end

```

## Output

Example 4 for SPLINE\_FITTING is correct.

---

## SURFACE\_CONSTRAINTS

To further shape a surface defined by a tensor product of B-splines, the routine `SURFACE_FITTING` will least squares fit data with equality, inequality and periodic constraints. These can apply to the surface function or its partial derivatives. Each constraint is packaged in the derived type `?_SURFACE_CONSTRAINTS`. This function uses the data consisting of: the place where the constraint is to hold, the partial derivative indices, and the type of the constraint. This object is returned as the derived type function result `?_SURFACE_CONSTRAINTS`. The function itself has two required and two optional arguments. In a list of constraints, the *j*-th item will be:

```
?_SURFACE_CONSTRAINTS(j) = &
SURFACE_CONSTRAINTS&
 ([DERIVATIVE=DERIVATIVE_INDEX(1:2),] &
 POINT = WHERE_APPLIED(1:2), [VALUE=VALUE_APPLIED,], &
 TYPE = CONSTRAINT_INDICATOR, &
 [PERIODIC_POINT = PERIODIC_POINT(1:2)])
```

The square brackets enclose optional arguments. For each constraint the arguments `'value ='` and `'PERIODIC_POINT ='` are not used at the same time.

### Required Arguments

`POINT = WHERE_APPLIED (Input)`

The point in the data domain where a constraint is to be applied. Each point has an *x* and *y* coordinate, in that order.

`TYPE = CONSTRAINT_INDICATOR (Input)`

The indicator for the type of constraint the tensor product spline function or its partial derivatives is to satisfy at the point: `where_applied`. The choices are the character strings `'=='`, `'<='`, `'>='`, `'=..'`, and `'=-.'`. They respectively indicate that the spline value or its derivatives will be equal to, not greater than, not less than, equal to the value of the spline at another point, or equal to the negative of the spline value at another point. These last two constraints are called *periodic* and *negative-periodic*, respectively.

### Optional Arguments

`DERIVATIVE = DERIVATIVE_INDEX(1:2) (Input)`

These are the number of the partial derivatives for the tensor product spline to apply the constraint. The array `(/0,0/)` corresponds to the function, the value `(/1,0/)` to the first partial derivative with respect to *x*, etc. If this argument is not present in the list, the value `(/0,0/)` is substituted automatically. Thus a constraint without the derivatives listed applies to the tensor product spline function.

`PERIODIC = PERIODIC_POINT(1:2)`

This optional argument improves readability by identifying the second pair of independent variable values for periodic constraints.

## FORTRAN 90 Interface

Generic: `CALL SURFACE_CONSTRAINTS (POINT, TYPE [, ...])`

Specific: The specific interface names are `S_SURFACE_CONSTRAINTS` and `D_SURFACE_CONSTRAINTS`.

---

# SURFACE\_VALUES

This rank-2 array function returns a tensor product array result, given two arrays of independent variable values. Use the optional input argument for the covariance matrix when the square root of the variance function is evaluated. The result will be a scalar value when the input independent variable is scalar.

## Required Arguments

`DERIVATIVE = DERIVATIVE (1:2)` (Input)

The indices of the partial derivative evaluated. Use non-negative integer values. For the function itself use the array `(/0, 0/)`.

`VARIABLESX = VARIABLESX` (Input)

The independent variable values in the first or  $x$  dimension where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

`VARIABLESY = VARIABLESY` (Input)

The independent variable values in the second or  $y$  dimension where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

`KNOTSX = KNOTSX` (Input)

The derived type `?_spline_knots`, used when the array `coeffs(:, :)` was obtained with the function `SURFACE_FITTING`. This contains the polynomial spline degree and the number of knots and the knots themselves, in the  $x$  dimension.

`KNOTSY = KNOTSY` (Input)

The derived type `?_spline_knots`, used when the array `coeffs(:, :)` was obtained with the function `SURFACE_FITTING`. This contains the polynomial spline degree and the number of knots and the knots themselves, in the  $y$  dimension.

`COEFFS = C` (Input)

The coefficients in the representation for the spline function,

$$f(x, y) = \sum_{j=1}^N \sum_{i=1}^M c_{ij} B_i(y) B_j(x)$$

These result from the fitting process or array assignment

`C = SURFACE_FITTING(...)`, defined below.

The values  $M = \text{size}(C, 1)$  and  $N = \text{size}(C, 2)$  satisfies the respective identities

$N - 1 + \text{spline\_degree} = \text{size} (?\_knotsx)$ , and  
 $M - 1 + \text{spline\_degree} = \text{size} (?\_knotsy)$ , where the two right-most quantities in both equations refer to components of the arguments `knotsx` and `knotsy`. The same value of `spline_degree` must be used for both `knotsx` and `knotsy`.

## Optional Arguments

`COVARIANCE = G` (Input)

This argument, when present, results in the evaluation of the square root of the variance function

$$e(x, y) = \left( b(x, y)^T G b(x, y) \right)^{1/2}$$

where

$$b(x, y) = \left[ B_1(x)B_1(y), \dots, B_N(x)B_1(y), \dots \right]^T$$

and  $G$  is the covariance matrix associated with the coefficients of the spline

$$c = \left[ c_{11}, \dots, c_{N1}, \dots \right]^T$$

The argument  $G$  is an optional output from `SURFACE_FITTING`, described below. When the square root of the variance function is computed, the arguments `DERIVATIVE` and `C` are not used.

`IOPT = IOPT` (Input)

This optional argument, of derived type `?_options`, is not used in this release.

## FORTRAN 90 Interface

Generic: `CALL SURFACE_VALUES (DERIVATIVE, VARIABLESX, VARIABLESY, KNOTSX, KNOTSY, COEFFS [, ...])`

Specific: The specific interface names are `S_SURFACE_VALUES` and `D_SURFACE_VALUES`.

---

# SURFACE\_FITTING

Weighted least-squares fitting by tensor product B-splines to discrete two-dimensional data is performed. Constraints on the spline or its partial derivatives are optional. The spline function

$$f(x, y) = \sum_{j=1}^N \sum_{i=1}^M c_{ij} B_i(y) B_j(x)$$

its derivatives, or the square root of its variance function are evaluated after the fitting.

## Required Arguments

DATA = DATA(1:4, :) (Input/Output)

An assumed-shape array with `size(data,1) = 4`. The data are placed in the array:

$$\text{data}(1, i) = x_i,$$

$$\text{data}(2, i) = y_i,$$

$$\text{data}(3, i) = z_i,$$

$$\text{data}(4, i) = \sigma_i, \quad i = 1, \dots, \text{ndata}.$$

If the variances are not known, but are proportional to an unknown value, use

$$\text{data}(4, i) = 1, \quad i = 1, \dots, \text{ndata}.$$

KNOTSX = KNOTSX (Input)

A derived type, `?_SPLINE_KNOTS`, that defines the degree of the spline and the breakpoints for the data fitting domain, in the first dimension.

KNOTSY = KNOTSX (Input)

A derived type, `?_SPLINE_KNOTS`, that defines the degree of the spline and the breakpoints for the data fitting domain, in the second dimension.

## Optional Arguments

CONSTRAINTS = SURFACE\_CONSTRAINTS (Input)

A rank-1 array of derived type `?_SURFACE_CONSTRAINTS` that defines constraints the tensor product spline is to satisfy.

COVARIANCE = G (Output)

An assumed-shape rank-2 array of the same precision as the data. This output is the covariance matrix of the coefficients. It is optionally used to evaluate the square root of the variance function.

IOPT = IOPT(:) (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `SURFACE_FITTING`. The options are as follows:

| Packaged Options for SURFACE_FITTING |                           |              |
|--------------------------------------|---------------------------|--------------|
| Prefix = None                        | Option Name               | Option Value |
|                                      | SURFACE_FITTING_SMALLNESS | 1            |
|                                      | SURFACE_FITTING_FLATNESS  | 2            |
|                                      | SURFACE_FITTING_TOL_EQUAL | 3            |
|                                      | SURFACE_FITTING_TOL_LEAST | 4            |
|                                      | SURFACE_FITTING_RESIDUALS | 5            |

| Packaged Options for SURFACE_FITTING |                          |   |
|--------------------------------------|--------------------------|---|
|                                      | SURFACE_FITTING_PRINT    | 6 |
|                                      | SURFACE_FITTING_THINNESS | 7 |

IOPT(IO) = ?\_OPTIONS&

(surface\_fitting\_smallnes, ?\_value)

This resets the square root of the regularizing parameter multiplying the squared integral of the unknown function. The argument ?\_value is replaced by the default value. The default is ?\_value = 0.

IOPT(IO) = ?\_OPTIONS&

(SURFACE\_FITTING\_FLATNESS, ?\_VALUE)

This resets the square root of the regularizing parameter multiplying the squared integral of the partial derivatives of the unknown function. The argument ?\_VALUE is replaced by the default value. The default is

?\_VALUE = SQRT(EPSILON(?\_VALUE))\*SIZE, where

$$size = \sum |data(3,:)/data(4,:)| / (ndata + 1).$$

IOPT(IO) = ?\_OPTIONS&

(SURFACE\_FITTING\_TOL\_EQUAL, ?\_VALUE)

This resets the value for determining that equality constraint equations are rank-deficient. The default is ?\_VALUE = 10<sup>-4</sup>.

IOPT(IO) = ?\_OPTIONS&

(SURFACE\_FITTING\_TOL\_LEAST, ?\_VALUE)

This resets the value for determining that least-squares equations are rank-deficient. The default is ?\_VALUE = 10<sup>-4</sup>.

IOPT(IO) = ?\_OPTIONS&

(SURFACE\_FITTING\_RESIDUALS, DUMMY)

This option returns the *residuals* = *surface* - *data*, in data(4,:). That row of the array is overwritten by the residuals. The data is returned in the order of cell processing order, or left-to-right in *x* and then increasing in *y*. The allocation of a temporary for data(1:4,:) is avoided, which may be desirable for problems with large amounts of data. The default is to not evaluate the residuals and to leave data(1:4,:) as input.

IOPT(IO) = ?\_OPTIONS&

(SURFACE\_FITTING\_PRINT, DUMMY)

This option prints the knots or breakpoints for *x* and *y*, and the count of data points in cell processing order. The default is to not print these arrays.

```
IOPT(IO) = ?_OPTIONS&
```

```
(SURFACE_FITTING_THINNESS, ?_VALUE)
```

This resets the square root of the regularizing parameter multiplying the squared integral of the second partial derivatives of the unknown function. The argument `?_VALUE` is replaced by the default value. The default is `?_VALUE = 10-3 × SIZE`, where

$$size = \sum |data(3,:) / data(4,:)| / (ndata + 1)$$

## FORTRAN 90 Interface

Generic: `CALL SURFACE_FITTING (DATA, KNOTSX, KNOTSX, KNOTSY [, ...])`

Specific: The specific interface names are `S_SURFACE_FITTING` and `D_SURFACE_FITTING`.

## Description

The coefficients are obtained by solving a least-squares system of linear algebraic equations, subject to linear equality and inequality constraints. The system is the result of the weighted data equations and regularization. If there are no constraints, the solution is computed using a banded least-squares solver. Details are found in Hanson (1995).

## Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `SURFACE_FITTING`. These error messages are numbered 1151-1152, 1161-1162, 1370-1393.

## Example 1: Tensor Product Spline Fitting of Data

The function

$$g(x, y) = \exp(-x^2 - y^2)$$

is least-squares fit by a tensor product of cubic splines on the square

$$[0, 2] \otimes [0, 2]$$

There are *ndata* random pairs of values for the independent variables. Each datum is given unit uncertainty. The grid of knots in both *x* and *y* dimensions are equally spaced, in the interior cells, and identical to each other. After the coefficients are computed a check is made that the surface approximately agrees with *g(x,y)* at a tensor product grid of equally spaced values.

```
USE surface_fitting_int
USE rand_int
USE norm_int
```

```
implicit none
```

```
! This is Example 1 for SURFACE_FITTING, tensor product
```

```

! B-splines approximation. Use the function
! exp(-x**2-y**2) on the square (0, 2) x (0, 2) for samples.
! The spline order is "nord" and the number of cells is
! "(ngrid-1)**2". There are "ndata" data values in the square.

integer :: i
integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
 nbkpt=ngrid+2*ndegree, ndata = 2000, nvalues=100
real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
real(kind(1d0)), parameter :: TOLERANCE=1d-3
real(kind(1d0)), target :: spline_data (4, ndata), bkpt(nbkpt), &
 coeff(ngrid+ndegree-1,ngrid+ndegree-1), delta, sizev, &
 x(nvalues), y(nvalues), values(nvalues, nvalues)

real(kind(1d0)), pointer :: pointer_bkpt(:)
type (d_spline_knots) knotsx, knotsy

! Generate random (x,y) pairs and evaluate the
! example exponential function at these values.
spline_data(1:2,:)=two*rand(spline_data(1:2,:))
spline_data(3,:)=exp(-sum(spline_data(1:2,:)**2,dim=1))
spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
delta = two/(ngrid-1)
bkpt(1:ndegree) = zero
bkpt(nbkpt-ndegree+1:nbkpt) = two
bkpt(nord:nbkpt-ndegree)=(/(i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
knotsx=d_spline_knots(ndegree, pointer_bkpt)
knotsy=knotsx

! Fit the data and obtain the coefficients.
coeff = surface_fitting(spline_data, knotsx, knotsy)

! Evaluate the residual = spline - function
! at a grid of points inside the square.
delta=two/(nvalues+1)
x=(/(i*delta,i=1,nvalues)/); y=x

values=exp(-spread(x**2,1,nvalues)-spread(y**2,2,nvalues))
values=surface_values(/(0,0/), x, y, knotsx, knotsy, coeff)-&
 values

! Compute the R.M.S. error:
sizev=norm(pack(values, (values == values)))/nvalues

if (sizev <= TOLERANCE) then
 write(*,*) 'Example 1 for SURFACE_FITTING is correct.'
end if
end

```

## Output

Example 1 for SURFACE\_FITTING is correct.

## Additional Examples

### Example 2: Parametric Representation of a Sphere

From Struik (1961), the parametric representation of points  $(x,y,z)$  on the surface of a sphere of radius  $a > 0$  is expressed in terms of *spherical coordinates*,

$$\begin{aligned}x(u,v) &= a \cos(u) \cos(v), \quad -\pi \leq 2u \leq \pi \\y(u,v) &= a \cos(u) \sin(v), \quad -\pi \leq v \leq \pi \\z(u,v) &= a \sin(u)\end{aligned}$$

The parameters are radians of *latitude* ( $u$ ) and *longitude* ( $v$ ). The example program fits the same *ndata* random pairs of latitude and longitude in each coordinate. We have covered the sphere twice by allowing

$$-\pi \leq u \leq \pi$$

for latitude. We solve three data fitting problems, one for each coordinate function. Periodic constraints on the value of the spline are used for both  $u$  and  $v$ . We could reduce the computational effort by fitting a spline function in one variable for the  $z$  coordinate. To illustrate the representation of more general surfaces than spheres, we did not do this. When the surface is evaluated we compute latitude, moving from the South Pole to the North Pole,

$$-\pi \leq 2u \leq \pi$$

Our surface will approximately satisfy the equality

$$x^2 + y^2 + z^2 = a^2$$

These residuals are checked at a rectangular mesh of latitude and longitude pairs. To illustrate the use of some options, we have reset the three regularization parameters to the value zero, the least-squares system tolerance to a smaller value than the default, and obtained the residuals for each parametric coordinate function at the data points.

```
USE surface_fitting_int
USE rand_int
USE norm_int
USE Numerical_Libraries

implicit none

! This is Example 2 for SURFACE_FITTING, tensor product
! B-splines approximation. Fit x, y, z parametric functions
! for points on the surface of a sphere of radius "A".
! Random values of latitude and longitude are used to generate
! data. The functions are evaluated at a rectangular grid
! in latitude and longitude and checked to lie on the surface
! of the sphere.

integer :: i, j
```

```

integer, parameter :: ngrid=6, nord=6, ndegree=nord-1, &
 nbkpt=ngrid+2*ndegree, ndata =1000, nvalues=50, NOPT=5
real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
real(kind(1d0)), parameter :: TOLERANCE=1d-2
real(kind(1d0)), target :: spline_data (4, ndata, 3), bkpt(nbkpt), &
 coeff(ngrid+ndegree-1,ngrid+ndegree-1, 3), delta, sizev, &
 pi, A, x(nvalues), y(nvalues), values(nvalues, nvalues), &
 data(4,ndata)

real(kind(1d0)), pointer :: pointer_bkpt(:)
type (d_spline_knots) knotsx, knotsy
type (d_options) OPTIONS(NOPT)

! Get the constant "pi" and a random radius, > 1.
pi = DCONST("pi"); A=one+rand(A)

! Generate random (latitude, longitude) pairs and evaluate the
! surface parameters at these points.
spline_data(1:2, :, 1)=pi*(two*rand(spline_data(1:2, :, 1))-one)
spline_data(1:2, :, 2)=spline_data(1:2, :, 1)
spline_data(1:2, :, 3)=spline_data(1:2, :, 1)

! Evaluate x, y, z parametric points.
spline_data(3, :, 1)=A*cos(spline_data(1, :, 1))*cos(spline_data(2, :, 1))
spline_data(3, :, 2)=A*cos(spline_data(1, :, 2))*sin(spline_data(2, :, 2))
spline_data(3, :, 3)=A*sin(spline_data(1, :, 3))

! The values are equally uncertain.
spline_data(4, :, :)=one

! Define the knots for the tensor product data fitting problem.
delta = two*pi/(ngrid-1)
bkpt(1:ndegree) = -pi
bkpt(nbkpt-ndegree+1:nbkpt) = pi
bkpt(nord:nbkpt-ndegree)=((-pi+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
knotsx=d_spline_knots(ndegree, pointer_bkpt)
knotsy=knotsx

! Fit a data surface for each coordinate.
! Set default regularization parameters to zero and compute
! residuals of the individual points. These are returned
! in DATA(4, :).
do j=1,3
 data=spline_data(:, :, j)
 OPTIONS(1)=d_options(surface_fitting_thinness,zero)
 OPTIONS(2)=d_options(surface_fitting_flatness,zero)
 OPTIONS(3)=d_options(surface_fitting_smallness,zero)
 OPTIONS(4)=d_options(surface_fitting_tol_least,1d-5)
 OPTIONS(5)=surface_fitting_residuals
 coeff(:, :, j) = surface_fitting(data, knotsx, knotsy, &
 IOPT=OPTIONS)
end do

```

```

! Evaluate the function at a grid of points inside the rectangle of
! latitude and longitude covering the sphere just once. Add the
! sum of squares. They should equal "A**2" but will not due to
! truncation and rounding errors.
 delta=pi/(nvalues+1)
 x=(/(-pi/two+i*delta,i=1,nvalues)/); y=two*x
 values=zero
 do j=1,3
 values=values+&
 surface_values((/0,0/), x, y, knotsx, knotsy, coeff(:, :,j))**2
 end do
 values=values-A**2
! Compute the R.M.S. error:

 sizev=norm(pack(values, (values == values)))/nvalues

 if (sizev <= TOLERANCE) then
 write(*,*) "Example 2 for SURFACE_FITTING is correct."
 end if
end
end

```

## Output

Example 2 for SURFACE\_FITTING is correct.

### Example 3: Constraining Some Points using a Spline Surface

This example illustrates the use of discrete constraints to shape the surface. The data fitting problem of Example 1 is modified by requiring that the surface interpolate the value one at  $x = y = 0$ . The shape is constrained so first partial derivatives in both  $x$  and  $y$  are zero at  $x = y = 0$ . These constraints mimic some properties of the function  $g(x,y)$ . The size of the residuals at a grid of points and the residuals of the constraints are checked.

```

USE surface_fitting_int
USE rand_int
USE norm_int

implicit none

! This is Example 3 for SURFACE_FITTING, tensor product
! B-splines approximation, f(x,y). Use the function
! exp(-x**2-y**2) on the square (0, 2) x (0, 2) for samples.
! The spline order is "nord" and the number of cells is
! "(ngrid-1)**2". There are "ndata" data values in the square.
! Constraints are put on the surface at (0,0). Namely
! f(0,0) = 1, f_x(0,0) = 0, f_y(0,0) = 0.

integer :: i
integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
 nbkpt=ngrid+2*ndegree, ndata = 2000, nvalues=100, NC = 3
real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
real(kind(1d0)), parameter :: TOLERANCE=1d-3
real(kind(1d0)), target :: spline_data (4, ndata), bkpt(nbkpt), &
 coeff(ngrid+ndegree-1,ngrid+ndegree-1), delta, sizev, &
 x(nvalues), y(nvalues), values(nvalues, nvalues), &
 f_00, f_x00, f_y00

```

```

real(kind(ld0)), pointer :: pointer_bkpt(:)
type (d_spline_knots) knotsx, knotsy
type (d_surface_constraints) C(NC)
LOGICAL PASS

! Generate random (x,y) pairs and evaluate the
! example exponential function at these values.
spline_data(1:2,:)=two*rand(spline_data(1:2,:))
spline_data(3,:)=exp(-sum(spline_data(1:2,:)**2,dim=1))
spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
delta = two/(ngrid-1)
bkpt(1:ndegree) = zero
bkpt(nbkpt-ndegree+1:nbkpt) = two
bkpt(nord:nbkpt-ndegree)=(/(i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
pointer_bkpt => bkpt
knotsx=d_spline_knots(ndegree, pointer_bkpt)
knotsy=knotsx

! Define the constraints for the fitted surface.
C(1)=surface_constraints(point=(/zero,zero/),type='==',value=one)
C(2)=surface_constraints(derivative=(/1,0/),&
point=(/zero,zero/),type='==',value=zero)
C(3)=surface_constraints(derivative=(/0,1/),&
point=(/zero,zero/),type='==',value=zero)

! Fit the data and obtain the coefficients.

coeff = surface_fitting(spline_data, knotsx, knotsy,&
CONSTRAINTS=C)

! Evaluate the residual = spline - function
! at a grid of points inside the square.
delta=two/(nvalues+1)
x=(/(i*delta,i=1,nvalues)/); y=x

values=exp(-spread(x**2,1,nvalues)-spread(y**2,2,nvalues))
values=surface_values(/0,0/), x, y, knotsx, knotsy, coeff)-&
values
f_00 = surface_values(/0,0/), zero, zero, knotsx, knotsy, coeff)
f_x00= surface_values(/1,0/), zero, zero, knotsx, knotsy, coeff)
f_y00= surface_values(/0,1/), zero, zero, knotsx, knotsy, coeff)

! Compute the R.M.S. error:
sizev=norm(pack(values, (values == values)))/nvalues
PASS = sizev <= TOLERANCE
PASS = abs (f_00 - one) <= sqrt(epsilon(one)) .and. PASS
PASS = f_x00 <= sqrt(epsilon(one)) .and. PASS
PASS = f_y00 <= sqrt(epsilon(one)) .and. PASS

if (PASS) then
write(*,*) 'Example 3 for SURFACE_FITTING is correct.'
```

```

end if
end

```

## Output

Example 3 for SURFACE\_FITTING is correct.

### Example 4: Constraining a Spline Surface to be non-Negative

The review of interpolating methods by Franke (1982) uses a test data set originally due to James Ferguson. We use this data set of 25 points, with unit uncertainty for each dependent variable. Our algorithm does not interpolate the data values but approximately fits them in the least-squares sense. We reset the regularization parameter values of *flatness* and *thinness*, Hanson (1995). Then the surface is fit to the data and evaluated at a grid of points. Although the surface appears smooth and fits the data, the values are negative near one corner. Our scenario for the application assumes that the surface be non-negative at all points of the rectangle containing the independent variable data pairs. Our algorithm for constraining the surface is simple but effective in this case. The data fitting is repeated one more time but with positive constraints at the grid of points where it was previously negative.

```

USE surface_fitting_int
USE rand_int
USE surface_fitting_int
USE rand_int
USE norm_int

implicit none

! This is Example 4 for SURFACE_FITTING, tensor product
! B-splines approximation, f(x,y). Use the data set from
! Franke, due to Ferguson. Without constraints the function
! becomes negative in a corner. Constrain the surface
! at a grid of values so it is non-negative.

integer :: i, j, q
integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
 nbkpt=ngrid+2*ndegree, ndata = 25, nvalues=50
real(kind(ld0)), parameter :: zero=0d0, one=1d0
real(kind(ld0)), parameter :: TOLERANCE=1d-3
real(kind(ld0)), target :: spline_data (4, ndata), bkptx(nbkpt), &
 bkpty(nbkpt), coeff(ngrid+ndegree-1,ngrid+ndegree-1), &
 x(nvalues), y(nvalues), values(nvalues, nvalues), &
 delta
real(kind(ld0)), pointer :: pointer_bkpt(:)
type (d_spline_knots) knotsx, knotsy
type (d_surface_constraints), allocatable :: C(:)

real(kind(1e0)) :: data (3*ndata) = & ! This is Ferguson's data:
(/2.0 , 15.0 , 2.5 , 2.49 , 7.647, 3.2,&
 2.981 , 0.291, 3.4 , 3.471, -7.062, 3.5,&
 3.961 , -14.418, 3.5 , 7.45 , 12.003, 2.5,&
 7.35 , 6.012, 3.5 , 7.251, 0.018, 3.0,&
 7.151 , -5.973, 2.0 , 7.051, -11.967, 2.5,&
 10.901, 9.015, 2.0 , 10.751, 4.536, 1.925,&
 10.602, 0.06 , 1.85, 10.453, -4.419, 1.576,&

```

```

10.304, -8.895, 1.7 , 14.055, 10.509, 1.5,&
14.194, 6.783, 1.3 , 14.331, 3.054, 1.7,&
14.469, -0.672, 2.1 , 14.607, -4.398, 1.75,&
15.0 , 12.0 , 0.5 , 15.729, 8.067, 0.5,&
16.457, 4.134, 0.7 , 17.185, 0.198, 1.1,&
17.914, -3.735, 1.7/)

 spline_data(1:3,:)=reshape(data,(/3,ndata/)); spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
! Use the data limits to the knot sequences.
 bkptx(1:ndegree) = minval(spline_data(1,:))
 bkptx(nbkpt-ndegree+1:nbkpt) = maxval(spline_data(1,:))
 delta=(bkptx(nbkpt)-bkptx(ndegree))/(ngrid-1)
 bkptx(nord:nbkpt-ndegree)=(/(bkptx(1)+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots for x.
 pointer_bkpt => bkptx
 knotsx=d_spline_knots(ndegree, pointer_bkpt)
 bkpty(1:ndegree) = minval(spline_data(2,:))
 bkpty(nbkpt-ndegree+1:nbkpt) = maxval(spline_data(2,:))
 delta=(bkpty(nbkpt)-bkpty(ndegree))/(ngrid-1)
 bkpty(nord:nbkpt-ndegree)=(/(bkpty(1)+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots for y.
 pointer_bkpt => bkpty
 knotsy=d_spline_knots(ndegree, pointer_bkpt)

! Fit the data and obtain the coefficients.
 coeff = surface_fitting(spline_data, knotsx, knotsy)

 delta=(bkptx(nbkpt)-bkptx(1))/(nvalues+1)
 x=(/(bkptx(1)+i*delta,i=1,nvalues)/)
 delta=(bkpty(nbkpt)-bkpty(1))/(nvalues+1)
 y=(/(bkpty(1)+i*delta,i=1,nvalues)/)

! Evaluate the function at a rectangular grid.
! Use non-positive values to a constraint.
 values=surface_values(/0,0/), x, y, knotsx, knotsy, coeff)

! Count the number of values <= zero. Then constrain the spline
! so that it is >= TOLERANCE at those points where it was <= zero.
 q=count(values <= zero)
 allocate (C(q))
 DO I=1,nvalues
 DO J=1,nvalues
 IF(values(I,J) <= zero) THEN
 C(q)=surface_constraints(point=(/x(i),y(j)/), type='>=',&
 value=TOLERANCE)
 q=q-1
 END IF
 END DO
 END DO

! Fit the data with constraints and obtain the coefficients.
 coeff = surface_fitting(spline_data, knotsx, knotsy,&

```

```

 CONSTRAINTS=C)
 deallocate(C)

! Evaluate the surface at a grid and check, once again, for
! non-positive values. All values should now be positive.
 values=surface_values((/0,0/), x, y, knotsx, knotsy, coeff)
if (count(values <= zero) == 0) then
 write(*,*) 'Example 4 for SURFACE_FITTING is correct.'
end if

end

```

## Output

Example 4 for SURFACE\_FITTING is correct.

# CSIEZ

Computes the cubic spline interpolant with the ‘not-a-knot’ condition and return values of the interpolant at specified points.

## Required Arguments

*XDATA* — Array of length *NDATA* containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

*FDATA* — Array of length *NDATA* containing the data point ordinates. (Input)

*XVEC* — Array of length *N* containing the points at which the spline is to be evaluated.  
(Input)

*VALUE* — Array of length *N* containing the values of the spline at the points in *XVEC*.  
(Output)

## Optional Arguments

*NDATA* — Number of data points. (Input)  
*NDATA* must be at least 2.  
Default: *NDATA* = size(*XDATA*,1).

*N* — Length of vector *XVEC*. (Input)  
Default: *N* = size(*XVEC*,1).

## FORTRAN 90 Interface

Generic: CALL CSIEZ (XDATA, FDATA, XVEC, VALUE [, ...])

Specific: The specific interface names are S\_CSIEZ and D\_CSIEZ.

## FORTRAN 77 Interface

Single:      CALL CSIEZ (NDATA, XDATA, FDATA, N, XVEC, VALUE)

Double:     The double precision name is DCSIEZ.

## Description

This routine is designed to let the user easily compute the values of a cubic spline interpolant. The routine CSIEZ computes a spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, \text{NDATA}$ . The output for this routine consists of a vector of values of the computed cubic spline. Specifically, let  $n = N$ ,  $v = \text{XVEC}$ , and  $y = \text{VALUE}$ , then if  $s$  is the computed spline we set

$$y_j = s(v_j) \quad j = 1, \dots, n$$

Additional documentation can be found by referring to the IMSL routines [CSINT](#) or [SPLEZ](#).

## Comments

Workspace may be explicitly provided, if desired, by use of C2IEZ/DC2IEZ. The reference is:

```
CALL C2IEZ (NDATA, XDATA, FDATA, N, XVEC, VALUE, IWK, WK1, WK2)
```

The additional arguments are as follows:

**IWK** — Integer work array of length  $\text{MAX0}(N, \text{NDATA}) + N$ .

**WK1** — Real work array of length  $5 * \text{NDATA}$ .

**WK2** — Real work array of length  $2 * N$ .

## Example

In this example, a cubic spline interpolant to a function  $F$  is computed. The values of this spline are then compared with the exact function values.

```
USE CSIEZ_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=11)
!
INTEGER I, NOUT
REAL F, FDATA(NDATA), FLOAT, SIN, VALUE(2*NDATA-1), X, &
XDATA(NDATA), XVEC(2*NDATA-1)
INTRINSIC FLOAT, SIN
!
! Define function
F(X) = SIN(15.0*X)
!
! Set up a grid
DO 10 I=1, NDATA
```

```

 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
 DO 20 I=1, 2*NDATA - 1
 XVEC(I) = FLOAT(I-1)/FLOAT(2*NDATA-2)
20 CONTINUE
!
! Compute cubic spline interpolant
 CALL CSIEZ (XDATA, FDATA, XVEC, VALUE)
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Write heading
 WRITE (NOUT,99998)
99998 FORMAT (13X, 'X', 9X, 'INTERPOLANT', 5X, 'ERROR')
!
! Print the interpolant and the error
! on a finer grid
 DO 30 I=1, 2*NDATA - 1
 WRITE (NOUT,99999) XVEC(I), VALUE(I), F(XVEC(I)) - VALUE(I)
30 CONTINUE
99999 FORMAT(' ', 2F15.3, F15.6)
END

```

## Output

| X     | INTERPOLANT | ERROR     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.809       | -0.127025 |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.723       | 0.055214  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.549      | -0.022789 |
| 0.300 | -0.978      | 0.000000  |
| 0.350 | -0.843      | -0.016246 |
| 0.400 | -0.279      | 0.000000  |
| 0.450 | 0.441       | 0.009348  |
| 0.500 | 0.938       | 0.000000  |
| 0.550 | 0.903       | 0.019947  |
| 0.600 | 0.412       | 0.000000  |
| 0.650 | -0.315      | -0.004895 |
| 0.700 | -0.880      | 0.000000  |
| 0.750 | -0.938      | -0.029541 |
| 0.800 | -0.537      | 0.000000  |
| 0.850 | 0.148       | 0.034693  |
| 0.900 | 0.804       | 0.000000  |
| 0.950 | 1.086       | -0.092559 |
| 1.000 | 0.650       | 0.000000  |

---

## CSINT

Computes the cubic spline interpolant with the ‘not-a-knot’ condition.

### Required Arguments

*XDATA* — Array of length *NDATA* containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

**FDATA** — Array of length `NDATA` containing the data point ordinates. (Input)

**BREAK** — Array of length `NDATA` containing the breakpoints for the piecewise cubic representation. (Output)

**CSCOEF** — Matrix of size 4 by `NDATA` containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)

`NDATA` must be at least 2.

Default: `NDATA = size (XDATA,1)`.

### FORTRAN 90 Interface

Generic: `CALL CSINT (XDATA, FDATA, BREAK, CSCOEF [, ...])`

Specific: The specific interface names are `S_CSINT` and `D_CSINT`.

### FORTRAN 77 Interface

Single: `CALL CSINT (NDATA, XDATA, FDATA, BREAK, CSCOEF)`

Double: The double precision name is `DCSINT`.

### Description

The routine `CSINT` computes a  $C^2$  cubic spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, \text{NDATA} = N$ . The breakpoints of the spline are the abscissas. Endpoint conditions are automatically determined by the program. These conditions correspond to the “not-a-knot” condition (see de Boor 1978), which requires that the third derivative of the spline be continuous at the second and next-to-last breakpoint. If  $N$  is 2 or 3, then the linear or quadratic interpolating polynomial is computed, respectively.

If the data points arise from the values of a smooth (say  $C^4$ ) function  $f$ , i.e.  $f_i = f(x_i)$ , then the error will behave in a predictable fashion. Let  $\xi$  be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \leq C \|f^{(4)}\|_{[\xi_1, \xi_N]} |\xi|_5^4$$

where

$$|\xi| := \max_{i=2, \dots, N} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, pages 55 – 56).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2INT/DC2INT`. The reference is:

```
CALL C2INT (NDATA, XDATA, FDATA, BREAK, CSCOEFF, IWK)
```

The additional argument is

**IWK** — Work array of length `NDATA`.

2. The cubic spline can be evaluated using `CSVAL`; its derivative can be evaluated using `CSDER`.
3. Note that column `NDATA` of `CSCOEFF` is used as workspace.

## Example

In this example, a cubic spline interpolant to a function  $F$  is computed. The values of this spline are then compared with the exact function values.

```
USE CSINT_INT
USE UMACH_INT
USE CSVAL_INT

IMPLICIT NONE
! Specifications
INTEGER NDATA
PARAMETER (NDATA=11)

!
INTEGER I, NINTV, NOUT
REAL BREAK(NDATA), CSCOEFF(4,NDATA), F,&
 FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA)
INTRINSIC FLOAT, SIN
! Define function
F(X) = SIN(15.0*X)
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
! Compute cubic spline interpolant
CALL CSINT (XDATA, FDATA, BREAK, CSCOEFF)
! Get output unit number.
CALL UMACH (2, NOUT)
! Write heading
WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
NINTV = NDATA - 1
! Print the interpolant and the error
! on a finer grid
DO 20 I=1, 2*NDATA - 1
 X = FLOAT(I-1)/FLOAT(2*NDATA-2)
 WRITE (NOUT, '(2F15.3, F15.6)') X, CSVAL(X, BREAK, CSCOEFF), &
 F(X) - CSVAL(X, BREAK, &
```

20 CONTINUE  
END

CSCOE F)

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.809       | -0.127025 |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.723       | 0.055214  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.549      | -0.022789 |
| 0.300 | -0.978      | 0.000000  |
| 0.350 | -0.843      | -0.016246 |
| 0.400 | -0.279      | 0.000000  |
| 0.450 | 0.441       | 0.009348  |
| 0.500 | 0.938       | 0.000000  |
| 0.550 | 0.903       | 0.019947  |
| 0.600 | 0.412       | 0.000000  |
| 0.650 | -0.315      | -0.004895 |
| 0.700 | -0.880      | 0.000000  |
| 0.750 | -0.938      | -0.029541 |
| 0.800 | -0.537      | 0.000000  |
| 0.850 | 0.148       | 0.034693  |
| 0.900 | 0.804       | 0.000000  |
| 0.950 | 1.086       | -0.092559 |
| 1.000 | 0.650       | 0.000000  |

---

## CSDEC

Computes the cubic spline interpolant with specified derivative endpoint conditions.

### Required Arguments

*XDATA* — Array of length *NDATA* containing the data point abscissas. (Input) The data point abscissas must be distinct.

*FDATA* — Array of length *NDATA* containing the data point ordinates. (Input)

*ILEFT* — Type of end condition at the left endpoint. (Input)

| <b>ILEFT</b> | <b>Condition</b>                            |
|--------------|---------------------------------------------|
| 0            | “Not-a-knot” condition                      |
| 1            | First derivative specified by <i>DLEFT</i>  |
| 2            | Second derivative specified by <i>DLEFT</i> |

**DLEFT** — Derivative at left endpoint if ILEFT is equal to 1 or 2. (Input)  
If ILEFT = 0, then DLEFT is ignored.

**IRIGHT** — Type of end condition at the right endpoint. (Input)

| <b>IRIGHT</b> | <b>Condition</b>                      |
|---------------|---------------------------------------|
| 0             | “Not-a-knot” condition                |
| 1             | First derivative specified by DRIGHT  |
| 2             | Second derivative specified by DRIGHT |

**DRIGHT** — Derivative at right endpoint if IRIGHT is equal to 1 or 2. (Input) If IRIGHT = 0 then DRIGHT is ignored.

**BREAK** — Array of length NDATA containing the breakpoints for the piecewise cubic representation. (Output)

**CSCOEF** — Matrix of size 4 by NDATA containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)  
Default: NDATA = size (XDATA,1).

### FORTRAN 90 Interface

Generic: CALL CSDEC (XDATA, FDATA, ILEFT, DLEFT, IRIGHT, DRIGHT, BREAK,  
CSCOEF [, ...])

Specific: The specific interface names are S\_CSDEC and D\_CSDEC.

### FORTRAN 77 Interface

Single: CALL CSDEC (NDATA, XDATA, FDATA, ILEFT, DLEFT, IRIGHT, DRIGHT,  
BREAK, CSCOEF)

Double: The double precision name is DCSDEC.

### Description

The routine CSDEC computes a  $C^2$  cubic spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, NDATA = N$ . The breakpoints of the spline are the abscissas. Endpoint conditions are to be selected by the user. The user may specify not-a-knot, first derivative, or second derivative at each endpoint (see de Boor 1978, Chapter 4).

If the data (including the endpoint conditions) arise from the values of a smooth (say  $C^4$ ) function  $f$ , i.e.  $f_i = f(x_i)$ , then the error will behave in a predictable fashion. Let  $\xi$  be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \leq C \|f^{(4)}\|_{[\xi_1, \xi_N]} |\xi|^4$$

where

$$|\xi| := \max_{i=2, \dots, N} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, Chapter 4 and 5).

## Comments

1. Workspace may be explicitly provided, if desired, by use of C2DEC/DC2DEC. The reference is:

```
CALL C2DEC (NDATA, XDATA, FDATA, ILEFT, DLEFT, IRIGHT,
 DRIGHT, BREAK, CSCOEFF, IWK)
```

The additional argument is:

**IWK** — Work array of length NDATA.

2. The cubic spline can be evaluated using [CSVAL](#); its derivative can be evaluated using [CSDER](#).
3. Note that column NDATA of CSCOEFF is used as workspace.

## Example 1

In Example 1, a cubic spline interpolant to a function  $f$  is computed. The value of the derivative at the left endpoint and the value of the second derivative at the right endpoint are specified. The values of this spline are then compared with the exact function values.

```
USE CSDEC_INT
USE UMACH_INT
USE CSVAL_INT

IMPLICIT NONE
INTEGER ILEFT, IRIGHT, NDATA
PARAMETER (ILEFT=1, IRIGHT=2, NDATA=11)
!
INTEGER I, NINTV, NOUT
REAL BREAK(NDATA), COS, CSCOEFF(4, NDATA), DLEFT, &
 DRIGHT, F, FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA)
INTRINSIC COS, FLOAT, SIN
!
F(X) = SIN(15.0*X)
!
! Define function
! Initialize DLEFT and DRIGHT
```

```

DLEFT = 15.0*COS(15.0*0.0)
DRIGHT = -15.0*15.0*SIN(15.0*1.0)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute cubic spline interpolant
CALL CSDEC (XDATA, FDATA, ILEFT, DLEFT, IRIGHT, &
 DRIGHT, BREAK, CSCOEFF)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
NINTV = NDATA - 1
!
! Print the interpolant on a finer grid
DO 20 I=1, 2*NDATA - 1
 X = FLOAT(I-1)/FLOAT(2*NDATA-2)
 WRITE (NOUT, '(2F15.3, F15.6)') X, CSVAL(X, BREAK, CSCOEFF), &
 F(X) - CSVAL(X, BREAK, &
 CSCOEFF)
20 CONTINUE
END

```

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.675       | 0.006332  |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.759       | 0.019485  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.558      | -0.013227 |
| 0.300 | -0.978      | 0.000000  |
| 0.350 | -0.840      | -0.018765 |
| 0.400 | -0.279      | 0.000000  |
| 0.450 | 0.440       | 0.009859  |
| 0.500 | 0.938       | 0.000000  |
| 0.550 | 0.902       | 0.020420  |
| 0.600 | 0.412       | 0.000000  |
| 0.650 | -0.312      | -0.007301 |
| 0.700 | -0.880      | 0.000000  |
| 0.750 | -0.947      | -0.020391 |
| 0.800 | -0.537      | 0.000000  |
| 0.850 | 0.182       | 0.000497  |
| 0.900 | 0.804       | 0.000000  |
| 0.950 | 0.959       | 0.035074  |
| 1.000 | 0.650       | 0.000000  |

## Additional Examples

### Example 2

In Example 2, we compute the *natural* cubic spline interpolant to a function  $f$  by forcing the second derivative of the interpolant to be zero at both endpoints. As in the previous example, we compare the exact function values with the values of the spline.

```
 USE CSDEC_INT
 USE UMACH_INT

 IMPLICIT NONE
 INTEGER ILEFT, IRIGHT, NDATA, NOUT
 PARAMETER (ILEFT=2, IRIGHT=2, NDATA=11)
!
 INTEGER I, NINTV
 REAL BREAK(NDATA), CSCOEFF(4,NDATA), DLEFT, DRIGHT, &
 F, FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA), CSVAL
 INTRINSIC FLOAT, SIN
!
! Initialize DLEFT and DRIGHT
 DATA DLEFT/0./, DRIGHT/0./
!
! Define function
 F(X) = SIN(15.0*X)
!
! Set up a grid
 DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute cubic spline interpolant
 CALL CSDEC (XDATA, FDATA, ILEFT, DLEFT, IRIGHT, DRIGHT, &
 BREAK, CSCOEFF)
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Write heading
 WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
 NINTV = NDATA - 1
!
! Print the interpolant on a finer grid
 DO 20 I=1, 2*NDATA - 1
 X = FLOAT(I-1)/FLOAT(2*NDATA-2)
 WRITE (NOUT,'(2F15.3,F15.6)') X, CSVAL(X,BREAK,CSCOEFF), &
 F(X) - CSVAL(X,BREAK, &
 CSCOEFF)
20 CONTINUE
 END
```

### Output

| X     | Interpolant | Error    |
|-------|-------------|----------|
| 0.000 | 0.000       | 0.000000 |
| 0.050 | 0.667       | 0.015027 |
| 0.100 | 0.997       | 0.000000 |
| 0.150 | 0.761       | 0.017156 |
| 0.200 | 0.141       | 0.000000 |

|       |        |           |
|-------|--------|-----------|
| 0.250 | -0.559 | -0.012609 |
| 0.300 | -0.978 | 0.000000  |
| 0.350 | -0.840 | -0.018907 |
| 0.400 | -0.279 | 0.000000  |
| 0.450 | 0.440  | 0.009812  |
| 0.500 | 0.938  | 0.000000  |
| 0.550 | 0.902  | 0.020753  |
| 0.600 | 0.412  | 0.000000  |
| 0.650 | -0.311 | -0.008586 |
| 0.700 | -0.880 | 0.000000  |
| 0.750 | -0.952 | -0.015585 |
| 0.800 | -0.537 | 0.000000  |

---

## CSHER

Computes the Hermite cubic spline interpolant.

### Required Arguments

*XDATA* — Array of length *NDATA* containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

*FDATA* — Array of length *NDATA* containing the data point ordinates. (Input)

*DFDATA* — Array of length *NDATA* containing the values of the derivative. (Input)

*BREAK* — Array of length *NDATA* containing the breakpoints for the piecewise cubic representation. (Output)

*CSCOEF* — Matrix of size 4 by *NDATA* containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

*NDATA* — Number of data points. (Input)  
Default: *NDATA* = size(*XDATA*,1).

### FORTRAN 90 Interface

Generic:    CALL CSHER (*XDATA*, *FDATA*, *DFDATA*, *BREAK*, *CSCOEF* [, ...])

Specific:   The specific interface names are *S\_CSHER* and *D\_CSHER*.

### FORTRAN 77 Interface

Single:     CALL CSHER (*NDATA*, *XDATA*, *FDATA*, *BREAK*, *CSCOEF*)

Double:     The double precision name is *DCSHER*.

## Description

The routine `CSHER` computes a  $C^1$  cubic spline interpolant to the set of data points

$$(x_i, f_i) \text{ and } (x_i, f'_i)$$

for  $i = 1, \dots, \text{NDATA} = N$ . The breakpoints of the spline are the abscissas.

If the data points arise from the values of a smooth (say  $C^4$ ) function  $f$ , i.e.,

$$f_i = f(x_i) \text{ and } f'_i = f'(x_i)$$

then the error will behave in a predictable fashion. Let  $\xi$  be the

breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \leq C \|f^{(4)}\|_{[\xi_1, \xi_N]} |\xi|^4$$

where

$$|\xi| := \max_{i=2, \dots, N} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, page 51).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2HER/DC2HER`. The reference is:

```
CALL C2HER (NDATA, XDATA, FDATA, DFDATA, BREAK, CSCOE,
 IWK)
```

The additional argument is:

**IWK** — Work array of length `NDATA`.

2. Informational error  
Type Code  
4 2 The `XDATA` values must be distinct.
3. The cubic spline can be evaluated using [CSVAL](#); its derivative can be evaluated using [CSDER](#).
4. Note that column `NDATA` of `CSCOE` is used as workspace.

## Example

In this example, a cubic spline interpolant to a function  $f$  is computed. The value of the function  $f$  and its derivative  $f'$  are computed on the interpolation nodes and passed to CSHER. The values of this spline are then compared with the exact function values.

```
USE CSHER_INT
USE UMACH_INT
USE CSVAL_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=11)

!
INTEGER I, NINTV, NOUT
REAL BREAK(NDATA), COS, CSCOEF(4,NDATA), DF, &
 DFDATA(NDATA), F, FDATA(NDATA), FLOAT, SIN, X, &
 XDATA(NDATA)
INTRINSIC COS, FLOAT, SIN

! Define function and derivative
F(X) = SIN(15.0*X)
DF(X) = 15.0*COS(15.0*X)

! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
 DFDATA(I) = DF(XDATA(I))
10 CONTINUE

! Compute cubic spline interpolant
CALL CSHER (XDATA, FDATA, DFDATA, BREAK, CSCOEF)

! Get output unit number
CALL UMACH (2, NOUT)

! Write heading
WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
NINTV = NDATA - 1

! Print the interpolant on a finer grid
DO 20 I=1, 2*NDATA - 1
 X = FLOAT(I-1)/FLOAT(2*NDATA-2)
 WRITE (NOUT,'(2F15.3, F15.6)') X, CSVAL(X,BREAK,CSCOEF) &
 , F(X) - CSVAL(X,BREAK, &
 CSCOEF)
20 CONTINUE
END
```

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.673       | 0.008654  |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.768       | 0.009879  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.564      | -0.007257 |
| 0.300 | -0.978      | 0.000000  |

|       |        |           |
|-------|--------|-----------|
| 0.350 | -0.848 | -0.010906 |
| 0.400 | -0.279 | 0.000000  |
| 0.450 | 0.444  | 0.005714  |
| 0.500 | 0.938  | 0.000000  |
| 0.550 | 0.911  | 0.011714  |
| 0.600 | 0.412  | 0.000000  |
| 0.650 | -0.315 | -0.004057 |
| 0.700 | -0.880 | 0.000000  |
| 0.750 | -0.956 | -0.012288 |
| 0.800 | -0.537 | 0.000000  |
| 0.850 | 0.180  | 0.002318  |
| 0.900 | 0.804  | 0.000000  |
| 0.950 | 0.981  | 0.012616  |
| 1.000 | 0.650  | 0.000000  |

---

## CSAKM

Computes the Akima cubic spline interpolant.

### Required Arguments

***XDATA*** — Array of length *NDATA* containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

***FDATA*** — Array of length *NDATA* containing the data point ordinates. (Input)

***BREAK*** — Array of length *NDATA* containing the breakpoints for the piecewise cubic representation. (Output)

***CSCOEFL*** — Matrix of size 4 by *NDATA* containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

***NDATA*** — Number of data points. (Input)  
Default: *NDATA* = size(*XDATA*,1).

### FORTRAN 90 Interface

Generic:     CALL CSAKM (*XDATA*, *FDATA*, *BREAK*, *CSCOEFL* [, ...])

Specific:    The specific interface names are *S\_CSAKM* and *D\_CSAKM*.

### FORTRAN 77 Interface

Single:      CALL CSAKM (*NDATA*, *XDATA*, *FDATA*, *BREAK*, *CSCOEFL*)

Double:     The double precision name is *DCSAKM*.

## Description

The routine `CSAKM` computes a  $C^1$  cubic spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, \text{NDATA} = N$ . The breakpoints of the spline are the abscissas. Endpoint conditions are automatically determined by the program; see Akima (1970) or de Boor (1978).

If the data points arise from the values of a smooth (say  $C^4$ ) function  $f$ , i.e.  $f_i = f(x_i)$ , then the error will behave in a predictable fashion. Let  $\xi$  be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \leq C \|f^{(2)}\|_{[\xi_1, \xi_N]} |\xi|^2$$

where

$$|\xi| := \max_{i=2, \dots, N} |\xi_i - \xi_{i-1}|$$

The routine `CSAKM` is based on a method by Akima (1970) to combat wiggles in the interpolant. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.)

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2AKMD/C2AKM`. The reference is:

```
CALL C2AKM (NDATA, XDATA, FDATA, BREAK, CSCOE, IWK)
```

The additional argument is:

**IWK** — Work array of length `NDATA`.

2. The cubic spline can be evaluated using `CSVAL`; its derivative can be evaluated using `CSDER`.
3. Note that column `NDATA` of `CSCOE` is used as workspace.

## Example

In this example, a cubic spline interpolant to a function  $f$  is computed. The values of this spline are then compared with the exact function values.

```
USE CSAKM_INT
USE UMACH_INT
USE CSVAL_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=11)

!
INTEGER I, NINTV, NOUT
REAL BREAK (NDATA), CSCOE (4, NDATA), F, &
```

```

 FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA)
INTRINSIC FLOAT, SIN
!
! Define function
F(X) = SIN(15.0*X)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute cubic spline interpolant
CALL CSAKM (XDATA, FDATA, BREAK, CSCOEFF)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
NINTV = NDATA - 1
!
! Print the interpolant on a finer grid
DO 20 I=1, 2*NDATA - 1
 X = FLOAT(I-1)/FLOAT(2*NDATA-2)
 WRITE (NOUT,'(2F15.3,F15.6)') X, CSVAL(X,BREAK,CSCOEFF), &
 F(X) - CSVAL(X,BREAK,&
 CSCOEFF)
20 CONTINUE
END

```

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.818       | -0.135988 |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.615       | 0.163487  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.478      | -0.093376 |
| 0.300 | -0.978      | 0.000000  |
| 0.350 | -0.812      | -0.046447 |
| 0.400 | -0.279      | 0.000000  |
| 0.450 | 0.386       | 0.064491  |
| 0.500 | 0.938       | 0.000000  |
| 0.550 | 0.854       | 0.068274  |
| 0.600 | 0.412       | 0.000000  |
| 0.650 | -0.276      | -0.043288 |
| 0.700 | -0.880      | 0.000000  |
| 0.750 | -0.889      | -0.078947 |
| 0.800 | -0.537      | 0.000000  |
| 0.850 | 0.149       | 0.033757  |
| 0.900 | 0.804       | 0.000000  |
| 0.950 | 0.932       | 0.061260  |
| 1.000 | 0.650       | 0.000000  |

---

## CSCON

Computes a cubic spline interpolant that is consistent with the concavity of the data.

## Required Arguments

***XDATA*** — Array of length *N*DATA containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

***FDATA*** — Array of length *N*DATA containing the data point ordinates. (Input)

***IBREAK*** — The number of breakpoints. (Output)  
It will be less than  $2 * N$ DATA.

***BREAK*** — Array of length *IBREAK* containing the breakpoints for the piecewise cubic representation in its first *IBREAK* positions. (Output)  
The dimension of *BREAK* must be at least  $2 * N$ DATA.

***CSCOE****F* — Matrix of size 4 by *N* where *N* is the dimension of *BREAK*. (Output)  
The first *IBREAK* - 1 columns of *CSCOE**F* contain the local coefficients of the cubic pieces.

## Optional Arguments

***N*DATA** — Number of data points. (Input)  
*N*DATA must be at least 3.  
Default: *N*DATA = size (*XDATA*,1).

## FORTRAN 90 Interface

Generic:     CALL CSCON (*XDATA*, *FDATA*, *IBREAK*, *BREAK*, *CSCOE**F* [, ...])

Specific:    The specific interface names are *S\_CSCON* and *D\_CSCON*.

## FORTRAN 77 Interface

Single:      CALL CSCON (*N*DATA, *XDATA*, *FDATA*, *IBREAK*, *BREAK*, *CSCOE**F*)

Double:     The double precision name is *DCSCON*.

## Description

The routine *CSCON* computes a cubic spline interpolant to  $n = N$ DATA data points  $\{x_i, f_i\}$  for  $i = 1, \dots, n$ . For ease of explanation, we will assume that  $x_i < x_{i+1}$ , although it is not necessary for the user to sort these data values. If the data are strictly convex, then the computed spline is convex,  $C^2$ , and minimizes the expression

$$\int_{x_1}^{x_n} (g''')^2$$

over all convex  $C^1$  functions that interpolate the data. In the general case when the data have both convex and concave regions, the convexity of the spline is consistent with the data and the above

integral is minimized under the appropriate constraints. For more information on this interpolation scheme, we refer the reader to Micchelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this subroutine is that it is not possible, a priori, to predict the number of breakpoints of the resulting interpolant. In most cases, there will be breakpoints at places other than data locations. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.) This routine should be used when it is important to preserve the convex and concave regions implied by the data.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2CON/DC2CON`. The reference is:

```
CALL C2CON (NDATA, XDATA, FDATA, IBREAK, BREAK, CSCOE,
 ITMAX, XSRT, FSRT, A, Y, DIVD, ID, WK)
```

The additional arguments are as follows:

**ITMAX** — Maximum number of iterations of Newton's method. (Input)

**XSRT** — Work array of length `NDATA` to hold the sorted `XDATA` values.

**FSRT** — Work array of length `NDATA` to hold the sorted `FDATA` values.

**A** — Work array of length `NDATA`.

**Y** — Work array of length `NDATA - 2`.

**DIVD** — Work array of length `NDATA - 2`.

**ID** — Integer work array of length `NDATA`.

**WK** — Work array of length  $5 * (NDATA - 2)$ .

2. Informational errors

| Type | Code |                                                                                                                       |
|------|------|-----------------------------------------------------------------------------------------------------------------------|
| 3    | 16   | Maximum number of iterations exceeded, call <code>C2CON/DC2CON</code> to set a larger number for <code>ITMAX</code> . |
| 4    | 3    | The <code>XDATA</code> values must be distinct.                                                                       |

3. The cubic spline can be evaluated using `CSVAL`; its derivative can be evaluated using `CSDER`.

4. The default value for `ITMAX` is 25. This can be reset by calling `C2CON/DC2CON` directly.

## Example

We first compute the shape-preserving interpolant using `CSCON`, and display the coefficients and breakpoints. Second, we interpolate the same data using `CSINT` in a program not shown and overlay the two results. The graph of the result from `CSINT` is represented by the dashed line. Notice the extra inflection points in the curve produced by `CSINT`.

```
USE CSCON_INT
USE UMACH_INT
USE WRRRL_INT

IMPLICIT NONE
!
! Specifications
INTEGER NDATA
PARAMETER (NDATA=9)
!
INTEGER IBREAK, NOUT
REAL BREAK(2*NDATA), CSCOE(4,2*NDATA), FDATA(NDATA), &
 XDATA(NDATA)
CHARACTER CLABEL(14)*2, RLABEL(4)*2
!
DATA XDATA/0.0, .1, .2, .3, .4, .5, .6, .8, 1./
DATA FDATA/0.0, .9, .95, .9, .1, .05, .05, .2, 1./
DATA RLABEL/' 1', ' 2', ' 3', ' 4'/
DATA CLABEL/' ', ' 1', ' 2', ' 3', ' 4', ' 5', ' 6', &
 ' 7', ' 8', ' 9', '10', '11', '12', '13'/
!
! Compute cubic spline interpolant
CALL CSCON (XDATA, FDATA, IBREAK, BREAK, CSCOE)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Print the BREAK points and the
! coefficients (CSCOE) for
! checking purposes.
WRITE (NOUT, '(1X,A,I2)') 'IBREAK = ', IBREAK
CALL WRRRL ('BREAK', BREAK, RLABEL, CLABEL, 1, IBREAK, 1, &
 FMT='(F9.3)')
CALL WRRRL ('CSCOE', CSCOE, RLABEL, CLABEL, 4, IBREAK, 4, &
 FMT='(F9.3)')
END
```

## Output

```
IBREAK = 13
 BREAK
 1 2 3 4 5 6
1 0.000 0.100 0.136 0.200 0.259 0.300
 7 8 9 10 11 12
1 0.400 0.436 0.500 0.600 0.609 0.800
 13
1 1.000
 CSCOE
 1 2 3 4 5 6
1 0.000 0.900 0.942 0.950 0.958 0.900
```

|   |           |          |       |       |           |          |
|---|-----------|----------|-------|-------|-----------|----------|
| 2 | 11.886    | 3.228    | 0.131 | 0.131 | 0.131     | -4.434   |
| 3 | 0.000     | -173.170 | 0.000 | 0.000 | 0.000     | 220.218  |
| 4 | -1731.699 | 4841.604 | 0.000 | 0.000 | -5312.082 | 4466.875 |
|   | 7         | 8        | 9     | 10    | 11        | 12       |
| 1 | 0.100     | 0.050    | 0.050 | 0.050 | 0.050     | 0.200    |
| 2 | -4.121    | 0.000    | 0.000 | 0.000 | 0.000     | 2.356    |
| 3 | 226.470   | 0.000    | 0.000 | 0.000 | 0.000     | 24.664   |
| 4 | -6222.348 | 0.000    | 0.000 | 0.000 | 129.115   | 123.321  |
|   | 13        |          |       |       |           |          |
| 1 | 1.000     |          |       |       |           |          |
| 2 | 0.000     |          |       |       |           |          |
| 3 | 0.000     |          |       |       |           |          |
| 4 | 0.000     |          |       |       |           |          |

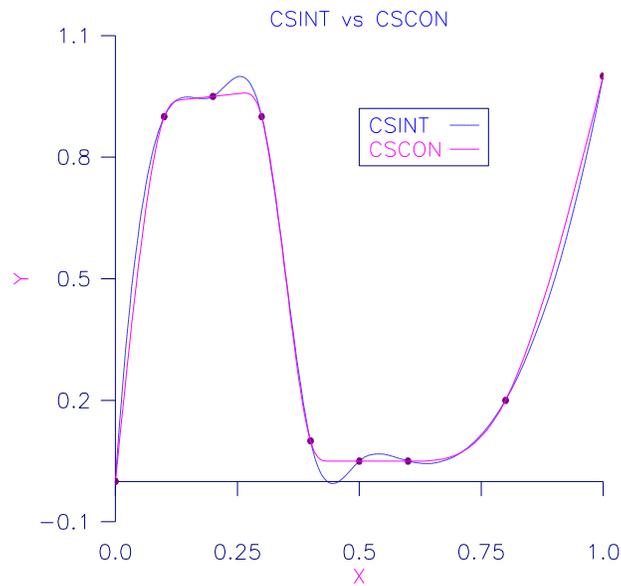


Figure 3-4 *CSCON* vs. *CSINT*

---

## CSPER

Computes the cubic spline interpolant with periodic boundary conditions.

### Required Arguments

***XDATA*** — Array of length *N**DATA* containing the data point abscissas. (Input)  
The data point abscissas must be distinct.

***FDATA*** — Array of length *N**DATA* containing the data point ordinates. (Input)

**BREAK** — Array of length `NDATA` containing the breakpoints for the piecewise cubic representation. (Output)

**CSCOEFF** — Matrix of size 4 by `NDATA` containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)  
NDATA must be at least 4.  
Default: `NDATA = size (XDATA,1)`.

### FORTRAN 90 Interface

Generic: `CALL CSPER (XDATA, FDATA, BREAK, CSCOEFF [, ...])`

Specific: The specific interface names are `S_CSPER` and `D_CSPER`.

### FORTRAN 77 Interface

Single: `CALL CSPER (NDATA, XDATA, FDATA, BREAK, CSCOEFF)`

Double: The double precision name is `DCSPER`.

### Description

The routine `CSPER` computes a  $C^2$  cubic spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, \text{NDATA} = N$ . The breakpoints of the spline are the abscissas. The program enforces periodic endpoint conditions. This means that the spline  $s$  satisfies  $s(a) = s(b)$ ,  $s'(a) = s'(b)$ , and  $s''(a) = s''(b)$ , where  $a$  is the leftmost abscissa and  $b$  is the rightmost abscissa. If the ordinate values corresponding to  $a$  and  $b$  are not equal, then a warning message is issued. The ordinate value at  $b$  is set equal to the ordinate value at  $a$  and the interpolant is computed.

If the data points arise from the values of a smooth (say  $C^4$ ) periodic function  $f$ , i.e.  $f_i = f(x_i)$ , then the error will behave in a predictable fashion. Let  $\xi$  be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \leq C \|f^{(4)}\|_{[\xi_1, \xi_N]} |\xi|^4$$

where

$$|\xi| := \max_{i=2, \dots, N} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, pages 320–322).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2PER/DC2PER`. The reference is:

```
CALL C2PER (NDATA, XDATA, FDATA, BREAK, CSCOE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $6 * \text{NDATA}$ .

**IWK** — Work array of length `NDATA`.

2. Informational error

Type Code

|   |   |                                                                                                                                                                                          |
|---|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The data set is not periodic, i.e., the function values at the smallest and largest <code>XDATA</code> points are not equal. The value at the smallest <code>XDATA</code> point is used. |
|---|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

3. The cubic spline can be evaluated using `CSVVAL` and its derivative can be evaluated using `CSDER`.

## Example

In this example, a cubic spline interpolant to a function  $f$  is computed. The values of this spline are then compared with the exact function values.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=11)

!
INTEGER I, NINTV, NOUT
REAL BREAK(NDATA), CSCOE(4,NDATA), F,&
 FDATA(NDATA), FLOAT, H, PI, SIN, X, XDATA(NDATA)
INTRINSIC FLOAT, SIN

!
! Define function
F(X) = SIN(15.0*X)

! Set up a grid
PI = CONST('PI')
H = 2.0*PI/15.0/10.0
DO 10 I=1, NDATA
 XDATA(I) = H*FLOAT(I-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE

! Round off will cause FDATA(11) to
! be nonzero; this would produce a
! warning error since FDATA(1) is zero.
! Therefore, the value of FDATA(1) is
```

```

! used rather than the value of
! FDATA(11).
 FDATA(NDATA) = FDATA(1)
!
! Compute cubic spline interpolant
 CALL CSPER (XDATA, FDATA, BREAK, CSCOE)
! Get output unit number
 CALL UMACH (2, NOUT)
! Write heading
 WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
 NINTV = NDATA - 1
 H = H/2.0
! Print the interpolant on a finer grid
 DO 20 I=1, 2*NDATA - 1
 X = H*FLOAT(I-1)
 WRITE (NOUT, '(2F15.3, F15.6)') X, CSVAL(X, BREAK, CSCOE), &
 F(X) - CSVAL(X, BREAK, &
 CSCOE)
20 CONTINUE
 END

```

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.021 | 0.309       | 0.000138  |
| 0.042 | 0.588       | 0.000000  |
| 0.063 | 0.809       | 0.000362  |
| 0.084 | 0.951       | 0.000000  |
| 0.105 | 1.000       | 0.000447  |
| 0.126 | 0.951       | 0.000000  |
| 0.147 | 0.809       | 0.000362  |
| 0.168 | 0.588       | 0.000000  |
| 0.188 | 0.309       | 0.000138  |
| 0.209 | 0.000       | 0.000000  |
| 0.230 | -0.309      | -0.000138 |
| 0.251 | -0.588      | 0.000000  |
| 0.272 | -0.809      | -0.000362 |
| 0.293 | -0.951      | 0.000000  |
| 0.314 | -1.000      | -0.000447 |
| 0.335 | -0.951      | 0.000000  |
| 0.356 | -0.809      | -0.000362 |
| 0.377 | -0.588      | 0.000000  |
| 0.398 | -0.309      | -0.000138 |
| 0.419 | 0.000       | 0.000000  |

---

## CSVAL

This function evaluates a cubic spline.

### Function Return Value

*CSVAL* — Value of the polynomial at *x*. (Output)

## Required Arguments

*X* — Point at which the spline is to be evaluated. (Input)

*BREAK* — Array of length  $NINTV + 1$  containing the breakpoints for the piecewise cubic representation. (Input)  
*BREAK* must be strictly increasing.

*CSCOEFF* — Matrix of size 4 by  $NINTV + 1$  containing the local coefficients of the cubic pieces. (Input)

## Optional Arguments

*NINTV* — Number of polynomial pieces. (Input)

## FORTRAN 90 Interface

Generic: `CSVAL (X, BREAK, CSCOEFF [, ...])`

Specific: The specific interface names are `S_CSVAL` and `D_CSVAL`.

## FORTRAN 77 Interface

Single: `CSVAL (X, NINTV, BREAK, CSCOEFF)`

Double: The double precision function name is `DCSVAL`.

## Description

The routine `CSVAL` evaluates a cubic spline at a given point. It is a special case of the routine `PPDER`, which evaluates the derivative of a piecewise polynomial. (The value of a piecewise polynomial is its zero-th derivative and a cubic spline is a piecewise polynomial of order 4.) The routine `PPDER` is based on the routine `PPVALU` in de Boor (1978, page 89).

## Example

For an example of the use of `CSVAL`, see IMSL routine `CSINT`.

---

# CSDER

This function evaluates the derivative of a cubic spline.

## Function Return Value

*CSDER* — Value of the `IDERIV`-th derivative of the polynomial at *X*. (Output)

## Required Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)  
In particular,  $IDERIV = 0$  returns the value of the polynomial.

**X** — Point at which the polynomial is to be evaluated. (Input)

**BREAK** — Array of length  $NINTV + 1$  containing the breakpoints for the piecewise cubic representation. (Input)  
**BREAK** must be strictly increasing.

**CSCOEF** — Matrix of size 4 by  $NINTV + 1$  containing the local coefficients of the cubic pieces. (Input)

## Optional Arguments

**NINTV** — Number of polynomial pieces. (Input)  
Default:  $NINTV = \text{size}(\text{BREAK}, 1) - 1$ .

## FORTRAN 90 Interface

Generic: `CSDER (IDERIV, X, BREAK, CSCOEF, CSDER [, ...])`

Specific: The specific interface names are `S_CSDER` and `D_CSDER`.

## FORTRAN 77 Interface

Single: `CSDER (IDERIV, X, NINTV, BREAK, CSCOEF)`

Double: The double precision function name is `DCSDER`.

## Description

The function `CSDER` evaluates the derivative of a cubic spline at a given point. It is a special case of the routine `PPDER`, which evaluates the derivative of a piecewise polynomial. (A cubic spline is a piecewise polynomial of order 4.) The routine `PPDER` is based on the routine `PPVALU` in de Boor (1978, page 89).

## Example

In this example, we compute a cubic spline interpolant to a function  $f$  using IMSL routine `CSINT`. The values of the spline and its first and second derivatives are computed using `CSDER`. These values can then be compared with the corresponding values of the interpolated function.

```
USE CSDER_INT
USE CSINT_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=10)
```

```

!
INTEGER I, NINTV, NOUT
REAL BREAK(NDATA), CDDE, CDF, CF, COS, CSCOE(4,NDATA), &
 DDF, DF, F, FDATA(NDATA), FLOAT, SIN, X, &
 XDATA(NDATA)
INTRINSIC COS, FLOAT, SIN
!
! Define function and derivatives
F(X) = SIN(15.0*X)
DF(X) = 15.0*COS(15.0*X)
DDF(X) = -225.0*SIN(15.0*X)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute cubic spline interpolant
CALL CSINT (XDATA, FDATA, BREAK, CSCOE)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
99999 FORMAT (9X, 'X', 8X, 'S(X)', 5X, 'Error', 6X, 'S''(X)', 5X, &
 'Error', 6X, 'S''''(X)', 4X, 'Error', /)
NINTV = NDATA - 1
!
! Print the interpolant on a finer grid
DO 20 I=1, 2*NDATA
 X = FLOAT(I-1)/FLOAT(2*NDATA-1)
 CF = CSDER(0,X,BREAK,CSCOE)
 CDF = CSDER(1,X,BREAK,CSCOE)
 CDDF = CSDER(2,X,BREAK,CSCOE)
 WRITE (NOUT,'(F11.3, 3(F11.3, F11.6))') X, CF, F(X) - CF, &
 CDF, DF(X) - CDF, &
 CDDF, DDF(X) - CDDF
20 CONTINUE
END

```

## Output

| X     | S(X)   | Error     | S'(X)   | Error      | S''(X)   | Error      |
|-------|--------|-----------|---------|------------|----------|------------|
| 0.000 | 0.000  | 0.000000  | 26.285  | -11.284739 | -379.458 | 379.457794 |
| 0.053 | 0.902  | -0.192203 | 8.841   | 1.722460   | -283.411 | 123.664734 |
| 0.105 | 1.019  | -0.019333 | -3.548  | 3.425718   | -187.364 | -37.628586 |
| 0.158 | 0.617  | 0.081009  | -10.882 | 0.146207   | -91.317  | -65.824875 |
| 0.211 | -0.037 | 0.021155  | -13.160 | -1.837700  | 4.730    | -1.062027  |
| 0.263 | -0.674 | -0.046945 | -10.033 | -0.355268  | 117.916  | 44.391640  |
| 0.316 | -0.985 | -0.015060 | -0.719  | 1.086203   | 235.999  | -11.066727 |
| 0.368 | -0.682 | -0.004651 | 11.314  | -0.409097  | 154.861  | -0.365387  |
| 0.421 | 0.045  | -0.011915 | 14.708  | 0.284042   | -25.887  | 18.552732  |
| 0.474 | 0.708  | 0.024292  | 9.508   | 0.702690   | -143.785 | -21.041260 |
| 0.526 | 0.978  | 0.020854  | 0.161   | -0.771948  | -211.402 | -13.411087 |
| 0.579 | 0.673  | 0.001410  | -11.394 | 0.322443   | -163.483 | 11.674103  |
| 0.632 | -0.064 | 0.015118  | -14.937 | -0.045511  | 28.856   | -17.856323 |
| 0.684 | -0.724 | -0.019246 | -8.859  | -1.170871  | 163.866  | 3.435547   |
| 0.737 | -0.954 | -0.044143 | 0.301   | 0.554493   | 184.217  | 40.417282  |

|       |        |           |         |           |          |            |
|-------|--------|-----------|---------|-----------|----------|------------|
| 0.789 | -0.675 | 0.012143  | 10.307  | 0.928152  | 166.021  | -16.939514 |
| 0.842 | 0.027  | 0.038176  | 15.015  | -0.047344 | 12.914   | -27.575521 |
| 0.895 | 0.764  | -0.010112 | 11.666  | -1.819128 | -140.193 | -29.538193 |
| 0.947 | 1.114  | -0.116304 | 0.258   | -1.357680 | -293.301 | 68.905701  |
| 1.000 | 0.650  | 0.000000  | -19.208 | 7.812407  | -446.408 | 300.092896 |

---

## CS1GD

Evaluates the derivative of a cubic spline on a grid.

### Required Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)

In particular, `IDERIV = 0` returns the values of the cubic spline.

**XVEC** — Array of length `N` containing the points at which the cubic spline is to be evaluated.

(Input)

The points in `XVEC` should be strictly increasing.

**BREAK** — Array of length `NINTV + 1` containing the breakpoints for the piecewise cubic representation. (Input)

`BREAK` must be strictly increasing.

**CSCOEF** — Matrix of size 4 by `NINTV + 1` containing the local coefficients of the cubic pieces. (Input)

**VALUE** — Array of length `N` containing the values of the `IDERIV`-th derivative of the cubic spline at the points in `XVEC`. (Output)

### Optional Arguments

**N** — Length of vector `XVEC`. (Input)

Default: `N = size (XVEC,1)`.

**NINTV** — Number of polynomial pieces. (Input)

Default: `NINTV = size (BREAK,1) - 1`.

### FORTRAN 90 Interface

Generic: `CALL CS1GD (IDERIV, XVEC, BREAK, CSCOEF, VALUE [, ...])`

Specific: The specific interface names are `S_CS1GD` and `D_CS1GD`.

### FORTRAN 77 Interface

Single: `CALL CS1GD (IDERIV, N, XVEC, NINTV, BREAK, CSCOEF, VALUE)`

Double: The double precision name is `DCS1GD`.

## Description

The routine `CS1GD` evaluates a cubic spline (or its derivative) at a vector of points. That is, given a vector  $x$  of length  $n$  satisfying  $x_i < x_{i+1}$  for  $i = 1, \dots, n - 1$ , a derivative value  $j$ , and a cubic spline  $s$  that is represented by a breakpoint sequence and coefficient matrix this routine returns the values

$$s^{(j)}(x_i) \quad i = 1, \dots, n$$

in the array `VALUE`. The functionality of this routine is the same as that of `CSDER` called in a loop, however `CS1GD` should be much more efficient.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C21GD/DC21GD`. The reference is:

```
CALL C21GD (IDERIV, N, XVEC, NINTV, BREAK, CSCOE, VALUE,
 IWK, WORK1, WORK2)
```

The additional arguments are as follows:

**IWK** — Array of length  $N$ .

**WORK1** — Array of length  $N$ .

**WORK2** — Array of length  $N$ .

2. Informational error

Type Code

4 4 The points in `XVEC` must be strictly increasing.

## Example

To illustrate the use of `CS1GD`, we modify the example program for `CSINT`. In this example, a cubic spline interpolant to  $F$  is computed. The values of this spline are then compared with the exact function values. The routine `CS1GD` is based on the routine `PPVALU` in de Boor (1978, page 89).

```
USE CS1GD_INT
USE CSINT_INT
USE UMACH_INT
USE CSVAL_INT

IMPLICIT NONE
!
! Specifications
INTEGER NDATA, N, IDERIV, J
PARAMETER (NDATA=11, N=2*NDATA-1)
!
INTEGER I, NINTV, NOUT
REAL BREAK (NDATA), CSCOE (4, NDATA), F, &
```

```

 FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA), &
 FVALUE(N), VALUE(N), XVEC(N)
INTRINSIC FLOAT, SIN
!
! Define function
F(X) = SIN(15.0*X)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute cubic spline interpolant
CALL CSINT (XDATA, FDATA, BREAK, CSCOEf)
DO 20 I=1, N
 XVEC(I) = FLOAT(I-1)/FLOAT(2*NDATA-2)
 FVALUE(I) = F(XVEC(I))
20 CONTINUE
IDERIV = 0
NINTV = NDATA - 1
CALL CS1GD (IDERIV, XVEC, BREAK, CSCOEf, VALUE)
!
! Get output unit number.
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
99999 FORMAT (13X, 'X', 9X, 'Interpolant', 5X, 'Error')
!
! Print the interpolant and the error
! on a finer grid
DO 30 J=1, N
 WRITE (NOUT,'(2F15.3,F15.6)') XVEC(J), VALUE(J), &
 FVALUE(J)-VALUE(J)
30 CONTINUE
END

```

## Output

| X     | Interpolant | Error     |
|-------|-------------|-----------|
| 0.000 | 0.000       | 0.000000  |
| 0.050 | 0.809       | -0.127025 |
| 0.100 | 0.997       | 0.000000  |
| 0.150 | 0.723       | 0.055214  |
| 0.200 | 0.141       | 0.000000  |
| 0.250 | -0.549      | -0.022789 |
| 0.300 | -0.978      | 0.000000  |
| 0.350 | -0.843      | -0.016246 |
| 0.400 | -0.279      | 0.000000  |
| 0.450 | 0.441       | 0.009348  |
| 0.500 | 0.938       | 0.000000  |
| 0.550 | 0.903       | 0.019947  |
| 0.600 | 0.412       | 0.000000  |
| 0.650 | -0.315      | -0.004895 |
| 0.700 | -0.880      | 0.000000  |
| 0.750 | -0.938      | -0.029541 |
| 0.800 | -0.537      | 0.000000  |
| 0.850 | 0.148       | 0.034693  |
| 0.900 | 0.804       | 0.000000  |
| 0.950 | 1.086       | -0.092559 |
| 1.000 | 0.650       | 0.000000  |

---

# CSITG

This function evaluates the integral of a cubic spline.

## Function Return Value

*CSITG* — Value of the integral of the spline from *A* to *B*. (Output)

## Required Arguments

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. (Input)

*BREAK* — Array of length  $NINTV + 1$  containing the breakpoints for the piecewise cubic representation. (Input)  
*BREAK* must be strictly increasing.

*CSCOEF* — Matrix of size 4 by  $NINTV + 1$  containing the local coefficients of the cubic pieces. (Input)

## Optional Arguments

*NINTV* — Number of polynomial pieces. (Input)  
Default:  $NINTV = \text{size}(\text{BREAK},1) - 1$ .

## FORTRAN 90 Interface

Generic: `CSITG (A, B, BREAK, CSCOEF [, ...])`

Specific: The specific interface names are `S_CSITG` and `D_CSITG`.

## FORTRAN 77 Interface

Single: `CSITG (A, B, NINTV, BREAK, CSCOEF)`

Double: The double precision function name is `DCSITG`.

## Description

The function `CSITG` evaluates the integral of a cubic spline over an interval. It is a special case of the routine `PPITG`, which evaluates the integral of a piecewise polynomial. (A cubic spline is a piecewise polynomial of order 4.)

## Example

This example computes a cubic spline interpolant to the function  $x^2$  using `CSINT` and evaluates its integral over the intervals  $[0., .5]$  and  $[0., 2.]$ . Since `CSINT` uses the not-a knot condition, the interpolant reproduces  $x^2$ , hence the integral values are  $1/24$  and  $8/3$ , respectively.

```
USE CSITG_INT
USE UMACH_INT
USE CSINT_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=10)

!
INTEGER I, NINTV, NOUT
REAL A, B, BREAK(NDATA), CSCOEEF(4,NDATA), ERROR, &
 EXACT, F, FDATA(NDATA), FI, FLOAT, VALUE, X, &
 XDATA(NDATA)
INTRINSIC FLOAT

! Define function and integral
F(X) = X*X
FI(X) = X*X*X/3.0

! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE

! Compute cubic spline interpolant
CALL CSINT (XDATA, FDATA, BREAK, CSCOEEF)

! Compute the integral of F over
! [0.0,0.5]
A = 0.0
B = 0.5
NINTV = NDATA - 1
VALUE = CSITG(A,B,BREAK,CSCOEEF)
EXACT = FI(B) - FI(A)
ERROR = EXACT - VALUE

! Get output unit number
CALL UMACH (2, NOUT)

! Print the result
WRITE (NOUT,99999) A, B, VALUE, EXACT, ERROR

! Compute the integral of F over
! [0.0,2.0]
A = 0.0
B = 2.0
VALUE = CSITG(A,B,BREAK,CSCOEEF)
EXACT = FI(B) - FI(A)
ERROR = EXACT - VALUE

! Print the result
WRITE (NOUT,99999) A, B, VALUE, EXACT, ERROR
99999 FORMAT (' On the closed interval (', F3.1, ',', F3.1, &
 ') we have :', /, 1X, 'Computed Integral = ', F10.5, /, &
 1X, 'Exact Integral = ', F10.5, /, 1X, 'Error ' &
 ', ' = ', F10.6, /, /)

END
```

## Output

On the closed interval (0.0,0.5) we have :  
Computed Integral = 0.04167  
Exact Integral = 0.04167  
Error = 0.000000

On the closed interval (0.0,2.0) we have :  
Computed Integral = 2.66666  
Exact Integral = 2.66667  
Error = 0.000006

---

## SPLEZ



Computes the values of a spline that either interpolates or fits user-supplied data.

### Required Arguments

***XDATA*** — Array of length *N*DATA containing the data point abscissae. (Input)  
The data point abscissas must be distinct.

***FDATA*** — Array of length *N*DATA containing the data point ordinates. (Input)

***XVEC*** — Array of length *N* containing the points at which the spline function values are desired. (Input)  
The entries of *XVEC* must be distinct.

***VALUE*** — Array of length *N* containing the spline values. (Output)  
 $VALUE(I) = S(XVEC(I))$  if *IDER* = 0,  $VALUE(I) = S'(XVEC(I))$  if *IDER* = 1, and so forth, where *S* is the computed spline.

### Optional Arguments

***N*DATA** — Number of data points. (Input)  
Default: *N*DATA = size(*XDATA*,1).

All choices of *I*TYPE are valid if *N*DATA is larger than 6. More specifically,

*N*DATA > *I*TYPE                      or *I*TYPE = 1.

*N*DATA > 3                              for *I*TYPE = 2, 3.

*N*DATA > (*I*TYPE - 3)              for *I*TYPE = 4, 5, 6, 7, 8.

NDATA > 3                    for ITYPE = 9, 10, 11, 12.  
NDATA > KORDER            for ITYPE = 13, 14, 15.

**ITYPE** — Type of interpolant desired. (Input)  
Default: ITYPE = 1.

**ITYPE**

- 1    yields CSINT
- 2    yields CSAKM
- 3    yields CSCON
- 4    yields BSINT-BSNAK K = 2
- 5    yields BSINT-BSNAK K = 3
- 6    yields BSINT-BSNAK K = 4
- 7    yields BSINT-BSNAK K = 5
- 8    yields BSINT-BSNAK K = 6
- 9    yields CSSCV
- 10   yields BSLSQ K = 2
- 11   yields BSLSQ K = 3
- 12   yields BSLSQ K = 4
- 13   yields BSVLS K = 2
- 14   yields BSVLS K = 3
- 15   yields BSVLS K = 4

**IDER** — Order of the derivative desired. (Input)  
Default: IDER = 0.

**N** — Number of function values desired. (Input)  
Default: N = size (XVEC,1).

### **FORTRAN 90 Interface**

Generic:    CALL SPLEZ (XDATA, FDATA, XVEC, VALUE [, ...])

Specific:    The specific interface names are S\_SPLEZ and D\_SPLEZ.

### **FORTRAN 77 Interface**

Single:    CALL SPLEZ (NDATA, XDATA, FDATA, ITYPE, IDER, N, XVEC, VALUE)

Double:    The double precision name is DSPLEZ.

## Description

This routine is designed to let the user experiment with various interpolation and smoothing routines in the library.

The routine SPLEZ computes a spline interpolant to a set of data points  $(x_i, f_i)$  for  $i = 1, \dots, \text{NDATA}$  if  $\text{ITYPE} = 1, \dots, 8$ . If  $\text{ITYPE} \geq 9$ , various smoothing or least squares splines are computed. The output for this routine consists of a vector of values of the computed spline or its derivatives. Specifically, let  $i = \text{IDER}$ ,  $n = N$ ,  $v = \text{XVEC}$ , and  $y = \text{VALUE}$ , then if  $s$  is the computed spline we set

$$y_j = s^{(i)}(v_j) \quad j = 1, \dots, n$$

The routines called are listed above under the  $\text{ITYPE}$  heading. Additional documentation can be found by referring to these routines.

## Example

In this example, all the  $\text{ITYPE}$  parameters are exercised. The values of the spline are then compared with the exact function values and derivatives.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NDATA, N
PARAMETER (NDATA=21, N=2*NDATA-1)
!
! Specifications for local variables
INTEGER I, IDER, ITYPE, NOUT
REAL FDATA(NDATA), FPVAL(N), FVALUE(N), &
VALUE(N), XDATA(NDATA), XVEC(N), EMAX1(15), &
EMAX2(15), X
!
! Specifications for intrinsics
INTRINSIC FLOAT, SIN, COS
REAL FLOAT, SIN, COS
!
! Specifications for subroutines
!
REAL F, FP
!
! Define a function
F(X) = SIN(X*X)
FP(X) = 2*X*COS(X*X)
!
CALL UMACH (2, NOUT)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = 3.0*(FLOAT(I-1)/FLOAT(NDATA-1))
 FDATA(I) = F(XDATA(I))
10 CONTINUE
DO 20 I=1, N
 XVEC(I) = 3.0*(FLOAT(I-1)/FLOAT(2*NDATA-2))
 FVALUE(I) = F(XVEC(I))
 FPVAL(I) = FP(XVEC(I))
20 CONTINUE
!
WRITE (NOUT,99999)
!
! Loop to call SPLEZ for each ITYPE
```

```

DO 40 ITYPE=1, 15
DO 30 IDER=0, 1
CALL SPLEZ (XDATA, FDATA, XVEC, VALUE, ITYPE=ITYPE, &
 IDER=IDER)
!
 Compute the maximum error
IF (IDER .EQ. 0) THEN
CALL SAXPY (N, -1.0, FVALUE, 1, VALUE, 1)
EMAX1(ITYPE) = ABS(VALUE(ISAMAX(N,VALUE,1)))
ELSE
CALL SAXPY (N, -1.0, FPVAL, 1, VALUE, 1)
EMAX2(ITYPE) = ABS(VALUE(ISAMAX(N,VALUE,1)))
END IF
30 CONTINUE
WRITE (NOUT,'(I7,2F20.6)') ITYPE, EMAX1(ITYPE), EMAX2(ITYPE)
40 CONTINUE
!
99999 FORMAT (4X, 'ITYPE', 6X, 'Max error for f', 5X,&
 'Max error for f'', /)
END

```

## Output

| ITYPE | Max error for f | Max error for f' |
|-------|-----------------|------------------|
| 1     | 0.014082        | 0.658018         |
| 2     | 0.024682        | 0.897757         |
| 3     | 0.020896        | 0.813228         |
| 4     | 0.083615        | 2.168083         |
| 5     | 0.010403        | 0.508043         |
| 6     | 0.014082        | 0.658020         |
| 7     | 0.004756        | 0.228858         |
| 8     | 0.001070        | 0.077159         |
| 9     | 0.020896        | 0.813228         |
| 10    | 0.392603        | 6.047916         |
| 11    | 0.162793        | 1.983959         |
| 12    | 0.045404        | 1.582624         |
| 13    | 0.588370        | 7.680381         |
| 14    | 0.752475        | 9.673786         |
| 15    | 0.049340        | 1.713031         |

---

## BSINT



Computes the spline interpolant, returning the B-spline coefficients.

### Required Arguments

*NDATA* — Number of data points. (Input)

*XDATA* — Array of length *NDATA* containing the data point abscissas. (Input)

**FDATA** — Array of length `NDATA` containing the data point ordinates. (Input)

**KORDER** — Order of the spline. (Input)  
`KORDER` must be less than or equal to `NDATA`.

**XKNOT** — Array of length `NDATA + KORDER` containing the knot sequence. (Input)  
`XKNOT` must be nondecreasing.

**BSCOEF** — Array of length `NDATA` containing the B-spline coefficients. (Output)

### **FORTRAN 90 Interface**

Generic:     `CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)`

Specific:    The specific interface names are `S_BSINT` and `D_BSINT`.

### **FORTRAN 77 Interface**

Single:      `CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)`

Double:     The double precision name is `DBSINT`.

### **Description**

Following the notation in de Boor (1978, page 108), let  $B_j = B_{j,k,t}$  denote the  $j$ -th B-spline of order  $k$  with respect to the knot sequence  $\mathbf{t}$ . Then, `BSINT` computes the vector  $\mathbf{a}$  satisfying

$$\sum_{j=1}^N a_j B_j(x_i) = f_i$$

and returns the result in `BSCOEF = a`. This linear system is banded with at most  $k - 1$  subdiagonals and  $k - 1$  superdiagonals. The matrix

$$A = (B_j(x_i))$$

is totally positive and is invertible if and only if the diagonal entries are nonzero. The routine `BSINT` is based on the routine `SPLINT` by de Boor (1978, page 204).

The routine `BSINT` produces the coefficients of the B-spline interpolant of order `KORDER` with knot sequence `XKNOT` to the data  $(x_i, f_i)$  for  $i = 1$  to `NDATA`, where  $x = \text{XDATA}$  and  $f = \text{FDATA}$ . Let  $\mathbf{t} = \text{XKNOT}$ ,  $k = \text{KORDER}$ , and  $N = \text{NDATA}$ . First, `BSINT` sorts the `XDATA` vector and stores the result in  $x$ . The elements of the `FDATA` vector are permuted appropriately and stored in  $f$ , yielding the equivalent data  $(x_i, f_i)$  for  $i = 1$  to  $N$ . The following preliminary checks are performed on the data. We verify that

$$\begin{aligned}
 x_i &< x_{i+1} & i = 1, \dots, N-1 \\
 \mathbf{t}_i &< \mathbf{t}_{i+1} & i = 1, \dots, N \\
 \mathbf{t}_i &\leq \mathbf{t}_{i+k} & i = 1, \dots, N+k-1
 \end{aligned}$$

The first test checks to see that the abscissas are distinct. The second and third inequalities verify that a valid knot sequence has been specified.

In order for the interpolation matrix to be nonsingular, we also check  $\mathbf{t}_k \leq x \leq \mathbf{t}_{N+1}$  for  $i = 1$  to  $N$ . This first inequality in the last check is necessary since the method used to generate the entries of the interpolation matrix requires that the  $k$  possibly nonzero B-splines at  $x_i$ ,

$$B_{j-k+1}, \dots, B_j \text{ where } j \text{ satisfies } \mathbf{t}_j \leq x_i < \mathbf{t}_{j+1}$$

be well-defined (that is,  $j - k + 1 \geq 1$ ).

General conditions are not known for the exact behavior of the error in spline interpolation, however, if  $\mathbf{t}$  and  $x$  are selected properly and the data points arise from the values of a smooth (say  $C^k$ ) function  $f$ , i.e.  $f_i = f(x_i)$ , then the error will behave in a predictable fashion. The maximum absolute error satisfies

$$\|f - s\|_{[\mathbf{t}_k, \mathbf{t}_{N+1}]} \leq C \|f^{(k)}\|_{[\mathbf{t}_k, \mathbf{t}_{N+1}]} |\mathbf{t}|^k$$

where

$$|\mathbf{t}| := \max_{i=k, \dots, N} |\mathbf{t}_{i+1} - \mathbf{t}_i|$$

For more information on this problem, see de Boor (1978, Chapter 13) and the references therein. This routine can be used in place of the IMSL routine `CSINT` by calling `BSNAK` to obtain the proper knots, then calling `BSINT` yielding the B-spline coefficients, and finally calling IMSL routine `BSCPP` to convert to piecewise polynomial form.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2INT/DB2INT`. The reference is:

```
CALL B2INT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOE,
 WK1, WK2, WK3, IWK)
```

The additional arguments are as follows:

**WK1** — Work array of length  $(5 * KORDER - 2) * NDATA$ .

**WK2** — Work array of length  $NDATA$ .

**WK3** — Work array of length  $NDATA$ .

**IWK** — Work array of length  $NDATA$ .



```

CALL BSNAK (NDATA, XDATA, KORDER, XKNOT)
! Interpolate
CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOE)
! Get output unit number
CALL UMACH (2, NOUT)
! Write heading
WRITE (NOUT,99999)
! Print on a finer grid
NCOEF = NDATA
XT = XDATA(1)
! Evaluate spline
BT = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOE)
WRITE (NOUT,99998) XT, BT, F(XT) - BT
DO 20 I=2, NDATA
 XT = (XDATA(I-1)+XDATA(I))/2.0
! Evaluate spline
 BT = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOE)
 WRITE (NOUT,99998) XT, BT, F(XT) - BT
 XT = XDATA(I)
! Evaluate spline
 BT = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOE)
 WRITE (NOUT,99998) XT, BT, F(XT) - BT
20 CONTINUE
99998 FORMAT (' ', F6.4, 15X, F8.4, 12X, F11.6)
99999 FORMAT ('/', 6X, 'X', 19X, 'S(X)', 18X, 'Error', '/')
END

```

## Output

| X      | S(X)   | Error     |
|--------|--------|-----------|
| 0.0000 | 0.0000 | 0.000000  |
| 0.1250 | 0.2918 | 0.061781  |
| 0.2500 | 0.5000 | 0.000000  |
| 0.3750 | 0.6247 | -0.012311 |
| 0.5000 | 0.7071 | 0.000000  |
| 0.6250 | 0.7886 | 0.002013  |
| 0.7500 | 0.8660 | 0.000000  |
| 0.8750 | 0.9365 | -0.001092 |
| 1.0000 | 1.0000 | 0.000000  |

---

## BSNAK

Computes the “not-a-knot” spline knot sequence.

### Required Arguments

**NDATA** — Number of data points. (Input)

**XDATA** — Array of length *NDATA* containing the location of the data points. (Input)

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length *NDATA* + *KORDER* containing the knot sequence. (Output)

## FORTRAN 90 Interface

Generic:     CALL BSNAK (NDATA, XDATA, KORDER, XKNOT)

Specific:    The specific interface names are S\_BSNAK and D\_BSNAK.

## FORTRAN 77 Interface

Single:     CALL BSNAK (NDATA, XDATA, KORDER, XKNOT)

Double:     The double precision name is DBSNAK.

## Description

Given the data points  $x = XDATA$ , the order of the spline  $k = KORDER$ , and the number  $N = NDATA$  of elements in  $XDATA$ , the subroutine BSNAK returns in  $\mathbf{t} = XKNOT$  a knot sequence that is appropriate for interpolation of data on  $x$  by splines of order  $k$ . The vector  $\mathbf{t}$  contains the knot sequence in its first  $N + k$  positions. If  $k$  is even and we assume that the entries in the input vector  $x$  are increasing, then  $\mathbf{t}$  is returned as

$$\mathbf{t}_i = x_1 \quad \text{for } i = 1, \dots, k$$

$$\mathbf{t}_i = x_{i-k/2} \quad \text{for } i = k + 1, \dots, N$$

$$\mathbf{t}_i = x_N + \varepsilon \quad \text{for } i = N + 1, \dots, N + k$$

where  $\varepsilon$  is a small positive constant. There is some discussion concerning this selection of knots in de Boor (1978, page 211). If  $k$  is odd, then  $\mathbf{t}$  is returned as

$$\mathbf{t}_i = x_1 \quad \text{for } i = 1, \dots, k$$

$$\mathbf{t}_i = (x_{i-\frac{k-1}{2}} + x_{i-1-\frac{k-1}{2}}) / 2 \quad \text{for } i = k + 1, \dots, N$$

$$\mathbf{t}_i = x_N + \varepsilon \quad \text{for } i = N + 1, \dots, N + k$$

It is not necessary to sort the values in  $x$  since this is done in the routine BSNAK.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of B2NAK/DB2NAK. The reference is:

CALL B2NAK (NDATA, XDATA, KORDER, XKNOT, XSRT, IWK)

The additional arguments are as follows:

**XSRT** — Work array of length NDATA to hold the sorted XDATA values. If XDATA is not needed, XSRT may be the same as XDATA.

**IWK** — Work array of length `NDATA` to hold the permutation of `XDATA`.

2. Informational error

Type Code

4 4 The `XDATA` values must be distinct.

3. The first knot is at the left endpoint and the last knot is slightly beyond the last endpoint. Both endpoints have multiplicity `KORDER`.

4. Interior knots have multiplicity one.

### Example

In this example, we compute (for  $k = 3, \dots, 8$ ) six spline interpolants  $s_k$  to  $F(x) = \sin(10x^3)$  on the interval  $[0,1]$ . The routine `BSNAK` is used to generate the knot sequences for  $s_k$  and then `BSINT` is called to obtain the interpolant. We evaluate the absolute error

$$|s_k - F|$$

at 100 equally spaced points and print the maximum error for each  $k$ .

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KMAX, KMIN, NDATA
PARAMETER (KMAX=8, KMIN=3, NDATA=20)
!
INTEGER I, K, KORDER, NOUT
REAL ABS, AMAX1, BSCOEFF(NDATA), DIF, DIFMAX, F, &
 FDATA(NDATA), FLOAT, FT, SIN, ST, T, X, XDATA(NDATA), &
 XKNOT(KMAX+NDATA), XT
INTRINSIC ABS, AMAX1, FLOAT, SIN
!
F(X) = SIN(10.0*X*X*X)
T(X) = 1.0 - X*X
!
DO 10 I=1, NDATA
 XT = FLOAT(I-1)/FLOAT(NDATA-1)
 XDATA(I) = T(XT)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
CALL UMACH (2, NOUT)
!
WRITE (NOUT,99999)
!
DO 30 K=KMIN, KMAX
 KORDER = K
!
 CALL BSNAK (NDATA, XDATA, KORDER, XKNOT)
!
 CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEFF)
 DIFMAX = 0.0
```

```

 DO 20 I=1, 100
 XT = FLOAT(I-1)/99.0
! Evaluate spline
 ST = BSVAL(XT,KORDER,XKNOT,NDATA,BSCOEF)
 FT = F(XT)
 DIF = ABS(FT-ST)
! Compute maximum difference
 DIFMAX = AMAX1(DIF,DIFMAX)
20 CONTINUE
! Print maximum difference
 WRITE (NOUT,99998) KORDER, DIFMAX
30 CONTINUE
!
99998 FORMAT (' ', I3, 5X, F9.4)
99999 FORMAT (' KORDER', 5X, 'Maximum difference', /)
 END

```

## Output

| KORDER | Maximum difference |
|--------|--------------------|
| 3      | 0.0080             |
| 4      | 0.0026             |
| 5      | 0.0004             |
| 6      | 0.0008             |
| 7      | 0.0010             |
| 8      | 0.0004             |

---

## BSOPK



Computes the “optimal” spline knot sequence.

### Required Arguments

**NDATA** — Number of data points. (Input)

**XDATA** — Array of length *NDATA* containing the location of the data points. (Input)

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length *NDATA* + *KORDER* containing the knot sequence. (Output)

### FORTRAN 90 Interface

Generic:     CALL BSOPK (NDATA, XDATA, KORDER, XKNOT)

Specific:    The specific interface names are S\_BSOPK and D\_BSOPK.

## FORTRAN 77 Interface

Single:      CALL BSOPK (NDATA, XDATA, KORDER, XKNOT)

Double:     The double precision name is DBSOPK.

## Description

Given the abscissas  $x = \text{XDATA}$  for an interpolation problem and the order of the spline interpolant  $k = \text{KORDER}$ , BSOPK returns the knot sequence  $\mathbf{t} = \text{XKNOT}$  that minimizes the constant in the error estimate

$$\|f - s\| \leq c \|f^{(k)}\|$$

In the above formula,  $f$  is any function in  $C^k$  and  $s$  is the spline interpolant to  $f$  at the abscissas  $x$  with knot sequence  $\mathbf{t}$ .

The algorithm is based on a routine described in de Boor (1978, page 204), which in turn is based on a theorem of Micchelli, Rivlin and Winograd (1976).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of B2OPK/DB2OPK. The reference is:

```
CALL B2OPK (NDATA, XDATA, KORDER, XKNOT, MAXIT, WK, IWK)
```

The additional arguments are as follows:

**MAXIT** — Maximum number of iterations of Newton's Method. (Input) A suggested value is 10.

**WK** — Work array of length  $(\text{NDATA} - \text{KORDER}) * (3 * \text{KORDER} - 2) + 6 * \text{NDATA} + 2 * \text{KORDER} + 5$ .

**IWK** — Work array of length NDATA.

2.    Informational errors

Type   Code

3      6      Newton's method iteration did not converge.

4      3      The XDATA values must be distinct.

4      4      Interpolation matrix is singular. The XDATA values may be too close together.

3.    The default value for MAXIT is 10, this can be overridden by calling B2OPK/DB2OPK directly with a larger value.

## Example

In this example, we compute (for  $k = 3, \dots, 8$ ) six spline interpolants  $s_k$  to  $F(x) = \sin(10x^3)$  on the interval  $[0, 1]$ . The routine BSOPK is used to generate the knot sequences for  $s_k$  and then BSINT is called to obtain the interpolant. We evaluate the absolute error

$$|s_k - F|$$

at 100 equally spaced points and print the maximum error for each  $k$ .

```
USE BSOPK_INT
USE BSINT_INT
USE UMACH_INT
USE BSVAL_INT

IMPLICIT NONE
INTEGER KMAX, KMIN, NDATA
PARAMETER (KMAX=8, KMIN=3, NDATA=20)
!
INTEGER I, K, KORDER, NOUT
REAL ABS, AMAX1, BSCOEFF(NDATA), DIF, DIFMAX, F, &
 FDATA(NDATA), FLOAT, FT, SIN, ST, T, X, XDATA(NDATA), &
 XKNOT(KMAX+NDATA), XT
INTRINSIC ABS, AMAX1, FLOAT, SIN
! Define function and tau function
F(X) = SIN(10.0*X*X*X)
T(X) = 1.0 - X*X
!
! Set up data
DO 10 I=1, NDATA
 XT = FLOAT(I-1)/FLOAT(NDATA-1)
 XDATA(I) = T(XT)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
!
! Loop over different orders
DO 30 K=KMIN, KMAX
 KORDER = K
!
! Generate knots
CALL BSOPK (NDATA, XDATA, KORDER, XKNOT)
! Interpolate
CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEFF)
DIFMAX = 0.0
DO 20 I=1, 100
 XT = FLOAT(I-1)/99.0
!
! Evaluate spline
 ST = BSVAL(XT, KORDER, XKNOT, NDATA, BSCOEFF)
 FT = F(XT)
 DIF = ABS(FT-ST)
!
! Compute maximum difference
 DIFMAX = AMAX1(DIF, DIFMAX)
20 CONTINUE
!
! Print maximum difference
```

```

 WRITE (NOUT,99998) KORDER, DIFMAX
 30 CONTINUE
!
99998 FORMAT (' ', I3, 5X, F9.4)
99999 FORMAT (' KORDER', 5X, 'Maximum difference', /)
END

```

## Output

| KORDER | Maximum difference |
|--------|--------------------|
| 3      | 0.0096             |
| 4      | 0.0018             |
| 5      | 0.0005             |
| 6      | 0.0004             |
| 7      | 0.0007             |
| 8      | 0.0035             |

---

## BS2IN



Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

### Required Arguments

***XDATA*** — Array of length *NXDATA* containing the data points in the X-direction. (Input)  
*XDATA* must be strictly increasing.

***YDATA*** — Array of length *NYDATA* containing the data points in the Y-direction. (Input)  
*YDATA* must be strictly increasing.

***FDATA*** — Array of size *NXDATA* by *NYDATA* containing the values to be interpolated.  
 (Input)  
*FDATA* (*I*, *J*) is the value at (*XDATA* (*I*), *YDATA*(*J*)).

***KXORD*** — Order of the spline in the X-direction. (Input)  
*KXORD* must be less than or equal to *NXDATA*.

***KYORD*** — Order of the spline in the Y-direction. (Input)  
*KYORD* must be less than or equal to *NYDATA*.

***XKNOT*** — Array of length *NXDATA* + *KXORD* containing the knot sequence in the X-direction.  
 (Input)  
*XKNOT* must be nondecreasing.

**YKNOT** — Array of length  $NYDATA + KYORD$  containing the knot sequence in the Y-direction. (Input)

YKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXDATA * NYDATA$  containing the tensor-product B-spline coefficients. (Output)

BSCOEF is treated internally as a matrix of size  $NXDATA$  by  $NYDATA$ .

### Optional Arguments

**NXDATA** — Number of data points in the X-direction. (Input)

Default:  $NXDATA = \text{size}(XDATA, 1)$ .

**NYDATA** — Number of data points in the Y-direction. (Input)

Default:  $NYDATA = \text{size}(YDATA, 1)$ .

**LDF** — The leading dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDF = \text{size}(FDATA, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL BS2IN (XDATA, YDATA, FDATA, KXORD, KYORD, XKNOT, YKNOT,  
                  BSCOEF [, ...])

Specific:    The specific interface names are S\_BS2IN and D\_BS2IN.

### FORTRAN 77 Interface

Single:      CALL BS2IN (NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF, KXORD, KYORD,  
                  XKNOT, YKNOT, BSCOEF)

Double:     The double precision name is DBS2IN.

### Description

The routine BS2IN computes a tensor product spline interpolant. The tensor product spline interpolant to data  $\{(x_i, y_j, f_{ij})\}$ , where  $1 \leq i \leq N_x$  and  $1 \leq j \leq N_y$ , has the form

$$\sum_{m=1}^{N_y} B_{n, k_x, \mathbf{t}_x}(x) B_{m, k_y, \mathbf{t}_y}(y)$$

where  $k_x$  and  $k_y$  are the orders of the splines. (These numbers are passed to the subroutine in KXORD and KYORD, respectively.) Likewise,  $\mathbf{t}_x$  and  $\mathbf{t}_y$  are the corresponding knot sequences (XKNOT and YKNOT). The algorithm requires that

$$\mathbf{t}_x(k_x) \leq x_i \leq \mathbf{t}_x(N_x + 1) \quad 1 \leq i \leq N_x$$

$$t_y(k_y) \leq y_j \leq t_y(N_y + 1) \quad 1 \leq j \leq N_y$$

Tensor product spline interpolants in two dimensions can be computed quite efficiently by solving (repeatedly) two univariate interpolation problems. The computation is motivated by the following observations. It is necessary to solve the system of equations

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}(x_i)$$

we note that for each fixed  $i$  from 1 to  $N_x$ , we have  $N_y$  linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=1}^{N_y} h_{mi} B_{m,k_y,t_y}(y_j) = f_{ij}$$

The same matrix appears in all of the equations above:

$$\left[ B_{m,k_y,t_y}(y_j) \right]_{1 \leq m, j \leq N_y}$$

Thus, we need only factor this matrix once and then apply this factorization to the  $N_x$  righthand sides. Once this is done and we have computed  $h_{mi}$ , then we must solve for the coefficients  $c_{nm}$  using the relation

$$\sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}(x_i) = h_{mi}$$

for  $m$  from 1 to  $N_y$ , which again involves one factorization and  $N_y$  solutions to the different righthand sides. The routine `BS2IN` is based on the routine `SPLI2D` by de Boor (1978, page 347).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B22IN/DB22IN`. The reference is:

```
CALL B22IN (NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF,
 KXORD, KYORD, XKNOT, YKNOT, BSCOE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $NXDATA * NYDATA + \max((2 * KXORD - 1) NXDATA, (2 * KYORD - 1) * NYDATA) + \max((3 * KXORD - 2) * NXDATA, (3 * KYORD - 2) * NYDATA) + 2 * \max(NXDATA, NYDATA)$ .

**IWK** — Work array of length  $\text{MAX}(\text{NXDATA}, \text{NYDATA})$ .

2. Informational errors

| Type | Code | Description                                                                                                                                            |
|------|------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Interpolation matrix is nearly singular. LU factorization failed.                                                                                      |
| 3    | 2    | Interpolation matrix is nearly singular. Iterative refinement failed.                                                                                  |
| 4    | 6    | The XDATA values must be strictly increasing.                                                                                                          |
| 4    | 7    | The YDATA values must be strictly increasing.                                                                                                          |
| 4    | 13   | Multiplicity of the knots cannot exceed the order of the spline.                                                                                       |
| 4    | 14   | The knots must be nondecreasing.                                                                                                                       |
| 4    | 15   | The $I$ -th smallest element of the data point array must be greater than the $I$ -th knot and less than the $(I + K\_ORD)$ -th knot.                  |
| 4    | 16   | The largest element of the data point array must be greater than the $(N\_DATA)$ -th knot and less than or equal to the $(N\_DATA + K\_ORD)$ -th knot. |
| 4    | 17   | The smallest element of the data point array must be greater than or equal to the first knot and less than the $(K\_ORD + 1)$ st knot.                 |

**Example**

In this example, a tensor product spline interpolant to a function  $f$  is computed. The values of the interpolant and the error on a  $4 \times 4$  grid are displayed.

```

USE BS2IN_INT
USE BSNAK_INT
USE BS2VL_INT
USE UMACH_INT

IMPLICIT NONE
!
! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, LDF, NXDATA, NXKNOT, NXVEC, NYDATA, &
 NYKNOT, NYVEC
PARAMETER (KXORD=5, KYORD=2, NXDATA=21, NXVEC=4, NYDATA=6, &
 NYVEC=4, LDF=NXDATA, NXKNOT=NXDATA+KXORD, &
 NYKNOT=NYDATA+KYORD)
!
INTEGER I, J, NOUT, NXCOEF, NYCOEF
REAL BSCOEFF(NXDATA,NYDATA), F, FDATA(LDF,NYDATA), FLOAT, &
 X, XDATA(NXDATA), XKNOT(NXKNOT), XVEC(NXVEC), Y, &
 YDATA(NYDATA), YKNOT(NYKNOT), YVEC(NYVEC), VL
INTRINSIC FLOAT
!
! Define function
F(X,Y) = X*X*X + X*Y
!
! Set up interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-1)/10.0

```

```

10 CONTINUE
!
! Generate knot sequence
 CALL BSNK (NXDATA, XDATA, KXORD, XKNOT)
!
! Set up interpolation points
 DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/5.0
20 CONTINUE
!
! Generate knot sequence
 CALL BSNK (NYDATA, YDATA, KYORD, YKNOT)
!
! Generate FDATA
 DO 40 I=1, NYDATA
 DO 30 J=1, NXDATA
 FDATA(J,I) = F(XDATA(J),YDATA(I))
30 CONTINUE
40 CONTINUE
!
! Interpolate
 CALL BS2IN (XDATA, YDATA, FDATA, KXORD, KYORD, XKNOT, YKNOT, &
 BSCOE)
 NXCOEF = NXDATA
 NYCOEF = NYDATA
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Write heading
 WRITE (NOUT,99999)
!
! Print over a grid of
! [0.0,1.0] x [0.0,1.0] at 16 points.
 DO 50 I=1, NXVEC
 XVEC(I) = FLOAT(I-1)/3.0
50 CONTINUE
 DO 60 I=1, NYVEC
 YVEC(I) = FLOAT(I-1)/3.0
60 CONTINUE
!
! Evaluate spline
 DO 80 I=1, NXVEC
 DO 70 J=1, NYVEC
 VL = BS2VL (XVEC(I), YVEC(J), KXORD, KYORD, XKNOT, &
 YKNOT, NXCOEF, NYCOEF, BSCOE)
 WRITE (NOUT, '(3F15.4,F15.6)') XVEC(I), YVEC(J), &
 VL, (F(XVEC(I),YVEC(J))-VL)
70 CONTINUE
80 CONTINUE
99999 FORMAT (13X, 'X', 14X, 'Y', 10X, 'S(X,Y)', 9X, 'Error')
END

```

## Output

| X      | Y      | S(X,Y) | Error    |
|--------|--------|--------|----------|
| 0.0000 | 0.0000 | 0.0000 | 0.000000 |
| 0.0000 | 0.3333 | 0.0000 | 0.000000 |
| 0.0000 | 0.6667 | 0.0000 | 0.000000 |
| 0.0000 | 1.0000 | 0.0000 | 0.000000 |
| 0.3333 | 0.0000 | 0.0370 | 0.000000 |
| 0.3333 | 0.3333 | 0.1481 | 0.000000 |
| 0.3333 | 0.6667 | 0.2593 | 0.000000 |
| 0.3333 | 1.0000 | 0.3704 | 0.000000 |
| 0.6667 | 0.0000 | 0.2963 | 0.000000 |

|        |        |        |          |
|--------|--------|--------|----------|
| 0.6667 | 0.3333 | 0.5185 | 0.000000 |
| 0.6667 | 0.6667 | 0.7407 | 0.000000 |
| 0.6667 | 1.0000 | 0.9630 | 0.000000 |
| 1.0000 | 0.0000 | 1.0000 | 0.000000 |
| 1.0000 | 0.3333 | 1.3333 | 0.000000 |
| 1.0000 | 0.6667 | 1.6667 | 0.000000 |
| 1.0000 | 1.0000 | 2.0000 | 0.000000 |

---

## BS3IN



Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

### Required Arguments

***XDATA*** — Array of length  $NXDATA$  containing the data points in the  $x$ -direction. (Input)  
*XDATA* must be increasing.

***YDATA*** — Array of length  $NYDATA$  containing the data points in the  $y$ -direction. (Input)  
*YDATA* must be increasing.

***ZDATA*** — Array of length  $NZDATA$  containing the data points in the  $z$ -direction. (Input)  
*ZDATA* must be increasing.

***FDATA*** — Array of size  $NXDATA$  by  $NYDATA$  by  $NZDATA$  containing the values to be interpolated. (Input)  
*FDATA* ( $I, J, K$ ) contains the value at ( $XDATA(I), YDATA(J), ZDATA(K)$ ).

***KXORD*** — Order of the spline in the  $x$ -direction. (Input)  
*KXORD* must be less than or equal to  $NXDATA$ .

***KYORD*** — Order of the spline in the  $y$ -direction. (Input)  
*KYORD* must be less than or equal to  $NYDATA$ .

***KZORD*** — Order of the spline in the  $z$ -direction. (Input)  
*KZORD* must be less than or equal to  $NZDATA$ .

***XKNOT*** — Array of length  $NXDATA + KXORD$  containing the knot sequence in the  $x$ -direction. (Input)  
*XKNOT* must be nondecreasing.

***YKNOT*** — Array of length  $NYDATA + KYORD$  containing the knot sequence in the  $y$ -direction. (Input)  
*YKNOT* must be nondecreasing.

**ZKNOT** — Array of length  $NZDATA + KZORD$  containing the knot sequence in the  $z$ -direction. (Input)  
ZKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXDATA * NYDATA * NZDATA$  containing the tensor-product B-spline coefficients. (Output)  
BSCOEF is treated internally as a matrix of size  $NXDATA$  by  $NYDATA$  by  $NZDATA$ .

### Optional Arguments

**NXDATA** — Number of data points in the  $x$ -direction. (Input)  
Default:  $NXDATA = \text{size}(XDATA,1)$ .

**NYDATA** — Number of data points in the  $y$ -direction. (Input)  
Default:  $NYDATA = \text{size}(YDATA,1)$ .

**NZDATA** — Number of data points in the  $z$ -direction. (Input)  
Default:  $NZDATA = \text{size}(ZDATA,1)$ .

**LDF** — Leading dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDF = \text{size}(FDATA,1)$ .

**MDF** — Middle dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDF = \text{size}(FDATA,2)$ .

### FORTRAN 90 Interface

Generic:     CALL BS3IN (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD, KZORD, XKNOT,  
                          YKNOT, ZKNOT, BSCOEF [, ...])

Specific:    The specific interface names are S\_BS3IN and D\_BS3IN.

### FORTRAN 77 Interface

Single:     CALL BS3IN (NXDATA, XDATA, NYDATA, YDATA, NZDATA, ZDATA, FDATA, LDF,  
                          MDF, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, BSCOEF)

Double:     The double precision name is DBS3IN.

### Description

The routine BS3IN computes a tensor-product spline interpolant. The tensor-product spline interpolant to data  $\{(x_i, y_j, z_k, f_{ijk})\}$ , where  $1 \leq i \leq N_x$ ,  $1 \leq j \leq N_y$ , and  $1 \leq k \leq N_z$  has the form

$$\sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}(x) B_{m,k_y,t_y}(y) B_{l,k_z,t_z}(z)$$

where  $k_x$ ,  $k_y$ , and  $k_z$  are the orders of the splines (these numbers are passed to the subroutine in `KXORD`, `KYORD`, and `KZORD`, respectively). Likewise,  $\mathbf{t}_x$ ,  $\mathbf{t}_y$ , and  $\mathbf{t}_z$  are the corresponding knot sequences (`XKNOT`, `YKNOT`, and `ZKNOT`). The algorithm requires that

$$\begin{aligned} \mathbf{t}_x(k_x) &\leq x_i \leq \mathbf{t}_x(N_x + 1) & 1 \leq i \leq N_x \\ \mathbf{t}_y(k_y) &\leq y_j \leq \mathbf{t}_y(N_y + 1) & 1 \leq j \leq N_y \\ \mathbf{t}_z(k_z) &\leq z_k \leq \mathbf{t}_z(N_z + 1) & 1 \leq k \leq N_z \end{aligned}$$

Tensor-product spline interpolants can be computed quite efficiently by solving (repeatedly) three univariate interpolation problems. The computation is motivated by the following observations. It is necessary to solve the system of equations

$$\sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) B_{l,k_z,t_z}(z_k) = f_{ijk}$$

Setting

$$h_{ij} = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j)$$

we note that for each fixed pair  $ij$  we have  $N_z$  linear equations in the same number of unknowns as can be seen below:

$$\sum_{l=1}^{N_z} h_{ij} B_{l,k_z,t_z}(z_k) = f_{ijk}$$

The same interpolation matrix appears in all of the equations above:

$$\left[ B_{l,k_z,t_z}(z_k) \right] \quad 1 \leq l, k \leq N_z$$

Thus, we need only factor this matrix once and then apply it to the  $N_x N_y$  right-hand sides. Once this is done and we have computed  $h_{ij}$ , then we must solve for the coefficients  $c_{nml}$  using the relation

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) = h_{ij}$$

that is the *bivariate* tensor-product problem addressed by the IMSL routine `BS2IN`. The interested reader should consult the algorithm description in the two-dimensional routine if more detail is desired. The routine `BS3IN` is based on the routine `SPLI2D` by de Boor (1978, page 347).

## Comments

1. Workspace may be explicitly provided, if desired, by use of B23IN/DB23IN. The reference is:

```
CALL B23IN (NXDATA, XDATA, NYDATA, YDATA, NZDAYA, ZDATA,
 FDATA, LDF, MDF, KXORD, KYORD, KZORD, XKNOT, YKNOT,
 ZKNOT, BSCEF, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $\text{MAX}((2 * \text{KXORD} - 1) * \text{NXDATA}, (2 * \text{KYORD} - 1) * \text{NYDATA}, (2 * \text{KZORD} - 1) * \text{NZDATA}) + \text{MAX}((3 * \text{KXORD} - 2) * \text{NXDATA}, (3 * \text{KYORD} - 2) * \text{NYDATA} + (3 * \text{KZORD} - 2) * \text{NZDATA}) + \text{NXDATA} * \text{NYDATA} * \text{NZDATA} + 2 * \text{MAX}(\text{NXDATA}, \text{NYDATA}, \text{NZDATA})$ .

**IWK** — Work array of length  $\text{MAX}(\text{NXDATA}, \text{NYDATA}, \text{NZDATA})$ .

2. Informational errors

| Type | Code |                                                                                                                                                        |
|------|------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Interpolation matrix is nearly singular. LU factorization failed.                                                                                      |
| 3    | 2    | Interpolation matrix is nearly singular. Iterative refinement failed.                                                                                  |
| 4    | 13   | Multiplicity of the knots cannot exceed the order of the spline.                                                                                       |
| 4    | 14   | The knots must be nondecreasing.                                                                                                                       |
| 4    | 15   | The $I$ -th smallest element of the data point array must be greater than the $I$ th knot and less than the $(I + K\_ORD)$ -th knot.                   |
| 4    | 16   | The largest element of the data point array must be greater than the $(N\_DATA)$ -th knot and less than or equal to the $(N\_DATA + K\_ORD)$ -th knot. |
| 4    | 17   | The smallest element of the data point array must be greater than or equal to the first knot and less than the $(K\_ORD + 1)$ st knot.                 |
| 4    | 18   | The $XDATA$ values must be strictly increasing.                                                                                                        |
| 4    | 19   | The $YDATA$ values must be strictly increasing.                                                                                                        |
| 4    | 20   | The $ZDATA$ values must be strictly increasing.                                                                                                        |

## Example

In this example, a tensor-product spline interpolant to a function  $f$  is computed. The values of the interpolant and the error on a  $4 \times 4 \times 2$  grid are displayed.

```
USE BS3IN_INT
USE BSNAK_INT
USE UMACH_INT
```

```

USE BS3GD_INT

IMPLICIT NONE

!
! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, KZORD, LDF, MDF, NXDATA, NXKNOT, NXVEC, &
NYDATA, NYKNOT, NYVEC, NZDATA, NZKNOT, NZVEC
PARAMETER (KXORD=5, KYORD=2, KZORD=3, NXDATA=21, NXVEC=4, &
, NYDATA=6, NYVEC=4, NZDATA=8, NZVEC=2, LDF=NXDATA, &
MDF=NYDATA, NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD, &
NZKNOT=NZDATA+KZORD)

!
INTEGER I, J, K, NOUT, NXCOEF, NYCOEF, NZCOEF
REAL BSCOEF(NXDATA,NYDATA,NZDATA), F, &
FDATA(LDF,MDF,NZDATA), FLOAT, VALUE(NXVEC,NYVEC,NZVEC) &
, X, XDATA(NXDATA), XKNOT(NXKNOT), XVEC(NXVEC), Y, &
YDATA(NYDATA), YKNOT(NYKNOT), YVEC(NYVEC), Z, &
ZDATA(NZDATA), ZKNOT(NZKNOT), ZVEC(NZVEC)

INTRINSIC FLOAT

!
! Define function.
F(X,Y,Z) = X*X*X + X*Y*Z

!
! Set up X-interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-1)/10.0
10 CONTINUE

!
! Set up Y-interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/FLOAT(NYDATA-1)
20 CONTINUE

!
! Set up Z-interpolation points
DO 30 I=1, NZDATA
 ZDATA(I) = FLOAT(I-1)/FLOAT(NZDATA-1)
30 CONTINUE

!
! Generate knots
CALL BSNAK (NXDATA, XDATA, KXORD, XKNOT)
CALL BSNAK (NYDATA, YDATA, KYORD, YKNOT)
CALL BSNAK (NZDATA, ZDATA, KZORD, ZKNOT)

!
! Generate FDATA
DO 50 K=1, NZDATA
 DO 40 I=1, NYDATA
 DO 40 J=1, NXDATA
 FDATA(J,I,K) = F(XDATA(J),YDATA(I),ZDATA(K))
40 CONTINUE
50 CONTINUE

!
! Get output unit number
CALL UMACH (2, NOUT)

!
! Interpolate
CALL BS3IN (XDATA, YDATA, ZDATA, FDATA, KXORD, &
KYORD, KZORD, XKNOT, YKNOT, ZKNOT, BSCOEF)

!
NXCOEF = NXDATA
NYCOEF = NYDATA
NZCOEF = NZDATA

!
! Write heading
WRITE (NOUT,99999)

!
! Print over a grid of
! [-1.0,1.0] x [0.0,1.0] x [0.0,1.0]

```

```

! at 32 points.
 DO 60 I=1, NXVEC
 XVEC(I) = 2.0*(FLOAT(I-1)/3.0) - 1.0
60 CONTINUE
 DO 70 I=1, NYVEC
 YVEC(I) = FLOAT(I-1)/3.0
70 CONTINUE
 DO 80 I=1, NZVEC
 ZVEC(I) = FLOAT(I-1)
80 CONTINUE
! Call the evaluation routine.
 CALL BS3GD (0, 0, 0, XVEC, YVEC, ZVEC, &
 KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, BSCOEFF, VALUE)
 DO 110 I=1, NXVEC
 DO 100 J=1, NYVEC
 DO 90 K=1, NZVEC
 WRITE (NOUT, '(4F13.4, F13.6)') XVEC(I), YVEC(K), &
 ZVEC(K), VALUE(I, J, K), &
 F(XVEC(I), YVEC(J), ZVEC(K)) &
 - VALUE(I, J, K)
 90 CONTINUE
 100 CONTINUE
 110 CONTINUE
99999 FORMAT (10X, 'X', 11X, 'Y', 10X, 'Z', 10X, 'S(X,Y,Z)', 7X, &
 'Error')
 END

```

## Output

| X       | Y      | Z      | S(X,Y,Z) | Error    |
|---------|--------|--------|----------|----------|
| -1.0000 | 0.0000 | 0.0000 | -1.0000  | 0.000000 |
| -1.0000 | 0.3333 | 1.0000 | -1.0000  | 0.000000 |
| -1.0000 | 0.0000 | 0.0000 | -1.0000  | 0.000000 |
| -1.0000 | 0.3333 | 1.0000 | -1.3333  | 0.000000 |
| -1.0000 | 0.0000 | 0.0000 | -1.0000  | 0.000000 |
| -1.0000 | 0.3333 | 1.0000 | -1.6667  | 0.000000 |
| -1.0000 | 0.0000 | 0.0000 | -1.0000  | 0.000000 |
| -1.0000 | 0.3333 | 1.0000 | -2.0000  | 0.000000 |
| -0.3333 | 0.0000 | 0.0000 | -0.0370  | 0.000000 |
| -0.3333 | 0.3333 | 1.0000 | -0.0370  | 0.000000 |
| -0.3333 | 0.0000 | 0.0000 | -0.0370  | 0.000000 |
| -0.3333 | 0.3333 | 1.0000 | -0.1481  | 0.000000 |
| -0.3333 | 0.0000 | 0.0000 | -0.0370  | 0.000000 |
| -0.3333 | 0.3333 | 1.0000 | -0.2593  | 0.000000 |
| -0.3333 | 0.0000 | 0.0000 | -0.0370  | 0.000000 |
| -0.3333 | 0.3333 | 1.0000 | -0.3704  | 0.000000 |
| 0.3333  | 0.0000 | 0.0000 | 0.0370   | 0.000000 |
| 0.3333  | 0.3333 | 1.0000 | 0.0370   | 0.000000 |
| 0.3333  | 0.0000 | 0.0000 | 0.0370   | 0.000000 |
| 0.3333  | 0.3333 | 1.0000 | 0.1481   | 0.000000 |
| 0.3333  | 0.0000 | 0.0000 | 0.0370   | 0.000000 |
| 0.3333  | 0.3333 | 1.0000 | 0.2593   | 0.000000 |
| 0.3333  | 0.0000 | 0.0000 | 0.0370   | 0.000000 |
| 0.3333  | 0.3333 | 1.0000 | 0.3704   | 0.000000 |
| 1.0000  | 0.0000 | 0.0000 | 1.0000   | 0.000000 |
| 1.0000  | 0.3333 | 1.0000 | 1.0000   | 0.000000 |
| 1.0000  | 0.0000 | 0.0000 | 1.0000   | 0.000000 |

|        |        |        |        |          |
|--------|--------|--------|--------|----------|
| 1.0000 | 0.3333 | 1.0000 | 1.3333 | 0.000000 |
| 1.0000 | 0.0000 | 0.0000 | 1.0000 | 0.000000 |
| 1.0000 | 0.3333 | 1.0000 | 1.6667 | 0.000000 |
| 1.0000 | 0.0000 | 0.0000 | 1.0000 | 0.000000 |
| 1.0000 | 0.3333 | 1.0000 | 2.0000 | 0.000000 |

---

## BSVAL

This function evaluates a spline, given its B-spline representation.

### Function Return Value

*BSVAL* — Value of the spline at  $x$ . (Output)

### Required Arguments

*X* — Point at which the spline is to be evaluated. (Input)

*KORDER* — Order of the spline. (Input)

*XKNOT* — Array of length  $KORDER + NCOEF$  containing the knot sequence. (Input)  
*XKNOT* must be nondecreasing.

*NCOEF* — Number of B-spline coefficients. (Input)

*BSCOEF* — Array of length  $NCOEF$  containing the B-spline coefficients. (Input)

### FORTRAN 90 Interface

Generic: `BSVAL (X, KORDER, XKNOT, NCOEF, BSCOEF)`

Specific: The specific interface names are `S_BSVAL` and `D_BSVAL`.

### FORTRAN 77 Interface

Single: `BSVAL (X, KORDER, XKNOT, NCOEF, BSCOEF)`

Double: The double precision function name is `DBSVAL`.

### Description

The function `BSVAL` evaluates a spline (given its B-spline representation) at a specific point. It is a special case of the routine `BSDER`, which evaluates the derivative of a spline given its B-spline representation. The routine `BSDER` is based on the routine `BVALUE` by de Boor (1978, page 144).

Specifically, given the knot vector  $\mathbf{t}$ , the number of coefficients  $N$ , the coefficient vector  $a$ , and a point  $x$ , `BSVAL` returns the number

$$\sum_{j=1}^N a_j B_{j,k}(x)$$

where  $B_{j,k}$  is the  $j$ -th B-spline of order  $k$  for the knot sequence  $\mathbf{t}$ . Note that this function routine arbitrarily treats these functions as if they were right continuous near  $\text{XKNOT}(\text{KORDER})$  and left continuous near  $\text{XKNOT}(\text{NCOEF} + 1)$ . Thus, if we have  $\text{KORDER}$  knots stacked at the left or right end point, and if we try to evaluate at these end points, then we will get the value of the limit from the interior of the interval.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2VAL/DB2VAL`. The reference is:

```
CALL B2VAL(X, KORDER, XKNOT, NCOEF, BSCOEFF, WK1, WK2, WK3)
```

The additional arguments are as follows:

**WK1** — Work array of length `KORDER`.

**WK2** — Work array of length `KORDER`.

**WK3** — Work array of length `KORDER`.

2. Informational errors

| Type | Code |                                                                  |
|------|------|------------------------------------------------------------------|
| 4    | 4    | Multiplicity of the knots cannot exceed the order of the spline. |
| 4    | 5    | The knots must be nondecreasing.                                 |

## Example

For an example of the use of `BSVAL`, see IMSL routine [BSINT](#).

# BSDER

This function evaluates the derivative of a spline, given its B-spline representation.

## Function Return Value

**BSDER** — Value of the `IDERIV`-th derivative of the spline at  $x$ . (Output)

## Required Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)

In particular, `IDERIV = 0` returns the value of the spline.

**X** — Point at which the spline is to be evaluated. (Input)

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length  $NCOEF + KORDER$  containing the knot sequence. (Input)  
 $XKNOT$  must be nondecreasing.

**NCOEF** — Number of B-spline coefficients. (Input)

**BSCOEF** — Array of length  $NCOEF$  containing the B-spline coefficients. (Input)

### **FORTRAN 90 Interface**

Generic: `BSDER (IDERIV, X, KORDER, XKNOT, NCOEF, BSCOEF)`

Specific: The specific interface names are `S_BSDER` and `D_BSDER`.

### **FORTRAN 77 Interface**

Single: `BSDER (IDERIV, X, KORDER, XKNOT, NCOEF, BSCOEF)`

Double: The double precision function name is `DBSDER`.

### **Description**

The function `BSDER` produces the value of a spline or one of its derivatives (given its B-spline representation) at a specific point. The function `BSDER` is based on the routine `BVALUE` by de Boor (1978, page 144).

Specifically, given the knot vector  $\mathbf{t}$ , the number of coefficients  $N$ , the coefficient vector  $a$ , the order of the derivative  $i$  and a point  $x$ , `BSDER` returns the number

$$\sum_{j=1}^N a_j B_{j,k}^{(i)}(x)$$

where  $B_{j,k}$  is the  $j$ -th B-spline of order  $k$  for the knot sequence  $\mathbf{t}$ . Note that this function routine arbitrarily treats these functions as if they were right continuous near  $XKNOT(KORDER)$  and left continuous near  $XKNOT(NCOEF + 1)$ . Thus, if we have  $KORDER$  knots stacked at the left or right end point, and if we try to evaluate at these end points, then we will get the value of the limit from the interior of the interval.

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `B2DER/DB2DER`. The reference is:

```
CALL B2DER (IDERIV, X, KORDER, XKNOT, NCOEF, BSCOEF, WK1, WK2, WK3)
```

The additional arguments are as follows:

**WK1** — Array of length KORDER.

**WK2** — Array of length KORDER.

**WK3** — Array of length KORDER.

2. Informational errors

| Type | Code |                                                                  |
|------|------|------------------------------------------------------------------|
| 4    | 4    | Multiplicity of the knots cannot exceed the order of the spline. |
| 4    | 5    | The knots must be nondecreasing.                                 |

### Example

A spline interpolant to the function

$$f(x) = \sqrt{x}$$

is constructed using `BSINT`. The B-spline representation, which is returned by the IMSL routine `BSINT`, is then used by `BSDER` to compute the value and derivative of the interpolant. The output consists of the interpolation values and the error at the data points and the midpoints. In addition, we display the value of the derivative and the error at these same points.

```
USE BSDER_INT
USE BSINT_INT
USE BSNK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KORDER, NDATA, NKNOT
PARAMETER (KORDER=3, NDATA=5, NKNOT=NDATA+KORDER)
!
INTEGER I, NCOEF, NOUT
REAL BSCOEF(NDATA), BT0, BT1, DF, F, FDATA(NDATA), &
 FLOAT, SQRT, X, XDATA(NDATA), XKNOT(NKNOT), XT
INTRINSIC FLOAT, SQRT
!
! Define function and derivative
F(X) = SQRT(X)
DF(X) = 0.5/SQRT(X)
!
! Set up interpolation points
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I)/FLOAT(NDATA)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Generate knot sequence
CALL BSNK (NDATA, XDATA, KORDER, XKNOT)
!
! Interpolate
CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99999)
!
! Print on a finer grid
NCOEF = NDATA
```

```

 XT = XDATA(1)
!
! Evaluate spline
BT0 = BSDER(0,XT,KORDER,XKNOT,NCOEF,BSCOEF)
BT1 = BSDER(1,XT,KORDER,XKNOT,NCOEF,BSCOEF)
WRITE (NOUT,99998) XT, BT0, F(XT) - BT0, BT1, DF(XT) - BT1
DO 20 I=2, NDATA
 XT = (XDATA(I-1)+XDATA(I))/2.0
!
! Evaluate spline
BT0 = BSDER(0,XT,KORDER,XKNOT,NCOEF,BSCOEF)
BT1 = BSDER(1,XT,KORDER,XKNOT,NCOEF,BSCOEF)
WRITE (NOUT,99998) XT, BT0, F(XT) - BT0, BT1, DF(XT) - BT1
 XT = XDATA(I)
!
! Evaluate spline
BT0 = BSDER(0,XT,KORDER,XKNOT,NCOEF,BSCOEF)
BT1 = BSDER(1,XT,KORDER,XKNOT,NCOEF,BSCOEF)
WRITE (NOUT,99998) XT, BT0, F(XT) - BT0, BT1, DF(XT) - BT1
20 CONTINUE
99998 FORMAT (' ', F6.4, 5X, F7.4, 3X, F10.6, 5X, F8.4, 3X, F10.6)

99999 FORMAT (6X, 'X', 8X, 'S(X)', 7X, 'Error', 8X, 'S'(X)', 8X, &
 'Error', /)
 END

```

## Output

| X      | S(X)   | Error     | S'(X)  | Error     |
|--------|--------|-----------|--------|-----------|
| 0.2000 | 0.4472 | 0.000000  | 1.0423 | 0.075738  |
| 0.3000 | 0.5456 | 0.002084  | 0.9262 | -0.013339 |
| 0.4000 | 0.6325 | 0.000000  | 0.8101 | -0.019553 |
| 0.5000 | 0.7077 | -0.000557 | 0.6940 | 0.013071  |
| 0.6000 | 0.7746 | 0.000000  | 0.6446 | 0.000869  |
| 0.7000 | 0.8366 | 0.000071  | 0.5952 | 0.002394  |
| 0.8000 | 0.8944 | 0.000000  | 0.5615 | -0.002525 |
| 0.9000 | 0.9489 | -0.000214 | 0.5279 | -0.000818 |
| 1.0000 | 1.0000 | 0.000000  | 0.4942 | 0.005814  |

---

## BS1GD

Evaluates the derivative of a spline on a grid, given its B-spline representation.

### Required Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)

In particular, IDERIV = 0 returns the value of the spline.

**XVEC** — Array of length N containing the points at which the spline is to be evaluated.

(Input)

XVEC should be strictly increasing.

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length  $NCOEF + KORDER$  containing the knot sequence. (Input)  
XKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NCOEF$  containing the B-spline coefficients. (Input)

**VALUE** — Array of length  $N$  containing the values of the  $IDERIV$ -th derivative of the spline at the points in  $XVEC$ . (Output)

### Optional Arguments

**N** — Length of vector  $XVEC$ . (Input)  
Default:  $N = \text{size}(XVEC, 1)$ .

**NCOEF** — Number of B-spline coefficients. (Input)  
Default:  $NCOEF = \text{size}(BSCOEF, 1)$ .

### FORTRAN 90 Interface

Generic:    CALL BS1GD (IDERIV, XVEC, KORDER, XKNOT, BSCOEF, VALUE [, ...])

Specific:   The specific interface names are S\_BS1GD and D\_BS1GD.

### FORTRAN 77 Interface

Single:     CALL BS1GD (IDERIV, N, XVEC, KORDER, XKNOT, NCOEF, BSCOEF, VALUE)

Double:    The double precision name is DBS1GD.

### Description

The routine BS1GD evaluates a B-spline (or its derivative) at a vector of points. That is, given a vector  $x$  of length  $n$  satisfying  $x_i < x_{i+1}$  for  $i = 1, \dots, n - 1$ , a derivative value  $j$ , and a B-spline  $s$  that is represented by a knot sequence and coefficient sequence, this routine returns the values

$$s^{(j)}(x_i) \quad i = 1, \dots, n$$

in the array VALUE. The functionality of this routine is the same as that of [BSDER](#) called in a loop, however BS1GD should be much more efficient. This routine converts the B-spline representation to piecewise polynomial form using the IMSL routine [BSCPP](#), and then uses the IMSL routine [PPVAL](#) for evaluation.

### Comments

1.    Workspace may be explicitly provided, if desired, by use of B21GD/DB21GD. The reference is:

```
CALL B21GD (IDERIV, N, XVEC, KORDER, XKNOT, NCOEF, BSCOEF,
VALUE, RWK1, RWK2, IWK3, RWK4, RWK5, RWK6)
```

The additional arguments are as follows:

**RWK1** — Real array of length  $KORDER * (NCOEF - KORDER + 1)$ .

**RWK2** — Real array of length  $NCOEF - KORDER + 2$ .

**IWK3** — Integer array of length  $N$ .

**RWK4** — Real array of length  $N$ .

**RWK5** — Real array of length  $N$ .

**RWK6** — Real array of length  $(KORDER + 3) * KORDER$ .

2. Informational error

Type Code

4 5 The points in *XVEC* must be strictly increasing.

### Example

To illustrate the use of `BS1GD`, we modify the example program for `BSDER`. In this example, a quadratic (order 3) spline interpolant to  $F$  is computed. The values and derivatives of this spline are then compared with the exact function and derivative values. The routine `BS1GD` is based on the routines `BSPLPP` and `PPVALU` in de Boor (1978, page 89).

```
USE BS1GD_INT
USE BSINT_INT
USE BSNAK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KORDER, NDATA, NKNOT, NFGRID
PARAMETER (KORDER=3, NDATA=5, NKNOT=NDATA+KORDER, NFGRID = 9)
! SPECIFICATIONS FOR LOCAL VARIABLES

INTEGER I, NCOEF, NOUT
REAL ANS0(NFGRID), ANS1(NFGRID), BSCOEFF(NDATA), &
 FDATA(NDATA), &
 X, XDATA(NDATA), XKNOT(NKNOT), XVEC(NFGRID)
! SPECIFICATIONS FOR INTRINSICS

INTRINSIC FLOAT, SQRT
REAL FLOAT, SQRT
! SPECIFICATIONS FOR SUBROUTINES

REAL DF, F
!

F(X) = SQRT(X)
DF(X) = 0.5/SQRT(X)
!

CALL UMACH (2, NOUT)
! Set up interpolation points

DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I)/FLOAT(NDATA)
 FDATA(I) = F(XDATA(I))
```

```

10 CONTINUE
 CALL BSNK (NDATA, XDATA, KORDER, XKNOT)
! Interpolate
 CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOE)
 WRITE (NOUT,99999)
!
! Print on a finer grid
 NCOEF = NDATA
 XVEC(1) = XDATA(1)
 DO 20 I=2, 2*NDATA - 2, 2
 XVEC(I) = (XDATA(I/2+1)+XDATA(I/2))/2.0
 XVEC(I+1) = XDATA(I/2+1)
20 CONTINUE
 CALL BS1GD (0, XVEC, KORDER, XKNOT, BSCOE, ANS0)
 CALL BS1GD (1, XVEC, KORDER, XKNOT, BSCOE, ANS1)
 DO 30 I=1, 2*NDATA - 1
 WRITE (NOUT,99998) XVEC(I), ANS0(I), F(XVEC(I)) - ANS0(I), &
 ANS1(I), DF(XVEC(I)) - ANS1(I)
30 CONTINUE
99998 FORMAT (' ', F6.4, 5X, F7.4, 5X, F8.4, 5X, F8.4, 5X, F8.4)
99999 FORMAT (6X, 'X', 8X, 'S(X)', 7X, 'Error', 8X, 'S'(X)', 8X, &
 'Error', /)
END

```

## Output

| X      | S(X)   | Error   | S'(X)  | Error   |
|--------|--------|---------|--------|---------|
| 0.2000 | 0.4472 | 0.0000  | 1.0423 | 0.0757  |
| 0.3000 | 0.5456 | 0.0021  | 0.9262 | -0.0133 |
| 0.4000 | 0.6325 | 0.0000  | 0.8101 | -0.0196 |
| 0.5000 | 0.7077 | -0.0006 | 0.6940 | 0.0131  |
| 0.6000 | 0.7746 | 0.0000  | 0.6446 | 0.0009  |
| 0.7000 | 0.8366 | 0.0001  | 0.5952 | 0.0024  |
| 0.8000 | 0.8944 | 0.0000  | 0.5615 | -0.0025 |
| 0.9000 | 0.9489 | -0.0002 | 0.5279 | -0.0008 |
| 1.0000 | 1.0000 | 0.0000  | 0.4942 | 0.0058  |

---

## BSITG

This function evaluates the integral of a spline, given its B-spline representation.

### Function Return Value

*BSITG* — Value of the integral of the spline from A to B. (Output)

### Required Arguments

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. (Input)

*KORDER* — Order of the spline. (Input)

**XKNOT** — Array of length `KORDER + NCOEF` containing the knot sequence. (Input)  
`XKNOT` must be nondecreasing.

**NCOEF** — Number of B-spline coefficients. (Input)

**BSCOEF** — Array of length `NCOEF` containing the B-spline coefficients. (Input)

### **FORTRAN 90 Interface**

Generic: `BSITG (A, B, KORDER, XKNOT, NCOEF, BSCOEF)`

Specific: The specific interface names are `S_BSITG` and `D_BSITG`.

### **FORTRAN 77 Interface**

Single: `BSITG (A, B, KORDER, XKNOT, NCOEF, BSCOEF)`

Double: The double precision function name is `DBSITG`.

### **Description**

The function `BSITG` computes the integral of a spline given its B-spline representation. Specifically, given the knot sequence  $\mathbf{t} = \text{XKNOT}$ , the order  $k = \text{KORDER}$ , the coefficients  $a = \text{BSCOEF}$ ,  $n = \text{NCOEF}$  and an interval  $[a, b]$ , `BSITG` returns the value

$$\int_a^b \sum_{i=1}^n a_i B_{i,k,\mathbf{t}}(x) dx$$

This routine uses the identity (22) on page 151 of de Boor (1978), and it assumes that  $\mathbf{t}_1 = \dots = \mathbf{t}_k$  and  $\mathbf{t}_{n+1} = \dots = \mathbf{t}_{n+k}$ .

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `B2ITG/DB2ITG`. The reference is:

```
CALL B2ITG(A, B, KORDER, XKNOT, NCOEF, BSCOEF, TCOEF, AJ,
 DL, DR)
```

The additional arguments are as follows:

**TCOEF** — Work array of length `KORDER + 1`.

**AJ** — Work array of length `KORDER + 1`.

**DL** — Work array of length `KORDER + 1`.

**DR** — Work array of length  $KORDER + 1$ .

2. Informational errors

| Type | Code |                                                                     |
|------|------|---------------------------------------------------------------------|
| 3    | 7    | The upper and lower endpoints of integration are equal.             |
| 3    | 8    | The lower limit of integration is less than $XKNOT(KORDER)$ .       |
| 3    | 9    | The upper limit of integration is greater than $XKNOT(NCOEF + 1)$ . |
| 4    | 4    | Multiplicity of the knots cannot exceed the order of the spline.    |
| 4    | 5    | The knots must be nondecreasing.                                    |

### Example

We integrate the quartic ( $k = 5$ ) spline that interpolates  $x^3$  at the points  $\{i/10 : i = -10, \dots, 10\}$  over the interval  $[0, 1]$ . The exact answer is  $1/4$  since the interpolant reproduces cubic polynomials.

```
USE BSITG_INT
USE BSNK_INT
USE BSINT_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KORDER, NDATA, NKNOT
PARAMETER (KORDER=5, NDATA=21, NKNOT=NDATA+KORDER)
!
INTEGER I, NCOEF, NOUT
REAL A, B, BSCOEF(NDATA), ERROR, EXACT, F, &
 FDATA(NDATA), FI, FLOAT, VAL, X, XDATA(NDATA), &
 XKNOT(NKNOT)
INTRINSIC FLOAT
!
! Define function and integral
F(X) = X*X*X
FI(X) = X**4/4.0
!
! Set up interpolation points
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-11)/10.0
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Generate knot sequence
CALL BSNK (NDATA, XDATA, KORDER, XKNOT)
!
! Interpolate
CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
NCOEF = NDATA
A = 0.0
B = 1.0
!
! Integrate from A to B
VAL = BSITG(A, B, KORDER, XKNOT, NCOEF, BSCOEF)
EXACT = FI(B) - FI(A)
```

```

 ERROR = EXACT - VAL
!
 Print results
 WRITE (NOUT,99999) A, B, VAL, EXACT, ERROR
99999 FORMAT (' On the closed interval (' , F3.1, ', ', F3.1, &
 ') we have :', /, 1X, 'Computed Integral = ', F10.5, /, &
 1X, 'Exact Integral = ', F10.5, /, 1X, 'Error ' &
 , ' = ', F10.6, /, /)
 END

```

## Output

```

On the closed interval (0.0,1.0) we have :
Computed Integral = 0.25000
Exact Integral = 0.25000
Error = 0.000000

```

---

## BS2VL

This function evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

### Function Return Value

*BS2VL* — Value of the spline at  $(x, y)$ . (Output)

### Required Arguments

*X* —  $x$ -coordinate of the point at which the spline is to be evaluated. (Input)

*Y* —  $y$ -coordinate of the point at which the spline is to be evaluated. (Input)

*KXORD* — Order of the spline in the  $x$ -direction. (Input)

*KYORD* — Order of the spline in the  $y$ -direction. (Input)

*XKNOT* — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the  $x$ -direction. (Input)  
*XKNOT* must be nondecreasing.

*YKNOT* — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the  $y$ -direction. (Input)  
*YKNOT* must be nondecreasing.

*NXCOEF* — Number of B-spline coefficients in the  $x$ -direction. (Input)

*NYCOEF* — Number of B-spline coefficients in the  $y$ -direction. (Input)

**BSCOE***F* — Array of length  $NXCOEF * NYCOEF$  containing the tensor-product B-spline coefficients. (Input)  
 BSCOE*F* is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$ .

### FORTRAN 90 Interface

Generic: BS2VL (X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOE*F*)  
 Specific: The specific interface names are S\_BS2VL and D\_BS2VL.

### FORTRAN 77 Interface

Single: BS2VL (X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOE*F*)  
 Double: The double precision function name is DBS2VL.

### Description

The function BS2VL evaluates a bivariate tensor product spline (represented as a linear combination of tensor product B-splines) at a given point. This routine is a special case of the routine BS2DR, which evaluates partial derivatives of such a spline. (The value of a spline is its zero-th derivative.) For more information see de Boor (1978, pages 351 – 353).

This routine returns the value of the function  $s$  at a point  $(x, y)$  given the coefficients  $c$  by computing

$$s(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}(x) B_{m,k_y,t_y}(y)$$

where  $k_x$  and  $k_y$  are the orders of the splines. (These numbers are passed to the subroutine in KXORD and KYORD, respectively.) Likewise,  $t_x$  and  $t_y$  are the corresponding knot sequences (XKNOT and YKNOT).

### Comments

Workspace may be explicitly provided, if desired, by use of B22VL/DB22VL. The reference is:

```
CALL B22VL(X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF,
 BSCOEF, WK)
```

The additional argument is

**WK** — Work array of length  $3 * \text{MAX}(KXORD, KYORD) + KYORD$ .

### Example

For an example of the use of BS2VL, see IMSL routine BS2IN.

---

## BS2DR

This function evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

### Function Return Value

*BS2DR* — Value of the (*IXDER*, *IYDER*) derivative of the spline at (*X*, *Y*). (Output)

### Required Arguments

*IXDER* — Order of the derivative in the *X*-direction. (Input)

*IYDER* — Order of the derivative in the *Y*-direction. (Input)

*X* — *X*-coordinate of the point at which the spline is to be evaluated. (Input)

*Y* — *Y*-coordinate of the point at which the spline is to be evaluated. (Input)

*KXORD* — Order of the spline in the *X*-direction. (Input)

*KYORD* — Order of the spline in the *Y*-direction. (Input)

*XKNOT* — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the *X*-direction. (Input)  
*XKNOT* must be nondecreasing.

*YKNOT* — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the *Y*-direction. (Input)  
*YKNOT* must be nondecreasing.

*NXCOEF* — Number of B-spline coefficients in the *X*-direction. (Input)

*NYCOEF* — Number of B-spline coefficients in the *Y*-direction. (Input)

*BSCOEF* — Array of length  $NXCOEF * NYCOEF$  containing the tensor-product B-spline coefficients. (Input)  
*BSCOEF* is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$ .

### FORTRAN 90 Interface

Generic:    *BS2DR* (*IXDER*, *IYDER*, *X*, *Y*, *KXORD*, *KYORD*, *XKNOT*, *YKNOT*, *NXCOEF*, *NYCOEF*,  
                  *BSCOEF*)

Specific:    The specific interface names are *S\_BS2DR* and *D\_BS2DR*.

## FORTRAN 77 Interface

Single: BS2DR (IXDER, IYDER, X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOEF)

Double: The double precision function name is DBS2DR.

## Description

The routine BS2DR evaluates a partial derivative of a bivariate tensor-product spline (represented as a linear combination of tensor product B-splines) at a given point; see de Boor (1978, pages 351–353).

This routine returns the value of  $s^{(p,q)}$  at a point  $(x, y)$  given the coefficients  $c$  by computing

$$s^{(p,q)}(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}^{(p)}(x) B_{m,k_y,t_y}^{(q)}(y)$$

where  $k_x$  and  $k_y$  are the orders of the splines. (These numbers are passed to the subroutine in KXORD and KYORD, respectively.) Likewise,  $t_x$  and  $t_y$  are the corresponding knot sequences (XKNOT and YKNOT).

## Comments

1. Workspace may be explicitly provided, if desired, by use of B22DR/DB22DR. The reference is:

```
CALL B22DR(IXDER, IYDER, X, Y, KXORD, KYORD, XKNOT, YKNOT,
 NXCOEF, NYCOEF, BSCOEF, WK)
```

The additional argument is:

**WK** — Work array of length  $3 * \text{MAX}(KXORD, KYORD) + KYORD$ .

2. Informational errors

Type Code

|   |   |                                                                             |
|---|---|-----------------------------------------------------------------------------|
| 3 | 1 | The point X does not satisfy<br>XKNOT(KXORD) .LE. X .LE. XKNOT(NXCOEF + 1). |
| 3 | 2 | The point Y does not satisfy<br>YKNOT(KYORD) .LE. Y .LE. YKNOT(NYCOEF + 1). |

## Example

In this example, a spline interpolant  $s$  to a function  $f$  is constructed. We use the IMSL routine BS2IN to compute the interpolant and then BS2DR is employed to compute  $s^{(2,1)}(x, y)$ . The values of this partial derivative and the error are computed on a  $4 \times 4$  grid and then displayed.

```
USE BS2DR_INT
```

```

USE BSNAK_INT
USE UMACH_INT
USE BS2IN_INT

IMPLICIT NONE

! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, LDF, NXDATA, NXKNOT, NYDATA, NYKNOT
PARAMETER (KXORD=5, KYORD=3, NXDATA=21, NYDATA=6, LDF=NXDATA, &
 NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD)

!
INTEGER I, J, NOUT, NXCOEF, NYCOEF
REAL BSCOEF(NXDATA,NYDATA), F, F21, &
 FDATA(LDF,NYDATA), FLOAT, S21, X, XDATA(NXDATA), &
 XKNOT(NXKNOT), Y, YDATA(NYDATA), YKNOT(NYKNOT)
INTRINSIC FLOAT

! Define function and (2,1) derivative
F(X,Y) = X*X*X*X + X*X*X*Y*Y
F21(X,Y) = 12.0*X*Y

! Set up interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-11)/10.0
10 CONTINUE

! Generate knot sequence
CALL BSNAK (NXDATA, XDATA, KXORD, XKNOT)

! Set up interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/5.0
20 CONTINUE

! Generate knot sequence
CALL BSNAK (NYDATA, YDATA, KYORD, YKNOT)

! Generate FDATA
DO 40 I=1, NYDATA
 DO 30 J=1, NXDATA
 FDATA(J,I) = F(XDATA(J),YDATA(I))
30 CONTINUE
40 CONTINUE

! Interpolate
CALL BS2IN (XDATA, YDATA, FDATA, KXORD, KYORD, XKNOT, &
 YKNOT, BSCOEF)
NXCOEF = NXDATA
NYCOEF = NYDATA

! Get output unit number
CALL UMACH (2, NOUT)

! Write heading
WRITE (NOUT,99999)

! Print (2,1) derivative over a
! grid of [0.0,1.0] x [0.0,1.0]
! at 16 points.
DO 60 I=1, 4
 DO 50 J=1, 4
 X = FLOAT(I-1)/3.0
 Y = FLOAT(J-1)/3.0
! Evaluate spline
S21 = BS2DR(2,1,X,Y,KXORD,KYORD,XKNOT,YKNOT,NXCOEF,NYCOEF, &
 BSCOEF)

```

```

WRITE (NOUT, '(3F15.4, F15.6)') X, Y, S21, F21(X,Y) - S21
50 CONTINUE
60 CONTINUE
99999 FORMAT (39X, '(2,1)', '/', 13X, 'X', 14X, 'Y', 10X, 'S (X,Y)', &
5X, 'Error')
END

```

## Output

| X      | Y      | (2,1)<br>S (X,Y) | Error     |
|--------|--------|------------------|-----------|
| 0.0000 | 0.0000 | 0.0000           | 0.000000  |
| 0.0000 | 0.3333 | 0.0000           | 0.000000  |
| 0.0000 | 0.6667 | 0.0000           | 0.000000  |
| 0.0000 | 1.0000 | 0.0000           | 0.000001  |
| 0.3333 | 0.0000 | 0.0000           | 0.000000  |
| 0.3333 | 0.3333 | 1.3333           | 0.000002  |
| 0.3333 | 0.6667 | 2.6667           | -0.000002 |
| 0.3333 | 1.0000 | 4.0000           | 0.000008  |
| 0.6667 | 0.0000 | 0.0000           | 0.000006  |
| 0.6667 | 0.3333 | 2.6667           | -0.000011 |
| 0.6667 | 0.6667 | 5.3333           | 0.000028  |
| 0.6667 | 1.0000 | 8.0001           | -0.000134 |
| 1.0000 | 0.0000 | -0.0004          | 0.000439  |
| 1.0000 | 0.3333 | 4.0003           | -0.000319 |
| 1.0000 | 0.6667 | 7.9996           | 0.000363  |
| 1.0000 | 1.0000 | 12.0005          | -0.000458 |

---

## BS2GD

Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

### Required Arguments

***IXDER*** — Order of the derivative in the X-direction. (Input)

***IYDER*** — Order of the derivative in the Y-direction. (Input)

***XVEC*** — Array of length *NX* containing the X-coordinates at which the spline is to be evaluated. (Input)

The points in *XVEC* should be strictly increasing.

***YVEC*** — Array of length *NY* containing the Y-coordinates at which the spline is to be evaluated. (Input)

The points in *YVEC* should be strictly increasing.

***KXORD*** — Order of the spline in the X-direction. (Input)

***KYORD*** — Order of the spline in the Y-direction. (Input)

**XKNOT** — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the X-direction. (Input)

XKNOT must be nondecreasing.

**YKNOT** — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the Y-direction. (Input)

YKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXCOEF * NYCOEF$  containing the tensor-product B-spline coefficients. (Input)

BSCOEF is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$ .

**VALUE** — Value of the  $(IXDER, IYDER)$  derivative of the spline on the  $NX$  by  $NY$  grid. (Output)

VALUE (I, J) contains the derivative of the spline at the point  $(XVEC(I), YVEC(J))$ .

### Optional Arguments

**NX** — Number of grid points in the X-direction. (Input)

Default:  $NX = \text{size}(XVEC, 1)$ .

**NY** — Number of grid points in the Y-direction. (Input)

Default:  $NY = \text{size}(YVEC, 1)$ .

**NXCOEF** — Number of B-spline coefficients in the X-direction. (Input)

Default:  $NXCOEF = \text{size}(XKNOT, 1) - KXORD$ .

**NYCOEF** — Number of B-spline coefficients in the Y-direction. (Input)

Default:  $NYCOEF = \text{size}(YKNOT, 1) - KYORD$ .

**LDVALU** — Leading dimension of VALUE exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDVALU = \text{SIZE}(VALUE, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL BS2GD (IXDER, IYDER, XVEC, YVEC, KXORD, KYORD, XKNOT, YKNOT,  
                  BSCOEF, VALUE [, ...])

Specific:    The specific interface names are S\_BS2GD and D\_BS2GD.

### FORTRAN 77 Interface

Single:     CALL BS2GD (IXDER, IYDER, NX, XVEC, NY, YVEC, KXORD, KYORD, XKNOT,  
                  YKNOT, NXCOEF, NYCOEF, BSCOEF, VALUE, LDVALU)

Double:     The double precision name is DBS2GD.

## Description

The routine `BS2GD` evaluates a partial derivative of a bivariate tensor-product spline (represented as a linear combination of tensor-product B-splines) on a grid of points; see de Boor (1978, pages 351–353).

This routine returns the values of  $s^{(p,q)}$  on the grid  $(x_i, y_j)$  for  $i = 1, \dots, nx$  and  $j = 1, \dots, ny$  given the coefficients  $c$  by computing (for all  $(x, y)$  in the grid)

$$s^{(p,q)}(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,t_x}^{(p)}(x) B_{m,k_y,t_y}^{(q)}(y)$$

where  $k_x$  and  $k_y$  are the orders of the splines. (These numbers are passed to the subroutine in `KXORD` and `KYORD`, respectively.) Likewise,  $\mathbf{t}_x$  and  $\mathbf{t}_y$  are the corresponding knot sequences (`XKNOT` and `YKNOT`). The grid must be ordered in the sense that  $x_i < x_{i+1}$  and  $y_j < y_{j+1}$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B22GD/DB22GD`. The reference is:

```
CALL B22GD (IXDER, IYDER, NX, XVEC, NY, YVEC, KXORD,
 KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOEF, VALUE,
 LDVALU, LEFTX, LEFTY, A, B, DBIATX, DBIATY, BX, BY)
```

The additional arguments are as follows:

**LEFTX** — Integer work array of length `NX`.

**LEFTY** — Integer work array of length `NY`.

**A** — Work array of length `KXORD * KXORD`.

**B** — Work array of length `KYORD * KYORD`.

**DBIATX** — Work array of length `KXORD * (IXDER + 1)`.

**DBIATY** — Work array of length `KYORD * (IYDER + 1)`.

**BX** — Work array of length `KXORD * NX`.

**BY** — Work array of length `KYORD * NY`.

2. Informational errors

Type Code

```
3 1 XVEC(I) does not satisfy
 XKNOT (KXORD) .LE. XVEC(I) .LE. XKNOT(NXCOEF + 1)
```

|   |   |                                                                               |
|---|---|-------------------------------------------------------------------------------|
| 3 | 2 | YVEC(I) does not satisfy<br>YKNOT (KYORD) .LE. YVEC(I) .LE. YKNOT(NYCOEF + 1) |
| 4 | 3 | XVEC is not strictly increasing.                                              |
| 4 | 4 | YVEC is not strictly increasing.                                              |

## Example

In this example, a spline interpolant  $s$  to a function  $f$  is constructed. We use the IMSL routine `BS2IN` to compute the interpolant and then `BS2GD` is employed to compute  $s^{(2,1)}(x, y)$  on a grid. The values of this partial derivative and the error are computed on a  $4 \times 4$  grid and then displayed.

```

USE BS2GD_INT
USE BS2IN_INT
USE BSNAK_INT
USE UMACH_INT

IMPLICIT NONE
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, J, KXORD, KYORD, LDF, NOUT, NXCOEF, NXDATA, &
NYCOEF, NYDATA
REAL DCCFD(21,6), DOCBSC(21,6), DOCXD(21), DOCXK(26), &
DOCYD(6), DOCYK(9), F, F21, FLOAT, VALUE(4,4), &
X, XVEC(4), Y, YVEC(4)
INTRINSIC FLOAT
!
! Define function and derivative
F(X,Y) = X*X*X*X + X*X*X*Y*Y
F21(X,Y) = 12.0*X*Y
!
! Initialize/Setup
CALL UMACH (2, NOUT)
KXORD = 5
KYORD = 3
NXDATA = 21
NYDATA = 6
LDF = NXDATA
!
! Set up interpolation points
DO 10 I=1, NXDATA
DOCXD(I) = FLOAT(I-1)/10.0
10 CONTINUE
!
! Set up interpolation points
DO 20 I=1, NYDATA
DOCYD(I) = FLOAT(I-1)/5.0
20 CONTINUE
!
! Generate knot sequence
CALL BSNAK (NXDATA, DOCXD, KXORD, DOCXK)
!
! Generate knot sequence
CALL BSNAK (NYDATA, DOCYD, KYORD, DOCYK)
!
! Generate FDATA
DO 40 I=1, NYDATA
DO 30 J=1, NXDATA
DCCFD(J, I) = F(DOCXD(J), DOCYD(I))
30 CONTINUE
40 CONTINUE

```

```

!
! Interpolate
CALL BS2IN (DOCXD, DOCYD, DCCFD, KXORD, KYORD, &
 DOCXK, DOCYK, DOCBSC)
! Print (2,1) derivative over a
! grid of [0.0,1.0] x [0.0,1.0]
! at 16 points.
!
NXCOEF = NXDATA
NYCOEF = NYDATA
WRITE (NOUT,99999)
DO 50 I=1, 4
 XVEC(I) = FLOAT(I-1)/3.0
 YVEC(I) = XVEC(I)
50 CONTINUE
CALL BS2GD (2, 1, XVEC, YVEC, KXORD, KYORD, DOCXK, DOCYK, &
 DOCBSC, VALUE)
DO 70 I=1, 4
 DO 60 J=1, 4
 WRITE (NOUT,'(3F15.4,F15.6)') XVEC(I), YVEC(J), &
 VALUE(I,J), &
 F21(XVEC(I),YVEC(J)) - &
 VALUE(I,J)
60 CONTINUE
70 CONTINUE
99999 FORMAT (39X, '(2,1)', /, 13X, 'X', 14X, 'Y', 10X, 'S (X,Y)', &
 5X, 'Error')
END

```

## Output

| X      | Y      | S<br>(2,1)<br>(X,Y) | Error     |
|--------|--------|---------------------|-----------|
| 0.0000 | 0.0000 | 0.0000              | 0.000000  |
| 0.0000 | 0.3333 | 0.0000              | 0.000000  |
| 0.0000 | 0.6667 | 0.0000              | 0.000000  |
| 0.0000 | 1.0000 | 0.0000              | 0.000001  |
| 0.3333 | 0.0000 | 0.0000              | -0.000001 |
| 0.3333 | 0.3333 | 1.3333              | 0.000001  |
| 0.3333 | 0.6667 | 2.6667              | -0.000004 |
| 0.3333 | 1.0000 | 4.0000              | 0.000008  |
| 0.6667 | 0.0000 | 0.0000              | -0.000001 |
| 0.6667 | 0.3333 | 2.6667              | -0.000008 |
| 0.6667 | 0.6667 | 5.3333              | 0.000038  |
| 0.6667 | 1.0000 | 8.0001              | -0.000113 |
| 1.0000 | 0.0000 | -0.0005             | 0.000488  |
| 1.0000 | 0.3333 | 4.0004              | -0.000412 |
| 1.0000 | 0.6667 | 7.9995              | 0.000488  |
| 1.0000 | 1.0000 | 12.0002             | -0.000244 |

---

## BS2IG

This function evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.

## Function Return Value

**BS2IG** — Integral of the spline over the rectangle (A, B) by (C, D).  
(Output)

## Required Arguments

**A** — Lower limit of the X-variable. (Input)

**B** — Upper limit of the X-variable. (Input)

**C** — Lower limit of the Y-variable. (Input)

**D** — Upper limit of the Y-variable. (Input)

**KXORD** — Order of the spline in the X-direction. (Input)

**KYORD** — Order of the spline in the Y-direction. (Input)

**XKNOT** — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the X-direction.  
(Input)  
XKNOT must be nondecreasing.

**YKNOT** — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the Y-direction.  
(Input)  
YKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXCOEF * NYCOEF$  containing the tensor-product B-spline coefficients. (Input)  
BSCOEF is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$ .

## Optional Arguments

**NXCOEF** — Number of B-spline coefficients in the X-direction. (Input)  
Default:  $NXCOEF = \text{size}(XKNOT,1) - KXORD$ .

**NYCOEF** — Number of B-spline coefficients in the Y-direction. (Input)  
Default:  $NYCOEF = \text{size}(YKNOT,1) - KYORD$ .

## FORTRAN 90 Interface

Generic:    `BS2IG (A, B, C, D, KXORD, KYORD, XKNOT, YKNOT, BSCOEF [, ...])`

Specific:    The specific interface names are `S_BS2IG` and `D_BS2IG`.

## FORTRAN 77 Interface

Single:    `BS2IG (A, B, C, D, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOEF)`

Double: The double precision function name is `DBS2IG`.

## Description

The function `BS2IG` computes the integral of a tensor-product two-dimensional spline given its B-spline representation. Specifically, given the knot sequence  $\mathbf{t}_x = \text{XKNOT}$ ,  $\mathbf{t}_y = \text{YKNOT}$ , the order  $k_x = \text{KXORD}$ ,  $k_y = \text{KYORD}$ , the coefficients  $\boldsymbol{\beta} = \text{BSCOEF}$ , the number of coefficients  $n_x = \text{NXCOEF}$ ,  $n_y = \text{NYCOEF}$  and a rectangle  $[a, b]$  by  $[c, d]$ , `BS2IG` returns the value

$$\int_a^b \int_c^d \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} \beta_{ij} B_{ij} \, dy \, dx$$

where

$$B_{i,j}(x, y) = B_{i,k_x,t_x}(x) B_{j,k_y,t_y}(y)$$

This routine uses the identity (22) on page 151 of de Boor (1978). It assumes (for all knot sequences) that the first and last  $k$  knots are stacked, that is,  $t_1 = \dots = t_k$  and  $t_{n+1} = \dots = t_{n+k}$ , where  $k$  is the order of the spline in the  $x$  or  $y$  direction.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B22IG/DB22IG`. The reference is:

```
CALL B22IG(A, B, C, D, KXORD, KYORD, XKNOT, YKNOT, NXCOEF,
 NYCOEF, BSCOEF, WK)
```

The additional argument is:

**WK** — Work array of length  $4 * (\text{MAX}(\text{KXORD}, \text{KYORD}) + 1) + \text{NYCOEF}$ .

2. Informational errors

| Type | Code |                                                                                       |
|------|------|---------------------------------------------------------------------------------------|
| 3    | 1    | The lower limit of the x-integration is less than <code>XKNOT(KXORD)</code> .         |
| 3    | 2    | The upper limit of the x-integration is greater than <code>XKNOT(NXCOEF + 1)</code> . |
| 3    | 3    | The lower limit of the y-integration is less than <code>YKNOT(KYORD)</code> .         |
| 3    | 4    | The upper limit of the y-integration is greater than <code>YKNOT(NYCOEF + 1)</code> . |
| 4    | 13   | Multiplicity of the knots cannot exceed the order of the spline.                      |
| 4    | 14   | The knots must be nondecreasing.                                                      |

## Example

We integrate the two-dimensional tensor-product quartic ( $k_x = 5$ ) by linear ( $k_y = 2$ ) spline that interpolates  $x^3 + xy$  at the points  $\{(i/10, j/5) : i = -10, \dots, 10 \text{ and } j = 0, \dots, 5\}$  over the rectangle  $[0, 1] \times [.5, 1]$ . The exact answer is  $5/16$ .

```
USE BS2IG_INT
USE BSNAK_INT
USE BS2IN_INT
USE UMACH_INT

IMPLICIT NONE

! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, LDF, NXDATA, NXKNOT, NYDATA, NYKNOT
PARAMETER (KXORD=5, KYORD=2, NXDATA=21, NYDATA=6, LDF=NXDATA, &
 NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD)

!
INTEGER I, J, NOUT, NXCOEF, NYCOEF
REAL A, B, BSCOEFF(NXDATA, NYDATA), C, D, F, &
 FDATA(LDF, NYDATA), FI, FLOAT, VAL, X, XDATA(NXDATA), &
 XKNOT(NXKNOT), Y, YDATA(NYDATA), YKNOT(NYKNOT)
INTRINSIC FLOAT

! Define function and integral
F(X, Y) = X*X*X + X*Y
FI(A, B, C, D) = .25*((B**4-A**4)*(D-C) + (B*B-A*A)*(D*D-C*C))

! Set up interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-11)/10.0
10 CONTINUE

! Generate knot sequence
CALL BSNAK(NXDATA, XDATA, KXORD, XKNOT)

! Set up interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/5.0
20 CONTINUE

! Generate knot sequence
CALL BSNAK(NYDATA, YDATA, KYORD, YKNOT)

! Generate FDATA
DO 40 I=1, NYDATA
 DO 30 J=1, NXDATA
 FDATA(J, I) = F(XDATA(J), YDATA(I))
 30 CONTINUE
40 CONTINUE

! Interpolate
CALL BS2IN(XDATA, YDATA, FDATA, KXORD, &
 KYORD, XKNOT, YKNOT, BSCOEFF)

! Integrate over rectangle
! [0.0,1.0] x [0.0,0.5]

NXCOEF = NXDATA
NYCOEF = NYDATA
A = 0.0
B = 1.0
C = 0.5
D = 1.0
```

```

 VAL = BS2IG(A,B,C ,D,KXORD,KYORD,XKNOT,YKNOT,BSCOEF)
!
 CALL UMACH (2, NOUT)
!
 WRITE (NOUT,99999) VAL, FI(A,B,C ,D), FI(A,B,C ,D) - VAL
99999 FORMAT (' Computed Integral = ', F10.5, '/', ' Exact Integral '&
 , '= ', F10.5, '/', ' Error '&
 , '= ', F10.6, '/')
 END

```

## Output

```

Computed Integral = 0.31250
Exact Integral = 0.31250
Error = 0.000000

```

---

## BS3VL

This function Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

### Function Return Value

*BS3VL* — Value of the spline at (X, Y, Z). (Output)

### Required Arguments

*X* — x-coordinate of the point at which the spline is to be evaluated. (Input)

*Y* — y-coordinate of the point at which the spline is to be evaluated. (Input)

*Z* — z-coordinate of the point at which the spline is to be evaluated. (Input)

*KXORD* — Order of the spline in the X-direction. (Input)

*KYORD* — Order of the spline in the Y-direction. (Input)

*KZORD* — Order of the spline in the Z-direction. (Input)

*XKNOT* — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the X-direction.  
(Input)  
*XKNOT* must be nondecreasing.

*YKNOT* — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the Y-direction.  
(Input)  
*YKNOT* must be nondecreasing.

*ZKNOT* — Array of length  $NZCOEF + KZORD$  containing the knot sequence in the Z-direction.  
(Input)  
*ZKNOT* must be nondecreasing.

*NXCOEF* — Number of B-spline coefficients in the x-direction. (Input)

*NYCOEF* — Number of B-spline coefficients in the y-direction. (Input)

*NZCOEF* — Number of B-spline coefficients in the z-direction. (Input)

*BSCOEF* — Array of length  $NXCOEF * NYCOEF * NZCOEF$  containing the tensor-product B-spline coefficients. (Input)

*BSCOEF* is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$  by  $NZCOEF$ .

### FORTRAN 90 Interface

Generic: `BS3VL (X, Y, Z, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)`

Specific: The specific interface names are `S_BS3VL` and `D_BS3VL`.

### FORTRAN 77 Interface

Single: `BS3VL (X, Y, Z, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)`

Double: The double precision function name is `DBS3VL`.

### Description

The function `BS3VL` evaluates a trivariate tensor-product spline (represented as a linear combination of tensor-product B-splines) at a given point. This routine is a special case of the IMSL routine `BS3DR`, which evaluates a partial derivative of such a spline. (The value of a spline is its zero-th derivative.) For more information, see de Boor (1978, pages 351 – 353).

This routine returns the value of the function  $s$  at a point  $(x, y, z)$  given the coefficients  $c$  by computing

$$s(x, y, z) = \sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}(x) B_{m,k_y,t_y}(y) B_{l,k_z,t_z}(z)$$

where  $k_x$ ,  $k_y$ , and  $k_z$  are the orders of the splines. (These numbers are passed to the subroutine in `KXORD`, `KYORD`, and `KZORD`, respectively.) Likewise,  $t_x$ ,  $t_y$ , and  $t_z$  are the corresponding knot sequences (`XKNOT`, `YKNOT`, and `ZKNOT`).

### Comments

Workspace may be explicitly provided, if desired, by use of `B23VL/DB23VL`. The reference is:

```
CALL B23VL (X, Y, Z, KXORD, KYORD, KZORD, XKNOT, YKNOT,
 ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF, WK)
```

The additional argument is:

**WK** — Work array of length  $3 * \text{MAX}(\text{KXORD}, \text{KYORD}, \text{KZORD}) + \text{KYORD} * \text{KZORD} + \text{KZORD}$ .

### Example

For an example of the use of `BS3VL`, see IMSL routine `BS3IN`.

---

## BS3DR

This function evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

### Function Return Value

**BS3DR** — Value of the  $(\text{IXDER}, \text{IYDER}, \text{IZDER})$  derivative of the spline at  $(X, Y, Z)$ .  
(Output)

### Required Arguments

**IXDER** — Order of the X-derivative. (Input)

**IYDER** — Order of the Y-derivative. (Input)

**IZDER** — Order of the Z-derivative. (Input)

**X** — X-coordinate of the point at which the spline is to be evaluated. (Input)

**Y** — Y-coordinate of the point at which the spline is to be evaluated. (Input)

**Z** — Z-coordinate of the point at which the spline is to be evaluated. (Input)

**KXORD** — Order of the spline in the X-direction. (Input)

**KYORD** — Order of the spline in the Y-direction. (Input)

**KZORD** — Order of the spline in the Z-direction. (Input)

**XKNOT** — Array of length  $\text{NXCOEF} + \text{KXORD}$  containing the knot sequence in the X-direction.  
(Input)  
KNOT must be nondecreasing.

**YKNOT** — Array of length  $\text{NYCOEF} + \text{KYORD}$  containing the knot sequence in the Y-direction.  
(Input)  
YKNOT must be nondecreasing.

**ZKNOT** — Array of length  $NZCOEF + KZORD$  containing the knot sequence in the z-direction. (Input)

ZKNOT must be nondecreasing.

**NXCOEF** — Number of B-spline coefficients in the x-direction. (Input)

**NYCOEF** — Number of B-spline coefficients in the y-direction. (Input)

**NZCOEF** — Number of B-spline coefficients in the z-direction. (Input)

**BSCOEF** — Array of length  $NXCOEF * NYCOEF * NZCOEF$  containing the tensor-product B-spline coefficients. (Input)

BSCOEF is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$  by  $NZCOEF$ .

### FORTRAN 90 Interface

Generic: BS3DR (IXDER, IYDER, IZDER, X, Y, Z, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)

Specific: The specific interface names are S\_BS3DR and D\_BS3DR.

### FORTRAN 77 Interface

Single: BS3DR (IXDER, IYDER, IZDER, X, Y, Z, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)

Double: The double precision function name is DBS3DR.

### Description

The function BS3DR evaluates a partial derivative of a trivariate tensor-product spline (represented as a linear combination of tensor-product B-splines) at a given point. For more information, see de Boor (1978, pages 351–353).

This routine returns the value of the function  $s^{(p, q, r)}$  at a point  $(x, y, z)$  given the coefficients  $c$  by computing

$$s^{(p, q, r)}(x, y, z) = \sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n, k_x, \mathbf{t}_x}^{(p)}(x) B_{m, k_y, \mathbf{t}_y}^{(q)}(y) B_{l, k_z, \mathbf{t}_z}^{(r)}(z)$$

where  $k_x$ ,  $k_y$ , and  $k_z$  are the orders of the splines. (These numbers are passed to the subroutine in KXORD, KYORD, and KZORD, respectively.) Likewise,  $\mathbf{t}_x$ ,  $\mathbf{t}_y$ , and  $\mathbf{t}_z$  are the corresponding knot sequences (XKNOT, YKNOT, and ZKNOT).

## Comments

1. Workspace may be explicitly provided, if desired, by use of B23DR/DB23DR. The reference is:

```
CALL B23DR(IXDER, IYDER, IZDER, X, Y, Z, KXORD, KYORD,
 KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF,
 BSCOE, WK)
```

The additional argument is:

**WK** — Work array of length  $3 * \text{MAX0}(KXORD, KYORD, KZORD) + KYORD * KZORD + KZORD$ .

2. Informational errors

| Type | Code |                                                                             |
|------|------|-----------------------------------------------------------------------------|
| 3    | 1    | The point X does not satisfy<br>XKNOT(KXORD) .LE. X .LE. XKNOT(NXCOEF + 1). |
| 3    | 2    | The point Y does not satisfy<br>YKNOT(KYORD) .LE. Y .LE. YKNOT(NYCOEF + 1). |
| 3    | 3    | The point Z does not satisfy<br>ZKNOT(KZORD) .LE. Z .LE. ZKNOT(NZCOEF + 1). |

## Example

In this example, a spline interpolant  $s$  to a function  $f(x, y, z) = x^4 + y(xz)^3$  is constructed using BS3IN. Next, BS3DR is used to compute  $s^{(2,0,1)}(x, y, z)$ . The values of this partial derivative and the error are computed on a  $4 \times 4 \times 2$  grid and then displayed.

```
USE BS3DR_INT
USE BS3IN_INT
USE BSNK_INT
USE UMACH_INT

IMPLICIT NONE
!
! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, KZORD, LDF, MDF, NXDATA, NXKNOT, &
 NYDATA, NYKNOT, NZDATA, NZKNOT
PARAMETER (KXORD=5, KYORD=2, KZORD=3, NXDATA=21, NYDATA=6, &
 NZDATA=8, LDF=NXDATA, MDF=NYDATA, &
 NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD, &
 NZKNOT=NZDATA+KZORD)
!
INTEGER I, J, K, L, NOUT, NXCOEF, NYCOEF, NZCOEF
REAL BSCOE(NXDATA, NYDATA, NZDATA), F, F201, &
 FDATA(LDF, MDF, NZDATA), FLOAT, S201, X, XDATA(NXDATA), &
 XKNOT(NXKNOT), Y, YDATA(NYDATA), YKNOT(NYKNOT), Z, &
 ZDATA(NZDATA), ZKNOT(NZKNOT)
INTRINSIC FLOAT
!
! Define function and (2,0,1)
! derivative
```

```

F(X,Y,Z) = X*X*X*X + X*X*X*Y*Z*Z*Z
F201(X,Y,Z) = 18.0*X*Y*Z
!
! Set up X-interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-1)/10.0
10 CONTINUE
!
! Set up Y-interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/FLOAT(NYDATA-1)
20 CONTINUE
!
! Set up Z-interpolation points
DO 30 I=1, NZDATA
 ZDATA(I) = FLOAT(I-1)/FLOAT(NZDATA-1)
30 CONTINUE
!
! Generate knots
CALL BSNK (NXDATA, XDATA, KXORD, XKNOT)
CALL BSNK (NYDATA, YDATA, KYORD, YKNOT)
CALL BSNK (NZDATA, ZDATA, KZORD, ZKNOT)
!
! Generate FDATA
DO 50 K=1, NZDATA
 DO 40 I=1, NYDATA
 DO 40 J=1, NXDATA
 FDATA(J,I,K) = F(XDATA(J),YDATA(I),ZDATA(K))
40 CONTINUE
50 CONTINUE
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Interpolate&
CALL BS3IN (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD, KZORD, XKNOT, &
 YKNOT, ZKNOT, BSCOEFF)
!
NXCOEF = NXDATA
NYCOEF = NYDATA
NZCOEF = NZDATA
!
! Write heading
WRITE (NOUT,99999)
!
! Print over a grid of
! [-1.0,1.0] x [0.0,1.0] x [0.0,1.0]
! at 32 points.
DO 80 I=1, 4
 DO 70 J=1, 4
 DO 60 L=1, 2
 X = 2.0*(FLOAT(I-1)/3.0) - 1.0
 Y = FLOAT(J-1)/3.0
 Z = FLOAT(L-1)
!
! Evaluate spline
S201 = BS3DR(2,0,1,X,Y,Z,KXORD,KYORD,KZORD,XKNOT,YKNOT,&
 ZKNOT,NXCOEF,NYCOEF,NZCOEF,BSCOEFF)
WRITE (NOUT,'(3F12.4,2F12.6)') X, Y, Z, S201,&
 F201(X,Y,Z) - S201
60 CONTINUE
70 CONTINUE
80 CONTINUE
99999 FORMAT (38X, '(2,0,1)', /, 9X, 'X', 11X,&
 'Y', 11X, 'Z', 4X, 'S (X,Y,Z) Error')
END

```

## Output

| X       | Y      | Z      | S          | (2,0,1)<br>(X,Y,Z) | Error     |
|---------|--------|--------|------------|--------------------|-----------|
| -1.0000 | 0.0000 | 0.0000 | -0.000107  | 0.000107           | 0.000107  |
| -1.0000 | 0.0000 | 1.0000 | 0.000053   | -0.000053          | -0.000053 |
| -1.0000 | 0.3333 | 0.0000 | 0.064051   | -0.064051          | -0.064051 |
| -1.0000 | 0.3333 | 1.0000 | -5.935941  | -0.064059          | -0.064059 |
| -1.0000 | 0.6667 | 0.0000 | 0.127542   | -0.127542          | -0.127542 |
| -1.0000 | 0.6667 | 1.0000 | -11.873034 | -0.126966          | -0.126966 |
| -1.0000 | 1.0000 | 0.0000 | 0.191166   | -0.191166          | -0.191166 |
| -1.0000 | 1.0000 | 1.0000 | -17.808527 | -0.191473          | -0.191473 |
| -0.3333 | 0.0000 | 0.0000 | -0.000002  | 0.000002           | 0.000002  |
| -0.3333 | 0.0000 | 1.0000 | 0.000000   | 0.000000           | 0.000000  |
| -0.3333 | 0.3333 | 0.0000 | 0.021228   | -0.021228          | -0.021228 |
| -0.3333 | 0.3333 | 1.0000 | -1.978768  | -0.021232          | -0.021232 |
| -0.3333 | 0.6667 | 0.0000 | 0.042464   | -0.042464          | -0.042464 |
| -0.3333 | 0.6667 | 1.0000 | -3.957536  | -0.042464          | -0.042464 |
| -0.3333 | 1.0000 | 0.0000 | 0.063700   | -0.063700          | -0.063700 |
| -0.3333 | 1.0000 | 1.0000 | -5.936305  | -0.063694          | -0.063694 |
| 0.3333  | 0.0000 | 0.0000 | -0.000003  | 0.000003           | 0.000003  |
| 0.3333  | 0.0000 | 1.0000 | 0.000000   | 0.000000           | 0.000000  |
| 0.3333  | 0.3333 | 0.0000 | -0.021229  | 0.021229           | 0.021229  |
| 0.3333  | 0.3333 | 1.0000 | 1.978763   | 0.021238           | 0.021238  |
| 0.3333  | 0.6667 | 0.0000 | -0.042465  | 0.042465           | 0.042465  |
| 0.3333  | 0.6667 | 1.0000 | 3.957539   | 0.042462           | 0.042462  |
| 0.3333  | 1.0000 | 0.0000 | -0.063700  | 0.063700           | 0.063700  |
| 0.3333  | 1.0000 | 1.0000 | 5.936304   | 0.063697           | 0.063697  |
| 1.0000  | 0.0000 | 0.0000 | -0.000098  | 0.000098           | 0.000098  |
| 1.0000  | 0.0000 | 1.0000 | 0.000053   | -0.000053          | -0.000053 |
| 1.0000  | 0.3333 | 0.0000 | -0.063855  | 0.063855           | 0.063855  |
| 1.0000  | 0.3333 | 1.0000 | 5.936146   | 0.063854           | 0.063854  |
| 1.0000  | 0.6667 | 0.0000 | -0.127631  | 0.127631           | 0.127631  |
| 1.0000  | 0.6667 | 1.0000 | 11.873067  | 0.126933           | 0.126933  |
| 1.0000  | 1.0000 | 0.0000 | -0.191442  | 0.191442           | 0.191442  |
| 1.0000  | 1.0000 | 1.0000 | 17.807940  | 0.192060           | 0.192060  |

---

## BS3GD

Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

### Required Arguments

*IXDER* — Order of the x-derivative. (Input)

*IYDER* — Order of the y-derivative. (Input)

*IZDER* — Order of the z-derivative. (Input)

*XVEC* — Array of length *NX* containing the x-coordinates at which the spline is to be evaluated. (Input)

The points in *XVEC* should be strictly increasing.

**YVEC** — Array of length *NY* containing the *y*-coordinates at which the spline is to be evaluated. (Input)

The points in *YVEC* should be strictly increasing.

**ZVEC** — Array of length *NY* containing the *y*-coordinates at which the spline is to be evaluated. (Input)

The points in *YVEC* should be strictly increasing.

**KXORD** — Order of the spline in the *x*-direction. (Input)

**KYORD** — Order of the spline in the *y*-direction. (Input)

**KZORD** — Order of the spline in the *z*-direction. (Input)

**XKNOT** — Array of length *NXCOEF* + *KXORD* containing the knot sequence in the *x*-direction. (Input)

*XKNOT* must be nondecreasing.

**YKNOT** — Array of length *NYCOEF* + *KYORD* containing the knot sequence in the *y*-direction. (Input)

*YKNOT* must be nondecreasing.

**ZKNOT** — Array of length *NZCOEF* + *KZORD* containing the knot sequence in the *z*-direction. (Input)

*ZKNOT* must be nondecreasing.

**BSCOEF** — Array of length *NXCOEF* \* *NYCOEF* \* *NZCOEF* containing the tensor-product B-spline coefficients. (Input)

*BSCOEF* is treated internally as a matrix of size *NXCOEF* by *NYCOEF* by *NZCOEF*.

**VALUE** — Array of size *NX* by *NY* by *NZ* containing the values of the (*IXDER*, *IYDER*, *IZDER*) derivative of the spline on the *NX* by *NY* by *NZ* grid. (Output)

*VALUE*(*I*, *J*, *K*) contains the derivative of the spline at the point (*XVEC*(*I*), *YVEC*(*J*), *ZVEC*(*K*)).

### Optional Arguments

***NX*** — Number of grid points in the *x*-direction. (Input)

Default: *NX* = size (*XVEC*,1).

***NY*** — Number of grid points in the *y*-direction. (Input)

Default: *NY* = size (*YVEC*,1).

***NZ*** — Number of grid points in the *z*-direction. (Input)

Default: *NZ* = size (*ZVEC*,1).

***NXCOEF*** — Number of B-spline coefficients in the *x*-direction. (Input)

Default: *NXCOEF* = size (*XKNOT*,1) - *KXORD*.

**NYCOEF** — Number of B-spline coefficients in the y-direction. (Input)

Default: NYCOEF = size (YKNOT,1) – KYORD.

**NZCOEF** — Number of B-spline coefficients in the z-direction. (Input)

Default: NZCOEF = size (ZKNOT,1) – KZORD.

**LDVALU** — Leading dimension of VALUE exactly as specified in the dimension statement of the calling program. (Input)

Default: LDVALU = SIZE (VALUE,1).

**MDVALU** — Middle dimension of VALUE exactly as specified in the dimension statement of the calling program. (Input)

Default: MDVALU = SIZE (VALUE,2).

## FORTRAN 90 Interface

Generic: CALL BS3GD (IXDER, IYDER, IZDER, XVEC, YVEC, ZVEC, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, BSCOEF, VALUE [, ...])

Specific: The specific interface names are S\_BS3GD and D\_BS3GD.

## FORTRAN 77 Interface

Single: CALL BS3GD (IXDER, IYDER, IZDER, NX, XVEC, NY, YVEC, NZ, ZVEC, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF, VALUE, LDVALU, MDVALU)

Double: The double precision name is DBS3GD.

## Description

The routine BS3GD evaluates a partial derivative of a trivariate tensor-product spline (represented as a linear combination of tensor-product B-splines) on a grid. For more information, see de Boor (1978, pages 351–353).

This routine returns the value of the function  $s^{(p,q,r)}$  on the grid  $(x_i, y_j, z_k)$  for  $i = 1, \dots, nx$ ,  $j = 1, \dots, ny$ , and  $k = 1, \dots, nz$  given the coefficients  $c$  by computing (for all  $(x, y, z)$  on the grid)

$$s^{(p,q,r)}(x, y, z) = \sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,t_x}^{(p)}(x) B_{m,k_y,t_y}^{(q)}(y) B_{l,k_z,t_z}^{(r)}(z)$$

where  $k_x$ ,  $k_y$ , and  $k_z$  are the orders of the splines. (These numbers are passed to the subroutine in KXORD, KYORD, and KZORD, respectively.) Likewise,  $t_x$ ,  $t_y$ , and  $t_z$  are the corresponding knot sequences (XKNOT, YKNOT, and ZKNOT). The grid must be ordered in the sense that  $x_i < x_{i+1}$ ,  $y_j < y_{j+1}$ , and  $z_k < z_{k+1}$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of B23GD/DB23GD. The reference is:

```
CALL B23GD ((IXDER, IYDER, IZDER, NX, XVEC, NY, YVEC, NZ,
 ZVEC, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT,
 NXCOEF, NYCOEF, NZCOEF, BSCOE, VALUE, LDVALU,
 MDVALU, LEFTX, LEFTY, LEFTZ, A, B, C, DBIATX, DBIATY,
 DBIATZ, BX, BY, BZ)
```

The additional arguments are as follows:

**LEFTX** — Work array of length  $NX$ .

**LEFTY** — Work array of length  $NY$ .

**LEFTZ** — Work array of length  $NZ$ .

**A** — Work array of length  $KXORD * KXORD$ .

**B** — Work array of length  $KYORD * KYORD$ .

**C** — Work array of length  $KZORD * KZORD$ .

**DBIATX** — Work array of length  $KXORD * (IXDER + 1)$ .

**DBIATY** — Work array of length  $KYORD * (IYDER + 1)$ .

**DBIATZ** — Work array of length  $KZORD * (IZDER + 1)$ .

**BX** — Work array of length  $KXORD * NX$ .

**BY** — Work array of length  $KYORD * NY$ .

**BZ** — Work array of length  $KZORD * NZ$ .

2. Informational errors

Type Code

|   |   |                                                                                  |
|---|---|----------------------------------------------------------------------------------|
| 3 | 1 | XVEC(I) does not satisfy<br>$XKNOT(KXORD) \leq XVEC(I) \leq XKNOT(NXCOEF + 1)$ . |
| 3 | 2 | YVEC(I) does not satisfy<br>$YKNOT(KYORD) \leq YVEC(I) \leq YKNOT(NYCOEF + 1)$ . |
| 3 | 3 | ZVEC(I) does not satisfy<br>$ZKNOT(KZORD) \leq ZVEC(I) \leq ZKNOT(NZCOEF + 1)$ . |

```

4 4 XVEC is not strictly increasing.
4 5 YVEC is not strictly increasing.
4 6 ZVEC is not strictly increasing.

```

## Example

In this example, a spline interpolant  $s$  to a function  $f(x, y, z) = x^4 + y(xz)^3$  is constructed using `BS3IN`. Next, `BS3GD` is used to compute  $s^{(2,0,1)}(x, y, z)$  on the grid. The values of this partial derivative and the error are computed on a  $4 \times 4 \times 2$  grid and then displayed.

```

USE BS3GD_INT
USE BS3IN_INT
USE BSNAK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KXORD, KYORD, KZORD, LDF, LDVAL, MDF, MDVAL, NXDATA, &
 NXKNOT, NYDATA, NYKNOT, NZ, NZDATA, NZKNOT
PARAMETER (KXORD=5, KYORD=2, KZORD=3, LDVAL=4, MDVAL=4, &
 NXDATA=21, NYDATA=6, NZ=2, NZDATA=8, LDF=NXDATA, &
 MDF=NYDATA, NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD, &
 NZKNOT=NZDATA+KZORD)
!
INTEGER I, J, K, L, NOUT, NXCOEF, NYCOEF, NZCOEF
REAL BSCOEFF(NXDATA, NYDATA, NZDATA), F, F201, &
 FDATA(LDF, MDF, NZDATA), FLOAT, VALUE(LDVAL, MDVAL, NZ), &
 X, XDATA(NXDATA), XKNOT(NXKNOT), XVEC(LDVAL), Y, &
 YDATA(NYDATA), YKNOT(NYKNOT), YVEC(MDVAL), Z, &
 ZDATA(NZDATA), ZKNOT(NZKNOT), ZVEC(NZ)
INTRINSIC FLOAT
!
!
!
F(X, Y, Z) = X*X*X*X + X*X*X*Y*Z*Z*Z
F201(X, Y, Z) = 18.0*X*Y*Z
!
CALL UMACH (2, NOUT)
!
! Set up X interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NXDATA-1)) - 1.0
10 CONTINUE
!
! Set up Y interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/FLOAT(NYDATA-1)
20 CONTINUE
!
! Set up Z interpolation points
DO 30 I=1, NZDATA
 ZDATA(I) = FLOAT(I-1)/FLOAT(NZDATA-1)
30 CONTINUE
!
! Generate knots
CALL BSNAK (NXDATA, XDATA, KXORD, XKNOT)
CALL BSNAK (NYDATA, YDATA, KYORD, YKNOT)
CALL BSNAK (NZDATA, ZDATA, KZORD, ZKNOT)
!
! Generate FDATA

```

```

DO 50 K=1, NZDATA
 DO 40 I=1, NYDATA
 DO 40 J=1, NXDATA
 FDATA(J,I,K) = F(XDATA(J),YDATA(I),ZDATA(K))
40 CONTINUE
50 CONTINUE
!
! Interpolate
CALL BS3IN (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD,&
 KZORD, XKNOT, YKNOT, ZKNOT, BSCOEUF)
!
NXCOEF = NXDATA
NYCOEF = NYDATA
NZCOEF = NZDATA
!
! Print over a grid of
! [-1.0,1.0] x [0.0,1.0] x [0.0,1.0]
! at 32 points.
DO 60 I=1, 4
 XVEC(I) = 2.0*(FLOAT(I-1)/3.0) - 1.0
60 CONTINUE
DO 70 J=1, 4
 YVEC(J) = FLOAT(J-1)/3.0
70 CONTINUE
DO 80 L=1, 2
 ZVEC(L) = FLOAT(L-1)
80 CONTINUE
CALL BS3GD (2, 0, 1, XVEC, YVEC, ZVEC, KXORD, KYORD,&
 KZORD, XKNOT, YKNOT, ZKNOT, BSCOEUF, VALUE)
!
!
WRITE (NOUT,99999)
DO 110 I=1, 4
 DO 100 J=1, 4
 DO 90 L=1, 2
 WRITE (NOUT,'(5F13.4)') XVEC(I), YVEC(J), ZVEC(L),&
 VALUE(I,J,L),&
 F201(XVEC(I),YVEC(J),ZVEC(L)) -&
 VALUE(I,J,L)
90 CONTINUE
100 CONTINUE
110 CONTINUE
99999 FORMAT (44X, '(2,0,1)', /, 10X, 'X', 11X, 'Y', 10X, 'Z', 10X,&
 'S (X,Y,Z) Error')
STOP
END

```

## Output

| X       | Y      | Z      | S (2,0,1)<br>S (X,Y,Z) | Error   |
|---------|--------|--------|------------------------|---------|
| -1.0000 | 0.0000 | 0.0000 | -0.0005                | 0.0005  |
| -1.0000 | 0.0000 | 1.0000 | 0.0002                 | -0.0002 |
| -1.0000 | 0.3333 | 0.0000 | 0.0641                 | -0.0641 |
| -1.0000 | 0.3333 | 1.0000 | -5.9360                | -0.0640 |
| -1.0000 | 0.6667 | 0.0000 | 0.1274                 | -0.1274 |
| -1.0000 | 0.6667 | 1.0000 | -11.8730               | -0.1270 |

|         |        |        |          |         |
|---------|--------|--------|----------|---------|
| -1.0000 | 1.0000 | 0.0000 | 0.1911   | -0.1911 |
| -1.0000 | 1.0000 | 1.0000 | -17.8086 | -0.1914 |
| -0.3333 | 0.0000 | 0.0000 | 0.0000   | 0.0000  |
| -0.3333 | 0.0000 | 1.0000 | 0.0000   | 0.0000  |
| -0.3333 | 0.3333 | 0.0000 | 0.0212   | -0.0212 |
| -0.3333 | 0.3333 | 1.0000 | -1.9788  | -0.0212 |
| -0.3333 | 0.6667 | 0.0000 | 0.0425   | -0.0425 |
| -0.3333 | 0.6667 | 1.0000 | -3.9575  | -0.0425 |
| -0.3333 | 1.0000 | 0.0000 | 0.0637   | -0.0637 |
| -0.3333 | 1.0000 | 1.0000 | -5.9363  | -0.0637 |
| 0.3333  | 0.0000 | 0.0000 | 0.0000   | 0.0000  |
| 0.3333  | 0.0000 | 1.0000 | 0.0000   | 0.0000  |
| 0.3333  | 0.3333 | 0.0000 | -0.0212  | 0.0212  |
| 0.3333  | 0.3333 | 1.0000 | 1.9788   | 0.0212  |
| 0.3333  | 0.6667 | 0.0000 | -0.0425  | 0.0425  |
| 0.3333  | 0.6667 | 1.0000 | 3.9575   | 0.0425  |
| 0.3333  | 1.0000 | 0.0000 | -0.0637  | 0.0637  |
| 0.3333  | 1.0000 | 1.0000 | 5.9363   | 0.0637  |
| 1.0000  | 0.0000 | 0.0000 | -0.0005  | 0.0005  |
| 1.0000  | 0.0000 | 1.0000 | 0.0000   | 0.0000  |
| 1.0000  | 0.3333 | 0.0000 | -0.0637  | 0.0637  |
| 1.0000  | 0.3333 | 1.0000 | 5.9359   | 0.0641  |
| 1.0000  | 0.6667 | 0.0000 | -0.1273  | 0.1273  |
| 1.0000  | 0.6667 | 1.0000 | 11.8733  | 0.1267  |
| 1.0000  | 1.0000 | 0.0000 | -0.1912  | 0.1912  |
| 1.0000  | 1.0000 | 1.0000 | 17.8096  | 0.1904  |

---

## BS3IG

This function evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensor-product B-spline representation.

### Function Return Value

*BS3IG* — Integral of the spline over the three-dimensional rectangle (A, B) by (C, D) by (E, F).  
(Output)

### Required Arguments

*A* — Lower limit of the x-variable. (Input)

*B* — Upper limit of the x-variable. (Input)

*C* — Lower limit of the y-variable. (Input)

*D* — Upper limit of the y-variable. (Input)

*E* — Lower limit of the z-variable. (Input)

*F* — Upper limit of the z-variable. (Input)

*KXORD* — Order of the spline in the x-direction. (Input)

**KYORD** — Order of the spline in the Y-direction. (Input)

**KZORD** — Order of the spline in the Z-direction. (Input)

**XKNOT** — Array of length  $NXCOEF + KXORD$  containing the knot sequence in the X-direction. (Input)  
XKNOT must be nondecreasing.

**YKNOT** — Array of length  $NYCOEF + KYORD$  containing the knot sequence in the Y-direction. (Input)  
YKNOT must be nondecreasing.

**ZKNOT** — Array of length  $NZCOEF + KZORD$  containing the knot sequence in the Z-direction. (Input)  
ZKNOT must be nondecreasing.

**NXCOEF** — Number of B-spline coefficients in the X-direction. (Input)

**NYCOEF** — Number of B-spline coefficients in the Y-direction. (Input)

**NZCOEF** — Number of B-spline coefficients in the Z-direction. (Input)

**BSCOEF** — Array of length  $NXCOEF * NYCOEF * NZCOEF$  containing the tensor-product B-spline coefficients. (Input)  
BSCOEF is treated internally as a matrix of size  $NXCOEF$  by  $NYCOEF$  by  $NZCOEF$ .

## **FORTRAN 90 Interface**

Generic: BS3IG (A, B, C, D, E, F, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)

Specific: The specific interface names are S\_BS3IG and D\_BS3IG.

## **FORTRAN 77 Interface**

Single: BS3IG (A, B, C, D, E, F, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOEF)

Double: The double precision function name is DBS3IG.

## **Description**

The routine BS3IG computes the integral of a tensor-product three-dimensional spline, given its B-spline representation. Specifically, given the knot sequence  $\mathbf{t}_x = \mathbf{XKNOT}$ ,  $\mathbf{t}_y = \mathbf{YKNOT}$ ,  $\mathbf{t}_z = \mathbf{ZKNOT}$ , the order  $k_x = KXORD$ ,  $k_y = KYORD$ ,  $k_z = KZORD$ , the coefficients  $\beta = \mathbf{BSCOEF}$ , the number of coefficients  $n_x = NXCOEF$ ,  $n_y = NYCOEF$ ,  $n_z = NZCOEF$ , and a three-dimensional rectangle  $[a, b]$  by  $[c, d]$  by  $[e, f]$ , BS3IG returns the value

$$\int_a^b \int_c^d \int_e^f \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} \sum_{m=1}^{n_z} \beta_{ijm} B_{ijm} dz dy dx$$

where

$$B_{ijm}(x, y, z) = B_{i,k_x,t_x}(x) B_{j,k_y,t_y}(y) B_{m,k_z,t_z}(z)$$

This routine uses the identity (22) on page 151 of de Boor (1978). It assumes (for all knot sequences) that the first and last  $k$  knots are stacked, that is,  $\mathbf{t}_1 = \dots = \mathbf{t}_k$  and  $\mathbf{t}_{n+1} = \dots = \mathbf{t}_{n+k}$ , where  $k$  is the order of the spline in the  $x$ ,  $y$ , or  $z$  direction.

## Comments

1. Workspace may be explicitly provided, if desired, by use of B23IG/DB23IG. The reference is:

```
CALL B23IG(A, B, C, D, E, F, KXORD, KYORD, KZORD, XKNOT,
 YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, BSCOE, WK)
```

The additional argument is:

**WK** — Work array of length  $4 * (\text{MAX}(\text{KXORD}, \text{KYORD}, \text{KZORD}) + 1) + \text{NYCOEF} + \text{NZCOEF}$ .

2. Informational errors

| Type | Code | Description                                                             |
|------|------|-------------------------------------------------------------------------|
| 3    | 1    | The lower limit of the X-integration is less than XKNOT(KXORD).         |
| 3    | 2    | The upper limit of the X-integration is greater than XKNOT(NXCOEF + 1). |
| 3    | 3    | The lower limit of the Y-integration is less than YKNOT(KYORD).         |
| 3    | 4    | The upper limit of the Y-integration is greater than YKNOT(NYCOEF + 1). |
| 3    | 5    | The lower limit of the Z- integration is less than ZKNOT(KZORD).        |
| 3    | 6    | The upper limit of the Z-integration is greater than ZKNOT(NZCOEF + 1). |
| 4    | 13   | Multiplicity of the knots cannot exceed the order of the spline.        |
| 4    | 14   | The knots must be nondecreasing.                                        |

## Example

We integrate the three-dimensional tensor-product quartic ( $k_x = 5$ ) by linear ( $k_y = 2$ ) by quadratic ( $k_z = 3$ ) spline which interpolates  $x^3 + xyz$  at the points

$$\{(i/10, j/5, m/7) : i = -10, \dots, 10, j = 0, \dots, 5, \text{ and } m = 0, \dots, 7\}$$

over the rectangle  $[0, 1] \times [0, 1] \times [0, .5]$ . The exact answer is  $11/128$ .

```

USE BS3IG_INT
USE BS3IN_INT
USE BSNK_INT
USE UMACH_INT

IMPLICIT NONE

! SPECIFICATIONS FOR PARAMETERS
INTEGER KXORD, KYORD, KZORD, LDF, MDF, NXDATA, NXKNOT, &
 NYDATA, NYKNOT, NZDATA, NZKNOT
PARAMETER (KXORD=5, KYORD=2, KZORD=3, NXDATA=21, NYDATA=6, &
 NZDATA=8, LDF=NXDATA, MDF=NYDATA, &
 NXKNOT=NXDATA+KXORD, NYKNOT=NYDATA+KYORD, &
 NZKNOT=NZDATA+KZORD)

!
INTEGER I, J, K, NOUT, NXCOEF, NYCOEF, NZCOEF
REAL A, B, BSCOEFF(NXDATA,NYDATA,NZDATA), C, D, E, &
 F, FDATA(LDF,MDF,NZDATA), FF, FIG, FLOAT, G, H, RI, &
 RJ, VAL, X, XDATA(NXDATA), XKNOT(NXKNOT), Y, &
 YDATA(NYDATA), YKNOT(NYKNOT), Z, ZDATA(NZDATA), &
 ZKNOT(NZKNOT)
INTRINSIC FLOAT

! Define function
F(X,Y,Z) = X*X*X + X*Y*Z

! Set up interpolation points
DO 10 I=1, NXDATA
 XDATA(I) = FLOAT(I-1)/10.0
10 CONTINUE

! Generate knot sequence
CALL BSNK (NXDATA, XDATA, KXORD, XKNOT)

! Set up interpolation points
DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/FLOAT(NYDATA-1)
20 CONTINUE

! Generate knot sequence
CALL BSNK (NYDATA, YDATA, KYORD, YKNOT)

! Set up interpolation points
DO 30 I=1, NZDATA
 ZDATA(I) = FLOAT(I-1)/FLOAT(NZDATA-1)
30 CONTINUE

! Generate knot sequence
CALL BSNK (NZDATA, ZDATA, KZORD, ZKNOT)

! Generate FDATA
DO 50 K=1, NZDATA
 DO 40 I=1, NYDATA
 DO 40 J=1, NXDATA
 FDATA(J,I,K) = F(XDATA(J),YDATA(I),ZDATA(K))
40 CONTINUE
50 CONTINUE

! Get output unit number
CALL UMACH (2, NOUT)

! Interpolate
CALL BS3IN (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD, KZORD, XKNOT, &
 YKNOT, ZKNOT, BSCOEFF)

```

```

!
 NXCOEF = NXDATA
 NYCOEF = NYDATA
 NZCOEF = NZDATA
 A = 0.0
 B = 1.0
 C = 0.5
 D = 1.0
 E = 0.0
 FF = 0.5
!
 Integrate
 VAL = BS3IG(A,B,C ,D,E,FF,KXORD,KYORD,KZORD,XKNOT,YKNOT,ZKNOT,&
 NXCOEF,NYCOEF,NZCOEF,BSCOEF)
!
 Calculate integral directly
 G = .5*(B**4-A**4)
 H = (B-A)*(B+A)
 RI = G*(D-C)
 RJ = .5*H*(D-C)*(D+C)
 FIG = .5*(RI*(FF-E)+.5*RJ*(FF-E)*(FF+E))
!
 Print results
 WRITE (NOUT,99999) VAL, FIG, FIG - VAL
99999 FORMAT (' Computed Integral = ', F10.5, '/', ' Exact Integral '&
 , '= ', F10.5, '/', ' Error '&
 , '= ', F10.6, '/')
 END

```

## Output

```

Computed Integral = 0.08594
Exact Integral = 0.08594
Error = 0.000000

```

---

## BSCPP

Converts a spline in B-spline representation to piecewise polynomial representation.

### Required Arguments

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length  $KORDER + NCOEF$  containing the knot sequence. (Input)  
 XKNOT must be nondecreasing.

**NCOEF** — Number of B-spline coefficients. (Input)

**BSCOEF** — Array of length  $NCOEF$  containing the B-spline coefficients. (Input)

**NPPCF** — Number of piecewise polynomial pieces. (Output)

NPPCF is always less than or equal to  $NCOEF - KORDER + 1$ .

**BREAK** — Array of length  $(NPPCF + 1)$  containing the breakpoints of the piecewise polynomial representation. (Output)

BREAK must be dimensioned at least  $NCOEF - KORDER + 2$ .

**PPCOEF** — Array of length  $KORDER * NPPCF$  containing the local coefficients of the polynomial pieces. (Output)

PPCOEF is treated internally as a matrix of size  $KORDER$  by  $NPPCF$ .

## FORTRAN 90 Interface

Generic:     CALL BSCPP (KORDER, XKNOT, NCOEF, BSCOEFF, NPPCF, BREAK, PPCOEF)

Specific:    The specific interface names are S\_BSCPP and D\_BSCPP.

## FORTRAN 77 Interface

Single:      CALL BSCPP (KORDER, XKNOT, NCOEF, BSCOEFF, NPPCF, BREAK, PPCOEF)

Double:     The double precision name is DBSCPP.

## Description

The routine BSCPP is based on the routine BSPLPP by de Boor (1978, page 140). This routine is used to convert a spline in B-spline representation to a piecewise polynomial (pp) representation which can then be evaluated more efficiently. There is some overhead in converting from the B-spline representation to the pp representation, but the conversion to pp form is recommended when 3 or more function values are needed per polynomial piece.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of B2CPP/DB2CPP. The reference is:

```
CALL B2CPP (KORDER, XKNOT, NCOEF, BSCOEFF, NPPCF, BREAK,
 PPCOEF, WK)
```

The additional argument is

**WK** — Work array of length  $(KORDER + 3) * KORDER$ .

2.    Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                  |
|---|---|------------------------------------------------------------------|
| 4 | 4 | Multiplicity of the knots cannot exceed the order of the spline. |
|---|---|------------------------------------------------------------------|

|   |   |                                  |
|---|---|----------------------------------|
| 4 | 5 | The knots must be nondecreasing. |
|---|---|----------------------------------|

## Example

For an example of the use of `BSCPP`, see [PPDER](#).

---

# PPVAL

This function evaluates a piecewise polynomial.

## Function Return Value

*PPVAL* — Value of the piecewise polynomial at  $x$ . (Output)

## Required Arguments

$X$  — Point at which the polynomial is to be evaluated. (Input)

*BREAK* — Array of length  $NINTV + 1$  containing the breakpoints of the piecewise polynomial representation. (Input)  
*BREAK* must be strictly increasing.

*PPCOEF* — Array of size  $KORDER * NINTV$  containing the local coefficients of the piecewise polynomial pieces. (Input)  
*PPCOEF* is treated internally as a matrix of size  $KORDER$  by  $NINTV$ .

## Optional Arguments

*KORDER* — Order of the polynomial. (Input)  
Default:  $KORDER = \text{size}(PPCOEF,1)$ .

*NINTV* — Number of polynomial pieces. (Input)  
Default:  $NINTV = \text{size}(PPCOEF,2)$ .

## FORTRAN 90 Interface

Generic: `PPVAL (X, BREAK, PPCOEF [, ...])`

Specific: The specific interface names are `S_PPVAL` and `D_PPVAL`.

## FORTRAN 77 Interface

Single: `PPVAL (X, KORDER, NINTV, BREAK, PPCOEF)`

Double: The double precision function name is `DPPVAL`.

## Description

The routine `PPVAL` evaluates a piecewise polynomial at a given point. This routine is a special case of the routine [PPDER](#), which evaluates the derivative of a piecewise polynomial. (The value of a piecewise polynomial is its zero-th derivative.)

The routine `PPDER` is based on the routine `PPVALU` in de Boor (1978, page 89).

### Example

In this example, a spline interpolant to a function  $f$  is computed using the IMSL routine `BSINT`. This routine represents the interpolant as a linear combination of B-splines. This representation is then converted to piecewise polynomial representation by calling the IMSL routine `BSCPP`. The piecewise polynomial is evaluated using `PPVAL`. These values are compared to the corresponding values of  $f$ .

```

 USE PPVAL_INT
 USE BSNK_INT
 USE BSCPP_INT
 USE BSINT_INT
 USE UMACH_INT

 IMPLICIT NONE
 INTEGER KORDER, NCOEF, NDATA, NKNOT
 PARAMETER (KORDER=4, NCOEF=20, NDATA=20, NKNOT=NDATA+KORDER)
!
 INTEGER I, NOUT, NPPCF
 REAL BREAK(NCOEF), BSCOEFF(NCOEF), EXP, F, FDATA(NDATA), &
 FLOAT, PPCOEFF(KORDER,NCOEF), S, X, XDATA(NDATA), &
 XKNOT(NKNOT)
 INTRINSIC EXP, FLOAT
!
 F(X) = X*EXP(X) Define function
!
 DO 30 I=1, NDATA Set up interpolation points
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
30 CONTINUE
!
 CALL BSNK(NDATA, XDATA, KORDER, XKNOT) Generate knot sequence
!
 CALL BSINT(NCOEF, XDATA, FDATA, KORDER, XKNOT, BSCOEFF)
 Compute the B-spline interpolant
!
 CALL BSCPP(KORDER, XKNOT, NCOEF, BSCOEFF, NPPCF, BREAK, PPCOEFF)
 Convert to piecewise polynomial
!
 CALL UMACH(2, NOUT) Get output unit number
!
 WRITE(NOUT,99999) Write heading
!
 DO 40 I=1, NDATA Print the interpolant on a uniform
 X = FLOAT(I-1)/FLOAT(NDATA-1) grid
!
 S = PPVAL(X,BREAK,PPCOEFF) Compute value of the piecewise
 WRITE(NOUT,'(2F12.3, E14.3)') X, S, F(X) - S
 polynomial
!
40 CONTINUE
99999 FORMAT (11X, 'X', 8X, 'S(X)', 7X, 'Error')
 END

```

## Output

| X     | S(X)  | Error      |
|-------|-------|------------|
| 0.000 | 0.000 | 0.000E+00  |
| 0.053 | 0.055 | -0.745E-08 |
| 0.105 | 0.117 | 0.000E+00  |
| 0.158 | 0.185 | 0.000E+00  |
| 0.211 | 0.260 | -0.298E-07 |
| 0.263 | 0.342 | 0.298E-07  |
| 0.316 | 0.433 | 0.000E+00  |
| 0.368 | 0.533 | 0.000E+00  |
| 0.421 | 0.642 | 0.000E+00  |
| 0.474 | 0.761 | 0.596E-07  |
| 0.526 | 0.891 | 0.000E+00  |
| 0.579 | 1.033 | 0.000E+00  |
| 0.632 | 1.188 | 0.000E+00  |
| 0.684 | 1.356 | 0.000E+00  |
| 0.737 | 1.540 | -0.119E-06 |
| 0.789 | 1.739 | 0.000E+00  |
| 0.842 | 1.955 | 0.000E+00  |
| 0.895 | 2.189 | 0.238E-06  |
| 0.947 | 2.443 | 0.238E-06  |
| 1.000 | 2.718 | 0.238E-06  |

---

## PPDER

This function evaluates the derivative of a piecewise polynomial.

### Function Return Value

**PPDER** — Value of the `IDERIV`-th derivative of the piecewise polynomial at `X`. (Output)

### Required Arguments

**X** — Point at which the polynomial is to be evaluated. (Input)

**BREAK** — Array of length `NINTV + 1` containing the breakpoints of the piecewise polynomial representation. (Input)  
`BREAK` must be strictly increasing.

**PPCOEF** — Array of size `KORDER * NINTV` containing the local coefficients of the piecewise polynomial pieces. (Input)  
`PPCOEF` is treated internally as a matrix of size `KORDER` by `NINTV`.

### Optional Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)  
In particular, `IDERIV = 0` returns the value of the polynomial.  
Default: `IDERIV = 1`.

**KORDER** — Order of the polynomial. (Input)  
Default: `KORDER = size (PPCOEF,1)`.

*NINTV* — Number of polynomial pieces. (Input)  
 Default:  $NINTV = \text{size}(\text{PPCOEF}, 2)$ .

### FORTRAN 90 Interface

Generic: `PPDER (X, BREAK, PPCOEF [, ...])`

Specific: The specific interface names are `S_PPDER` and `D_PPDER`.

### FORTRAN 77 Interface

Single: `PPDER (IDERIV, X, KORDER, NINTV, BREAK, PPCOEF)`

Double: The double precision function name is `DPPDER`.

### Description

The routine `PPDER` evaluates the derivative of a piecewise polynomial function  $f$  at a given point. This routine is based on the subroutine `PPVALU` by de Boor (1978, page 89). In particular, if the breakpoint sequence is stored in  $\xi$  (a vector of length  $N = NINTV + 1$ ), and if the coefficients of the piecewise polynomial representation are stored in  $\mathbf{c}$ , then the value of the  $j$ -th derivative of  $f$  at  $x$  in  $[\xi_i, \xi_{i+1})$  is

$$f^{(j)}(x) = \sum_{m=j}^{k-1} c_{m+1,i} \frac{(x - \xi_i)^{m-j}}{(m-j)!}$$

when  $j = 0$  to  $k - 1$  and zero otherwise. Notice that this representation forces the function to be right continuous. If  $x$  is less than  $\xi_1$ , then  $i$  is set to 1 in the above formula; if  $x$  is greater than or equal to  $\xi_N$ , then  $i$  is set to  $N - 1$ . This has the effect of extending the piecewise polynomial representation to the real axis by extrapolation of the first and last pieces.

### Example

In this example, a spline interpolant to a function  $f$  is computed using the IMSL routine `BSINT`. This routine represents the interpolant as a linear combination of B-splines. This representation is then converted to piecewise polynomial representation by calling the IMSL routine `BSCPP`. The piecewise polynomial's zero-th and first derivative are evaluated using `PPDER`. These values are compared to the corresponding values of  $f$ .

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, NCOEF, NDATA, NKNOT
PARAMETER (KORDER=4, NCOEF=20, NDATA=20, NKNOT=NDATA+KORDER)
!
INTEGER I, NOUT, NPPCF
REAL BREAK(NCOEF), BSCOEFF(NCOEF), DF, DS, EXP, F, &
```

```

 FDATA(NDATA), FLOAT, PPCOEF(KORDER,NCOEF), S,&
 X, XDATA(NDATA), XKNOT(NKNOT)
INTRINSIC EXP, FLOAT
!
F(X) = X*EXP(X)
DF(X) = (X+1.)*EXP(X)
!
 Set up interpolation points
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
 Generate knot sequence
CALL BSNAK (NDATA, XDATA, KORDER, XKNOT)
!
 Compute the B-spline interpolant
CALL BSINT (NCOEF, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
!
 Convert to piecewise polynomial
CALL BSCPP (KORDER, XKNOT, NCOEF, BSCOEF, NPPCF, BREAK, PPCOEF)
!
 Get output unit number
CALL UMACH (2, NOUT)
!
 Write heading
WRITE (NOUT,99999)
!
 Print the interpolant on a uniform
! grid
DO 20 I=1, NDATA
 X = FLOAT(I-1)/FLOAT(NDATA-1)
!
 Compute value of the piecewise
! polynomial
 S = PPDER(X,BREAK,PPCOEF, IDERIV=0, NINTV=NPPCF)
!
 Compute derivative of the piecewise
! polynomial
 DS = PPDER(X,BREAK,PPCOEF, IDERIV=1, NINTV=NPPCF)
 WRITE (NOUT,'(2F12.3,F12.6,F12.3,F12.6)') X, S, F(X) - S, DS,&
 DF(X), DS
20 CONTINUE
99999 FORMAT (11X, 'X', 8X, 'S(X)', 7X, 'Error', 7X, 'S''(X)', 7X,&
'Error')
END

```

## Output

| X     | S(X)  | Error    | S'(X) | Error     |
|-------|-------|----------|-------|-----------|
| 0.000 | 0.000 | 0.000000 | 1.000 | -0.000112 |
| 0.053 | 0.055 | 0.000000 | 1.109 | 0.000030  |
| 0.105 | 0.117 | 0.000000 | 1.228 | -0.000008 |
| 0.158 | 0.185 | 0.000000 | 1.356 | 0.000002  |
| 0.211 | 0.260 | 0.000000 | 1.494 | 0.000000  |
| 0.263 | 0.342 | 0.000000 | 1.643 | 0.000000  |
| 0.316 | 0.433 | 0.000000 | 1.804 | -0.000001 |
| 0.368 | 0.533 | 0.000000 | 1.978 | 0.000002  |
| 0.421 | 0.642 | 0.000000 | 2.165 | 0.000001  |
| 0.474 | 0.761 | 0.000000 | 2.367 | 0.000000  |
| 0.526 | 0.891 | 0.000000 | 2.584 | -0.000001 |
| 0.579 | 1.033 | 0.000000 | 2.817 | 0.000001  |
| 0.632 | 1.188 | 0.000000 | 3.068 | 0.000001  |
| 0.684 | 1.356 | 0.000000 | 3.338 | 0.000001  |
| 0.737 | 1.540 | 0.000000 | 3.629 | 0.000001  |
| 0.789 | 1.739 | 0.000000 | 3.941 | 0.000000  |

|       |       |          |       |           |
|-------|-------|----------|-------|-----------|
| 0.842 | 1.955 | 0.000000 | 4.276 | -0.000006 |
| 0.895 | 2.189 | 0.000000 | 4.636 | 0.000024  |
| 0.947 | 2.443 | 0.000000 | 5.022 | -0.000090 |
| 1.000 | 2.718 | 0.000000 | 5.436 | 0.000341  |

---

## PP1GD

Evaluates the derivative of a piecewise polynomial on a grid.

### Required Arguments

**XVEC** — Array of length  $N$  containing the points at which the piecewise polynomial is to be evaluated. (Input)

The points in **XVEC** should be strictly increasing.

**BREAK** — Array of length  $N_{INTV} + 1$  containing the breakpoints for the piecewise polynomial representation. (Input)

**BREAK** must be strictly increasing.

**PPCOEF** — Matrix of size  $KORDER$  by  $N_{INTV}$  containing the local coefficients of the polynomial pieces. (Input)

**VALUE** — Array of length  $N$  containing the values of the  $IDERIV$ -th derivative of the piecewise polynomial at the points in **XVEC**. (Output)

### Optional Arguments

**IDERIV** — Order of the derivative to be evaluated. (Input)

In particular,  $IDERIV = 0$  returns the values of the piecewise polynomial.

Default:  $IDERIV = 1$ .

**N** — Length of vector **XVEC**. (Input)

Default:  $N = \text{size}(\text{XVEC}, 1)$ .

**KORDER** — Order of the polynomial. (Input)

Default:  $KORDER = \text{size}(\text{PPCOEF}, 1)$ .

**NINTV** — Number of polynomial pieces. (Input)

Default:  $N_{INTV} = \text{size}(\text{PPCOEF}, 2)$ .

### FORTRAN 90 Interface

Generic:    CALL PP1GD (XVEC, BREAK, PPCOEF, VALUE [, ...])

Specific:    The specific interface names are S\_PP1GD and D\_PP1GD.

## FORTRAN 77 Interface

Single:      CALL PP1GD (IDERIV, N, XVEC, KORDER, NINTV, BREAK, PPCOEF, VALUE)

Double:      The double precision name is DPP1GD.

## Description

The routine PP1GD evaluates a piecewise polynomial function  $f$  (or its derivative) at a vector of points. That is, given a vector  $x$  of length  $n$  satisfying  $x_i < x_{i+1}$  for  $i = 1, \dots, n - 1$ , a derivative value  $j$ , and a piecewise polynomial function  $f$  that is represented by a breakpoint sequence and coefficient matrix this routine returns the values

$$f^{(j)}(x_i) \quad i = 1, \dots, n$$

in the array VALUE. The functionality of this routine is the same as that of PPDER called in a loop, however PP1GD is much more efficient.

## Comments

1.      Workspace may be explicitly provided, if desired, by use of P21GD/DP21GD. The reference is:

```
CALL P21GD (IDERIV, N, XVEC, KORDER, NINTV, BREAK, PPCOEF,
 VALUE, IWK, WORK1, WORK2)
```

The additional arguments are as follows:

**IWK** — Array of length N.

**WORK1** — Array of length N.

**WORK2** — Array of length N.

2.      Informational error

Type   Code

4      4      The points in XVEC must be strictly increasing.

## Example

To illustrate the use of PP1GD, we modify the example program for PPDER. In this example, a piecewise polynomial interpolant to  $F$  is computed. The values of this polynomial are then compared with the exact function values. The routine PP1GD is based on the routine PPVALU in de Boor (1978, page 89).

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, N, NCOEF, NDATA, NKNOT
PARAMETER (KORDER=4, N=20, NCOEF=20, NDATA=20, &
```

```

 NKNOT=NDATA+KORDER)
!
INTEGER I, NINTV, NOUT, NPPCF
REAL BREAK(NCOEF), BSCOEF(NCOEF), DF, EXP, F,&
 FDATA(NDATA), FLOAT, PPCOEF(KORDER,NCOEF), VALUE1(N),&
 VALUE2(N), X, XDATA(NDATA), XKNOT(NKNOT), XVEC(N)
INTRINSIC EXP, FLOAT
!
F(X) = X*EXP(X)
DF(X) = (X+1.)*EXP(X)
!
 Set up interpolation points
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
 Generate knot sequence
CALL BSNK (NDATA, XDATA, KORDER, XKNOT)
!
 Compute the B-spline interpolant
CALL BSINT (NCOEF, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
!
 Convert to piecewise polynomial
CALL BSCPP (KORDER, XKNOT, NCOEF, BSCOEF, NPPCF, BREAK, PPCOEF)
!
 Compute evaluation points
DO 20 I=1, N
 XVEC(I) = FLOAT(I-1)/FLOAT(N-1)
20 CONTINUE
!
 Compute values of the piecewise
! polynomial
NINTV = NPPCF
CALL PP1GD (XVEC, BREAK, PPCOEF, VALUE1, IDERIV=0, NINTV=NINTV)
!
 Compute the values of the first
! derivative of the piecewise
! polynomial
CALL PP1GD (XVEC, BREAK, PPCOEF, VALUE2, IDERIV=1, NINTV=NINTV)
!
 Get output unit number
CALL UMACH (2, NOUT)
!
 Write heading
WRITE (NOUT,99998)
!
 Print the results on a uniform
! grid
DO 30 I=1, N
 WRITE (NOUT,99999) XVEC(I), VALUE1(I), F(XVEC(I)) - VALUE1(I)&
 , VALUE2(I), DF(XVEC(I)) - VALUE2(I)
30 CONTINUE
99998 FORMAT (11X, 'X', 8X, 'S(X)', 7X, 'Error', 7X, 'S'(X)', 7X,&
 'Error')
99999 FORMAT (' ', 2F12.3, F12.6, F12.3, F12.6)
END

```

## Output

| X     | S(X)  | Error    | S'(X) | Error     |
|-------|-------|----------|-------|-----------|
| 0.000 | 0.000 | 0.000000 | 1.000 | -0.000112 |
| 0.053 | 0.055 | 0.000000 | 1.109 | 0.000030  |
| 0.105 | 0.117 | 0.000000 | 1.228 | -0.000008 |
| 0.158 | 0.185 | 0.000000 | 1.356 | 0.000002  |

|       |       |          |       |           |
|-------|-------|----------|-------|-----------|
| 0.211 | 0.260 | 0.000000 | 1.494 | 0.000000  |
| 0.263 | 0.342 | 0.000000 | 1.643 | 0.000000  |
| 0.316 | 0.433 | 0.000000 | 1.804 | -0.000001 |
| 0.368 | 0.533 | 0.000000 | 1.978 | 0.000002  |
| 0.421 | 0.642 | 0.000000 | 2.165 | 0.000001  |
| 0.474 | 0.761 | 0.000000 | 2.367 | 0.000000  |
| 0.526 | 0.891 | 0.000000 | 2.584 | -0.000001 |
| 0.579 | 1.033 | 0.000000 | 2.817 | 0.000001  |
| 0.632 | 1.188 | 0.000000 | 3.068 | 0.000001  |
| 0.684 | 1.356 | 0.000000 | 3.338 | 0.000001  |
| 0.737 | 1.540 | 0.000000 | 3.629 | 0.000001  |
| 0.789 | 1.739 | 0.000000 | 3.941 | 0.000000  |
| 0.842 | 1.955 | 0.000000 | 4.276 | -0.000006 |
| 0.895 | 2.189 | 0.000000 | 4.636 | 0.000024  |
| 0.947 | 2.443 | 0.000000 | 5.022 | -0.000090 |
| 1.000 | 2.718 | 0.000000 | 5.436 | 0.000341  |

---

## PPITG

This function evaluates the integral of a piecewise polynomial.

### Function Return Value

*PPITG* — Value of the integral from A to B of the piecewise polynomial. (Output)

### Required Arguments

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. (Input)

*BREAK* — Array of length  $NINTV + 1$  containing the breakpoints for the piecewise polynomial. (Input)  
*BREAK* must be strictly increasing.

*PPCOEF* — Array of size  $KORDER * NINTV$  containing the local coefficients of the piecewise polynomial pieces. (Input)  
*PPCOEF* is treated internally as a matrix of size *KORDER* by *NINTV*.

### Optional Arguments

*KORDER* — Order of the polynomial. (Input)  
 Default: *KORDER* = size (*PPCOEF*,1).

*NINTV* — Number of piecewise polynomial pieces. (Input)  
 Default: *NINTV* = size (*PPCOEF*,2).

### FORTRAN 90 Interface

Generic: `PPITG (A, B, BREAK, PPCOEF [, ...])`

Specific: The specific interface names are S\_PP1TG and D\_PP1TG.

## FORTRAN 77 Interface

Single: PP1TG (A, B, KORDER, NINTV, BREAK, PPCOEF)

Double: The double precision function name is DPP1TG.

## Description

The routine PP1TG evaluates the integral of a piecewise polynomial over an interval.

## Example

In this example, we compute a quadratic spline interpolant to the function  $x^2$  using the IMSL routine BSINT. We then evaluate the integral of the spline interpolant over the intervals  $[0, 1/2]$  and  $[0, 2]$ . The interpolant reproduces  $x^2$ , and hence, the values of the integrals are  $1/24$  and  $8/3$ , respectively.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, NDATA, NKNOT
PARAMETER (KORDER=3, NDATA=10, NKNOT=NDATA+KORDER)
!
INTEGER I, NOUT, NPPCF
REAL A, B, BREAK (NDATA), BSCOE (NDATA), EXACT, F, &
 FDATA (NDATA), FI, FLOAT, PPCOEF (KORDER, NDATA), &
 VALUE, X, XDATA (NDATA), XKNOT (NKNOT)
INTRINSIC FLOAT
!
F(X) = X*X
FI(X) = X*X*X/3.0
!
! Set up interpolation points
DO 10 I=1, NDATA
 XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Generate knot sequence
CALL BSNK (NDATA, XDATA, KORDER, XKNOT)
!
! Interpolate
CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOE)
!
! Convert to piecewise polynomial
CALL BSCPP (KORDER, XKNOT, NDATA, BSCOE, NPPCF, BREAK, PPCOEF)
!
! Compute the integral of F over
! [0.0,0.5]
A = 0.0
B = 0.5
VALUE = PP1TG (A, B, BREAK, PPCOEF, NINTV=NPPCF)
EXACT = FI (B) - FI (A)
!
! Get output unit number
CALL UMACH (2, NOUT)
```

```

! Print the result
WRITE (NOUT,99999) A, B, VALUE, EXACT, EXACT - VALUE
! Compute the integral of F over
! [0.0,2.0]
A = 0.0
B = 2.0
VALUE = PPITG(A,B,BREAK,PPCOEF,NINTV=NPPCF)
EXACT = FI(B) - FI(A)
!
! Print the result
WRITE (NOUT,99999) A, B, VALUE, EXACT, EXACT - VALUE
99999 FORMAT (' On the closed interval (' , F3.1, ', ', F3.1, &
 ') we have :', /, 1X, 'Computed Integral = ', F10.5, /, &
 1X, 'Exact Integral = ', F10.5, /, 1X, 'Error ' &
 ', ' = ', F10.6, /, /)
!
END

```

## Output

```

On the closed interval (0.0,0.5) we have :
Computed Integral = 0.04167
Exact Integral = 0.04167
Error = 0.000000

```

```

On the closed interval (0.0,2.0) we have :
Computed Integral = 2.66667
Exact Integral = 2.66667
Error = 0.000001

```

---

## QDVAL

This function evaluates a function defined on a set of points using quadratic interpolation.

### Function Return Value

*QDVAL* — Value of the quadratic interpolant at  $x$ . (Output)

### Required Arguments

$X$  — Coordinate of the point at which the function is to be evaluated. (Input)

*XDATA* — Array of length *NDATA* containing the location of the data points. (Input)  
*XDATA* must be strictly increasing.

*FDATA* — Array of length *NDATA* containing the function values. (Input)  
*FDATA(I)* is the value of the function at *XDATA(I)*.

### Optional Arguments

*NDATA* — Number of data points. (Input)  
*NDATA* must be at least 3.  
Default: *NDATA* = size(*XDATA*,1).

**CHECK**— Logical variable that is `.TRUE.` if checking of `XDATA` is required or `.FALSE.` if checking is not required. (Input)  
 Default: `CHECK = .TRUE.`

### FORTRAN 90 Interface

Generic: `QDVAL (X, XDATA, FDATA [, ...])`

Specific: The specific interface names are `S_QDVAL` and `D_QDVAL`.

### FORTRAN 77 Interface

Single: `QDVAL (X, NDATA, XDATA, FDATA, CHECK)`

Double: The double precision name is `DQDVAL`.

### Description

The function `QDVAL` interpolates a table of values, using quadratic polynomials, returning an approximation to the tabulated function. Let  $(x_i, f_i)$  for  $i = 1, \dots, n$  be the tabular data. Given a number  $x$  at which an interpolated value is desired, we first find the nearest interior grid point  $x_i$ . A quadratic interpolant  $q$  is then formed using the three points  $(x_{i-1}, f_{i-1})$ ,  $(x_i, f_i)$ , and  $(x_{i+1}, f_{i+1})$ . The number returned by `QDVAL` is  $q(x)$ .

### Comments

Informational error

Type Code

4 3 The `XDATA` values must be strictly increasing.

### Example

In this example, the value of  $\sin x$  is approximated at  $\pi/4$  by using `QDVAL` on a table of 33 equally spaced values.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=33)
!
INTEGER I, NOUT
REAL F, FDATA(NDATA), H, PI, QT, SIN, X, &
 XDATA(NDATA)
INTRINSIC SIN
! Define function
F(X) = SIN(X)
! Generate data points

```

```

XDATA(1) = 0.0
FDATA(1) = F(XDATA(1))
H = 1.0/32.0
DO 10 I=2, NDATA
 XDATA(I) = XDATA(I-1) + H
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Get value of PI and set X
 PI = CONST('PI')
 X = PI/4.0
!
! Evaluate at PI/4
 QT = QDVAL(X,XDATA,FDATA)
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Print results
 WRITE (NOUT,99999) X, F(X), QT, (F(X)-QT)
!
99999 FORMAT (15X, 'X', 6X, 'F(X)', 6X, 'QDVAL', 5X, 'ERROR', //, 6X,&
 4F10.3, /)
END

```

### Output

| X     | F(X)  | QDVAL | ERROR |
|-------|-------|-------|-------|
| 0.785 | 0.707 | 0.707 | 0.000 |

---

## QDDER

This function evaluates the derivative of a function defined on a set of points using quadratic interpolation.

### Function Return Value

**QDDER** — Value of the *IDERIV*-th derivative of the quadratic interpolant at *X*. (Output)

### Required Arguments

**IDERIV** — Order of the derivative. (Input)

**X** — Coordinate of the point at which the function is to be evaluated. (Input)

**XDATA** — Array of length *NDATA* containing the location of the data points. (Input) *XDATA* must be strictly increasing.

**FDATA** — Array of length *NDATA* containing the function values. (Input)  
*FDATA(I)* is the value of the function at *XDATA(I)*.

## Optional Arguments

**NDATA** — Number of data points. (Input)

NDATA must be at least three.

Default: NDATA = size (XDATA,1).

**CHECK** — Logical variable that is `.TRUE.` if checking of XDATA is required or `.FALSE.` if checking is not required. (Input)

Default: CHECK = `.TRUE.`.

## FORTRAN 90 Interface

Generic: QDDER (IDERIV, X, XDATA, FDATA [, ...])

Specific: The specific interface names are `S_QDVAL` and `D_QDVAL`.

## FORTRAN 77 Interface

Single: QDDER (IDERIV, X, NDATA, XDATA, FDATA, CHECK)

Double: The double precision function name is `DQDVAL`.

## Description

The function QDDER interpolates a table of values, using quadratic polynomials, returning an approximation to the derivative of the tabulated function. Let  $(x_i, f_i)$  for  $i = 1, \dots, n$  be the tabular data. Given a number  $x$  at which an interpolated value is desired, we first find the nearest interior grid point  $x_i$ . A quadratic interpolant  $q$  is then formed using the three points  $(x_{i-1}, f_{i-1})$ ,  $(x_i, f_i)$ , and  $(x_{i+1}, f_{i+1})$ . The number returned by QDDER is  $q^{(j)}(x)$ , where  $j = \text{IDERIV}$ .

## Comments

1. Informational error

Type Code

4 3 The XDATA values must be strictly increasing.

2. Because quadratic interpolation is used, if the order of the derivative is greater than two, then the returned value is zero.

## Example

In this example, the value of  $\sin x$  and its derivatives are approximated at  $\pi/4$  by using QDDER on a table of 33 equally spaced values.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
```

```

 INTEGER NDATA
 PARAMETER (NDATA=33)
!
 INTEGER I, IDERIV, NOUT
 REAL COS, F, F1, F2, FDATA(NDATA), H, PI, &
 QT, SIN, X, XDATA(NDATA)
 LOGICAL CHECK
 INTRINSIC COS, SIN
!
 Define function and derivatives
 F(X) = SIN(X)
 F1(X) = COS(X)
 F2(X) = -SIN(X)
!
 Generate data points
 XDATA(1) = 0.0
 FDATA(1) = F(XDATA(1))
 H = 1.0/32.0
 DO 10 I=2, NDATA
 XDATA(I) = XDATA(I-1) + H
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
 Get value of PI and set X
 PI = CONST('PI')
 X = PI/4.0
!
 Check XDATA
 CHECK = .TRUE.
!
 Get output unit number
 CALL UMACH (2, NOUT)
!
 Write heading
 WRITE (NOUT,99998)
!
 Evaluate quadratic at PI/4
 IDERIV = 0
 QT = QDDER(IDERIV,X,XDATA,FDATA, CHECK=CHECK)
 WRITE (NOUT,99999) X, IDERIV, F(X), QT, (F(X)-QT)
 CHECK = .FALSE.
!
 Evaluate first derivative at PI/4
 IDERIV = 1
 QT = QDDER(IDERIV,X,XDATA,FDATA)
 WRITE (NOUT,99999) X, IDERIV, F1(X), QT, (F1(X)-QT)
!
 Evaluate second derivative at PI/4
 IDERIV = 2
 QT = QDDER(IDERIV,X,XDATA,FDATA, CHECK=CHECK)
 WRITE (NOUT,99999) X, IDERIV, F2(X), QT, (F2(X)-QT)
!
99998 FORMAT (33X, 'IDER', /, 15X, 'X', 6X, 'IDER', 6X, 'F (X)', &
 5X, 'QDDER', 6X, 'ERROR', //)
99999 FORMAT (7X, F10.3, I8, 3F12.3/)
 END

```

## Output

| X     | IDER | IDER  |       |       |
|-------|------|-------|-------|-------|
|       |      | F     | (X)   | QDDER |
| 0.785 | 0    | 0.707 | 0.707 | 0.000 |
| 0.785 | 1    | 0.707 | 0.707 | 0.000 |

0.785          2          -0.707          -0.704          -0.003

---

## QD2VL

This function evaluates a function defined on a rectangular grid using quadratic interpolation.

### Function Return Value

*QD2VL* — Value of the function at (x, y). (Output)

### Required Arguments

*X* — x-coordinate of the point at which the function is to be evaluated. (Input)

*Y* — y-coordinate of the point at which the function is to be evaluated. (Input)

*XDATA* — Array of length *NXDATA* containing the location of the data points in the x-direction. (Input)  
*XDATA* must be increasing.

*YDATA* — Array of length *NYDATA* containing the location of the data points in the y-direction. (Input)  
*YDATA* must be increasing.

*FDATA* — Array of size *NXDATA* by *NYDATA* containing function values. (Input)  
*FDATA* (I, J) is the value of the function at (*XDATA* (I), *YDATA*(J)).

### Optional Arguments

*NXDATA* — Number of data points in the x-direction. (Input)  
*NXDATA* must be at least three.  
Default: *NXDATA* = size (*XDATA*,1).

*NYDATA* — Number of data points in the y-direction. (Input)  
*NYDATA* must be at least three.  
Default: *NYDATA* = size (*YDATA*,1).

*LDF* — Leading dimension of *FDATA* exactly as specified in the dimension statement of the calling program. (Input)  
*LDF* must be at least as large as *NXDATA*.  
Default: *LDF* = size (*FDATA*,1).

*CHECK* — Logical variable that is `.TRUE.` if checking of *XDATA* and *YDATA* is required or `.FALSE.` if checking is not required. (Input)  
Default: *CHECK* = `.TRUE.`

## FORTRAN 90 Interface

Generic: QD2VL (X, Y, XDATA, YDATA, FDATA [, ...])

Specific: The specific interface names are S\_QD2VL and D\_QD2VL.

## FORTRAN 77 Interface

Single: QD2VL (X, Y, NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF, CHECK)

Double: The double precision function name is DQD2VL.

## Description

The function QD2VL interpolates a table of values, using quadratic polynomials, returning an approximation to the tabulated function. Let  $(x_i, y_j, f_{ij})$  for  $i = 1, \dots, n_x$  and  $j = 1, \dots, n_y$  be the tabular data. Given a point  $(x, y)$  at which an interpolated value is desired, we first find the nearest interior grid point  $(x_i, y_j)$ . A bivariate quadratic interpolant  $q$  is then formed using six points near  $(x, y)$ . Five of the six points are  $(x_i, y_j)$ ,  $(x_{i\pm 1}, y_j)$ , and  $(x_i, y_{j\pm 1})$ . The sixth point is the nearest point to  $(x, y)$  of the grid points  $(x_{i\pm 1}, y_{j\pm 1})$ . The value  $q(x, y)$  is returned by QD2VL.

## Comments

Informational errors

Type Code

4 6 The XDATA values must be strictly increasing.

4 7 The YDATA values must be strictly increasing.

## Example

In this example, the value of  $\sin(x + y)$  at  $x = y = \pi/4$  is approximated by using QDVAL on a table of size  $21 \times 42$  equally spaced values on the unit square.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDF, NXDATA, NYDATA
PARAMETER (NXDATA=21, NYDATA=42, LDF=NXDATA)
!
INTEGER I, J, NOUT
REAL F, FDATA(LDF,NYDATA), FLOAT, PI, Q, &
 SIN, X, XDATA(NXDATA), Y, YDATA(NYDATA)
INTRINSIC FLOAT, SIN
!
F(X,Y) = SIN(X+Y) Define function
!
DO 10 I=1, NXDATA Set up X-grid
 XDATA(I) = FLOAT(I-1)/FLOAT(NXDATA-1)
10 CONTINUE
!
Set up Y-grid
```

```

 DO 20 I=1, NYDATA
 YDATA(I) = FLOAT(I-1)/FLOAT(NYDATA-1)
20 CONTINUE
!
! Evaluate function on grid
 DO 30 I=1, NXDATA
 DO 30 J=1, NYDATA
 FDATA(I,J) = F(XDATA(I),YDATA(J))
30 CONTINUE
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Write heading
 WRITE (NOUT,99999)
!
! Get value for PI and set X and Y
 PI = CONST('PI')
 X = PI/4.0
 Y = PI/4.0
!
! Evaluate quadratic at (X,Y)
 Q = QD2VL(X,Y,XDATA,YDATA,FDATA)
!
! Print results
 WRITE (NOUT,'(5F12.4)') X, Y, F(X,Y), Q, (Q-F(X,Y))
99999 FORMAT (10X, 'X', 11X, 'Y', 7X, 'F(X,Y)', 7X, 'QD2VL', 9X,&
 'DIF')
 END

```

## Output

| X      | Y      | F(X,Y) | QD2VL  | DIF    |
|--------|--------|--------|--------|--------|
| 0.7854 | 0.7854 | 1.0000 | 1.0000 | 0.0000 |

---

## QD2DR

This function evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.

### Function Return Value

*QD2DR* — Value of the (*IXDER*, *IYDER*) derivative of the function at (*X*, *Y*). (Output)

### Required Arguments

*IXDER* — Order of the *x*-derivative. (Input)

*IYDER* — Order of the *y*-derivative. (Input)

*X* — *x*-coordinate of the point at which the function is to be evaluated. (Input)

*Y* — *y*-coordinate of the point at which the function is to be evaluated. (Input)

*XDATA* — Array of length *NXDATA* containing the location of the data points in the *x*-direction. (Input)  
*XDATA* must be increasing.

**YDATA** — Array of length NYDATA containing the location of the data points in the y-direction. (Input)  
YDATA must be increasing.

**FDATA** — Array of size NXDATA by NYDATA containing function values. (Input)  
FDATA(I, J) is the value of the function at (XDATA(I), YDATA(J)).

### Optional Arguments

**NXDATA** — Number of data points in the x-direction. (Input)  
NXDATA must be at least three.  
Default: NXDATA = size (XDATA,1).

**NYDATA** — Number of data points in the y-direction. (Input)  
NYDATA must be at least three.  
Default: NYDATA = size (YDATA,1).

**LDF** — Leading dimension of FDATA exactly as specified in the dimension statement of the calling program. (Input)  
LDF must be at least as large as NXDATA.  
Default: LDF = size (FDATA,1).

**CHECK** — Logical variable that is .TRUE. if checking of XDATA and YDATA is required or .FALSE. if checking is not required. (Input)  
Default: CHECK = .TRUE.

### FORTRAN 90 Interface

Generic: QD2DR (IXDER, IYDER, X, Y, XDATA, YDATA, FDATA [, ...])

Specific: The specific interface names are S\_QD2DR and D\_QD2DR.

### FORTRAN 77 Interface

Single: QD2DR (IXDER, IYDER, X, Y, NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF, CHECK)

Double: The double precision function name is DQD2DR.

### Description

The function QD2DR interpolates a table of values, using quadratic polynomials, returning an approximation to the tabulated function. Let  $(x_i, y_j, f_{ij})$  for  $i = 1, \dots, n_x$  and  $j = 1, \dots, n_y$  be the tabular data. Given a point  $(x, y)$  at which an interpolated value is desired, we first find the nearest interior grid point  $(x_i, y_j)$ . A bivariate quadratic interpolant  $q$  is then formed using six points near  $(x, y)$ . Five of the six points are  $(x_i, y_j)$ ,  $(x_{i\pm 1}, y_j)$ , and  $(x_i, y_{j\pm 1})$ . The sixth point is the nearest point to  $(x, y)$  of the grid points  $(x_{i\pm 1}, y_{j\pm 1})$ . The value  $q^{(p, r)}(x, y)$  is returned by QD2DR, where  $p = \text{IXDER}$  and  $r = \text{IYDER}$ .

## Comments

1. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                               |
|---|---|-----------------------------------------------|
| 4 | 6 | The XDATA values must be strictly increasing. |
|---|---|-----------------------------------------------|

|   |   |                                               |
|---|---|-----------------------------------------------|
| 4 | 7 | The YDATA values must be strictly increasing. |
|---|---|-----------------------------------------------|

2. Because quadratic interpolation is used, if the order of any derivative is greater than two, then the returned value is zero.

## Example

In this example, the partial derivatives of  $\sin(x + y)$  at  $x = y = \pi/3$  are approximated by using QD2DR on a table of size  $21 \times 42$  equally spaced values on the rectangle  $[0, 2] \times [0, 2]$ .

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDF, NXDATA, NYDATA
PARAMETER (NXDATA=21, NYDATA=42, LDF=NXDATA)
!
INTEGER I, IXDER, IYDER, J, NOUT
REAL F, FDATA(LDF,NYDATA), FLOAT, FU, FUNC, PI, Q, &
 SIN, X, XDATA(NXDATA), Y, YDATA(NYDATA)
INTRINSIC FLOAT, SIN
EXTERNAL FUNC
!
! Define function
F(X,Y) = SIN(X+Y)
!
! Set up X-grid
DO 10 I=1, NXDATA
 XDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NXDATA-1))
10 CONTINUE
!
! Set up Y-grid
DO 20 I=1, NYDATA
 YDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NYDATA-1))
20 CONTINUE
!
! Evaluate function on grid
DO 30 I=1, NXDATA
 DO 30 J=1, NYDATA
 FDATA(I,J) = F(XDATA(I),YDATA(J))
30 CONTINUE
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99998)
!
! Check XDATA and YDATA
! Get value for PI and set X and Y
PI = CONST('PI')
X = PI/3.0
Y = PI/3.0
!
! Evaluate and print the function
```

```

! and its derivatives at X=PI/3 and
! Y=PI/3.
DO 40 IXDER=0, 1
 DO 40 IYDER=0, 1
 Q = QD2DR (IXDER, IYDER, X, Y, XDATA, YDATA, FDATA)
 FU = FUNC (IXDER, IYDER, X, Y)
 WRITE (NOUT, 99999) X, Y, IXDER, IYDER, FU, Q, (FU-Q)
 40 CONTINUE
!
99998 FORMAT (32X, '(IDX, IDY)', /, 8X, 'X', 8X, 'Y', 3X, 'IDX', 2X, &
'IDY', 3X, 'F' (X, Y)', 3X, 'QD2DR', 6X, 'ERROR')
99999 FORMAT (2F9.4, 2I5, 3X, F9.4, 2X, 2F11.4)
END
REAL FUNCTION FUNC (IX, IY, X, Y)
INTEGER IX, IY
REAL X, Y
!
REAL COS, SIN
INTRINSIC COS, SIN
!
IF (IX.EQ.0 .AND. IY.EQ.0) THEN
! Define (0,0) derivative
 FUNC = SIN(X+Y)
ELSE IF (IX.EQ.0 .AND. IY.EQ.1) THEN
! Define (0,1) derivative
 FUNC = COS(X+Y)
ELSE IF (IX.EQ.1 .AND. IY.EQ.0) THEN
! Define (1,0) derivative
 FUNC = COS(X+Y)
ELSE IF (IX.EQ.1 .AND. IY.EQ.1) THEN
! Define (1,1) derivative
 FUNC = -SIN(X+Y)
ELSE
 FUNC = 0.0
END IF
RETURN
END

```

## Output

| X      | Y      | IDX | IDY | (IDX, IDY)<br>F (X, Y) | QD2DR   | ERROR   |
|--------|--------|-----|-----|------------------------|---------|---------|
| 1.0472 | 1.0472 | 0   | 0   | 0.8660                 | 0.8661  | -0.0001 |
| 1.0472 | 1.0472 | 0   | 1   | -0.5000                | -0.4993 | -0.0007 |
| 1.0472 | 1.0472 | 1   | 0   | -0.5000                | -0.4995 | -0.0005 |
| 1.0472 | 1.0472 | 1   | 1   | -0.8660                | -0.8634 | -0.0026 |

---

## QD3VL

This function evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.

### Function Return Value

*QD3VL* — Value of the function at (X, Y, Z). (Output)

## Required Arguments

***X*** —  $x$ -coordinate of the point at which the function is to be evaluated. (Input)

***Y*** —  $y$ -coordinate of the point at which the function is to be evaluated. (Input)

***Z*** —  $z$ -coordinate of the point at which the function is to be evaluated. (Input)

***XDATA*** — Array of length  $NXDATA$  containing the location of the data points in the  $x$ -direction. (Input)  
 $XDATA$  must be increasing.

***YDATA*** — Array of length  $NYDATA$  containing the location of the data points in the  $y$ -direction. (Input)  
 $YDATA$  must be increasing.

***ZDATA*** — Array of length  $NZDATA$  containing the location of the data points in the  $z$ -direction. (Input)  
 $ZDATA$  must be increasing.

***FDATA*** — Array of size  $NXDATA$  by  $NYDATA$  by  $NZDATA$  containing function values. (Input)  
 $FDATA(I, J, K)$  is the value of the function at  $(XDATA(I), YDATA(J), ZDATA(K))$ .

## Optional Arguments

***NXDATA*** — Number of data points in the  $x$ -direction. (Input)  
 $NXDATA$  must be at least three.  
Default:  $NXDATA = \text{size}(XDATA,1)$ .

***NYDATA*** — Number of data points in the  $y$ -direction. (Input)  
 $NYDATA$  must be at least three.  
Default:  $NYDATA = \text{size}(YDATA,1)$ .

***NZDATA*** — Number of data points in the  $z$ -direction. (Input)  
 $NZDATA$  must be at least three.  
Default:  $NZDATA = \text{size}(ZDATA,1)$ .

***LDF*** — Leading dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
 $LDF$  must be at least as large as  $NXDATA$ .  
Default:  $LDF = \text{size}(FDATA,1)$ .

***MDF*** — Middle (second) dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
 $MDF$  must be at least as large as  $NYDATA$ .  
Default:  $MDF = \text{size}(FDATA,2)$ .

**CHECK** — Logical variable that is `.TRUE.` if checking of `XDATA`, `YDATA`, and `ZDATA` is required or `.FALSE.` if checking is not required. (Input)  
 Default: `CHECK = .TRUE.`

### FORTRAN 90 Interface

Generic: `QD3VL (X, Y, Z, XDATA, YDATA, ZDATA, FDATA [, ...])`

Specific: The specific interface names are `S_QD3VL` and `D_QD3VL`.

### FORTRAN 77 Interface

Single: `QD3VL (X, Y, Z, NXDATA, XDATA, NYDATA, YDATA, NZDATA, ZDATA, FDATA, LDF, MDF, CHECK)`

Double: The double precision function name is `DQD3VL`.

### Description

The function `QD3VL` interpolates a table of values, using quadratic polynomials, returning an approximation to the tabulated function. Let  $(x_i, y_j, z_k, f_{ijk})$  for  $i = 1, \dots, n_x, j = 1, \dots, n_y,$  and  $k = 1, \dots, n_z$  be the tabular data. Given a point  $(x, y, z)$  at which an interpolated value is desired, we first find the nearest interior grid point  $(x_i, y_j, z_k)$ . A trivariate quadratic interpolant  $q$  is then formed. Ten points are needed for this purpose. Seven points have the form

$$(x_i, y_j, z_k), (x_{i\pm 1}, y_j, z_k), (x_i, y_{j\pm 1}, z_k) \text{ and } (x_i, y_j, z_{k\pm 1})$$

The last three points are drawn from the vertices of the octant containing  $(x, y, z)$ . There are four of these vertices remaining, and we choose to exclude the vertex farthest from the center. This has the slightly deleterious effect of not reproducing the tabular data at the eight exterior corners of the table. The value  $q(x, y, z)$  is returned by `QD3VL`.

### Comments

Informational errors

| Type | Code |                                                            |
|------|------|------------------------------------------------------------|
| 4    | 9    | The <code>XDATA</code> values must be strictly increasing. |
| 4    | 10   | The <code>YDATA</code> values must be strictly increasing. |
| 4    | 11   | The <code>ZDATA</code> values must be strictly increasing. |

### Example

In this example, the value of  $\sin(x + y + z)$  at  $x = y = z = \pi/3$  is approximated by using `QD3VL` on a grid of size  $21 \times 42 \times 18$  equally spaced values on the cube  $[0, 2]^3$ .

```
USE IMSL_LIBRARIES
```

```

 IMPLICIT NONE
 INTEGER LDF, MDF, NXDATA, NYDATA, NZDATA
 PARAMETER (NXDATA=21, NYDATA=42, NZDATA=18, LDF=NXDATA, &
 MDF=NYDATA)
!
 INTEGER I, J, K, NOUT
 REAL F, FDATA(LDF,MDF,NZDATA), FLOAT, PI, Q, &
 SIN, X, XDATA(NXDATA), Y, YDATA(NYDATA), Z, &
 ZDATA(NZDATA)
 INTRINSIC FLOAT, SIN
!
! Define function
 F(X,Y,Z) = SIN(X+Y+Z)
!
! Set up X-grid
 DO 10 I=1, NXDATA
 XDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NXDATA-1))
10 CONTINUE
!
! Set up Y-grid
 DO 20 J=1, NYDATA
 YDATA(J) = 2.0*(FLOAT(J-1)/FLOAT(NYDATA-1))
20 CONTINUE
!
! Set up Z-grid
 DO 30 K=1, NZDATA
 ZDATA(K) = 2.0*(FLOAT(K-1)/FLOAT(NZDATA-1))
30 CONTINUE
!
! Evaluate function on grid
 DO 40 I=1, NXDATA
 DO 40 J=1, NYDATA
 DO 40 K=1, NZDATA
 FDATA(I,J,K) = F(XDATA(I),YDATA(J),ZDATA(K))
40 CONTINUE
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Write heading
 WRITE (NOUT,99999)
!
! Get value for PI and set values
! for X, Y, and Z
 PI = CONST('PI')
 X = PI/3.0
 Y = PI/3.0
 Z = PI/3.0
!
! Evaluate quadratic at (X,Y,Z)
 Q = QD3VL(X,Y,Z,XDATA,YDATA,ZDATA,FDATA)
!
! Print results
 WRITE (NOUT,'(6F11.4)') X, Y, Z, F(X,Y,Z), Q, (Q-F(X,Y,Z))
99999 FORMAT (10X, 'X', 10X, 'Y', 10X, 'Z', 5X, 'F(X,Y,Z)', 4X, &
 'QD3VL', 6X, 'ERROR')
 END

```

## Output

| X      | Y      | Z      | F(X, Y, Z) | QD3VL  | ERROR  |
|--------|--------|--------|------------|--------|--------|
| 1.0472 | 1.0472 | 1.0472 | 0.0000     | 0.0001 | 0.0001 |

---

## QD3DR

This function evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation.

### Function Return Value

*QD3DR* — Value of the appropriate derivative of the function at (X, Y, Z). (Output)

### Required Arguments

*IXDER* — Order of the *x*-derivative. (Input)

*IYDER* — Order of the *y*-derivative. (Input)

*IZDER* — Order of the *z*-derivative. (Input)

*X* — *x*-coordinate of the point at which the function is to be evaluated. (Input)

*Y* — *y*-coordinate of the point at which the function is to be evaluated. (Input)

*Z* — *z*-coordinate of the point at which the function is to be evaluated. (Input)

*XDATA* — Array of length *NXDATA* containing the location of the data points in the *x*-direction. (Input)  
*XDATA* must be increasing.

*YDATA* — Array of length *NYDATA* containing the location of the data points in the *y*-direction. (Input)  
*YDATA* must be increasing.

*ZDATA* — Array of length *NZDATA* containing the location of the data points in the *z*-direction. (Input)  
*ZDATA* must be increasing.

*FDATA* — Array of size *NXDATA* by *NYDATA* by *NZDATA* containing function values. (Input)  
*FDATA*(I, J, K) is the value of the function at (*XDATA*(I), *YDATA*(J), *ZDATA*(K)).

### Optional Arguments

*NXDATA* — Number of data points in the *x*-direction. (Input)  
*NXDATA* must be at least three.  
Default: *NXDATA* = size(*XDATA*,1).

**NYDATA** — Number of data points in the y-direction. (Input)

NYDATA must be at least three.

Default: NYDATA = size (YDATA,1).

**NZDATA** — Number of data points in the z-direction. (Input)

NZDATA must be at least three.

Default: NZDATA = size (ZDATA,1).

**LDF** — Leading dimension of FDATA exactly as specified in the dimension statement of the calling program. (Input)

LDF must be at least as large as NXDATA.

Default: LDF = size (FDATA,1).

**MDF** — Middle (second) dimension of FDATA exactly as specified in the dimension statement of the calling program. (Input)

MDF must be at least as large as NYDATA.

Default: MDF = size (FDATA,2).

**CHECK** — Logical variable that is .TRUE. if checking of XDATA, YDATA, and ZDATA is required or .FALSE. if checking is not required. (Input)

Default: CHECK = .TRUE.

## **FORTRAN 90 Interface**

Generic: QD3DR (IXDER, IYDER, IZDER, X, Y, Z, XDATA, YDATA, ZDATA, FDATA [, ...])

Specific: The specific interface names are S\_QD3DR and D\_QD3DR.

## **FORTRAN 77 Interface**

Single: QD3DR (IXDER, IYDER, IZDER, X, Y, Z, NXDATA, XDATA, NYDATA, YDATA, NZDATA, ZDATA, FDATA, LDF, MDF, CHECK)

Double: The double precision function name is DQD3DR.

## **Description**

The function QD3DR interpolates a table of values, using quadratic polynomials, returning an approximation to the partial derivatives of the tabulated function. Let

$$(x_i, y_j, z_k, f_{ijk})$$

for  $i = 1, \dots, n_x, j = 1, \dots, n_y,$  and  $k = 1, \dots, n_z$  be the tabular data. Given a point  $(x, y, z)$  at which an interpolated value is desired, we first find the nearest interior grid point  $(x_i, y_j, z_k)$ . A trivariate quadratic interpolant  $q$  is then formed. Ten points are needed for this purpose. Seven points have the form

$$(x_i, y_j, z_k), (x_{i\pm 1}, y_j, z_k), (x_i, y_{j\pm 1}, z_k) \text{ and } (x_i, y_j, z_{k\pm 1})$$

The last three points are drawn from the vertices of the octant containing  $(x, y, z)$ . There are four of these vertices remaining, and we choose to exclude the vertex farthest from the center. This has the slightly deleterious effect of not reproducing the tabular data at the eight exterior corners of the table. The value  $q^{(p,r,t)}(x, y, z)$  is returned by QD3DR, where  $p = \text{IXDER}$ ,  $r = \text{IYDER}$ , and  $t = \text{IZDER}$ .

## Comments

### 1. Informational errors

| Type | Code |                                               |
|------|------|-----------------------------------------------|
| 4    | 9    | The XDATA values must be strictly increasing. |
| 4    | 10   | The YDATA values must be strictly increasing. |
| 4    | 11   | The ZDATA values must be strictly increasing. |

### 2. Because quadratic interpolation is used, if the order of any derivative is greater than two, then the returned value is zero.

## Example

In this example, the derivatives of  $\sin(x + y + z)$  at  $x = y = z = \pi/5$  are approximated by using QD3DR on a grid of size  $21 \times 42 \times 18$  equally spaced values on the cube  $[0, 2]^3$ .

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER LDF, MDF, NXDATA, NYDATA, NZDATA
PARAMETER (NXDATA=21, NYDATA=42, NZDATA=18, LDF=NXDATA, &
 MDF=NYDATA)
!
INTEGER I, IXDER, IYDER, IZDER, J, K, NOUT
REAL F, FDATA(NXDATA,NYDATA,NZDATA), FLOAT, FU, &
 FUNC, PI, Q, SIN, X, XDATA(NXDATA), Y, &
 YDATA(NYDATA), Z, ZDATA(NZDATA)
INTRINSIC FLOAT, SIN
EXTERNAL FUNC
!
! Define function
F(X,Y,Z) = SIN(X+Y+Z)
!
! Set up X-grid
DO 10 I=1, NXDATA
 XDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NXDATA-1))
10 CONTINUE
!
! Set up Y-grid
DO 20 J=1, NYDATA
 YDATA(J) = 2.0*(FLOAT(J-1)/FLOAT(NYDATA-1))
20 CONTINUE
!
! Set up Z-grid
DO 30 K=1, NZDATA
 ZDATA(K) = 2.0*(FLOAT(K-1)/FLOAT(NZDATA-1))
30 CONTINUE

```

```

! Evaluate function on grid
DO 40 I=1, NXDATA
 DO 40 J=1, NYDATA
 DO 40 K=1, NZDATA
 FDATA(I,J,K) = F(XDATA(I),YDATA(J),ZDATA(K))
40 CONTINUE
! Get output unit number
CALL UMACH (2, NOUT)
! Write heading
WRITE (NOUT,99999)
! Get value for PI and set X, Y, and Z
PI = CONST('PI')
X = PI/5.0
Y = PI/5.0
Z = PI/5.0
! Compute derivatives at (X,Y,Z)
! and print results
DO 50 IXDER=0, 1
 DO 50 IYDER=0, 1
 DO 50 IZDER=0, 1
 Q = QD3DR (IXDER, IYDER, IZDER, X, Y, Z, XDATA, YDATA, ZDATA, FDATA)
 FU = FUNC (IXDER, IYDER, IZDER, X, Y, Z)
 WRITE (NOUT,99998) X, Y, Z, IXDER, IYDER, IZDER, FU, Q, &
 (FU-Q)
50 CONTINUE
!
99998 FORMAT (3F7.4, 3I5, 4X, F7.4, 8X, 2F10.4)
99999 FORMAT (39X, '(IDX, IDY, IDZ)', /, 6X, 'X', 6X, 'Y', 6X, &
 'Z', 3X, 'IDX', 2X, 'IDY', 2X, 'IDZ', 2X, 'F', &
 '(X,Y,Z)', 3X, 'QD3DR', 5X, 'ERROR')
END
!
REAL FUNCTION FUNC (IX, IY, IZ, X, Y, Z)
INTEGER IX, IY, IZ
REAL X, Y, Z
!
REAL COS, SIN
INTRINSIC COS, SIN
!
IF (IX.EQ.0 .AND. IY.EQ.0 .AND. IZ.EQ.0) THEN
! Define (0,0,0) derivative
 FUNC = SIN(X+Y+Z)
ELSE IF (IX.EQ.0 .AND. IY.EQ.0 .AND. IZ.EQ.1) THEN
! Define (0,0,1) derivative
 FUNC = COS(X+Y+Z)
ELSE IF (IX.EQ.0 .AND. IY.EQ.1 .AND. IZ.EQ.0) THEN
! Define (0,1,0,) derivative
 FUNC = COS(X+Y+Z)
ELSE IF (IX.EQ.0 .AND. IY.EQ.1 .AND. IZ.EQ.1) THEN
! Define (0,1,1) derivative
 FUNC = -SIN(X+Y+Z)
ELSE IF (IX.EQ.1 .AND. IY.EQ.0 .AND. IZ.EQ.0) THEN
! Define (1,0,0) derivative
 FUNC = COS(X+Y+Z)
ELSE IF (IX.EQ.1 .AND. IY.EQ.0 .AND. IZ.EQ.1) THEN
! Define (1,0,1) derivative

```

```

 FUNC = -SIN(X+Y+Z)
ELSE IF (IX.EQ.1 .AND. IY.EQ.1 .AND. IZ.EQ.0) THEN
!
 Define (1,1,0) derivative
 FUNC = -SIN(X+Y+Z)
ELSE IF (IX.EQ.1 .AND. IY.EQ.1 .AND. IZ.EQ.1) THEN
!
 Define (1,1,1) derivative
 FUNC = -COS(X+Y+Z)
ELSE
 FUNC = 0.0
END IF
RETURN
END

```

## Output

|        |        |        | (IDX, IDY, IDZ) |     |     |         |           |         |         |
|--------|--------|--------|-----------------|-----|-----|---------|-----------|---------|---------|
| X      | Y      | Z      | IDX             | IDY | IDZ | F       | (X, Y, Z) | QD3DR   | ERROR   |
| 0.6283 | 0.6283 | 0.6283 | 0               | 0   | 0   | 0.9511  |           | 0.9511  | -0.0001 |
| 0.6283 | 0.6283 | 0.6283 | 0               | 0   | 1   | -0.3090 |           | -0.3080 | -0.0010 |
| 0.6283 | 0.6283 | 0.6283 | 0               | 1   | 0   | -0.3090 |           | -0.3088 | 0.0002  |
| 0.6283 | 0.6283 | 0.6283 | 0               | 1   | 1   | -0.9511 |           | -0.9587 | 0.0077  |
| 0.6283 | 0.6283 | 0.6283 | 1               | 0   | 0   | -0.3090 |           | -0.3078 | -0.0012 |
| 0.6283 | 0.6283 | 0.6283 | 1               | 0   | 1   | -0.9511 |           | -0.9348 | -0.0162 |
| 0.6283 | 0.6283 | 0.6283 | 1               | 1   | 0   | -0.9511 |           | -0.9613 | 0.0103  |
| 0.6283 | 0.6283 | 0.6283 | 1               | 1   | 1   | 0.3090  |           | 0.0000  | 0.3090  |

---

## SURF



Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

### Required Arguments

**XYDATA** — A 2 by *NDATA* array containing the coordinates of the interpolation points. (Input)

These points must be distinct. The *x*-coordinate of the *I*-th data point is stored in *XYDATA*(1, *I*) and the *y*-coordinate of the *I*-th data point is stored in *XYDATA*(2, *I*).

**FDATA** — Array of length *NDATA* containing the interpolation values. (Input) *FDATA*(*I*) contains the value at (*XYDATA*(1, *I*), *XYDATA*(2, *I*)).

**XOUT** — Array of length *NXOUT* containing an increasing sequence of points. (Input)  
These points are the *x*-coordinates of a grid on which the interpolated surface is to be evaluated.

**YOUT** — Array of length `NYOUT` containing an increasing sequence of points. (Input)  
These points are the y-coordinates of a grid on which the interpolated surface is to be evaluated.

**SUR** — Matrix of size `NXOUT` by `NYOUT`. (Output)  
This matrix contains the values of the surface on the `XOUT` by `YOUT` grid, i.e. `SUR(I, J)` contains the interpolated value at `(XOUT(I), YOUT(J))`.

### Optional Arguments

**NDATA** — Number of data points. (Input)  
`NDATA` must be at least four.  
Default: `NDATA = size (FDATA,1)`.

**NXOUT** — The number of elements in `XOUT`. (Input)  
Default: `NXOUT = size (XOUT,1)`.

**NYOUT** — The number of elements in `YOUT`. (Input)  
Default: `NYOUT = size (YOUT,1)`.

**LDSUR** — Leading dimension of `SUR` exactly as specified in the dimension statement of the calling program. (Input)  
`LDSUR` must be at least as large as `NXOUT`.  
Default: `LDSUR = size (SUR,1)`.

### FORTRAN 90 Interface

Generic: `CALL SURF (XYDATA, FDATA, XOUT, YOUT, SUR [, ...])`

Specific: The specific interface names are `S_SURF` and `D_SURF`.

### FORTRAN 77 Interface

Single: `CALL SURF (NDATA, XYDATA, FDATA, NXOUT, NYOUT, XOUT, YOUT, SUR, LDSUR)`

Double: The double precision name is `DSURF`.

### Description

This routine is designed to compute a  $C^1$  interpolant to scattered data in the plane. Given the data points

$$\{(x_i, y_i, f_i)\}_{i=1}^N \text{ in } \mathbf{R}^3$$

`SURF` returns (in `SUR`, the user-specified grid) the values of the interpolant  $s$ . The computation of  $s$  is as follows: First the Delaunay triangulation of the points

$$\{(x_i, y_i)\}_{i=1}^N$$

is computed. On each triangle  $T$  in this triangulation,  $s$  has the form

$$s(x, y) = \sum_{m+n \leq 5} c_{mn}^T x^m y^n \quad \forall x, y \in T$$

Thus,  $s$  is a bivariate quintic polynomial on each triangle of the triangulation. In addition, we have

$$s(x_i, y_i) = f_i \quad \text{for } i = 1, \dots, N$$

and  $s$  is continuously differentiable across the boundaries of neighboring triangles. These conditions do not exhaust the freedom implied by the above representation. This additional freedom is exploited in an attempt to produce an interpolant that is faithful to the global shape properties implied by the data. For more information on this routine, we refer the reader to the article by Akima (1978). The grid is specified by the two integer variables `NXOUT`, `NYOUT` that represent, respectively, the number of grid points in the first (second) variable and by two real vectors that represent, respectively, the first (second) coordinates of the grid.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `S2RF/DS2RF`. The reference is:

```
CALL S2RF (NDATA, XYDATA, FDATA, NXOUT, NYOUT, XOUT, YOUT,
 SUR, LDSUR, IWK, WK)
```

The additional arguments are as follows:

**IWK** — Work array of length  $31 * \text{NDATA} + 2 * (\text{NXOUT} * \text{NYOUT})$ .

**WK** — Work array of length  $6 * \text{NDATA}$ .

2. Informational errors

| Type | Code | Description                                               |
|------|------|-----------------------------------------------------------|
| 4    | 5    | The data point values must be distinct.                   |
| 4    | 6    | The <code>XOUT</code> values must be strictly increasing. |
| 4    | 7    | The <code>YOUT</code> values must be strictly increasing. |

3. This method of interpolation reproduces linear functions.

## Example

In this example, the interpolant to the linear function  $3 + 7x + 2y$  is computed from 20 data points equally spaced on the circle of radius 3. We then print the values on a  $3 \times 3$  grid.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
```

```

INTEGER LDSUR, NDATA, NXOUT, NYOUT
PARAMETER (NDATA=20, NXOUT=3, NYOUT=3, LDSUR=NXOUT)
!
INTEGER I, J, NOUT
REAL ABS, COS, F, FDATA(NDATA), FLOAT, PI, &
 SIN, SUR(LDSUR,NYOUT), X, XOUT(NXOUT), &
 XYDATA(2,NDATA), Y, YOUT(NYOUT)
INTRINSIC ABS, COS, FLOAT, SIN
!
! Define function
F(X,Y) = 3.0 + 7.0*X + 2.0*Y
!
! Get value for PI
PI = CONST('PI')
!
! Set up X, Y, and F data on a circle
DO 10 I=1, NDATA
 XYDATA(1,I) = 3.0*SIN(2.0*PI*FLOAT(I-1)/FLOAT(NDATA))
 XYDATA(2,I) = 3.0*COS(2.0*PI*FLOAT(I-1)/FLOAT(NDATA))
 FDATA(I) = F(XYDATA(1,I),XYDATA(2,I))
10 CONTINUE
!
! Set up XOUT and YOUT data on [0,1] by
! [0,1] grid.
DO 20 I=1, NXOUT
 XOUT(I) = FLOAT(I-1)/FLOAT(NXOUT-1)
20 CONTINUE
DO 30 I=1, NYOUT
 YOUT(I) = FLOAT(I-1)/FLOAT(NYOUT-1)
30 CONTINUE
!
! Interpolate scattered data
CALL SURF (XYDATA, FDATA, XOUT, YOUT, SUR)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99998)
!
! Print results
DO 40 I=1, NYOUT
 DO 40 J=1, NXOUT
 WRITE (NOUT,99999) XOUT(J), YOUT(I), SUR(J,I), &
 F(XOUT(J),YOUT(I)), &
 ABS(SUR(J,I)-F(XOUT(J),YOUT(I)))
 40 CONTINUE
99998 FORMAT (' ', 10X, 'X', 11X, 'Y', 9X, 'SURF', 6X, 'F(X,Y)', 7X, &
 'ERROR', /)
99999 FORMAT (1X, 5F12.4)
END

```

## Output

| X      | Y      | SURF    | F(X,Y)  | ERROR  |
|--------|--------|---------|---------|--------|
| 0.0000 | 0.0000 | 3.0000  | 3.0000  | 0.0000 |
| 0.5000 | 0.0000 | 6.5000  | 6.5000  | 0.0000 |
| 1.0000 | 0.0000 | 10.0000 | 10.0000 | 0.0000 |
| 0.0000 | 0.5000 | 4.0000  | 4.0000  | 0.0000 |
| 0.5000 | 0.5000 | 7.5000  | 7.5000  | 0.0000 |
| 1.0000 | 0.5000 | 11.0000 | 11.0000 | 0.0000 |
| 0.0000 | 1.0000 | 5.0000  | 5.0000  | 0.0000 |

|        |        |         |         |        |
|--------|--------|---------|---------|--------|
| 0.5000 | 1.0000 | 8.5000  | 8.5000  | 0.0000 |
| 1.0000 | 1.0000 | 12.0000 | 12.0000 | 0.0000 |

---

## SURFND

Performs multidimensional interpolation and differentiation for up to 7 dimensions.

The dimension,  $n$ , of the problem is determined by the rank of `FDATA`, and cannot be greater than seven. The number of gridpoints in the  $i$ -th direction,  $d_i$ , is determined by the corresponding dimension for `FDATA`.

### Function Return Value

*SURFND* — Interpolated value of the function.

### Required Arguments

*X* — Array of length  $n$  containing the point at which interpolation is to be done. (Input)  
An interpolant is to be calculated at the point:

$$(x_1, x_2, \dots, x_n)$$

*XDATA* — Array of size  $n$  by  $\max(d_1, \dots, d_n)$  giving the gridpoint values for the function to be interpolated. (Input)

The gridpoints need not be uniformly spaced. See `FDATA` for more details.

*FDATA* —  $n$  dimensional array, dimensioned  $d_1 \times d_2 \times \dots \times d_n$  giving the values at the gridpoints of the function to be interpolated. (Input)

`FDATA( $i, j, k, \dots$ )` is the value of the function at

$$(XDATA_{1,i}, XDATA_{2,j}, XDATA_{3,k}, \dots)$$

$$\text{for } i=1, \dots, d_1, j=1, \dots, d_2, k=1, \dots, d_3, \dots$$

### Optional Arguments

*NDEG* — Array of length  $n$ , giving the degree of polynomial interpolation to be used in each dimension. (Input)

`NDEG( $i$ )` must be less than or equal to 15.

Default: `NDEG( $i$ ) = 5`, for  $i = 1, \dots, n$ .

*NDERS* — Maximum order of derivatives to be computed with respect to each variable. (Input)

`NDERS` cannot be larger than  $\max(7 - n, 2)$ . See `DERIV` for more details.

Default: `NDERS = 0`.

*DERIV* —  $n$  dimensional array, dimensioned  $(NDERS+1) \times (NDERS+1) \times \dots$  containing derivative estimates at the interpolation point. (Output)

`DERIV( $i, j, \dots$ )` will hold an estimate of the derivative with respect to  $x_1$   $i$  times, and with respect to  $x_2$   $j$  times, etc. where  $i = 0, \dots, NDERS$ ,  $j = 0, \dots, NDERS, \dots$ . The 0-th derivative means the function value, thus, `DERIV(0, 0, ...) = SURFND`.

**ERROR** — Estimate of the error in SURFND. (Output)

## FORTRAN 90 Interface

Generic: SURFND (X, XDATA, FDATA [, ...])

Specific: The specific interface names are  $sn\_SURFND$  and  $dn\_SURFND$ , where “ $n$ ” indicates the dimension of the problem ( $n = 1, 2, 3, 4, 5, 6$  or  $7$ ).

## Description

The function SURFND interpolates a function of up to 7 variables, defined on a (possibly nonuniform) grid. It fits a polynomial of up to degree 15 in each variable through the grid points nearest the interpolation point. Approximations of partial derivatives are calculated, if requested. If derivatives are desired, high precision is strongly recommended. For more details, see Krogh (1970).

## Comments

Informational errors

Type Code

|   |   |                                                                                                     |
|---|---|-----------------------------------------------------------------------------------------------------|
| 3 | 1 | NDERS is too large, it has been reset to $\max(7-n, 2)$ .                                           |
| 3 | 2 | The interpolation point is outside the domain of the table, so extrapolation is used.               |
| 4 | 3 | Too many derivatives requested for the polynomial degree used.                                      |
| 4 | 4 | One of the polynomial degrees requested is too large for the number of gridlines in that direction. |

## Example

The 3D function  $f(x, y, z) = \exp(x + 2y + 3z)$ , defined on a 20 by 30 by 40 uniform grid, is interpolated.

```
USE SURFND_INT
USE UMACH_INT
IMPLICIT NONE

INTEGER, PARAMETER :: N=3, ND1=20, ND2=30, ND3=40, NDEGS=1
REAL X(N), DEROUT(0:NDEGS, 0:NDEGS, 0:NDEGS), &
 XDATA(N, MAX(ND1, ND2, ND3)), FDATA(ND1, ND2, ND3), &
 ERROR, XX, YY, ZZ, TRUE, RELERR, YOUT
INTEGER NDEG(N), I, J, K, NOUT
CHARACTER*1 ORDER(3)
INTRINSIC EXP, MAX

! 20 by 30 by 40 uniform grid used for
! interpolation of F(x,y,z) = exp(x+2*y+3*z)
NDEG(1) = 8
```

```

NDEG(2) = 7
NDEG(3) = 9

DO I=1,ND1
 XDATA(1,I) = 0.05*(I-1)
END DO

DO J=1,ND2
 XDATA(2,J) = 0.03*(J-1)
END DO

DO K=1,ND3
 XDATA(3,K) = 0.025*(K-1)
END DO

DO I=1,ND1
 DO J=1,ND2
 DO K=1,ND3
 XX = XDATA(1,I)
 YY = XDATA(2,J)
 ZZ = XDATA(3,K)
 FDATA(I,J,K) = EXP(XX+2*YY+3*ZZ)
 END DO
 END DO
END DO

! Interpolate at (0.18,0.43,0.35)
X(1) = 0.18
X(2) = 0.43
X(3) = 0.35

! Call SURFND
YOUT = SURFND(X,XDATA,FDATA,NDEG=NDEG,DERIV=DEROUT,ERROR=ERROR, &
 NDERS=NDERS)

! Output F,Fx,Fy,Fz,Fxy,Fxz,Fyz,Fxyz at
! interpolation point
XX = X(1)
YY = X(2)
ZZ = X(3)
CALL UMACH (2, NOUT)
WRITE(NOUT, 10) YOUT,ERROR

DO K=0,NDERS
 DO J=0,NDERS
 DO I=0,NDERS
 ORDER(1:3) = ' '
 IF (I.EQ.1) ORDER(1) = 'x'
 IF (J.EQ.1) ORDER(2) = 'y'
 IF (K.EQ.1) ORDER(3) = 'z'
 TRUE = 2**J*3**K*EXP(XX+2*YY+3*ZZ)
 RELERR = (DEROUT(I,J,K)-TRUE)/TRUE
 WRITE(NOUT, 20) ORDER,DEROUT(I,J,K),TRUE,RELERR
 END DO
 END DO
END DO
10 FORMAT (' EST. VALUE = ',F10.6,', EST. ERROR = ',E11.3, '//, &

```

```

11X, 'Computed Der.', 5X, 'True Der.', 4X, 'Rel. Err')
20 FORMAT (2X, 'F', 3A1, 2F15.6, E15.3)
END

```

## Output

EST. VALUE = 8.084915, EST. ERROR = 0.419E-05

|      | Computed Der. | True Der. | Rel. Err   |
|------|---------------|-----------|------------|
| F    | 8.084915      | 8.084914  | 0.118E-06  |
| Fx   | 8.084907      | 8.084914  | -0.944E-06 |
| F y  | 16.169882     | 16.169828 | 0.330E-05  |
| Fxy  | 16.171101     | 16.169828 | 0.787E-04  |
| F z  | 24.254705     | 24.254742 | -0.149E-05 |
| Fx z | 24.255133     | 24.254742 | 0.161E-04  |
| F yz | 48.505203     | 48.509483 | -0.882E-04 |
| Fxyz | 48.464718     | 48.509483 | -0.923E-03 |

---

## RLINE

Fits a line to a set of data points using least squares.

### Required Arguments

*XDATA* — Vector of length *NOBS* containing the *x*-values. (Input)

*YDATA* — Vector of length *NOBS* containing the *y*-values. (Input)

*B0* — Estimated intercept of the fitted line. (Output)

*B1* — Estimated slope of the fitted line. (Output)

### Optional Arguments

*NOBS* — Number of observations. (Input)

Default: *NOBS* = size(*XDATA*,1).

*STAT* — Vector of length 12 containing the statistics described below. (Output)

| <b>I</b> | <b>STAT (I)</b>                 |
|----------|---------------------------------|
| 1        | Mean of <i>XDATA</i>            |
| 2        | Mean of <i>YDATA</i>            |
| 3        | Sample variance of <i>XDATA</i> |
| 4        | Sample variance of <i>YDATA</i> |

|          |                                                                                  |
|----------|----------------------------------------------------------------------------------|
| <b>I</b> | <b>STAT (I)</b>                                                                  |
| 5        | Correlation                                                                      |
| 6        | Estimated standard error of B0                                                   |
| 7        | Estimated standard error of B1                                                   |
| 8        | Degrees of freedom for regression                                                |
| 9        | Sum of squares for regression                                                    |
| 10       | Degrees of freedom for error                                                     |
| 11       | Sum of squares for error                                                         |
| 12       | Number of (x, y) points containing NaN (not a number) as either the x or y value |

### **FORTRAN 90 Interface**

Generic:    CALL RLINE (XDATA, YDATA, B0, B1 [, ...])

Specific:   The specific interface names are S\_RLINE and D\_RLINE.

### **FORTRAN 77 Interface**

Single:     CALL RLINE (NOBS, XDATA, YDATA, B0, B1, STAT)

Double:     The double precision name is DRLINE.

### **Description**

Routine RLINE fits a line to a set of (x, y) data points using the method of least squares. Draper and Smith (1981, pages 1– 69) discuss the method. The fitted model is

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

where  $\hat{\beta}_0$  (stored in B0) is the estimated intercept and  $\hat{\beta}_1$  (stored in B1) is the estimated slope. In addition to the fit, RLINE produces some summary statistics, including the means, sample variances, correlation, and the error (residual) sum of squares. The estimated standard errors of  $\hat{\beta}_0$  and  $\hat{\beta}_1$  are computed under the simple linear regression model. The errors in the model are assumed to be uncorrelated and with constant variance.

If the x values are all equal, the model is degenerate. In this case, RLINE sets  $\hat{\beta}_1$  to zero and  $\hat{\beta}_0$  to the mean of the y values.

## Comments

Informational error

Type Code

4 1 Each (x, y) point contains NaN (not a number). There are no valid data.

## Example

This example fits a line to a set of data discussed by Draper and Smith (1981, Table 1.1, pages 9–33). The response  $y$  is the amount of steam used per month (in pounds), and the independent variable  $x$  is the average atmospheric temperature (in degrees Fahrenheit).

```
USE RLINE_INT
USE UMACH_INT
USE WRRRL_INT

IMPLICIT NONE
INTEGER NOBS
PARAMETER (NOBS=25)
!
INTEGER NOUT
REAL B0, B1, STAT(12), XDATA(NOBS), YDATA(NOBS)
CHARACTER CLABEL(13)*15, RLABEL(1)*4
!
DATA XDATA/35.3, 29.7, 30.8, 58.8, 61.4, 71.3, 74.4, 76.7, 70.7,&
 57.5, 46.4, 28.9, 28.1, 39.1, 46.8, 48.5, 59.3, 70.0, 70.0,&
 74.5, 72.1, 58.1, 44.6, 33.4, 28.6/
DATA YDATA/10.98, 11.13, 12.51, 8.4, 9.27, 8.73, 6.36, 8.5,&
 7.82, 9.14, 8.24, 12.19, 11.88, 9.57, 10.94, 9.58, 10.09,&
 8.11, 6.83, 8.88, 7.68, 8.47, 8.86, 10.36, 11.08/
DATA RLABEL/'NONE', CLABEL/' ', 'Mean of X', 'Mean of Y',&
 'Variance X', 'Variance Y', 'Corr.', 'Std. Err. B0',&
 'Std. Err. B1', 'DF Reg.', 'SS Reg.', 'DF Error',&
 'SS Error', 'Pts. with NaN'/
!
CALL RLINE (XDATA, YDATA, B0, B1, STAT=STAT)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) B0, B1
99999 FORMAT (' B0 = ', F7.2, ' B1 = ', F9.5)
CALL WRRRL ('%/STAT', STAT, RLABEL, CLABEL, 1, 12, 1, &
 FMT = '(12W10.4)')
!
END
```

## Output

B0 = 13.62 B1 = -0.07983

|           |           |            | STAT       |         |              |  |
|-----------|-----------|------------|------------|---------|--------------|--|
| Mean of X | Mean of Y | Variance X | Variance Y | Corr.   | Std. Err. B0 |  |
| 52.6      | 9.424     | 298.1      | 2.659      | -0.8452 | 0.5815       |  |

| Std. Err. B1 | DF Reg. | SS Reg. | DF Error | SS Error | Pts. with NaN |
|--------------|---------|---------|----------|----------|---------------|
| 0.01052      | 1       | 45.59   | 23       | 18.22    | 0             |

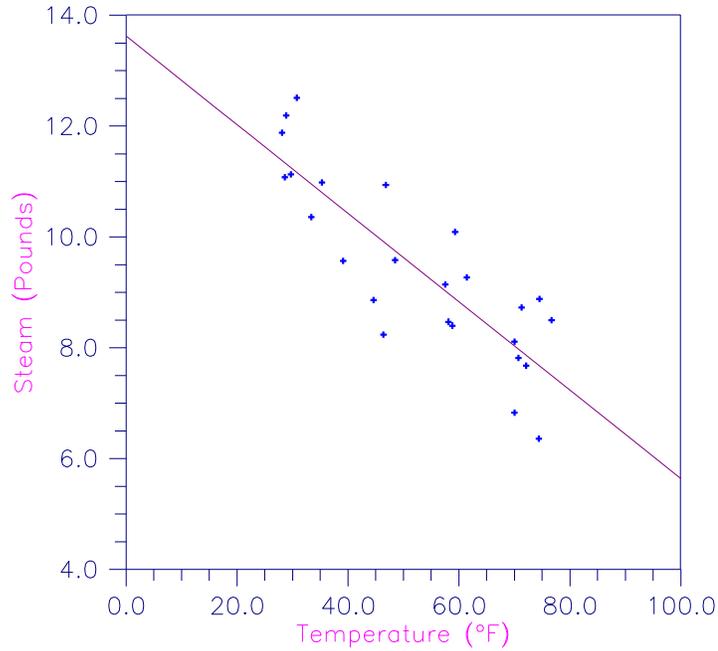


Figure 3- 5 Plot of the Data and the Least Squares Line

---

## RCURV

Fits a polynomial curve using least squares.

### Required Arguments

*XDATA* — Vector of length *NOBS* containing the *x* values. (Input)

*YDATA* — Vector of length *NOBS* containing the *y* values. (Input)

*B* — Vector of length *NDEG* + 1 containing the coefficients  $\hat{\beta}$ .  
(Output)

The fitted polynomial is

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \dots + \hat{\beta}_k x^k$$

### Optional Arguments

*NOBS* — Number of observations. (Input)  
Default: *NOBS* = size(*XDATA*,1).

**NDEG** — Degree of polynomial. (Input)

Default:  $NDEG = \text{size}(B,1) - 1$ .

**SSPOLY** — Vector of length  $NDEG + 1$  containing the sequential sums of squares. (Output)

$SSPOLY(1)$  contains the sum of squares due to the mean. For  $i = 1, 2, \dots, NDEG$ ,

$SSPOLY(i + 1)$  contains the sum of squares due to  $x^i$  adjusted for the mean,  $x, x^2, \dots$ , and  $x^{i-1}$ .

**STAT** — Vector of length 10 containing statistics described below. (Output)

***i*      Statistics**

1      Mean of  $x$

2      Mean of  $y$

3      Sample variance of  $x$

4      Sample variance of  $y$

5      R-squared (in percent)

6      Degrees of freedom for regression

7      Regression sum of squares

8      Degrees of freedom for error

9      Error sum of squares

10     Number of data points  $(x, y)$  containing NaN (not a number) as a  $x$  or  $y$  value

### **FORTRAN 90 Interface**

Generic:      `CALL RCURV (XDATA, YDATA, B [, ...])`

Specific:     The specific interface names are `S_RCURV` and `D_RCURV`.

### **FORTRAN 77 Interface**

Single:        `CALL RCURV (NOBS, XDATA, YDATA, NDEG, B, SSPOLY, STAT)`

Double:       The double precision name is `DRCURV`.

## Description

Routine `RCURV` computes estimates of the regression coefficients in a polynomial (curvilinear) regression model. In addition to the computation of the fit, `RCURV` computes some summary statistics. Sequential sums of squares attributable to each power of the independent variable (stored in `SSPOLY`) are computed. These are useful in assessing the importance of the higher order powers in the fit. Draper and Smith (1981, pages 101–102) and Neter and Wasserman (1974, pages 278–287) discuss the interpretation of the sequential sums of squares. The statistic  $R^2$  (stored in `STAT(5)`) is the percentage of the sum of squares of  $y$  about its mean explained by the polynomial curve. Specifically,

$$R^2 = \frac{\sum_{i=1}^n (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^n (y_i - \bar{y})^2} 100\%$$

where

$$\hat{y}_i$$

is the fitted  $y$  value at  $x_i$  and

$$\bar{y}$$

(stored in `STAT(2)`) is the mean of  $y$ . This statistic is useful in assessing the overall fit of the curve to the data.  $R^2$  must be between 0% and 100%, inclusive.  $R^2 = 100\%$  indicates a perfect fit to the data.

Routine `RCURV` computes estimates of the regression coefficients in a polynomial model using orthogonal polynomials as the regressor variables. This reparameterization of the polynomial model in terms of orthogonal polynomials has the advantage that the loss of accuracy resulting from forming powers of the  $x$ -values is avoided. All results are returned to the user for the original model.

The routine `RCURV` is based on the algorithm of Forsythe (1957). A modification to Forsythe's algorithm suggested by Shampine (1975) is used for computing the polynomial coefficients. A discussion of Forsythe's algorithm and Shampine's modification appears in Kennedy and Gentle (1980, pages 342–347).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `R2URV/DR2URV`. The reference is:

```
CALL R2URV (NOBS, XDATA, YDATA, NDEG, B, SSPOLY, STAT, WK,
 IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $11 * \text{NOBS} + 11 * \text{NDEG} + 5 + (\text{NDEG} + 1) * (\text{NDEG} + 3)$ .

**IWK** — Work vector of length `NOBS`.

## 2. Informational errors

| Type | Code |                                                                                                                    |
|------|------|--------------------------------------------------------------------------------------------------------------------|
| 4    | 3    | Each $(x, y)$ point contains NaN (not a number). There are no valid data.                                          |
| 4    | 7    | The $x$ values are constant. At least $NDEG + 1$ distinct $x$ values are needed to fit a $NDEG$ polynomial.        |
| 3    | 4    | The $y$ values are constant. A zero order polynomial is fit. High order coefficients are set to zero.              |
| 3    | 5    | There are too few observations to fit the desired degree polynomial. High order coefficients are set to zero.      |
| 3    | 6    | A perfect fit was obtained with a polynomial of degree less than $NDEG$ . High order coefficients are set to zero. |

3. If  $NDEG$  is greater than 10, the accuracy of the results may be questionable.

## Example

A polynomial model is fitted to data discussed by Neter and Wasserman (1974, pages 279– 285). The data set contains the response variable  $y$  measuring coffee sales (in hundred gallons) and the number of self-service coffee dispensers. Responses for fourteen similar cafeterias are in the data set.

```
USE RCURV_INT
USE WRRRL_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER NDEG, NOBS
PARAMETER (NDEG=2, NOBS=14)
!
REAL B(NDEG+1), SSPOLY(NDEG+1), STAT(10), XDATA(NOBS), &
 YDATA(NOBS)
CHARACTER CLABEL(11)*15, RLABEL(1)*4
!
DATA RLABEL/'NONE'/, CLABEL/' ', 'Mean of X', 'Mean of Y', &
 'Variance X', 'Variance Y', 'R-squared', &
 'DF Reg.', 'SS Reg.', 'DF Error', 'SS Error', &
 'Pts. with NaN'/
DATA XDATA/0., 0., 1., 1., 2., 2., 4., 4., 5., 5., 6., 6., 7., &
 7./
DATA YDATA/508.1, 498.4, 568.2, 577.3, 651.7, 657.0, 755.3, &
 758.9, 787.6, 792.1, 841.4, 831.8, 854.7, 871.4/
!
CALL RCURV (XDATA, YDATA, B, SSPOLY=SSPOLY, STAT=STAT)
!
CALL WRRRN ('B', B, 1, NDEG+1, 1)
CALL WRRRN ('SSPOLY', SSPOLY, 1, NDEG+1, 1)

CALL WRRRL ('%/STAT', STAT, RLABEL, CLABEL, 1, 10, 1, &
```

FMT='(2W10.4)'

END

## Output

| B     |      |      |
|-------|------|------|
| 1     | 2    | 3    |
| 503.3 | 78.9 | -4.0 |

| SSPOLY    |          |        |
|-----------|----------|--------|
| 1         | 2        | 3      |
| 7077152.0 | 220644.2 | 4387.7 |

| STAT      |           |            |            |           |         |
|-----------|-----------|------------|------------|-----------|---------|
| Mean of X | Mean of Y | Variance X | Variance Y | R-squared | DF Reg. |
| 3.571     | 711.0     | 6.418      | 17364.8    | 99.69     | 2       |

|          |          |          |               |
|----------|----------|----------|---------------|
| SS Reg.  | DF Error | SS Error | Pts. with NaN |
| 225031.9 | 11       | 710.5    | 0             |

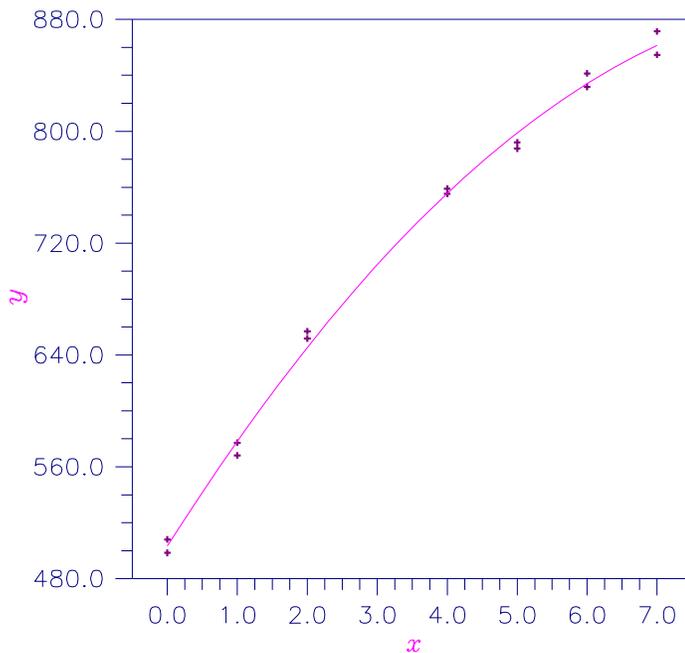


Figure 3- 6 Plot of Data and Second Degree Polynomial Fit

---

## FNLSQ

Computes a least-squares approximation with user-supplied basis functions.

## Required Arguments

***F*** — User-supplied function to evaluate basis functions. The form is  $F(K, X)$ , where

***K*** — Number of the basis function. (Input)

***K*** may be equal to 1, 2, ..., ***NBASIS***.

***X*** — Argument for evaluation of the *K*-th basis function. (Input)

***F*** — The function value. (Output)

***F*** must be declared `EXTERNAL` in the calling program. The data ***FDATA*** is approximated by  $A(1) * F(1, X) + A(2) * F(2, X) + \dots + A(NBASIS) * F(NBASIS, X)$  if ***INTCEP*** = 0 and is approximated by  $A(1) + A(2) * F(1, X) + \dots + A(NBASIS + 1) * F(NBASIS, X)$  if ***INTCEP*** = 1.

***XDATA*** — Array of length ***NDATA*** containing the abscissas of the data points. (Input)

***FDATA*** — Array of length ***NDATA*** containing the ordinates of the data points. (Input)

***A*** — Array of length ***INTCEP*** + ***NBASIS*** containing the coefficients of the approximation. (Output)

If ***INTCEP*** = 1, ***A***(1) contains the intercept. ***A***(***INTCEP*** + 1) contains the coefficient of the *I*-th basis function.

***SSE*** — Sum of squares of the errors. (Output)

## Optional Arguments

***INTCEP*** — Intercept option. (Input)

Default: ***INTCEP*** = 0.

### ***INTCEP*** Action

0 No intercept is automatically included in the model.

1 An intercept is automatically included in the model.

***NBASIS*** — Number of basis functions. (Input)

Default: ***NBASIS*** = size (***A***,1)

***NDATA*** — Number of data points. (Input)

Default: ***NDATA*** = size (***XDATA***,1).

***IWT*** — Weighting option. (Input)

Default: ***IWT*** = 0.

**IWT Action**

- 0 Weights of one are assumed.
- 1 Weights are supplied in WEIGHT.

**WEIGHT** — Array of length NDATA containing the weights. (Input if IWT = 1)  
 If IWT = 0, WEIGHT is not referenced and may be dimensioned of length one.

**FORTRAN 90 Interface**

Generic: CALL FNLSQ (F, XDATA, FDATA, A, SSE [, ...])

Specific: The specific interface names are S\_FNLSQ and D\_FNLSQ.

**FORTRAN 77 Interface**

Single: CALL FNLSQ (F, INTCEP, NBASIS, NDATA, XDATA, FDATA, IWT, WEIGHT, A, SSE)

Double: The double precision name is DFNLSQ.

**Description**

The routine FNLSQ computes a best least-squares approximation to given univariate data of the form

$$\{(x_i, f_i)\}_{i=1}^N$$

by  $M$  basis functions

$$\{F_j\}_{j=1}^M$$

(where  $M = \text{NBASIS}$ ). In particular, if INTCEP = 0, this routine returns the error sum of squares SSE and the coefficients  $a$  which minimize

$$\sum_{i=1}^N w_i \left( f_i - \sum_{j=1}^M a_j F_j(x_i) \right)^2$$

where  $w = \text{WEIGHT}$ ,  $N = \text{NDATA}$ ,  $x = \text{XDATA}$ , and  $f = \text{FDATA}$ .

If INTCEP = 1, then an intercept is placed in the model; and the coefficients  $a$ , returned by FNLSQ, minimize the error sum of squares as indicated below.

$$\sum_{i=1}^N w_i \left( f_i - a_1 - \sum_{j=1}^M a_{j+1} F_j(x_i) \right)^2$$

That is, the first element of the vector  $a$  is now the coefficient of the function that is identically 1 and the coefficients of the  $F_j$ 's are now  $a_{j+1}$ .

One additional parameter in the calling sequence for `FNLSQ` is `IWT`. If `IWT` is set to 0, then  $w_i = 1$  is assumed. If `IWT` is set to 1, then the user must supply the weights.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2LSQ/DF2LSQ`. The reference is:

```
CALL F2LSQ (F, INTCEP, NBASIS, NDATA, XDATA, FDATA, IWT,
 WEIGHT, A, SSE, WK)
```

The additional argument is

**WK** — Work vector of length  $(\text{INTCEP} + \text{NBASIS})**2 + 4 * (\text{INTCEP} + \text{NBASIS}) + \text{IWT} + 1$ . On output, the first  $(\text{INTCEP} + \text{NBASIS})**2$  elements of `WK` contain the `R` matrix from a `QR` decomposition of the matrix containing a column of ones (if `INTCEP = 1`) and the evaluated basis functions in columns `INTCEP + 1` through `INTCEP + NBASIS`.

2. Informational errors

Type Code

|   |   |                                                                                                                                  |
|---|---|----------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | Linear dependence of the basis functions exists. One or more components of <code>A</code> are set to zero.                       |
| 3 | 2 | Linear dependence of the constant function and basis functions exists. One or more components of <code>A</code> are set to zero. |
| 4 | 1 | Negative weight encountered.                                                                                                     |

## Example

In this example, we fit the following two functions (indexed by  $\delta$ )

$$1 + \sin x + 7 \sin 3x + \delta \epsilon$$

where  $\epsilon$  is random uniform deviate over the range  $[-1, 1]$ , and  $\delta$  is 0 for the first function and 1 for the second. These functions are evaluated at 90 equally spaced points on the interval  $[0, 6]$ . We use 4 basis functions,  $\sin kx$  for  $k = 1, \dots, 4$ , with and without the intercept.

```
USE FNLSQ_INT
USE RNSET_INT
USE UMACH_INT
USE RNUNF_INT

IMPLICIT NONE
INTEGER NBASIS, NDATA
PARAMETER (NBASIS=4, NDATA=90)
```

```

!
INTEGER I, INTCEP, NOUT
REAL A(NBASIS+1), F, FDATA(NDATA), FLOAT, G, RNOISE, &
 SIN, SSE, X, XDATA(NDATA)
INTRINSIC FLOAT, SIN
EXTERNAL F
!
G(X) = 1.0 + SIN(X) + 7.0*SIN(3.0*X)
!
CALL RNSET (1234579) Set random number seed
!
CALL RSET (1234579) Set up data values
DO 10 I=1, NDATA
 XDATA(I) = 6.0*(FLOAT(I-1)/FLOAT(NDATA-1))
 FDATA(I) = G(XDATA(I))
10 CONTINUE
!
! Compute least squares fit with no
! intercept
CALL FNLSQ (F, XDATA, FDATA, A, SSE, INTCEP=INTCEP, &
 NBASIS=NBASIS)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99996)
!
! Write output
WRITE (NOUT,99999) SSE, (A(I),I=1,NBASIS)
!
INTCEP = 1
!
! Compute least squares fit with
! intercept
CALL FNLSQ (F, XDATA, FDATA, A, SSE, INTCEP=INTCEP, &
 NBASIS=NBASIS)
!
! Write output
WRITE (NOUT,99998) SSE, A(1), (A(I),I=2,NBASIS+1)
!
! Introduce noise
DO 20 I=1, NDATA
 RNOISE = RNUNF()
 RNOISE = 2.0*RNOISE - 1.0
 FDATA(I) = FDATA(I) + RNOISE
20 CONTINUE
INTCEP = 0
!
! Compute least squares fit with no
! intercept
CALL FNLSQ (F, XDATA, FDATA, A, SSE, INTCEP=INTCEP, &
 NBASIS=NBASIS)
!
! Write heading
WRITE (NOUT,99997)
!
! Write output
WRITE (NOUT,99999) SSE, (A(I),I=1,NBASIS)
!
INTCEP = 1
!
! Compute least squares fit with
! intercept
CALL FNLSQ (F, XDATA, FDATA, A, SSE, INTCEP=INTCEP, &
 NBASIS=NBASIS)
!
! Write output
WRITE (NOUT,99998) SSE, A(1), (A(I),I=2,NBASIS+1)

```

```

!
99996 FORMAT (//, ' Without error introduced we have :', /,&
' SSE Intercept Coefficients ', /)
99997 FORMAT (//, ' With error introduced we have :', /, ' SSE '&
', ' Intercept Coefficients ', /)
99998 FORMAT (1X, F8.4, 5X, F9.4, 5X, 4F9.4, /)
99999 FORMAT (1X, F8.4, 14X, 5X, 4F9.4, /)
END
REAL FUNCTION F (K, X)
INTEGER K
REAL X
!
REAL SIN
INTRINSIC SIN
!
F = SIN(K*X)
RETURN
END

```

## Output

Without error introduced we have :

| SSE     | Intercept | Coefficients |        |        |        |
|---------|-----------|--------------|--------|--------|--------|
| 89.8776 |           | 1.0101       | 0.0199 | 7.0291 | 0.0374 |
| 0.0000  | 1.0000    | 1.0000       | 0.0000 | 7.0000 | 0.0000 |

With error introduced we have :

| SSE      | Intercept | Coefficients |         |        |         |
|----------|-----------|--------------|---------|--------|---------|
| 112.4662 |           | 0.9963       | -0.0675 | 6.9825 | 0.0133  |
| 30.9831  | 0.9522    | 0.9867       | -0.0864 | 6.9548 | -0.0223 |

---

## BSLSQ

Computes the least-squares spline approximation, and return the B-spline coefficients.

### Required Arguments

**XDATA** — Array of length *NDATA* containing the data point abscissas. (Input)

**FDATA** — Array of length *NDATA* containing the data point ordinates. (Input)

**KORDER** — Order of the spline. (Input)

*KORDER* must be less than or equal to *NDATA*.

**XKNOT** — Array of length *NCOEF* + *KORDER* containing the knot sequence. (Input)

*XKNOT* must be nondecreasing.

**NCOEF** — Number of B-spline coefficients. (Input)

*NCOEF* cannot be greater than *NDATA*.

**BSCOEF** — Array of length `NCOEF` containing the B-spline coefficients. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)

Default: `NDATA = size(XDATA, 1)`

**WEIGHT** — Array of length `NDATA` containing the weights. (Input)

Default: `WEIGHT = 1.0`.

### FORTRAN 90 Interface

Generic: `CALL BSLSQ (XDATA, FDATA, KORORDER, XKNOT, NCOEF, BSCOEF [, ...])`

Specific: The specific interface names are `S_BSLSQ` and `D_BSLSQ`.

### FORTRAN 77 Interface

Single: `CALL BSLSQ (NDATA, XDATA, FDATA, WEIGHT, KORORDER, XKNOT, NCOEF, BSCOEF)`

Double: The double precision name is `DBSLSQ`.

### Description

The routine `BSLSQ` is based on the routine `L2APPR` by de Boor (1978, page 255). The IMSL routine `BSLSQ` computes a weighted discrete  $L_2$  approximation from a spline subspace to a given data set  $(x_i, f_i)$  for  $i = 1, \dots, N$  (where  $N = \text{NDATA}$ ). In other words, it finds B-spline coefficients,  $a = \text{BSCOEF}$ , such that

$$\sum_{i=1}^N \left| f_i - \sum_{j=1}^m a_j B_j(x_i) \right|^2 w_i$$

is a minimum, where  $m = \text{NCOEF}$  and  $B_j$  denotes the  $j$ -th B-spline for the given order, `KORORDER`, and knot sequence, `XKNOT`. This linear least squares problem is solved by computing and solving the normal equations. While the normal equations can sometimes cause numerical difficulties, their use here should not cause a problem because the B-spline basis generally leads to well-conditioned banded matrices.

The choice of weights depends on the problem. In some cases, there is a natural choice for the weights based on the relative importance of the data points. To approximate a continuous function (if the location of the data points can be chosen), then the use of Gauss quadrature weights and points is reasonable. This follows because `BSLSQ` is minimizing an approximation to the integral

$$\int |F - s|^2 dx$$

The Gauss quadrature weights and points can be obtained using the IMSL routine `GQRUL` (see [Chapter 4, Integration and Differentiation](#)).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LSQ/DB2LSQ`. The reference is:

```
CALL B2LSQ (NDATA, XDATA, FDATA, WEIGHT, KORDER, XKNOT,
 NCOEF, BSCOEF, WK1, WK2, WK3, WK4, IWK)
```

The additional arguments are as follows:

**WK1** — Work array of length  $(3 + \text{NCOEF}) * \text{KORDER}$ .

**WK2** — Work array of length `NDATA`.

**WK3** — Work array of length `NDATA`.

**WK4** — Work array of length `NDATA`.

**IWK** — Work array of length `NDATA`.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                  |
|---|---|------------------------------------------------------------------|
| 4 | 5 | Multiplicity of the knots cannot exceed the order of the spline. |
|---|---|------------------------------------------------------------------|

|   |   |                                  |
|---|---|----------------------------------|
| 4 | 6 | The knots must be nondecreasing. |
|---|---|----------------------------------|

|   |   |                                        |
|---|---|----------------------------------------|
| 4 | 7 | All weights must be greater than zero. |
|---|---|----------------------------------------|

|   |   |                                                                                                             |
|---|---|-------------------------------------------------------------------------------------------------------------|
| 4 | 8 | The smallest element of the data point array must be greater than or equal to the <code>KORDth</code> knot. |
|---|---|-------------------------------------------------------------------------------------------------------------|

|   |   |                                                                                                             |
|---|---|-------------------------------------------------------------------------------------------------------------|
| 4 | 9 | The largest element of the data point array must be less than or equal to the $(\text{NCOEF} + 1)$ st knot. |
|---|---|-------------------------------------------------------------------------------------------------------------|

3. The B-spline representation can be evaluated using `BSVAL`, and its derivative can be evaluated using `BSDER`.

## Example

In this example, we try to recover a quadratic polynomial using a quadratic spline with one interior knot from two different data sets. The first data set is generated by evaluating the quadratic at 50 equally spaced points in the interval  $(0, 1)$  and then adding uniformly distributed noise to the data. The second data set includes the first data set, and, additionally, the values at 0 and at 1 with no noise added. Since the first and last data points are uncontaminated by noise, we have chosen weights equal to  $10^5$  for these two points in this second problem. The quadratic, the first approximation, and the second approximation are then evaluated at 11 equally spaced points. This example illustrates the use of the weights to enforce interpolation at certain of the data points.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, NCOEF
PARAMETER (KORDER=3, NCOEF=4)
!
INTEGER I, NDATA, NOUT
REAL ABS, BSCOF1(NCOEF), BSCOF2(NCOEF), F,&
 FDATA1(50), FDATA2(52), FLOAT, RNOISE, S1,&
 S2, WEIGHT(52), X, XDATA1(50), XDATA2(52),&
 XKNOT(KORDER+NCOEF), XT, YT
INTRINSIC ABS, FLOAT
!
DATA WEIGHT/52*1.0/
!
F(X) = 8.0*X*(1.0-X)
!
CALL RNSET (12345679)
NDATA = 50
!
DO 10 I=1, NCOEF - KORDER + 2
 XKNOT(I+KORDER-1) = FLOAT(I-1)/FLOAT(NCOEF-KORDER+1)
10 CONTINUE
!
DO 20 I=1, KORDER - 1
 XKNOT(I) = XKNOT(KORDER)
 XKNOT(I+NCOEF+1) = XKNOT(NCOEF+1)
20 CONTINUE
!
DO 30 I=1, NDATA
 XDATA1(I) = FLOAT(I)/51.0
 RNOISE = RNUNF()
 RNOISE = RNOISE - 0.5
 FDATA1(I) = F(XDATA1(I)) + RNOISE
30 CONTINUE
!
CALL BSLSQ (XDATA1, FDATA1, KORDER, XKNOT, NCOEF, BSCOF1)
!
CALL SCOPY (50, XDATA1, 1, XDATA2(2:), 1)
CALL SCOPY (50, FDATA1, 1, FDATA2(2:), 1)
!
WEIGHT(1) = 1.0E5
XDATA2(1) = 0.0
FDATA2(1) = F(XDATA2(1))
WEIGHT(NDATA) = 1.0E5
XDATA2(NDATA) = 1.0
FDATA2(NDATA) = F(XDATA2(NDATA))
!

```

```

CALL BSLSQ (XDATA2, FDATA2, KORDER, XKNOT, NCOEF, BSCOF2, &
 WEIGHT=WEIGHT)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99998)
!
! Print the two interpolants
! at 11 points.
DO 40 I=1, 11
 XT = FLOAT(I-1)/10.0
 YT = F(XT)
!
! Evaluate splines
 S1 = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOF1)
 S2 = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOF2)
 WRITE (NOUT,99999) XT, YT, S1, S2, (S1-YT), (S2-YT)
40 CONTINUE
!
99998 FORMAT (7X, 'X', 9X, 'F(X)', 6X, 'S1(X)', 5X, 'S2(X)', 7X, &
 'F(X)-S1(X)', 7X, 'F(X)-S2(X)')
99999 FORMAT (' ', 4F10.4, 4X, F10.4, 7X, F10.4)
END

```

### Output

| X      | F(X)   | S1(X)   | S2(X)  | F(X)-S1(X) | F(X)-S2(X) |
|--------|--------|---------|--------|------------|------------|
| 0.0000 | 0.0000 | 0.0515  | 0.0000 | 0.0515     | 0.0000     |
| 0.1000 | 0.7200 | 0.7594  | 0.7490 | 0.0394     | 0.0290     |
| 0.2000 | 1.2800 | 1.3142  | 1.3277 | 0.0342     | 0.0477     |
| 0.3000 | 1.6800 | 1.7158  | 1.7362 | 0.0358     | 0.0562     |
| 0.4000 | 1.9200 | 1.9641  | 1.9744 | 0.0441     | 0.0544     |
| 0.5000 | 2.0000 | 2.0593  | 2.0423 | 0.0593     | 0.0423     |
| 0.6000 | 1.9200 | 1.9842  | 1.9468 | 0.0642     | 0.0268     |
| 0.7000 | 1.6800 | 1.7220  | 1.6948 | 0.0420     | 0.0148     |
| 0.8000 | 1.2800 | 1.2726  | 1.2863 | -0.0074    | 0.0063     |
| 0.9000 | 0.7200 | 0.6360  | 0.7214 | -0.0840    | 0.0014     |
| 1.0000 | 0.0000 | -0.1878 | 0.0000 | -0.1878    | 0.0000     |

---

## BSVLS

Computes the variable knot B-spline least squares approximation to given data.

### Required Arguments

**XDATA** — Array of length *NDATA* containing the data point abscissas. (Input)

**FDATA** — Array of length *NDATA* containing the data point ordinates. (Input)

**KORDER** — Order of the spline. (Input)  
 KORDER must be less than or equal to *NDATA*.

**NCOEF** — Number of B-spline coefficients. (Input)  
 NCOEF must be less than or equal to *NDATA*.

**XGUESS** — Array of length  $N_{\text{COEF}} + K_{\text{ORDER}}$  containing the initial guess of knots. (Input)  
XGUESS must be nondecreasing.

**XKNOT** — Array of length  $N_{\text{COEF}} + K_{\text{ORDER}}$  containing the (nondecreasing) knot sequence.  
(Output)

**BSCOEF** — Array of length  $N_{\text{COEF}}$  containing the B-spline representation. (Output)

**SSQ** — The square root of the sum of the squares of the error. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)  
NDATA must be at least 2.  
Default:  $NDATA = \text{size}(XDATA, 1)$

**WEIGHT** — Array of length  $NDATA$  containing the weights. (Input)  
Default:  $WEIGHT = 1.0$ .

### FORTRAN 90 Interface

Generic:    CALL BSVLS (NDATA, XDATA, FDATA, WEIGHT, KORDER, NCOEF, XGUESS,  
                  XKNOT, BSCOEF, SSQ)

Specific:    The specific interface names are S\_BSVLS and D\_BSVLS.

### FORTRAN 77 Interface

Single:     CALL BSVLS (XDATA, FDATA, KORDER, NCOEF, XGUESS, XKNOT, BSCOEF,  
                  SSQ [, ...])

Double:     The double precision name is DBSVLS.

### Description

The routine BSVLS attempts to find the best placement of knots that will minimize the leastsquares error to given data by a spline of order  $k = KORDER$  with  $N = NCOEF$  coefficients. The user provides the order  $k$  of the spline and the number of coefficients  $N$ . For this problem to make sense, it is necessary that  $N > k$ . We then attempt to find the minimum of the functional

$$F(\mathbf{a}, \mathbf{t}) = \sum_{i=1}^M w_i \left( f_i - \sum_{j=1}^N a_j B_{j,k,t}(x_j) \right)^2$$

The user must provide the weights  $w = WEIGHT$ , the data  $x_i = XDATA$  and  $f_i = FDATA$ , and  $M = NDATA$ . The minimum is taken over all admissible knot sequences  $\mathbf{t}$ .

The technique employed in `BSVLS` uses the fact that for a fixed knot sequence  $\mathbf{t}$  the minimization in  $a$  is a linear least-squares problem that can be solved by calling the IMSL routine `BSLSQ`. Thus, we can think of our objective function  $F$  as a function of just  $\mathbf{t}$  by setting

$$G(\mathbf{t}) = \min_a F(a, \mathbf{t})$$

A Gauss-Seidel (cyclic coordinate) method is then used to reduce the value of the new objective function  $G$ . In addition to this local method, there is a global heuristic built into the algorithm that will be useful if the data arise from a smooth function. This heuristic is based on the routine `NEWNOT` of de Boor (1978, pages 184 and 258–261).

The user must input an initial guess,  $\mathbf{t}^g = \text{XGUESS}$ , for the knot sequence. This guess must be a *valid* knot sequence for the splines of order  $k$  with

$$\mathbf{t}_1^g \leq \dots \leq \mathbf{t}_k^g \leq x_i \leq \mathbf{t}_{N+1}^g \leq \dots \leq \mathbf{t}_{N+k}^g, \quad i = 1, \dots, M$$

with  $\mathbf{t}^g$  nondecreasing, and

$$\mathbf{t}_i^g < \mathbf{t}_{i+k}^g \quad i = 1, \dots, N$$

The routine `BSVLS` returns the B-spline representation of the best fit found by the algorithm as well as the square root of the sum of squares error in `SSQ`. If this answer is unsatisfactory, you may reinitialize `BSVLS` with the return from `BSVLS` to see if an improvement will occur. We have found that this option does not usually (substantially) improve the result. In regard to execution speed, this routine can be several orders of magnitude slower than one call to the least-squares routine `BSLSQ`.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2VLS/DB2VLS`. The reference is:

```
CALL B2VLS (NDATA, XDATA, FDATA, WEIGHT, KORDER, NCOEF,
 XGUESS, XKNOT, BSCOEFF, SSQ, IWK, WK)
```

The additional arguments are as follows:

**IWK** — Work array of length `NDATA`.

**WK** — Work array of length `NCOEF * (6 + 2 * KORDER) + KORDER * (7 - KORDER) + 3 * NDATA + 3`.

2. Informational errors

| Type | Code | Description                                                                                                                                                                              |
|------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 12   | The knots found to be optimal are stacked more than <code>KORDER</code> . This indicates fewer knots will produce the same error sum of squares. The knots have been separated slightly. |

- 4 9 The multiplicity of the knots in XGUESS cannot exceed the order of the spline.
- 4 10 XGUESS must be nondecreasing.

## Example

In this example, we try to fit the function  $|x - .33|$  evaluated at 100 equally spaced points on  $[0, 1]$ . We first use quadratic splines with 2 interior knots initially at .2 and .8. The eventual error should be zero since the function is a quadratic spline with two knots stacked at .33. As a second example, we try to fit the same data with cubic splines with three interior knots initially located at .1, .2, and, .5. Again, the theoretical error is zero when the three knots are stacked at .33.

We include a graph of the initial least-squares fit using the IMSL routine [BSLSQ](#) for the above quadratic spline example with knots at .2 and .8. This graph overlays the graph of the spline computed by [BSVLS](#), which is indistinguishable from the data.

```

USE BSVLS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KORD1, KORD2, NCOEF1, NCOEF2, NDATA
PARAMETER (KORD1=3, KORD2=4, NCOEF1=5, NCOEF2=7, NDATA=100)
!
INTEGER I, NOUT
REAL ABS, BSCOEF(NCOEF2), F, FDATA(NDATA), FLOAT, SSQ, &
WEIGHT(NDATA), X, XDATA(NDATA), XGUES1(NCOEF1+KORD1), &
XGUES2(KORD2+NCOEF2), XKNOT(NCOEF2+KORD2)
INTRINSIC ABS, FLOAT
!
DATA XGUES1/3*0.0, .2, .8, 3*1.0001/
DATA XGUES2/4*0.0, .1, .2, .5, 4*1.0001/
DATA WEIGHT/NDATA*.01/
!
! Define function
F(X) = ABS(X-.33)
!
! Set up data
DO 10 I=1, NDATA
XDATA(I) = FLOAT(I-1)/FLOAT(NDATA)
FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Compute least squares B-spline
! representation with KORD1, NCOEF1,
! and XGUES1.
CALL BSVLS (XDATA, FDATA, KORD1, NCOEF1, XGUES1, &
XKNOT, BSCOEF, SSQ, WEIGHT=WEIGHT)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Print heading
WRITE (NOUT,99998) 'quadratic'
!
! Print SSQ and the knots
WRITE (NOUT,99999) SSQ, (XKNOT(I),I=1,KORD1+NCOEF1)
!
! Compute least squares B-spline
! representation with KORD2, NCOEF2,
! and XGUES2.
CALL BSVLS (XDATA, FDATA, KORD2, NCOEF2, XGUES2, &
XKNOT, BSCOEF, SSQ, WEIGHT=WEIGHT)

```

```

! Print SSQ and the knots
 WRITE (NOUT,99998) 'cubic'
 WRITE (NOUT,99999) SSQ, (XKNOT(I),I=1,KORD2+NCOEF2)
!
99998 FORMAT (' Piecewise ', A, /)
99999 FORMAT (' Square root of the sum of squares : ', F9.4, /, &
 ' Knot sequence : ', /, 1X, 11(F9.4,/,1X))
 END

```

## Output

Piecewise quadratic

```

Square root of the sum of squares : 0.0008
Knot sequence :
 0.0000
 0.0000
 0.0000
 0.3137
 0.3464
 1.0001
 1.0001
 1.0001

```

Piecewise cubic

```

Square root of the sum of squares : 0.0005
Knot sequence :
 0.0000
 0.0000
 0.0000
 0.0000
 0.3167
 0.3273
 0.3464
 1.0001
 1.0001
 1.0001
 1.0001

```

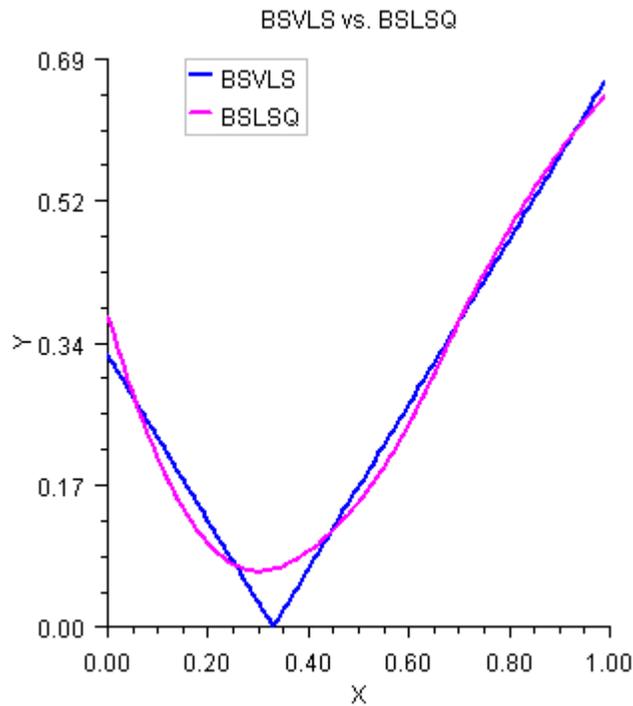


Figure 3-7 BSVLS vs. BSLSQ

---

## CONFIT



Computes the least-squares constrained spline approximation, returning the B-spline coefficients.

### Required Arguments

**XDATA** — Array of length `NDATA` containing the data point abscissas. (Input)

**FDATA** — Array of size `NDATA` containing the values to be approximated. (Input)  
`FDATA(I)` contains the value at `XDATA(I)`.

**XVAL** — Array of length `NXVAL` containing the abscissas at which the fit is to be constrained. (Input)

**NHARD** — Number of entries of `XVAL` involved in the ‘hard’ constraints. (Input)

Note: ( $0 \leq \text{NHARD} \leq \text{NXVAL}$ ). Setting `NHARD` to zero always results in a fit, while setting `NHARD` to `NXVAL` forces all constraints to be met. The ‘hard’ constraints must be

satisfied or else the routine signals failure. The ‘soft’ constraints need not be satisfied, but there will be an attempt to satisfy the ‘soft’ constraints. The constraints must be ordered in terms of priority with the most important constraints first. Thus, all of the ‘hard’ constraints must precede the ‘soft’ constraints. If infeasibility is detected among the soft constraints, we satisfy (in order) as many of the soft constraints as possible.

**IDER** — Array of length `NXVAL` containing the derivative value of the spline that is to be constrained. (Input)

If we want to constrain the integral of the spline over the closed interval  $(c, d)$ , then we set `IDER(I) = IDER(I + 1) = - 1` and `XVAL(I) = c` and `XVAL(I + 1) = d`. For consistency, we insist that `ITYPE(I) = ITYPE(I + 1) .GE. 0` and `c .LE. d`. Note that every entry in `IDER` must be at least  $- 1$ .

**ITYPE** — Array of length `NXVAL` indicating the types of general constraints. (Input)

| ITYPE(I)      | I-th Constraint                          |
|---------------|------------------------------------------|
| 1             | $BL(I) = f^{(d_i)}(x_i)$                 |
| 2             | $f^{(d_i)}(x_i) \leq BU(I)$              |
| 3             | $f^{(d_i)}(x_i) \geq BL(I)$              |
| 4             | $BL(I) \leq f^{(d_i)}(x_i) \leq BU(I)$   |
| $(d_i = -1)1$ | $BL(I) = \int_c^d f(t) dt$               |
| $(d_i = -1)2$ | $\int_c^d f(t) dt \leq BU(I)$            |
| $(d_i = -1)3$ | $\int_c^d f(t) dt \geq BL(I)$            |
| $(d_i = -1)4$ | $BL(I) \leq \int_c^d f(t) dt \leq BU(I)$ |
| 10            | periodic end conditions                  |
| 99            | disregard this constraint                |

In order to set two point constraints, we must have `ITYPE(I) = ITYPE(I + 1)` and `ITYPE(I)` must be negative.

| ITYPE(I) | I-th Constraint                                                 |
|----------|-----------------------------------------------------------------|
| -1       | $BL(I) = f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1})$               |
| -2       | $f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \leq BU(I)$            |
| -3       | $f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \geq BL(I)$            |
| -4       | $BL(I) \leq f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \leq BU(I)$ |

**BL** — Array of length NXVAL containing the lower limit of the general constraints, if there is no lower limit on the I-th constraint, then BL(I) is not referenced. (Input)

**BU** — Array of length NXVAL containing the upper limit of the general constraints, if there is no upper limit on the I-th constraint, then BU(I) is not referenced; if there is no range constraint, BL and BU can share the same storage locations. (Input)  
If the I-th constraint is an equality constraint, BU(I) is not referenced.

**KORDER** — Order of the spline. (Input)

**XKNOT** — Array of length NCOEF + KORDER containing the knot sequence. (Input)  
The entries of XKNOT must be nondecreasing.

**BSCOEF** — Array of length NCOEF containing the B-spline coefficients. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)  
Default: NDATA = size(XDATA,1).

**WEIGHT** — Array of length NDATA containing the weights. (Input)  
Default: WEIGHT = 1.0.

**NXVAL** — Number of points in the vector XVAL. (Input)  
Default: NXVAL = size(XVAL,1).

**NCOEF** — Number of B-spline coefficients. (Input)  
Default: NCOEF = size(BSCOEF,1).

### FORTRAN 90 Interface

Generic: CALL CONFT (XDATA, FDATA, XVAL, NHARD, IDER, ITYPE, BL, BU, KORDER,  
XKNOT, BSCOEF [, ...])

Specific: The specific interface names are S\_CONFT and D\_CONFT.

## FORTRAN 77 Interface

Single:      CALL CONFT (NDATA, XDATA, FDATA, WEIGHT, NXVAL, XVAL, NHARD, IDER, ITYPE, BL, BU, KORDER, XKNOT, NCOEF, BSCOEf)

Double:     The double precision name is DCONFT.

## Description

The routine CONFT produces a constrained, weighted least-squares fit to data from a spline subspace. Constraints involving one point, two points, or integrals over an interval are allowed. The types of constraints supported by the routine are of four types.

$$\begin{aligned}
 E_p[f] &= f^{(j_p)}(y_p) \\
 \text{or} &= f^{(j_p)}(y_p) - f^{(j_{p+1})}(y_{p+1}) \\
 \text{or} &= \int_{y_p}^{y_{p+1}} f(t) dt \\
 \text{or} &= \text{periodic end conditions}
 \end{aligned}$$

An interval,  $I_p$ , (which may be a point, a finite interval, or semi-infinite interval) is associated with each of these constraints.

The input for this routine consists of several items, first, the data set  $(x_i, f_i)$  for  $i = 1, \dots, N$  (where  $N = \text{NDATA}$ ), that is the data which is to be fit. Second, we have the weights to be used in the least squares fit ( $w = \text{WEIGHT}$ ). The vector XVAL of length NXVAL contains the abscissas of the points involved in specifying the constraints. The algorithm tries to satisfy all the constraints, but if the constraints are inconsistent then it will drop constraints, in the reverse order specified, until either a consistent set of constraints is found or the "hard" constraints are determined to be inconsistent (the "hard" constraints are those involving XVAL(1), ..., XVAL(NHARD)). Thus, the algorithm satisfies as many constraints as possible in the order specified by the user. In the case when constraints are dropped, the user will receive a message explaining how many constraints had to be dropped to obtain the fit. The next several arguments are related to the type of constraint and the constraint interval. The last four arguments determine the spline solution. The user chooses the spline subspace (KORDER, XKNOT, and NCOEF), and the routine returns the B-spline coefficients in BSCOEf.

Let  $n_f$  denote the number of feasible constraints as described above. Then, the routine solves the problem.

$$\sum_{i=1}^N \left| f_i - \sum_{j=1}^m a_j B_j(x_i) \right|^2 w_i$$

subject to

$$E_p \left[ \sum_{j=1}^m a_j B_j \right] \in I_p \quad p = 1, \dots, n_f$$

This linearly constrained least-squares problem is treated as a quadratic program and is solved by invoking the IMSL routine `QPROG` (see Chapter 8, Optimization).

The choice of weights depends on the data uncertainty in the problem. In some cases, there is a natural choice for the weights based on the estimates of errors in the data points.

Determining feasibility of linear constraints is a numerically sensitive task. If you encounter difficulties, a quick fix would be to widen the constraint intervals  $I_p$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2NFT/DC2NFT`. The reference is:

```
CALL C2NFT (NDATA, XDATA, FDATA, WEIGHT, NXVAL, XVAL,
 NHARD, IDER, ITYPE, BL, BU, KORDER, XKNOT, NCOEF,
 BSCOEf, H, G, A, RHS, WK, IPERM, IWK)
```

The additional arguments are as follows:

**H** — Work array of size `NCOEF` by `NCOEF`. Upon output, `H` contains the Hessian matrix of the objective function used in the call to `QPROG` (see Chapter 8, Optimization).

**G** — Work array of size `NCOEF`. Upon output, `G` contains the coefficients of the linear term used in the call to `QPROG`.

**A** — Work array of size  $(2 * NXVAL + KORDER)$  by  $(NCOEF + 1)$ . Upon output, `A` contains the constraint matrix used in the call `QPROG`. The last column of `A` is used to keep record of the original order of the constraints.

**RHS** — Work array of size  $2 * NXVAL + KORDER$ . Upon output, `RHS` contains the right hand side of the constraint matrix `A` used in the call to `QPROG`.

**WK** — Work array of size  $(KORDER + 1) * (2 * KORDER + 1) + (3 * NCOEF * NCOEF + 13 * NCOEF)/2 + (2 * NXVAL + KORDER + 30) * (2 * NXVAL + KORDER) + NDATA + 1$ .

**IPERM** — Work array of size `NXVAL`. Upon output, `IPERM` contains the permutation of the original constraints used to generate the matrix `A`.

**IWK** — Work array of size  $NDATA + 30 * (2 * NXVAL + KORDER) + 4 * NCOEF$ .

2. Informational errors

| Type | Code | Description                                                      |
|------|------|------------------------------------------------------------------|
| 3    | 11   | Soft constraints had to be removed in order to get a fit.        |
| 4    | 12   | Multiplicity of the knots cannot exceed the order of the spline. |

|   |    |                                                                                                                                                 |
|---|----|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 13 | The knots must be nondecreasing.                                                                                                                |
| 4 | 14 | The smallest element of the data point array must be greater than or equal to the <code>KORD</code> -th knot.                                   |
| 4 | 15 | The largest element of the data point array must be less than or equal to the <code>(NCOEF + 1)</code> st knot.                                 |
| 4 | 16 | All weights must be greater than zero.                                                                                                          |
| 4 | 17 | The hard constraints could not be met.                                                                                                          |
| 4 | 18 | The abscissas of the constrained points must lie within knot interval.                                                                          |
| 4 | 19 | The upperbound must be greater than or equal to the lowerbound for a range constraint.                                                          |
| 4 | 20 | The upper limit of integration must be greater than the lower limit of integration for constraints involving the integral of the approximation. |

### Example 1

This is a simple application of `CONFIT`. We generate data from the function

$$\frac{x}{2} + \sin\left(\frac{x}{2}\right)$$

contaminated with random noise and fit it with cubic splines. The function is increasing so we would hope that our least-squares fit would also be increasing. This is not the case for the unconstrained least squares fit generated by `BSLSQ`. We then force the derivative to be greater than 0 at `NXVAL = 15` equally spaced points and call `CONFIT`. The resulting curve is monotone. We print the error for the two fits averaged over 100 equally spaced points.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, NCOEF, NDATA, NXVAL
PARAMETER (KORDER=4, NCOEF=8, NDATA=15, NXVAL=15)
!
INTEGER I, IDER(NXVAL), ITYPE(NXVAL), NHARD, NOUT
REAL ABS, BL(NXVAL), BSCLSQ(NDATA), BSCNFT(NDATA), &
 BU(NXVAL), ERRLSQ, ERNFT, F1, FDATA(NDATA), FLOAT, &
 GRDSIZ, SIN, WEIGHT(NDATA), X, XDATA(NDATA), &
 XKNOT(KORDER+NDATA), XVAL(NXVAL)
INTRINSIC ABS, FLOAT, SIN
!
F1(X) = .5*X + SIN(.5*X)
!
! Initialize random number generator
! and get output unit number.
CALL RNSET (234579)
CALL UMACH (2, NOUT)
!
! Use default weights of one.
!
! Compute original XDATA and FDATA

```

```

! with random noise.
GRDSIZ = 10.0
DO 10 I=1, NDATA
 XDATA(I) = GRDSIZ*((FLOAT(I-1)/FLOAT(NDATA-1)))
 FDATA(I) = RNUNF()
 FDATA(I) = F1(XDATA(I)) + (FDATA(I)-.5)
10 CONTINUE
! Compute knots
DO 20 I=1, NCOEF - KORDER + 2
 XKNOT(I+KORDER-1) = GRDSIZ*((FLOAT(I-1)/FLOAT(NCOEF-KORDER+1)) &
)
20 CONTINUE
DO 30 I=1, KORDER - 1
 XKNOT(I) = XKNOT(KORDER)
 XKNOT(I+NCOEF+1) = XKNOT(NCOEF+1)
30 CONTINUE
! Compute BSLSQ fit.
! Construct the constraints for
! CONFIT.
DO 40 I=1, NXVAL
 XVAL(I) = GRDSIZ*FLOAT(I-1)/FLOAT(NXVAL-1)
 ITYPE(I) = 3
 IDER(I) = 1
 BL(I) = 0.0
40 CONTINUE
! Call CONFIT
NHARD = 0
CALL CONFIT (XDATA, FDATA, XVAL, NHARD, IDER, ITYPE, BL, BU, KORDER, &
 XKNOT, BSCNFIT, NCOEF=NCOEF)
! Compute the average error
! of 100 points in the interval.
ERRLSQ = 0.0
ERRNFT = 0.0
DO 50 I=1, 100
 X = GRDSIZ*FLOAT(I-1)/99.0
 ERRNFT = ERRNFT + ABS(F1(X)-BSVAL(X,KORDER,XKNOT,NCOEF,BSCNFIT) &
)
 ERRLSQ = ERRLSQ + ABS(F1(X)-BSVAL(X,KORDER,XKNOT,NCOEF,BSCLSQ) &
)
50 CONTINUE
! Print results
WRITE (NOUT,99998) ERRLSQ/100.0
WRITE (NOUT,99999) ERRNFT/100.0
!
99998 FORMAT (' Average error with BSLSQ fit: ', F8.5)
99999 FORMAT (' Average error with CONFIT fit: ', F8.5)
END

```

## Output

Average error with BSLSQ fit: 0.20250

Average error with CONFT fit: 0.14334

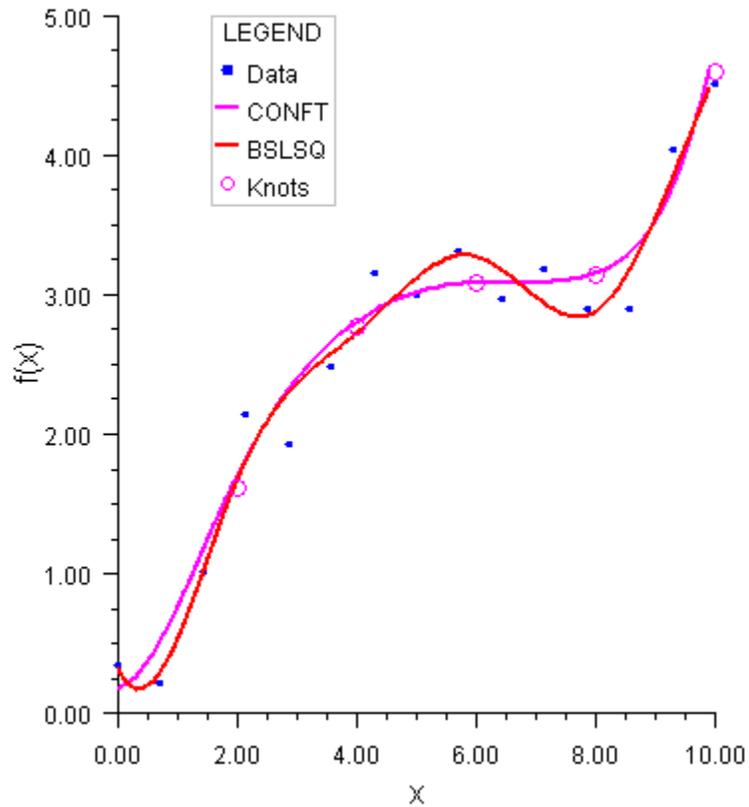


Figure 3- 8 CONFT vs. BSLSQ Forcing Monotonicity

## Additional Examples

### Example 2

We now try to recover the function

$$\frac{1}{1+x^4}$$

from noisy data. We first try the unconstrained least-squares fit using `BSLSQ`. Finding that fit somewhat unsatisfactory, we apply several constraints using `CONFT`. First, notice that the unconstrained fit oscillates through the true function at both ends of the interval. This is common for flat data. To remove this oscillation, we constrain the cubic spline to have zero second

derivative at the first and last four knots. This forces the cubic spline to reduce to a linear polynomial on the first and last three knot intervals. In addition, we constrain the fit (which we will call  $s$ ) as follows:

$$\begin{aligned} s(-7) &\geq 0 \\ \int_{-7}^7 s(x) dx &\leq 2.3 \\ s(-7) &= s(7) \end{aligned}$$

Notice that the last constraint was generated using the periodic option (requiring only the zeroth derivative to be periodic). We print the error for the two fits averaged over 100 equally spaced points.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KORDER, NCOEF, NDATA, NXVAL
PARAMETER (KORDER=4, NCOEF=13, NDATA=51, NXVAL=12)
!
INTEGER I, IDER(NXVAL), ITYPE(NXVAL), NHARPT, NOUT
REAL ABS, BL(NXVAL), BSCLSQ(NDATA), BSCNFT(NDATA), &
 BU(NXVAL), ERRLSQ, ERRNFT, F1, FDATA(NDATA), FLOAT, &
 GRDSIZ, WEIGHT(NDATA), X, XDATA(NDATA), &
 XKNOT(KORDER+NDATA), XVAL(NXVAL)
INTRINSIC ABS, FLOAT
!
F1(X) = 1.0/(1.0+X**4)
!
! Initialize random number generator
! and get output unit number.
CALL UMACH (2, NOUT)
CALL RNSET (234579)
!
! Use default weights of one.
!
! Compute original XDATA and FDATA
! with random noise.
GRDSIZ = 14.0
DO 10 I=1, NDATA
 XDATA(I) = GRDSIZ*((FLOAT(I-1)/FLOAT(NDATA-1))) - GRDSIZ/2.0
 FDATA(I) = RNUNF()
 FDATA(I) = F1(XDATA(I)) + 0.125*(FDATA(I)-.5)
10 CONTINUE
!
! Compute KNOTS
DO 20 I=1, NCOEF - KORDER + 2
 XKNOT(I+KORDER-1) = GRDSIZ*((FLOAT(I-1)/FLOAT(NCOEF-KORDER+1)) &
) - GRDSIZ/2.0
20 CONTINUE
DO 30 I=1, KORDER - 1
 XKNOT(I) = XKNOT(KORDER)
 XKNOT(I+NCOEF+1) = XKNOT(NCOEF+1)
30 CONTINUE
!
! Compute BSLSQ fit
CALL BSLSQ (XDATA, FDATA, KORDER, XKNOT, NCOEF, BSCLSQ)
!
! Construct the constraints for
! CONFIT
DO 40 I=1, 4

```

```

 XVAL(I) = XKNOT(KORDER+I-1)
 XVAL(I+4) = XKNOT(NCOEF-3+I)
 ITYPE(I) = 1
 ITYPE(I+4) = 1
 IDER(I) = 2
 IDER(I+4) = 2
 BL(I) = 0.0
 BL(I+4) = 0.0
40 CONTINUE
!
 XVAL(9) = -7.0
 ITYPE(9) = 3
 IDER(9) = 0
 BL(9) = 0.0
!
 XVAL(10) = -7.0
 ITYPE(10) = 2
 IDER(10) = -1
 BU(10) = 2.3
!
 XVAL(11) = 7.0
 ITYPE(11) = 2
 IDER(11) = -1
 BU(11) = 2.3
!
 XVAL(12) = -7.0
 ITYPE(12) = 10
 IDER(12) = 0
!
 Call CONF T
 CALL CONF T (XDATA, FDATA, XVAL, NHARPT, IDER, ITYPE, BL, BU, &
 KORDER, XKNOT, BSCNFT, NCOEF=NCOEF)
!
 Compute the average error
! of 100 points in the interval.
 ERRLSQ = 0.0
 ERRNFT = 0.0
 DO 50 I=1, 100
 X = GRDSIZ*FLOAT(I-1)/99.0 - GRDSIZ/2.0
 ERRNFT = ERRNFT + ABS(F1(X)-BSVAL(X,KORDER,XKNOT,NCOEF,BSCNFT) &
)
 ERRLSQ = ERRLSQ + ABS(F1(X)-BSVAL(X,KORDER,XKNOT,NCOEF,BSCLSQ) &
)
50 CONTINUE
!
 Print results
 WRITE (NOUT,99998) ERRLSQ/100.0
 WRITE (NOUT,99999) ERRNFT/100.0
!
99998 FORMAT (' Average error with BSLSQ fit: ', F8.5)
99999 FORMAT (' Average error with CONF T fit: ', F8.5)
END

```

## Output

Average error with BSLSQ fit: 0.01783  
Average error with CONFT fit: 0.01339

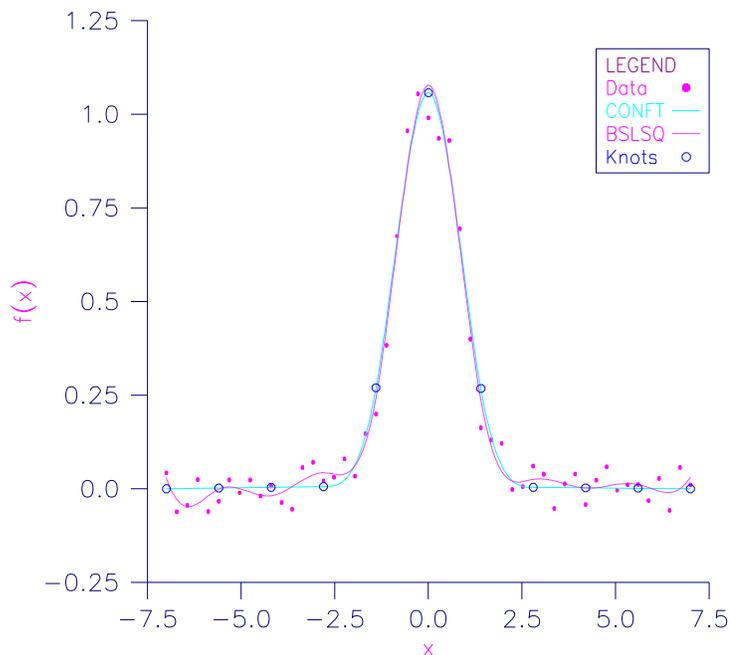


Figure 3-9 CONFT vs. BSLSQ Approximating  $1/(1+x^4)$

---

## BSLS2

Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

### Required Arguments

**XDATA** — Array of length `NXDATA` containing the data points in the `X`-direction. (Input)  
XDATA must be nondecreasing.

**YDATA** — Array of length `NYDATA` containing the data points in the `Y`-direction. (Input)  
YDATA must be nondecreasing.

**FDATA** — Array of size `NXDATA` by `NYDATA` containing the values on the `X` – `Y` grid to be interpolated. (Input)  
FDATA(`I`, `J`) contains the value at (XDATA(`I`), YDATA(`I`)).

**KXORD** — Order of the spline in the `X`-direction. (Input)

**KYORD** — Order of the spline in the `Y`-direction. (Input)

**XKNOT** — Array of length  $KXORD + NXCOEF$  containing the knots in the X-direction. (Input)  
XKNOT must be nondecreasing.

**YKNOT** — Array of length  $KYORD + NYCOEF$  containing the knots in the Y-direction. (Input)  
YKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXCOEF * NYCOEF$  that contains the tensor product B-spline coefficients. (Output)  
BSCOEF is treated internally as an array of size  $NXCOEF$  by  $NYCOEF$ .

### Optional Arguments

**NXDATA** — Number of data points in the X-direction. (Input)  
Default:  $NXDATA = \text{size}(XDATA,1)$ .

**NYDATA** — Number of data points in the Y-direction. (Input)  
Default:  $NYDATA = \text{size}(YDATA,1)$ .

**LDF** — Leading dimension of FDATA exactly as specified in the dimension statement of calling program. (Input)  
Default:  $LDF = \text{size}(FDATA,1)$ .

**NXCOEF** — Number of B-spline coefficients in the X-direction. (Input)  
Default:  $NXCOEF = \text{size}(XKNOT,1) - KXORD$ .

**NYCOEF** — Number of B-spline coefficients in the Y-direction. (Input)  
Default:  $NYCOEF = \text{size}(YKNOT,1) - KYORD$ .

**XWEIGH** — Array of length  $NXDATA$  containing the positive weights of XDATA. (Input)  
Default:  $XWEIGH = 1.0$ .

**YWEIGH** — Array of length  $NYDATA$  containing the positive weights of YDATA. (Input)  
Default:  $YWEIGH = 1.0$ .

### FORTRAN 90 Interface

Generic:    CALL BSLS2 (XDATA, YDATA, FDATA, KXORD, KYORD, XKNOT, YKNOT,  
                  BSCOEF [, ...])

Specific:    The specific interface names are S\_BSLS2 and D\_BSLS2.

### FORTRAN 77 Interface

Single:     CALL BSLS2 (NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF, KXORD, KYORD,  
                  XKNOT, YKNOT, NXCOEF, NYCOEF, XWEIGH, YWEIGH, BSCOEF)

Double:     The double precision name is DBSLS2.

## Description

The routine `BLS2` computes the coefficients of a tensor-product spline least-squares approximation to weighted tensor-product data. The input for this subroutine consists of data vectors to specify the tensor-product grid for the data, two vectors with the weights, the values of the surface on the grid, and the specification for the tensor-product spline. The grid is specified by the two vectors  $x = \text{XDATA}$  and  $y = \text{YDATA}$  of length  $n = \text{NXDATA}$  and  $m = \text{NYDATA}$ , respectively. A two-dimensional array  $f = \text{FDATA}$  contains the data values that are to be fit. The two vectors  $w_x = \text{XWEIGH}$  and  $w_y = \text{YWEIGH}$  contain the weights for the weighted least-squares problem. The information for the approximating tensor-product spline must also be provided. This information is contained in  $k_x = \text{KXORD}$ ,  $\mathbf{t}_x = \text{XKNOT}$ , and  $N = \text{NXCOEF}$  for the spline in the first variable, and in  $k_y = \text{KYORD}$ ,  $\mathbf{t}_y = \text{YKNOT}$  and  $M = \text{NYCOEF}$  for the spline in the second variable. The coefficients of the resulting tensor-product spline are returned in  $c = \text{BSCOEF}$ , which is an  $N * M$  array. The procedure computes coefficients by solving the normal equations in tensor-product form as discussed

in de Boor (1978, Chapter 17). The interested reader might also want to study the paper by E. Grosse (1980).

The final result produces coefficients  $c$  minimizing

$$\sum_{i=1}^n \sum_{j=1}^m w_x(i) w_y(j) \left[ \sum_{k=1}^N \sum_{l=1}^M c_{kl} B_{kl}(x_i, y_j) - f_{ij} \right]^2$$

where the function  $B_{kl}$  is the tensor-product of two B-splines of order  $k_x$  and  $k_y$ . Specifically, we have

$$B_{kl}(x, y) = B_{k, k_x, \mathbf{t}_x}(x) B_{l, k_y, \mathbf{t}_y}(y)$$

The spline

$$\sum_{k=1}^N \sum_{l=1}^M c_{kl} B_{kl}$$

can be evaluated using `BS2VL` and its partial derivatives can be evaluated using `BS2DR`.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LS2/DB2LS2`. The reference is:

```
CALL B2LS2 (NXDATA, XDATA, NYDATA, YDATA, FDATA, LDF,
 KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, XWEIGH,
 YWEIGH, BSCOEF, WK)
```

The additional argument is:

**WK** — Work array of length  $(\text{NXCOEF} + 1) * \text{NYDATA} + \text{KXORD} * \text{NXCOEF} + \text{KYORD} * \text{NYCOEF} + 3 * \text{MAX}(\text{KXORD}, \text{KYORD})$ .

2. Informational errors

| Type | Code |                                                                                                                |
|------|------|----------------------------------------------------------------------------------------------------------------|
| 3    | 14   | There may be less than one digit of accuracy in the least squares fit. Try using higher precision if possible. |
| 4    | 5    | Multiplicity of the knots cannot exceed the order of the spline.                                               |
| 4    | 6    | The knots must be nondecreasing.                                                                               |
| 4    | 7    | All weights must be greater than zero.                                                                         |
| 4    | 9    | The data point abscissae must be nondecreasing.                                                                |
| 4    | 10   | The smallest element of the data point array must be greater than or equal to the $K\_ORD$ th knot.            |
| 4    | 11   | The largest element of the data point array must be less than or equal to the $(N\_COEF + 1)$ st knot.         |

## Example

The data for this example arise from the function  $e^x \sin(x + y) + \varepsilon$  on the rectangle  $[0, 3] \times [0, 5]$ . Here,  $\varepsilon$  is a uniform random variable with range  $[-1, 1]$ . We sample this function on a  $100 \times 50$  grid and then try to recover it by using cubic splines in the  $x$  variable and quadratic splines in the  $y$  variable. We print out the values of the function  $e^x \sin(x + y)$  on a  $3 \times 5$  grid and compare these values with the values of the tensor-product spline that was computed using the IMSL routine BSL2.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER KXORD, KYORD, LDF, NXCOEF, NXDATA, NXVEC, NYCOEF, &
NYDATA, NYVEC
PARAMETER (KXORD=4, KYORD=3, NXCOEF=15, NXDATA=100, NXVEC=4, &
NYCOEF=7, NYDATA=50, NYVEC=6, LDF=NXDATA)
!
INTEGER I, J, NOUT
REAL BSCOEF(NXCOEF,NYCOEF), EXP, F, FDATA(NXDATA,NYDATA), &
FLOAT, RNOISE, SIN, VALUE(NXVEC,NYVEC), X, &
XDATA(NXDATA), XKNOT(NXCOEF+KXORD), XVEC(NXVEC), &
XWEIGH(NXDATA), Y, YDATA(NYDATA), &
YKNOT(NYCOEF+KYORD), YVEC(NYVEC), YWEIGH(NYDATA)
INTRINSIC EXP, FLOAT, SIN
!
F(X,Y) = EXP(X)*SIN(X+Y)
!
CALL RNSET (1234579)
!
DO 10 I=1, NXCOEF - KXORD + 2
 XKNOT(I+KXORD-1) = 3.0*(FLOAT(I-1)/FLOAT(NXCOEF-KXORD+1))
10 CONTINUE
XKNOT(NXCOEF+1) = XKNOT(NXCOEF+1) + 0.001
!
DO 20 I=1, KXORD - 1
 XKNOT(I) = XKNOT(KXORD)

```

```

 XKNOT(I+NXCOEF+1) = XKNOT(NXCOEF+1)
20 CONTINUE
! Set up Y knot sequence.
 DO 30 I=1, NYCOEF - KYORD + 2
 YKNOT(I+KYORD-1) = 5.0*(FLOAT(I-1)/FLOAT(NYCOEF-KYORD+1))
30 CONTINUE
 YKNOT(NYCOEF+1) = YKNOT(NYCOEF+1) + 0.001
! Stack knots.
 DO 40 I=1, KYORD - 1
 YKNOT(I) = YKNOT(KYORD)
 YKNOT(I+NYCOEF+1) = YKNOT(NYCOEF+1)
40 CONTINUE
! Set up X-grid.
 DO 50 I=1, NXDATA
 XDATA(I) = 3.0*(FLOAT(I-1)/FLOAT(NXDATA-1))
50 CONTINUE
! Set up Y-grid.
 DO 60 I=1, NYDATA
 YDATA(I) = 5.0*(FLOAT(I-1)/FLOAT(NYDATA-1))
60 CONTINUE
! Evaluate function on grid and
! introduce random noise in [1,-1].
 DO 70 I=1, NYDATA
 DO 70 J=1, NXDATA
 RNOISE = RNUNF()
 RNOISE = 2.0*RNOISE - 1.0
 FDATA(J,I) = F(XDATA(J),YDATA(I)) + RNOISE
70 CONTINUE
! Use default weights equal to 1.
!
! Compute least squares approximation.
 CALL BLS2 (XDATA, YDATA, FDATA, KXORD, KYORD, &
 XKNOT, YKNOT, BSCOEF)
! Get output unit number
 CALL UMACH (2, NOUT)
! Write heading
 WRITE (NOUT,99999)
! Print interpolated values
! on [0,3] x [0,5].
 DO 80 I=1, NXVEC
 XVEC(I) = FLOAT(I-1)
80 CONTINUE
 DO 90 I=1, NYVEC
 YVEC(I) = FLOAT(I-1)
90 CONTINUE
! Evaluate spline
 CALL BS2GD (0, 0, XVEC, YVEC, KXORD, KYORD, XKNOT, &
 YKNOT, BSCOEF, VALUE)
 DO 110 I=1, NXVEC
 DO 100 J=1, NYVEC
 WRITE (NOUT,'(5F15.4)') XVEC(I), YVEC(J), &
 F(XVEC(I),YVEC(J)), VALUE(I,J), &
 (F(XVEC(I),YVEC(J))-VALUE(I,J))
100 CONTINUE
110 CONTINUE
99999 FORMAT (13X, 'X', 14X, 'Y', 10X, 'F(X,Y)', 9X, 'S(X,Y)', 10X, &
 'Error')

```

END

## Output

| X      | Y      | F(X, Y)  | S(X, Y)  | Error   |
|--------|--------|----------|----------|---------|
| 0.0000 | 0.0000 | 0.0000   | 0.2782   | -0.2782 |
| 0.0000 | 1.0000 | 0.8415   | 0.7762   | 0.0653  |
| 0.0000 | 2.0000 | 0.9093   | 0.8203   | 0.0890  |
| 0.0000 | 3.0000 | 0.1411   | 0.1391   | 0.0020  |
| 0.0000 | 4.0000 | -0.7568  | -0.5705  | -0.1863 |
| 0.0000 | 5.0000 | -0.9589  | -1.0290  | 0.0701  |
| 1.0000 | 0.0000 | 2.2874   | 2.2678   | 0.0196  |
| 1.0000 | 1.0000 | 2.4717   | 2.4490   | 0.0227  |
| 1.0000 | 2.0000 | 0.3836   | 0.4947   | -0.1111 |
| 1.0000 | 3.0000 | -2.0572  | -2.0378  | -0.0195 |
| 1.0000 | 4.0000 | -2.6066  | -2.6218  | 0.0151  |
| 1.0000 | 5.0000 | -0.7595  | -0.7274  | -0.0321 |
| 2.0000 | 0.0000 | 6.7188   | 6.6923   | 0.0265  |
| 2.0000 | 1.0000 | 1.0427   | 0.8492   | 0.1935  |
| 2.0000 | 2.0000 | -5.5921  | -5.5885  | -0.0035 |
| 2.0000 | 3.0000 | -7.0855  | -7.0955  | 0.0099  |
| 2.0000 | 4.0000 | -2.0646  | -2.1588  | 0.0942  |
| 2.0000 | 5.0000 | 4.8545   | 4.7339   | 0.1206  |
| 3.0000 | 0.0000 | 2.8345   | 2.5971   | 0.2373  |
| 3.0000 | 1.0000 | -15.2008 | -15.1079 | -0.0929 |
| 3.0000 | 2.0000 | -19.2605 | -19.1698 | -0.0907 |
| 3.0000 | 3.0000 | -5.6122  | -5.5820  | -0.0302 |
| 3.0000 | 4.0000 | 13.1959  | 12.6659  | 0.5300  |
| 3.0000 | 5.0000 | 19.8718  | 20.5170  | -0.6452 |

---

## BSLS3

Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

### Required Arguments

***XDATA*** — Array of length *NXDATA* containing the data points in the *x*-direction. (Input)  
*XDATA* must be nondecreasing.

***YDATA*** — Array of length *NYDATA* containing the data points in the *y*-direction. (Input)  
*YDATA* must be nondecreasing.

***ZDATA*** — Array of length *NZDATA* containing the data points in the *z*-direction. (Input)  
*ZDATA* must be nondecreasing.

***FDATA*** — Array of size *NXDATA* by *NYDATA* by *NZDATA* containing the values to be interpolated. (Input)  
*FDATA*(*I*, *J*, *K*) contains the value at (*XDATA*(*I*), *YDATA*(*J*), *ZDATA*(*K*)).

***KXORD*** — Order of the spline in the *x*-direction. (Input)

**KYORD** — Order of the spline in the  $y$ -direction. (Input)

**KZORD** — Order of the spline in the  $z$ -direction. (Input)

**XKNOT** — Array of length  $KXORD + NXCOEF$  containing the knots in the  $x$ -direction. (Input)  
XKNOT must be nondecreasing.

**YKNOT** — Array of length  $KYORD + NYCOEF$  containing the knots in the  $y$ -direction. (Input)  
YKNOT must be nondecreasing.

**ZKNOT** — Array of length  $KZORD + NZCOEF$  containing the knots in the  $z$ -direction. (Input)  
ZKNOT must be nondecreasing.

**BSCOEF** — Array of length  $NXCOEF * NYCOEF * NZCOEF$  that contains the tensor product B-spline coefficients. (Output)

### Optional Arguments

**NXDATA** — Number of data points in the  $x$ -direction. (Input)  
NXDATA must be greater than or equal to  $NXCOEF$ .  
Default:  $NXDATA = \text{size}(XDATA, 1)$ .

**NYDATA** — Number of data points in the  $y$ -direction. (Input)  
NYDATA must be greater than or equal to  $NYCOEF$ .  
Default:  $NYDATA = \text{size}(YDATA, 1)$ .

**NZDATA** — Number of data points in the  $z$ -direction. (Input)  
NZDATA must be greater than or equal to  $NZCOEF$ .  
Default:  $NZDATA = \text{size}(ZDATA, 1)$ .

**LDFDAT** — Leading dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFDAT = \text{size}(FDATA, 1)$ .

**MDFDAT** — Second dimension of  $FDATA$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDFDAT = \text{size}(FDATA, 2)$ .

**NXCOEF** — Number of B-spline coefficients in the  $x$ -direction. (Input)  
Default:  $NXCOEF = \text{size}(XKNOT, 1) - KXORD$ .

**NYCOEF** — Number of B-spline coefficients in the  $y$ -direction. (Input)  
Default:  $NYCOEF = \text{size}(YKNOT, 1) - KYORD$ .

**NZCOEF** — Number of B-spline coefficients in the  $z$ -direction. (Input)  
Default:  $NZCOEF = \text{size}(ZKNOT, 1) - KZORD$ .

**XWEIGH** — Array of length  $NXDATA$  containing the positive weights of  $XDATA$ . (Input)  
Default:  $XWEIGH = 1.0$ .

***YWEIGH*** — Array of length NYDATA containing the positive weights of YDATA. (Input)  
 Default: YWEIGH = 1.0.

***ZWEIGH*** — Array of length NZDATA containing the positive weights of ZDATA. (Input)  
 Default: ZWEIGH = 1.0.

### **FORTRAN 90 Interface**

Generic: CALL BSLS3 (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, BSCOEF [, ...])

Specific: The specific interface names are S\_BSLS3 and D\_BSLS3.

### **FORTRAN 77 Interface**

Single: CALL BSLS3 (NXDATA, XDATA, NYDATA, YDATA, NZDATA, ZDATA, FDATA, LDFDAT, MDFDAT, KXORD, KYORD, KZORD, XKNOT, YKNOT, ZKNOT, NXCOEF, NYCOEF, NZCOEF, XWEIGH, YWEIGH, ZWEIGH, BSCOEF)

Double: The double precision name is DBSLS3.

### **Description**

The routine BSLS3 computes the coefficients of a tensor-product spline least-squares approximation to weighted tensor-product data. The input for this subroutine consists of data vectors to specify the tensor-product grid for the data, three vectors with the weights, the values of the surface on the grid, and the specification for the tensor-product spline. The grid is specified by the three vectors  $x = XDATA$ ,  $y = YDATA$ , and  $z = ZDATA$  of length  $k = NXDATA$ ,  $l = NYDATA$ , and  $m = NYDATA$ , respectively. A three-dimensional array  $f = FDATA$  contains the data values which are to be fit. The three vectors  $w_x = XWEIGH$ ,  $w_y = YWEIGH$ , and  $w_z = ZWEIGH$  contain the weights for the weighted least-squares problem. The information for the approximating tensor-product spline must also be provided. This information is contained in  $k_x = KXORD$ ,  $t_x = XKNOT$ , and  $K = NXCOEF$  for the spline in the first variable, in  $k_y = KYORD$ ,  $t_y = YKNOT$  and  $L = NYCOEF$  for the spline in the second variable, and in  $k_z = KZORD$ ,  $t_z = ZKNOT$  and  $M = NZCOEF$  for the spline in the third variable.

The coefficients of the resulting tensor product spline are returned in  $c = BSCOEF$ , which is an  $K \times L \times M$  array. The procedure computes coefficients by solving the normal equations in tensor-product form as discussed in de Boor (1978, Chapter 17). The interested reader might also want to study the paper by E. Grosse (1980).

The final result produces coefficients  $c$  minimizing

$$\sum_{i=1}^k \sum_{j=1}^l \sum_{p=1}^m w_x(i) w_y(j) w_z(p) \left[ \sum_{s=1}^K \sum_{t=1}^L \sum_{u=1}^M c_{stu} B_{stu}(x_i, y_j, z_p) - f_{ijp} \right]^2$$

where the function  $B_{stu}$  is the tensor-product of three B-splines of order  $k_x$ ,  $k_y$ , and  $k_z$ . Specifically, we have

$$B_{stu}(x, y, z) = B_{s,k_x,t_x}(x) B_{t,k_y,t_y}(y) B_{u,k_z,t_z}(z)$$

The spline

$$\sum_{s=1}^K \sum_{t=1}^L \sum_{u=1}^M c_{stu} B_{stu}$$

can be evaluated at one point using [BS3VL](#) and its partial derivatives can be evaluated using [BS3DR](#). If the values on a grid are desired then we recommend [BS3GD](#).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LS3/DB2LS3`. The reference is:

```
CALL B2LS3 (NXDATA, XDATA, NYDATA, NZDATA, ZDATA, YDATA,
 FDATA, LDFDAT, KXORD, KYORD, KZORD, XKNOT, YKNOT,
 ZKNOT, NXCOEF, NYCOEF, NZCOEF, XWEIGH, YWEIGH,
 ZWEIGH, BSCOEF, WK)
```

The additional argument is:

**WK** — Work array of length  $\text{NYCOEF} * (\text{NZDATA} + \text{KYORD} + \text{NZCOEF}) + \text{NZDATA} * (1 + \text{NYDATA}) + \text{NXCOEF} * (\text{KXORD} + \text{NYDATA} * \text{NZDATA}) + \text{KZORD} * \text{NZCOEF} + 3 * \text{MAX0}(\text{KXORD}, \text{KYORD}, \text{KZORD})$ .

2. Informational errors

| Type | Code |                                                                                                                |
|------|------|----------------------------------------------------------------------------------------------------------------|
| 3    | 13   | There may be less than one digit of accuracy in the least squares fit. Try using higher precision if possible. |
| 4    | 7    | Multiplicity of knots cannot exceed the order of the spline.                                                   |
| 4    | 8    | The knots must be nondecreasing.                                                                               |
| 4    | 9    | All weights must be greater than zero.                                                                         |
| 4    | 10   | The data point abscissae must be nondecreasing.                                                                |
| 4    | 11   | The smallest element of the data point array must be greater than or equal to the $K\_ORD$ th knot.            |
| 4    | 12   | The largest element of the data point array must be less than or equal to the $(N\_COEF + 1)$ st knot.         |

## Example

The data for this example arise from the function  $e^{(y-z)} \sin(x+y) + \epsilon$  on the rectangle  $[0, 3] \times [0, 2] \times [0, 1]$ . Here,  $\epsilon$  is a uniform random variable with range  $[-.5, .5]$ . We sample this

function on a  $4 \times 3 \times 2$  grid and then try to recover it by using tensor-product cubic splines in all variables. We print out the values of the function  $e^{(y-z)} \sin(x+y)$  on a  $4 \times 3 \times 2$  grid and compare these values with the values of the tensor-product spline that was computed using the IMSL routine BSLS3.

```

USE BSLS3_INT
USE RNSET_INT
USE RNUNF_INT
USE UMACH_INT
USE BS3GD_INT

IMPLICIT NONE
INTEGER KXORD, KYORD, KZORD, LDFDAT, MDFDAT, NXCOEF, NXDATA, &
NXVAL, NYCOEF, NYDATA, NYVAL, NZCOEF, NZDATA, NZVAL
PARAMETER (KXORD=4, KYORD=4, KZORD=4, NXCOEF=8, NXDATA=15, &
NXVAL=4, NYCOEF=8, NYDATA=15, NYVAL=3, NZCOEF=8, &
NZDATA=15, NZVAL=2, LDFDAT=NXDATA, MDFDAT=NYDATA)
!
INTEGER I, J, K, NOUT
REAL BSCOEFF(NXCOEF,NYCOEF,NZCOEF), EXP, F, &
FDATA(NXDATA,NYDATA,NZDATA), FLOAT, RNOISE, &
SIN, SPXYZ(NXVAL,NYVAL,NZVAL), X, XDATA(NXDATA), &
XKNOT(NXCOEF+KXORD), XVAL(NXVAL), XWEIGH(NXDATA), Y, &
YDATA(NYDATA), YKNOT(NYCOEF+KYORD), YVAL(NYVAL), &
YWEIGH(NYDATA), Z, ZDATA(NZDATA), &
ZKNOT(NZCOEF+KZORD), ZVAL(NZVAL), ZWEIGH(NZDATA)
INTRINSIC EXP, FLOAT, SIN
!
! Define a function
F(X,Y,Z) = EXP(Y-Z)*SIN(X+Y)
!
CALL RNSET (1234579)
CALL UMACH (2, NOUT)
!
! Set up knot sequences
! X-knots
DO 10 I=1, NXCOEF - KXORD + 2
XKNOT(I+KXORD-1) = 3.0*(FLOAT(I-1)/FLOAT(NXCOEF-KXORD+1))
10 CONTINUE
DO 20 I=1, KXORD - 1
XKNOT(I) = XKNOT(KXORD)
XKNOT(I+NXCOEF+1) = XKNOT(NXCOEF+1)
20 CONTINUE
!
! Y-knots
DO 30 I=1, NYCOEF - KYORD + 2
YKNOT(I+KYORD-1) = 2.0*(FLOAT(I-1)/FLOAT(NYCOEF-KYORD+1))
30 CONTINUE
DO 40 I=1, KYORD - 1
YKNOT(I) = YKNOT(KYORD)
YKNOT(I+NYCOEF+1) = YKNOT(NYCOEF+1)
40 CONTINUE
!
! Z-knots
DO 50 I=1, NZCOEF - KZORD + 2
ZKNOT(I+KZORD-1) = 1.0*(FLOAT(I-1)/FLOAT(NZCOEF-KZORD+1))
50 CONTINUE
DO 60 I=1, KZORD - 1

```

```

 ZKNOT(I) = ZKNOT(KZORD)
 ZKNOT(I+NZCOEF+1) = ZKNOT(NZCOEF+1)
60 CONTINUE
! Set up X-grid.
 DO 70 I=1, NXDATA
 XDATA(I) = 3.0*(FLOAT(I-1)/FLOAT(NXDATA-1))
70 CONTINUE
! Set up Y-grid.
 DO 80 I=1, NYDATA
 YDATA(I) = 2.0*(FLOAT(I-1)/FLOAT(NYDATA-1))
80 CONTINUE
! Set up Z-grid
 DO 90 I=1, NZDATA
 ZDATA(I) = 1.0*(FLOAT(I-1)/FLOAT(NZDATA-1))
90 CONTINUE
! Evaluate the function on the grid
! and add noise.
 DO 100 I=1, NXDATA
 DO 100 J=1, NYDATA
 DO 100 K=1, NZDATA
 RNOISE = RNUNF()
 RNOISE = RNOISE - 0.5
 FDATA(I,J,K) = F(XDATA(I),YDATA(J),ZDATA(K)) + RNOISE
100 CONTINUE
! Use default weights equal to 1.0
!
! Compute least-squares
 CALL BSLS3 (XDATA, YDATA, ZDATA, FDATA, KXORD, KYORD, KZORD, XKNOT, &
 YKNOT, ZKNOT, BSCOEF)
! Set up grid for evaluation.
 DO 110 I=1, NXVAL
 XVAL(I) = FLOAT(I-1)
110 CONTINUE
 DO 120 I=1, NYVAL
 YVAL(I) = FLOAT(I-1)
120 CONTINUE
 DO 130 I=1, NZVAL
 ZVAL(I) = FLOAT(I-1)
130 CONTINUE
! Evaluate on the grid.
 CALL BS3GD (0, 0, 0, XVAL, YVAL, ZVAL, KXORD, KYORD, KZORD, XKNOT, &
 YKNOT, ZKNOT, BSCOEF, SPXYZ)
! Print results.
 WRITE (NOUT,99998)
 DO 140 I=1, NXVAL
 DO 140 J=1, NYVAL
 DO 140 K=1, NZVAL
 WRITE (NOUT,99999) XVAL(I), YVAL(J), ZVAL(K), &
 F(XVAL(I),YVAL(J),ZVAL(K)), &
 SPXYZ(I,J,K), F(XVAL(I),YVAL(J),ZVAL(K) &
) - SPXYZ(I,J,K)
140 CONTINUE
99998 FORMAT (8X, 'X', 9X, 'Y', 9X, 'Z', 6X, 'F(X,Y,Z)', 3X, &
 'S(X,Y,Z)', 4X, 'Error')
99999 FORMAT (' ', 3F10.3, 3F11.4)
END

```

## Output

| X     | Y     | Z     | F(X, Y, Z) | S(X, Y, Z) | Error   |
|-------|-------|-------|------------|------------|---------|
| 0.000 | 0.000 | 0.000 | 0.0000     | 0.1987     | -0.1987 |
| 0.000 | 0.000 | 1.000 | 0.0000     | 0.1447     | -0.1447 |
| 0.000 | 1.000 | 0.000 | 2.2874     | 2.2854     | 0.0019  |
| 0.000 | 1.000 | 1.000 | 0.8415     | 1.0557     | -0.2142 |
| 0.000 | 2.000 | 0.000 | 6.7188     | 6.4704     | 0.2484  |
| 0.000 | 2.000 | 1.000 | 2.4717     | 2.2054     | 0.2664  |
| 1.000 | 0.000 | 0.000 | 0.8415     | 0.8779     | -0.0365 |
| 1.000 | 0.000 | 1.000 | 0.3096     | 0.2571     | 0.0524  |
| 1.000 | 1.000 | 0.000 | 2.4717     | 2.4015     | 0.0703  |
| 1.000 | 1.000 | 1.000 | 0.9093     | 0.8995     | 0.0098  |
| 1.000 | 2.000 | 0.000 | 1.0427     | 1.1330     | -0.0902 |
| 1.000 | 2.000 | 1.000 | 0.3836     | 0.4951     | -0.1115 |
| 2.000 | 0.000 | 0.000 | 0.9093     | 0.8269     | 0.0824  |
| 2.000 | 0.000 | 1.000 | 0.3345     | 0.3258     | 0.0087  |
| 2.000 | 1.000 | 0.000 | 0.3836     | 0.3564     | 0.0272  |
| 2.000 | 1.000 | 1.000 | 0.1411     | 0.1905     | -0.0494 |
| 2.000 | 2.000 | 0.000 | -5.5921    | -5.5362    | -0.0559 |
| 2.000 | 2.000 | 1.000 | -2.0572    | -1.9659    | -0.0913 |
| 3.000 | 0.000 | 0.000 | 0.1411     | 0.4841     | -0.3430 |
| 3.000 | 0.000 | 1.000 | 0.0519     | -0.4257    | 0.4776  |
| 3.000 | 1.000 | 0.000 | -2.0572    | -1.9710    | -0.0862 |
| 3.000 | 1.000 | 1.000 | -0.7568    | -0.8479    | 0.0911  |
| 3.000 | 2.000 | 0.000 | -7.0855    | -7.0957    | 0.0101  |
| 3.000 | 2.000 | 1.000 | -2.6066    | -2.1650    | -0.4416 |

---

## CSSD

Smooths one-dimensional data by error detection.

### Required Arguments

*XDATA* — Array of length *NDATA* containing the abscissas of the data points. (Input)

*FDATA* — Array of length *NDATA* containing the ordinates (function values) of the data points. (Input)

*DIS* — Proportion of the distance the ordinate in error is moved to its interpolating curve. (Input)

It must be in the range 0.0 to 1.0. A suggested value for *DIS* is one.

*SC* — Stopping criterion. (Input)

*SC* should be greater than or equal to zero. A suggested value for *SC* is zero.

*MAXIT* — Maximum number of iterations allowed. (Input)

*SDATA* — Array of length *NDATA* containing the smoothed data. (Output)

## Optional Arguments

*N*DATA — Number of data points. (Input)  
Default: *N*DATA = size (XDATA,1).

## FORTRAN 90 Interface

Generic: CALL CSSED (XDATA, FDATA, DIS, SC, MAXIT, SDATA [, ...])

Specific: The specific interface names are S\_CSSED and D\_CSSED.

## FORTRAN 77 Interface

Single: CALL CSSED (NDATA, XDATA, FDATA, DIS, SC, MAXIT, SDATA)

Double: The double precision name is DCSSED.

## Description

The routine CSSED is designed to smooth a data set that is mildly contaminated with isolated errors. In general, the routine will not work well if more than 25% of the data points are in error. The routine CSSED is based on an algorithm of Guerra and Tapia (1974).

Setting *N*DATA = *n*, FDATA = *f*, SDATA = *s* and XDATA = *x*, the algorithm proceeds as follows. Although the user need not input an ordered XDATA sequence, we will assume that *x* is increasing for simplicity. The algorithm first sorts the XDATA values into an increasing sequence and then continues. A cubic spline interpolant is computed for each of the 6-point data sets (initially setting *s* = *f*)

$$(x_j, s_j) \quad j = i - 3, \dots, i + 3 \quad j \neq i,$$

where  $i = 4, \dots, n - 3$  using CSAKM. For each *i* the interpolant, which we will call  $S_i$ , is compared with the current value of  $s_i$ , and a ‘point energy’ is computed as

$$pe_i = S_i(x_i) - s_i$$

Setting *sc* = SC, the algorithm terminates either if MAXIT iterations have taken place or if

$$|pe_i| \leq sc (x_{i+3} - x_{i-3}) / 6 \quad i = 4, \dots, n - 3$$

If the above inequality is violated for any *i*, then we update the *i*-th element of *s* by setting  $s_i = s_i + d(pe_i)$ , where *d* = DIS. Note that neither the first three nor the last three data points are changed. Thus, if these points are inaccurate, care must be taken to interpret the results.

The choice of the parameters *d*, *sc* and MAXIT are crucial to the successful usage of this subroutine. If the user has specific information about the extent of the contamination, then he should choose the parameters as follows: *d* = 1, *sc* = 0 and MAXIT to be the number of data points in error. On the other hand, if no such specific information is available, then choose *d* = .5, MAXIT  $\leq 2n$ , and

$$sc = .5 \frac{\max s - \min s}{(x_n - x_1)}$$

In any case, we would encourage the user to experiment with these values.

## Comments

1. Workspace may be explicitly provided, if desired, by use of C2SED/DC2SED. The reference is:

```
CALL C2SED (NDATA, XDATA, FDATA, DIS, SC, MAXIT, DATA, WK,
 IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $4 * \text{NDATA} + 30$ .

**IWK** — Work array of length  $2 * \text{NDATA}$ .

2. Informational error

Type Code

3 1 The maximum number of iterations allowed has been reached.

3. The arrays FDATA and SDATA may be the same.

## Example

We take 91 uniform samples from the function  $5 + (5 + t^2 \sin t)/t$  on the interval [1, 10]. Then, we contaminate 10 of the samples and try to recover the original function values.

```
USE CSSED_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=91)
!
INTEGER I, MAXIT, NOUT, ISB(10)
REAL DIS, F, FDATA(91), SC, SDATA(91), SIN, X, XDATA(91), &
 RNOISE(10)
INTRINSIC SIN
!
DATA ISB/6, 17, 26, 34, 42, 49, 56, 62, 75, 83/
DATA RNOISE/2.5, -3.0, -2.0, 2.5, 3.0, -2.0, -2.5, 2.0, -2.0, 3.0/
!
F(X) = (X*X*SIN(X)+5.0)/X + 5.0
!
! EX. #1; No specific information
! available
DIS = 0.5
SC = 0.56
```

```

MAXIT = 182
!
! Set values for XDATA and FDATA
XDATA(1) = 1.0
FDATA(1) = F(XDATA(1))
DO 10 I=2, NDATA
 XDATA(I) = XDATA(I-1) + .1
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Contaminate the data
DO 20 I=1, 10
 FDATA(ISB(I)) = FDATA(ISB(I)) + RNOISE(I)
20 CONTINUE
!
! Smooth data
CALL CSSED (XDATA, FDATA, DIS, SC, MAXIT, SDATA)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Write heading
WRITE (NOUT,99997)
!
! Write data
DO 30 I=1, 10
 WRITE (NOUT,99999) F(XDATA(ISB(I))), FDATA(ISB(I)), &
 SDATA(ISB(I))
30 CONTINUE
!
! EX. #2; Specific information
! available
DIS = 1.0
SC = 0.0
MAXIT = 10
!
! A warning message is produced
! because the maximum number of
! iterations is reached.
!
! Smooth data
CALL CSSED (XDATA, FDATA, DIS, SC, MAXIT, SDATA)
!
! Write heading
WRITE (NOUT,99998)
!
! Write data
DO 40 I=1, 10
 WRITE (NOUT,99999) F(XDATA(ISB(I))), FDATA(ISB(I)), &
 SDATA(ISB(I))
40 CONTINUE
!
99997 FORMAT (' Case A - No specific information available', /, &
 ' F(X) F(X)+NOISE SDATA(X)', /)
99998 FORMAT (' Case B - Specific information available', /, &
 ' F(X) F(X)+NOISE SDATA(X)', /)
99999 FORMAT (' ', F7.3, 8X, F7.3, 11X, F7.3)
END

```

## Output

```

Case A - No specific information available
F(X) F(X)+NOISE SDATA(X)

9.830 12.330 9.870
8.263 5.263 8.215
5.201 3.201 5.168

```

|        |        |        |
|--------|--------|--------|
| 2.223  | 4.723  | 2.264  |
| 1.259  | 4.259  | 1.308  |
| 3.167  | 1.167  | 3.138  |
| 7.167  | 4.667  | 7.131  |
| 10.880 | 12.880 | 10.909 |
| 12.774 | 10.774 | 12.708 |
| 7.594  | 10.594 | 7.639  |

\*\*\* WARNING ERROR 1 from CSSED. Maximum number of iterations limit MAXIT  
 \*\*\* =10 exceeded. The best answer found is returned.

Case B - Specific information available

| F(X)   | F(X)+NOISE | SDATA(X) |
|--------|------------|----------|
| 9.830  | 12.330     | 9.831    |
| 8.263  | 5.263      | 8.262    |
| 5.201  | 3.201      | 5.199    |
| 2.223  | 4.723      | 2.225    |
| 1.259  | 4.259      | 1.261    |
| 3.167  | 1.167      | 3.170    |
| 7.167  | 4.667      | 7.170    |
| 10.880 | 12.880     | 10.878   |
| 12.774 | 10.774     | 12.770   |
| 7.594  | 10.594     | 7.592    |

---

## CSSMH

Computes a smooth cubic spline approximation to noisy data.

### Required Arguments

**XDATA** — Array of length *NDATA* containing the data point abscissas. (Input)  
*XDATA* must be distinct.

**FDATA** — Array of length *NDATA* containing the data point ordinates. (Input)

**SMPAR** — A nonnegative number which controls the smoothing. (Input)  
 The spline function *S* returned is such that the sum from *I* = 1 to *NDATA* of  
 $((S(XDATA(I))FDATA(I)) / WEIGHT(I))^{*2}$  is less than or equal to *SMPAR*. It is  
 recommended that *SMPAR* lie in the confidence interval of this sum, i.e.,  
 $NDATA - SQRT(2 * NDATA) .LE. SMPAR .LE. NDATA + SQRT(2 * NDATA)$ .

**BREAK** — Array of length *NDATA* containing the breakpoints for the piecewise cubic  
 representation. (Output)

**CSCOE** — Matrix of size 4 by *NDATA* containing the local coefficients of the cubic pieces.  
 (Output)

## Optional Arguments

**NDATA** — Number of data points. (Input)

NDATA must be at least 2.

Default: NDATA = size (XDATA,1).

**WEIGHT** — Array of length NDATA containing estimates of the standard deviations of FDATA. (Input)

All elements of WEIGHT must be positive.

Default: WEIGHT = 1.0.

## FORTRAN 90 Interface

Generic: CALL CSSMH (XDATA, FDATA, SMPAR, BREAK, CSCOEFF [, ...])

Specific: The specific interface names are S\_CSSMH and D\_CSSMH.

## FORTRAN 77 Interface

Single: CALL CSSMH (NDATA, XDATA, FDATA, WEIGHT, SMPAR, BREAK, CSCOEFF)

Double: The double precision name is DCSSMH.

## Description

The routine CSSMH is designed to produce a  $C^2$  cubic spline approximation to a data set in which the function values are noisy. This spline is called a *smoothing spline*. It is a natural cubic spline with knots at all the data abscissas  $x = XDATA$ , but it does *not* interpolate the data  $(x_i, f_i)$ . The smoothing spline  $S$  is the unique  $C^2$  function which minimizes

$$\int_a^b S''(x)^2 dx$$

subject to the constraint

$$\sum_{i=1}^N \left| \frac{S(x_i) - f_i}{w_i} \right|^2 \leq \sigma$$

where  $w = WEIGHT$ ,  $\sigma = SMPAR$  is the smoothing parameter, and  $N = NDATA$ .

Recommended values for  $\sigma$  depend on the weights  $w$ . If an estimate for the standard deviation of the error in the value  $f_i$  is available, then  $w_i$  should be set to this value and the smoothing parameter  $\sigma$  should be chosen in the confidence interval corresponding to the left side of the above inequality. That is,

$$N - \sqrt{2N} \leq \sigma \leq N + \sqrt{2N}$$

The routine CSSMH is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pages 235–243).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2SMH/DC2SMH`. The reference is:

```
CALL C2SMH (NDATA, XDATA, FDATA, WEIGHT, SMPAR, BREAK,
 CSCOE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work array of length  $8 * \text{NDATA} + 5$ .

**IWK** — Work array of length `NDATA`.

2. Informational errors

Type Code

|   |   |                                                                                        |
|---|---|----------------------------------------------------------------------------------------|
| 3 | 1 | The maximum number of iterations has been reached. The best approximation is returned. |
| 4 | 3 | All weights must be greater than zero.                                                 |

3. The cubic spline can be evaluated using `CSVAL`; its derivative can be evaluated using `CSDER`.

## Example

In this example, function values are contaminated by adding a small “random” amount to the correct values. The routine `CSSMH` is used to approximate the original, uncontaminated data.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=300)
!
INTEGER I, NOUT
REAL BREAK(NDATA), CSCOE(4,NDATA), ERROR, F, &
 FDATA(NDATA), FLOAT, FVAL, SDEV, SMPAR, SQRT, &
 SVAL, WEIGHT(NDATA), X, XDATA(NDATA), XT, RN
INTRINSIC FLOAT, SQRT
!
F(X) = 1.0/(.1+(3.0*(X-1.0))**4)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = 3.0*(FLOAT(I-1)/FLOAT(NDATA-1))
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Set the random number seed
CALL RNSET (1234579)
!
! Contaminate the data
DO 20 I=1, NDATA
```

```

 RN = RNUNF()
 FDATA(I) = FDATA(I) + 2.0*RN - 1.0
20 CONTINUE
! Set the WEIGHT vector
 SDEV = 1.0/SQRT(3.0)
 CALL SSET (NDATA, SDEV, WEIGHT, 1)
 SMPAR = NDATA
! Smooth the data
 CALL CSSMH (XDATA, FDATA, SMPAR, BREAK, CSCOEf, WEIGHT=WEIGHT)
! Get output unit number
 CALL UMACH (2, NOUT)
! Write heading
 WRITE (NOUT,99999)
! Print 10 values of the function.
 DO 30 I=1, 10
 XT = 90.0*(FLOAT(I-1)/FLOAT(NDATA-1))
! Evaluate the spline
 SVAL = CSVAL(XT,BREAK,CSCOEf)
 FVAL = F(XT)
 ERROR = SVAL - FVAL
 WRITE (NOUT,'(4F15.4)') XT, FVAL, SVAL, ERROR
30 CONTINUE
!
99999 FORMAT (12X, 'X', 9X, 'Function', 7X, 'Smoothed', 10X,&
 'Error')
END

```

## Output

| X      | Function | Smoothed | Error   |
|--------|----------|----------|---------|
| 0.0000 | 0.0123   | 0.1118   | 0.0995  |
| 0.3010 | 0.0514   | 0.0646   | 0.0131  |
| 0.6020 | 0.4690   | 0.2972   | -0.1718 |
| 0.9030 | 9.3312   | 8.7022   | -0.6289 |
| 1.2040 | 4.1611   | 4.7887   | 0.6276  |
| 1.5050 | 0.1863   | 0.2718   | 0.0856  |
| 1.8060 | 0.0292   | 0.1408   | 0.1116  |
| 2.1070 | 0.0082   | 0.0826   | 0.0743  |
| 2.4080 | 0.0031   | 0.0076   | 0.0045  |
| 2.7090 | 0.0014   | -0.1789  | -0.1803 |

---

## CSSCV

Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.

### Required Arguments

**XDATA** — Array of length *NDATA* containing the data point abscissas. (Input) *XDATA* must be distinct.

**FDATA** — Array of length *NDATA* containing the data point ordinates. (Input)

**IEQUAL** — A flag alerting the subroutine that the data is equally spaced. (Input)

**BREAK** — Array of length `NDATA` containing the breakpoints for the piecewise cubic representation. (Output)

**CSCOEFL** — Matrix of size 4 by `NDATA` containing the local coefficients of the cubic pieces. (Output)

### Optional Arguments

**NDATA** — Number of data points. (Input)

`NDATA` must be at least 3.

Default: `NDATA = size (XDATA,1)`.

### FORTRAN 90 Interface

Generic: `CALL CSSCV (XDATA, FDATA, IEQUAL, BREAK, CSCOEFL [ , ... ])`

Specific: The specific interface names are `S_CSSCV` and `D_CSSCV`.

### FORTRAN 77 Interface

Single: `CALL CSSCV (NDATA, XDATA, FDATA, IEQUAL, BREAK, CSCOEFL)`

Double: The double precision name is `DCSSCV`.

### Description

The routine `CSSCV` is designed to produce a  $C^2$  cubic spline approximation to a data set in which the function values are noisy. This spline is called a *smoothing spline*. It is a natural cubic spline with knots at all the data abscissas  $x = XDATA$ , but it does *not* interpolate the data  $(x_i, f_i)$ . The smoothing spline  $S_\sigma$  is the unique  $C^2$  function that minimizes

$$\int_a^b S_\sigma''(x)^2 dx$$

subject to the constraint

$$\sum_{i=1}^N |S_\sigma(x_i) - f_i|^2 \leq \sigma$$

where  $\sigma$  is the smoothing parameter and  $N = NDATA$ . The reader should consult Reinsch (1967) for more information concerning smoothing splines. The IMSL subroutine `CSSMH` solves the above problem when the user provides the smoothing parameter  $\sigma$ . This routine attempts to find the 'optimal' smoothing parameter using the statistical technique known as cross-validation. This means that (in a very rough sense) one chooses the value of  $\sigma$  so that the smoothing spline ( $S_\sigma$ ) best approximates the value of the data at  $x_i$ , if it is computed using all the data *except* the  $i$ -th; this

is true for all  $i = 1, \dots, N$ . For more information on this topic, we refer the reader to Craven and Wahba (1979).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2SCV/DC2SCV`. The reference is:

```
CALL C2SCV (NDATA, XDATA, FDATA, IEQUAL, BREAK, CSCOE,
 WK, SDWK, IPVT)
```

The additional arguments are as follows:

**WK** — Work array of length  $7 * (NDATA + 2)$ .

**SDWK** — Work array of length  $2 * NDATA$ .

**IPVT** — Work array of length  $NDATA$ .

2. Informational error

| Type | Code |                                                                                  |
|------|------|----------------------------------------------------------------------------------|
| 4    | 2    | Points in the data point abscissas array, <code>XDATA</code> , must be distinct. |

## Example

In this example, function values are computed and are contaminated by adding a small “random” amount. The routine `CSSCV` is used to try to reproduce the original, uncontaminated data.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NDATA
PARAMETER (NDATA=300)

!
INTEGER I, IEQUAL, NOUT
REAL BREAK(NDATA), CSCOE(4,NDATA), ERROR, F,&
 FDATA(NDATA), FLOAT, FVAL, SVAL, X,&
 XDATA(NDATA), XT, RN
INTRINSIC FLOAT

!
F(X) = 1.0/(.1+(3.0*(X-1.0))**4)
!
CALL UMACH (2, NOUT)
!
! Set up a grid
DO 10 I=1, NDATA
 XDATA(I) = 3.0*(FLOAT(I-1)/FLOAT(NDATA-1))
 FDATA(I) = F(XDATA(I))
10 CONTINUE
!
! Introduce noise on [-.5,.5]
! Contaminate the data
CALL RNSET (1234579)
DO 20 I=1, NDATA
```

```

 RN = RNUNF ()
 FDATA(I) = FDATA(I) + 2.0*RN - 1.0
20 CONTINUE
!
! Set IEQUAL=1 for equally spaced data
 IEQUAL = 1
! Smooth data
 CALL CSSCV (XDATA, FDATA, IEQUAL, BREAK, CSCOE)
! Print results
 WRITE (NOUT,99999)
 DO 30 I=1, 10
 XT = 90.0*(FLOAT(I-1)/FLOAT(NDATA-1))
 SVAL = CSVAL(XT,BREAK,CSCOE)
 FVAL = F(XT)
 ERROR = SVAL - FVAL
 WRITE (NOUT,'(4F15.4)') XT, FVAL, SVAL, ERROR
30 CONTINUE
99999 FORMAT (12X, 'X', 9X, 'Function', 7X, 'Smoothed', 10X,&
 'Error')
 END

```

## Output

| X      | Function | Smoothed | Error   |
|--------|----------|----------|---------|
| 0.0000 | 0.0123   | 0.2528   | 0.2405  |
| 0.3010 | 0.0514   | 0.1054   | 0.0540  |
| 0.6020 | 0.4690   | 0.3117   | -0.1572 |
| 0.9030 | 9.3312   | 8.9461   | -0.3850 |
| 1.2040 | 4.1611   | 4.6847   | 0.5235  |
| 1.5050 | 0.1863   | 0.3819   | 0.1956  |
| 1.8060 | 0.0292   | 0.1168   | 0.0877  |
| 2.1070 | 0.0082   | 0.0658   | 0.0575  |
| 2.4080 | 0.0031   | 0.0395   | 0.0364  |
| 2.7090 | 0.0014   | -0.2155  | -0.2169 |

---

## RATCH



Computes a rational weighted Chebyshev approximation to a continuous function on an interval.

### Required Arguments

*F* — User-supplied FUNCTION to be approximated. The form is F(X), where

X — Independent variable. (Input)

F — The function value. (Output)

$F$  must be declared `EXTERNAL` in the calling program.

***PHI*** — User-supplied `FUNCTION` to supply the variable transformation which must be continuous and monotonic. The form is `PHI(X)`, where

$X$  — Independent variable. (Input)

`PHI` — The function value. (Output)

`PHI` must be declared `EXTERNAL` in the calling program.

***WEIGHT*** — User-supplied `FUNCTION` to scale the maximum error. It must be continuous and nonvanishing on the closed interval  $(A, B)$ . The form is `WEIGHT(X)`, where

$X$  — Independent variable. (Input)

`WEIGHT` — The function value. (Output)

`WEIGHT` must be declared `EXTERNAL` in the calling program.

***A*** — Lower end of the interval on which the approximation is desired. (Input)

***B*** — Upper end of the interval on which the approximation is desired. (Input)

***P*** — Vector of length  $N + 1$  containing the coefficients of the numerator polynomial.  
(Output)

***Q*** — Vector of length  $M + 1$  containing the coefficients of the denominator polynomial.  
(Output)

***ERROR*** — Min-max error of approximation. (Output)

### Optional Arguments

***N*** — The degree of the numerator. (Input)  
Default:  $N = \text{size}(P,1) - 1$ .

***M*** — The degree of the denominator. (Input)  
Default:  $M = \text{size}(Q,1) - 1$ .

### FORTRAN 90 Interface

Generic: `CALL RATCH (F, PHI, WEIGHT, A, B, P, Q, ERROR [, ...])`

Specific: The specific interface names are `S_RATCH` and `D_RATCH`.

### FORTRAN 77 Interface

Single: `CALL RATCH (F, PHI, WEIGHT, A, B, N, M, P, Q, ERROR)`

Double: The double precision name is DRATCH.

## Description

The routine RATCH is designed to compute the best weighted  $L_\infty$  (Chebyshev) approximant to a given function. Specifically, given a weight function  $w = \text{WEIGHT}$ , a monotone function  $\phi = \text{PHI}$ , and a function  $f$  to be approximated on the interval  $[a, b]$ , the subroutine RATCH returns the coefficients (in  $P$  and  $Q$ ) for a rational approximation to  $f$  on  $[a, b]$ . The user must supply the degree of the numerator  $N$  and the degree of the denominator  $M$  of the rational function

$$R_M^N$$

The goal is to produce coefficients which minimize the expression

$$\left\| \frac{f - R_M^N}{w} \right\| := \max_{x \in [a, b]} \left| \frac{f(x) - \frac{\sum_{i=1}^{N+1} P_i \phi^{i-1}(x)}{\sum_{i=1}^{M+1} Q_i \phi^{i-1}(x)}}{w(x)} \right|$$

Notice that setting  $\phi(x) = x$  yields ordinary rational approximation. A typical use of the function  $\phi$  occurs when one wants to approximate an even function on a symmetric interval, say  $[-a, a]$  using ordinary rational functions. In this case, it is known that the answer must be an even function. Hence, one can set  $\phi(x) = x^2$ , only approximate on  $[0, a]$ , and decrease by one half the degrees in the numerator and denominator.

The algorithm implemented in this subroutine is designed for fast execution. It assumes that the best approximant has precisely  $N + M + 2$  equi-oscillations. That is, that there exist  $N + M + 2$  points  $\mathbf{t}_1 < \dots < \mathbf{t}_{N+M+2}$  satisfying

$$e(\mathbf{t}_i) = -e(\mathbf{t}_{i+1}) = \pm \left\| \frac{f - R_M^N}{w} \right\|$$

Such points are called alternants. Unfortunately, there are many instances in which the best rational approximant to the given function has either fewer alternants or more alternants. In this case, it is not expected that this subroutine will perform well. For more information on rational Chebyshev approximation, the reader can consult Cheney (1966). The subroutine is based on work of Cody, Fraser, and Hart (1968).

## Comments

1. Workspace may be explicitly provided, if desired, by use of R2TCH/DR2TCH. The reference is:

```
CALL R2TCH (F, PHI, WEIGHT, A, B, N, M, P, Q, ERROR,
 ITMAX, IWK, WK)
```

The additional arguments are as follows:

**ITMAX** — Maximum number of iterations. (Input)  
The default value is 20.

**IWK** — Workspace vector of length  $(N + M + 2)$ . (Workspace)

**WK** — Workspace vector of length  $(N + M + 8) * (N + M + 2)$ . (Workspace)

## 2. Informational errors

| Type | Code |                                                                                                                                                                    |
|------|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | The maximum number of iterations has been reached. The routine R2TCH may be called directly to set a larger value for ITMAX.                                       |
| 3    | 2    | The error was reduced as far as numerically possible. A good approximation is returned in P and Q, but this does not necessarily give the Chebyshev approximation. |
| 4    | 3    | The linear system that defines P and Q was found to be algorithmically singular. This indicates the possibility of a degenerate approximation.                     |
| 4    | 4    | A sequence of critical points that was not monotonic generated. This indicates the possibility of a degenerate approximation.                                      |
| 4    | 5    | The value of the error curve at some critical point is too large. This indicates the possibility of poles in the rational function.                                |
| 4    | 6    | The weight function cannot be zero on the closed interval (A, B).                                                                                                  |

## Example

In this example, we compute the best rational approximation to the gamma function,  $\Gamma$ , on the interval  $[2, 3]$  with weight function  $w = 1$  and  $N = M = 2$ . We display the maximum error and the coefficients. This problem is taken from the paper of Cody, Fraser, and Hart (1968). We compute in double precision due to the conditioning of this problem.

```
USE RATCH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER M, N
PARAMETER (M=2, N=2)
!
INTEGER NOUT
DOUBLE PRECISION A, B, ERROR, F, P(N+1), PHI, Q(M+1), WEIGHT
EXTERNAL F, PHI, WEIGHT
!
A = 2.0D0
B = 3.0D0
!
! Compute double precision rational
! approximation
CALL RATCH (F, PHI, WEIGHT, A, B, P, Q, ERROR)
! Get output unit number
CALL UMACH (2, NOUT)
```

```

! Print P, Q and min-max error
 WRITE (NOUT,'(1X,A)') 'In double precision we have:'
 WRITE (NOUT,99999) 'P = ', P
 WRITE (NOUT,99999) 'Q = ', Q
 WRITE (NOUT,99999) 'ERROR = ', ERROR
99999 FORMAT (' ', A, 5X, 3F20.12, '/')
 END
! -----
!
 DOUBLE PRECISION FUNCTION F (X)
 DOUBLE PRECISION X
!
 DOUBLE PRECISION DGAMMA
 EXTERNAL DGAMMA
!
 F = DGAMMA (X)
 RETURN
 END
! -----
!
 DOUBLE PRECISION FUNCTION PHI (X)
 DOUBLE PRECISION X
!
 PHI = X
 RETURN
 END
! -----
!
 DOUBLE PRECISION FUNCTION WEIGHT (X)
 DOUBLE PRECISION X
!
 DOUBLE PRECISION DGAMMA
 EXTERNAL DGAMMA
!
 WEIGHT = DGAMMA (X)
 RETURN
 END

```

### Output

```

In double precision we have:
P = 1.265583562487 -0.650585004466 0.197868699191
Q = 1.000000000000 -0.064342721236 -0.028851461855
ERROR = -0.000026934190

```

# Chapter 4: Integration and Differentiation

---

## Routines

|                                                                                       |                        |     |
|---------------------------------------------------------------------------------------|------------------------|-----|
| <b>4.1. Univariate Quadrature</b>                                                     |                        |     |
| Adaptive general-purpose endpoint singularities.....                                  | <a href="#">QDAGS</a>  | 918 |
| Adaptive general purpose.....                                                         | <a href="#">QDAG</a>   | 922 |
| Adaptive general-purpose points of singularity.....                                   | <a href="#">QDAGP</a>  | 925 |
| Adaptive general-purpose with a possible internal or<br>endpoint singularity .....    | <a href="#">QDAG1D</a> | 929 |
| Adaptive general-purpose infinite interval .....                                      | <a href="#">QDAGI</a>  | 935 |
| Adaptive weighted oscillatory (trigonometric) .....                                   | <a href="#">QDAWO</a>  | 938 |
| Adaptive weighted Fourier (trigonometric).....                                        | <a href="#">QDAWF</a>  | 942 |
| Adaptive weighted algebraic endpoint singularities.....                               | <a href="#">QDAWS</a>  | 946 |
| Adaptive weighted Cauchy principal value .....                                        | <a href="#">QDAWC</a>  | 949 |
| Nonadaptive general purpose.....                                                      | <a href="#">QDNG</a>   | 953 |
| <b>4.2. Multidimensional Quadrature</b>                                               |                        |     |
| Two-dimensional quadrature (iterated integral).....                                   | <a href="#">TWODQ</a>  | 955 |
| Two-dimensional quadrature with a possible<br>internal or endpoint singularity.....   | <a href="#">QDAG2D</a> | 960 |
| Three-dimensional quadrature with a possible<br>internal or endpoint singularity..... | <a href="#">QDAG3D</a> | 966 |
| Adaptive N-dimensional quadrature<br>over a hyper-rectangle.....                      | <a href="#">QAND</a>   | 973 |
| Integrates a function over a hyperrectangle using a<br>quasi-Monte Carlo method ..... | <a href="#">QMC</a>    | 976 |
| <b>4.3. Gauss Rules and Three-term Recurrences</b>                                    |                        |     |
| Gauss quadrature rule for classical weights.....                                      | <a href="#">GQRUL</a>  | 979 |
| Gauss quadrature rule from recurrence coefficients .....                              | <a href="#">GQRCF</a>  | 983 |
| Recurrence coefficients for classical weights .....                                   | <a href="#">RECCF</a>  | 986 |
| Recurrence coefficients from quadrature rule .....                                    | <a href="#">RECQR</a>  | 988 |
| Fejer quadrature rule .....                                                           | <a href="#">FQRUL</a>  | 991 |
| <b>4.4. Differentiation</b>                                                           |                        |     |
| Approximation to first, second, or third derivative.....                              | <a href="#">DERIV</a>  | 995 |

---

# Usage Notes

## Univariate Quadrature

The first ten routines described in this chapter are designed to compute approximations to integrals of the form

$$\int_a^b f(x)w(x)dx$$

The weight function  $w$  is used to incorporate known singularities (either algebraic or logarithmic), to incorporate oscillations, or to indicate that a Cauchy principal value is desired. For general purpose integration, we recommend the use of [QDAGS](#) (even if no endpoint singularities are present). If more efficiency is desired, then the use of [QDAG](#) (or [QDAG\\*](#)) should be considered. These routines are organized as follows:

- $w = 1$ 
  - [QDAGS](#)
  - [QDAG](#)
  - [QDAGP](#)
  - [QDAG1D](#)
  - [QDAGI](#)
  - [QDNG](#)
- $w(x) = \sin \omega x$  or  $w(x) = \cos \omega x$ 
  - [QDAWO](#) (for a finite interval)
  - [QDAWF](#) (for an infinite interval)
- $w(x) = (x - a)^\alpha (b - x)^\beta \ln(x - a) \ln(b - x)$ , where the  $\ln$  factors are optional
  - [QDAWS](#)
- $w(x) = 1/(x - c)$  Cauchy principal value
  - [QDAWC](#)

The calling sequences for these routines are very similar. The function to be integrated is always `F`; the lower and upper limits are, respectively, `A` and `B`. The requested absolute error  $\epsilon$  is `ERRABS`, while the requested relative error  $\rho$  is `ERRREL`. These quadrature routines return two numbers of interest, namely, `RESULT` and `ERREST`, which are the approximate integral  $R$  and the error estimate  $E$ , respectively. These numbers are related as follows:

$$\left| \int_a^b f(x)w(x) dx - R \right| \leq E \leq \max \left\{ \varepsilon, \rho \left| \int_a^b f(x)w(x) dx \right| \right\}$$

The requested absolute and relative errors must be interpreted as ‘tuning knobs.’ The actual errors may be much larger than these values indicate if the sampling of the integrand function misses a peak. Coarse sampling of the integration interval occurs with larger values of `ERRABS` or `ERRREL`. We recommend experimenting with these values, starting with small positive values and then increasing them until the required accuracy is obtained.

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The routines described above do not directly address this question. However, the standard method for handling this problem is first to interpolate the data and then to integrate the interpolant. This can be accomplished by using the IMSL spline interpolation routines described in [Chapter 3, “Interpolation and Approximation”](#), with one of the integration routines `CSINT`, `BSINT`, or `PPITG`.

## Multivariate Quadrature

Four routines are described in this chapter that are of use in approximating certain multivariate integrals. In particular, the routine `TWODQ` and `QDAG2D` return an approximation to an iterated two-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y) dy dx$$

while `QDAG3D` returns an approximation to an iterated three-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} \int_{p(x,y)}^{q(x,y)} f(x, y, z) dz dy dx$$

The fourth routine `QAND` returns an approximation to the integral of a function of  $n$  variables over a hyper-rectangle

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \dots, x_n) dx_n \cdots dx_1$$

If one has two- or three-dimensional tensor-product tabular data, use the IMSL spline interpolation routines `BS2IN` or `BS3IN`, followed by the IMSL spline integration routines `BS2IG` and `BS3IG` that are described in [Chapter 3, Interpolation and Approximation](#).

## Gauss Rules and Three-term Recurrences

The routines described in this section deal with the constellation of problems encountered in Gauss quadrature. These problems arise when quadrature formulas, which integrate polynomials of the highest degree possible, are computed. Once a member of a family of seven weight functions is specified, the routine `GQRUL` produces the points  $\{x_i\}$  and weights  $\{w_i\}$  for  $i = 1, \dots, N$  that satisfy

$$\int_a^b f(x)w(x) dx = \sum_{i=1}^N f(x_i)w_i$$

for all functions  $f$  that are polynomials of degree less than  $2N$ . The weight functions  $w$  may be selected from the following table:

| $w(x)$                     | Interval            | Name                 |
|----------------------------|---------------------|----------------------|
| 1                          | $(-1, 1)$           | Legendre             |
| $1/\sqrt{1-x^2}$           | $(-1, 1)$           | Chebyshev 1st kind   |
| $\sqrt{1-x^2}$             | $(-1, 1)$           | Chebyshev 2nd kind   |
| $e^{-x^2}$                 | $(-\infty, \infty)$ | Hermite              |
| $(1+x)^\alpha (1-x)^\beta$ | $(-1, 1)$           | Jacobi               |
| $e^{-x} x^\alpha$          | $(0, \infty)$       | Generalized Laguerre |
| $1/\cosh(x)$               | $(-\infty, \infty)$ | Hyperbolic cosine    |

Where permissible, `GQRUL` will also compute Gauss-Radau and Gauss-Lobatto quadrature rules. The routine `RECCF` produces the three-term recurrence relation for the monic orthogonal polynomials with respect to the above weight functions.

Another routine, `GQRCF`, produces the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule from the three-term recurrence relation. This means Gauss rules for general weight functions may be obtained if the three-term recursion for the orthogonal polynomials is known. The routine `RECQR` is an inverse to `GQRCF` in the sense that it produces the recurrence coefficients given the Gauss quadrature formula.

The last routine described in this section, `FQRUL`, generates the Fejér quadrature rules for the following family of weights:

$$\begin{aligned}
 w(x) &= 1 \\
 w(x) &= 1/(x - \alpha) \\
 w(x) &= (b - x)^\alpha (x - a)^\beta \\
 w(x) &= (b - x)^\alpha (x - a)^\beta \ln(x - a) \\
 w(x) &= (b - x)^\alpha (x - a)^\beta \ln(b - x)
 \end{aligned}$$

## Numerical Differentiation

We provide one routine, `DERIV`, for numerical differentiation. This routine provides an estimate for the first, second, or third derivative of a user-supplied function.

---

## QDAGS

Integrates a function (which may have endpoint singularities).

## Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(x)$ , where

$x$  — Independent variable. (Input)

$F$  — The function value. (Output)

$F$  must be declared EXTERNAL in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)

**RESULT** — Estimate of the integral from  $A$  to  $B$  of  $F$ . (Output)

## Optional Required Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default:  $ERRABS = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Relative accuracy desired. (Input)

Default:  $ERRREL = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAGS (F, A, B, RESULT [, ...])`

Specific: The specific interface names are `S_QDAGS` and `D_QDAGS`.

## FORTRAN 77 Interface

Single: `CALL QDAGS (F, A, B, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDAGS`.

## Description

The routine `QDAGS` is a general-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It subdivides the interval  $[A, B]$  and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-point Gauss quadrature rule. This routine is designed to handle functions with endpoint singularities. However, the performance on functions, which are well-behaved at the endpoints, is quite good also. In addition to the general strategy described in `QDAG`, this routine uses an extrapolation procedure known as the  $\epsilon$ -algorithm. The routine `QDAGS` is an implementation of the routine `QAGS`, which is fully documented by Piessens et al. (1983). Should

QDAGS fail to produce acceptable results, then either IMSL routines QDAG or QDAG\* may be appropriate. These routines are documented in this chapter.

## Comments

1. Workspace may be explicitly provided, if desired, by use of Q2AGS/DQ2AGS. The reference is

```
CALL Q2AGS (F, A, B, ERRABS, ERRREL, RESULT, ERREST, MAXSUB,
 NEVAL, NSUBIN, ALIST, BLIST, RLIST, ELIST, IORD)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)

A value of 500 is used by QDAGS.

**NEVAL** — Number of evaluations of F. (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length MAXSUB containing a list of the NSUBIN left endpoints. (Output)

**BLIST** — Array of length MAXSUB containing a list of the NSUBIN right endpoints. (Output)

**RLIST** — Array of length MAXSUB containing approximations to the NSUBIN integrals over the intervals defined by ALIST, BLIST. (Output)

**ELIST** — Array of length MAXSUB containing the error estimates of the NSUBIN values in RLIST. (Output)

**IORD** — Array of length MAXSUB. (Output)

Let  $k$  be

NSUBIN if  $NSUBIN \leq (MAXSUB/2 + 2)$ ;

MAXSUB + 1 - NSUBIN otherwise.

The first  $k$  locations contain pointers to the error estimates over the subintervals such that  $ELIST(IORD(1)), \dots, ELIST(IORD(k))$  form a decreasing sequence.

2. Informational errors

| Type | Code |                                                                                            |
|------|------|--------------------------------------------------------------------------------------------|
| 4    | 1    | The maximum number of subintervals allowed has been reached.                               |
| 3    | 2    | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| 3    | 3    | A degradation in precision has been detected.                                              |

3 4 Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

4 5 Integral is probably divergent or slowly convergent.

3. If EXACT is the exact value, QDAGS attempts to find RESULT such that  $|EXACT - RESULT| \leq \max(ERRABS, ERRREL * |EXACT|)$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.

## Example

The value of

$$\int_0^1 \ln(x) x^{-1/2} dx = -4$$

is estimated. The values of the actual and estimated error are machine dependent.

```

USE QDAGS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
REAL A, ABS, B, ERRABS, ERREST, ERROR, ERRREL, EXACT, F, &
 RESULT
INTRINSIC ABS
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

! Set limits of integration
A = 0.0
B = 1.0

! Set error tolerances
ERRABS = 0.0
CALL QDAGS (F, A, B, RESULT, ERRABS=ERRABS, ERREST=ERREST)

! Print results
EXACT = -4.0
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, '/', '/', &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END

!
REAL FUNCTION F (X)
REAL X
REAL ALOG, SQRT
INTRINSIC ALOG, SQRT
F = ALOG(X)/SQRT(X)
RETURN
END

```

## Output

Computed = -4.000                      Exact = -4.000  
Error estimate = 1.519E-04              Error = 2.098E-05

---

## QDAG

Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.

### Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  
F(X), where

X — Independent variable. (Input)

F — The function value. (Output)

F must be declared EXTERNAL in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)

**RESULT** — Estimate of the integral from A to B of F. (Output)

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)  
Default: ERRABS = 1.e-3 for single precision and 1.d-8 for double precision.

**ERRREL** — Relative accuracy desired. (Input)  
Default: ERRREL = 1.e-3 for single precision and 1.d-8 for double precision.

**IRULE** — Choice of quadrature rule. (Input)  
Default: IRULE = 2.  
The Gauss-Kronrod rule is used with the following points:

| IRULE | Points |
|-------|--------|
| 1     | 7-15   |
| 2     | 10-21  |
| 3     | 15-31  |
| 4     | 20-41  |

| <b>IRULE</b> | <b>Points</b> |
|--------------|---------------|
| 5            | 25-51         |
| 6            | 30-61         |

`IRULE = 2` is recommended for most functions. If the function has a peak singularity, use `IRULE = 1`. If the function is oscillatory, use `IRULE = 6`.

**ERREST** — Estimate of the absolute value of the error. (Output)

### **FORTRAN 90 Interface**

Generic: `CALL QDAG (F, A, B, RESULT [, ...])`

Specific: The specific interface names are `S_QDAG` and `D_QDAG`.

### **FORTRAN 77 Interface**

Single: `CALL QDAG (F, A, B, ERRABS, ERRREL, IRULE, RESULT, ERREST)`

Double: The double precision name is `DQDAG`.

### **Description**

The routine `QDAG` is a general-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error. It subdivides the interval  $[A, B]$  and uses a  $(2k + 1)$ -point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the  $k$ -point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. The routine `QDAG` is based on the subroutine `QAG` by Piessens et al. (1983).

Should `QDAG` fail to produce acceptable results, then one of the IMSL routines `QDAG*` may be appropriate. These routines are documented in this chapter.

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `Q2AG/DQ2AG`. The reference is:

```
CALL Q2AG (F, A, B, ERRABS, ERRREL, IRULE, RESULT, ERREST,
 MAXSUB, NEVAL, NSUBIN, ALIST, BLIST, RLIST, ELIST,
 IORD)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)  
A value of 500 is used by QDAG.

**NEVAL** — Number of evaluations of  $F$ . (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** left endpoints. (Output)

**BLIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** right endpoints. (Output)

**RLIST** — Array of length **MAXSUB** containing approximations to the **NSUBIN** integrals over the intervals defined by **ALIST**, **BLIST**. (Output)

**ELIST** — Array of length **MAXSUB** containing the error estimates of the **NSUBIN** values in **RLIST**. (Output)

**IORD** — Array of length **MAXSUB**. (Output)

Let  $K$  be **NSUBIN** if **NSUBIN**.LE.(**MAXSUB**/2 + 2), **MAXSUB** + 1 - **NSUBIN** otherwise. The first  $K$  locations contain pointers to the error estimates over the corresponding subintervals, such that

**ELIST**(**IORD**(1)), ..., **ELIST**(**IORD**( $K$ )) form a decreasing sequence.

2. Informational errors

Type Code

4 1 The maximum number of subintervals allowed has been reached.

3 2 Roundoff error, preventing the requested tolerance from being achieved, has been detected.

3 3 A degradation in precision has been detected.

3. If **EXACT** is the exact value, QDAG attempts to find **RESULT** such that

**ABS**(**EXACT** - **RESULT**).LE.**MAX**(**ERRABS**, **ERRREL** \* **ABS**(**EXACT**)). To specify only a relative error, set **ERRABS** to zero. Similarly, to specify only an absolute error, set **ERRREL** to zero.

### Example

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is estimated. Since the integrand is not oscillatory, **IRULE** = 1 is used. The values of the actual and estimated error are machine dependent.

```

USE QDAG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IRULE, NOUT
REAL A, ABS, B, ERRABS, ERREST, ERROR, EXACT, EXP, &
 F, RESULT
INTRINSIC ABS, EXP
EXTERNAL F
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set limits of integration
A = 0.0
B = 2.0
!
! Set error tolerances
ERRABS = 0.0
!
! Parameter for non-oscillatory
! function
IRULE = 1
CALL QDAG (F, A, B, RESULT, ERRABS=ERRABS, IRULE=IRULE, ERREST=ERREST)
!
! Print results
EXACT = 1.0 + EXP(2.0)
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, '/', '/', &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END
!
REAL FUNCTION F (X)
REAL X
REAL EXP
INTRINSIC EXP
F = X*EXP(X)
RETURN
END

```

## Output

```

Computed = 8.389 Exact = 8.389
Error estimate = 5.000E-05 Error = 9.537E-07

```

---

## QDAGP

Integrates a function with singularity points given.

### Required Arguments

*F* — User-supplied FUNCTION to be integrated. The form is  
 $F(X)$ , where

*X* — Independent variable. (Input)

*F* — The function value. (Output)

*F* must be declared `EXTERNAL` in the calling program.

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. (Input)

*POINTS* — Array of length `NPTS` containing breakpoints in the range of integration. (Input)  
Usually these are points where the integrand has singularities.

*RESULT* — Estimate of the integral from *A* to *B* of *F*. (Output)

### Optional Arguments

*NPTS* — Number of break points given. (Input)

Default: `NPTS = size (POINTS,1)`.

*ERRABS* — Absolute accuracy desired. (Input)

Default: `ERRABS = 1.e-3` for single precision and `1.d-8` for double precision.

*ERRREL* — Relative accuracy desired. (Input)

Default: `ERRREL = 1.e-3` for single precision and `1.d-8` for double precision.

*ERREST* — Estimate of the absolute value of the error. (Output)

### FORTRAN 90 Interface

Generic: `CALL QDAGP (F, A, B, POINTS, RESULT [, ...])`

Specific: The specific interface names are `S_QDAGP` and `D_QDAGP`.

### FORTRAN 77 Interface

Single: `CALL QDAGP (F, A, B, NPTS, POINTS, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDAGP`.

### Description

The routine `QDAGP` uses a globally adaptive scheme in order to reduce the absolute error. It initially subdivides the interval  $[A, B]$  into `NPTS + 1` user-supplied subintervals and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-point Gauss quadrature rule. This routine is designed to handle endpoint as well as interior singularities. In addition to the general strategy described in the IMSL routine `QDAG`, this routine employs an extrapolation procedure known as the  $\epsilon$ -algorithm.

The routine QDAGP is an implementation of the subroutine QAGP, which is fully documented by Piessens et al. (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of Q2AGP/DQ2AGP. The reference is:

```
CALL Q2AGP (F, A, B, NPTS, POINTS, ERRABS, ERRREL, RESULT,
 ERREST, MAXSUB, NEVAL, NSUBIN, ALIST, BLIST, RLIST,
 ELIST, IORD, LEVEL, WK, IWK)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)  
A value of 450 is used by QDAGP.

**NEVAL** — Number of evaluations of F. (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length MAXSUB containing a list of the NSUBIN left endpoints. (Output)

**BLIST** — Array of length MAXSUB containing a list of the NSUBIN right endpoints. (Output)

**RLIST** — Array of length MAXSUB containing approximations to the NSUBIN integrals over the intervals defined by ALIST, BLIST. (Output)

**ELIST** — Array of length MAXSUB containing the error estimates of the NSUBIN values in RLIST. (Output)

**IORD** — Array of length MAXSUB. (Output)

Let  $K$  be  $NSUBIN$  if  $NSUBIN.LE.(MAXSUB/2 + 2)$ ,  $MAXSUB + 1 - NSUBIN$  otherwise. The first  $K$  locations contain pointers to the error estimates over the subintervals, such that  $ELIST(IORD(1)), \dots, ELIST(IORD(K))$  form a decreasing sequence.

**LEVEL** — Array of length MAXSUB, containing the subdivision levels of the subinterval. (Output)

That is, if  $(AA, BB)$  is a subinterval of  $(P1, P2)$  where  $P1$  as well as  $P2$  is a user-provided break point or integration limit, then  $(AA, BB)$  has level  $L$  if

$$ABS(BB - AA) = ABS(P2 - P1) * 2^{**}(-L).$$

**WK** — Work array of length NPTS + 2.

**IWK** — Work array of length NPTS + 2.

2. Informational errors

Type Code

|   |   |                                                                                                                       |
|---|---|-----------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The maximum number of subintervals allowed has been reached.                                                          |
| 3 | 2 | Roundoff error, preventing the requested tolerance from being achieved, has been detected.                            |
| 3 | 3 | A degradation in precision has been detected.                                                                         |
| 3 | 4 | Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected. |
| 4 | 5 | Integral is probably divergent or slowly convergent.                                                                  |

3. If EXACT is the exact value, QDAGP attempts to find RESULT such that  $\text{ABS}(\text{EXACT} - \text{RESULT}) \leq \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.

### Example

The value of

$$\int_0^3 x^3 \ln |(x^2 - 1)(x^2 - 2)| dx = 61 \ln 2 + \frac{77}{4} \ln 7 - 27$$

is estimated. The values of the actual and estimated error are machine dependent. Note that this subroutine never evaluates the user-supplied function at the user-supplied breakpoints.

```

USE QDAGP_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT, NPTS
REAL A, ABS, ALOG, B, ERRABS, ERREST, ERROR, ERRREL, &
 EXACT, F, POINTS(2), RESULT, SQRT
INTRINSIC ABS, ALOG, SQRT
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

! Set limits of integration
A = 0.0
B = 3.0

! Set error tolerances
ERRABS = 0.0
ERRREL = 0.01

! Set singularity parameters
NPTS = 2
POINTS(1) = 1.0
POINTS(2) = SQRT(2.0)
CALL QDAGP (F, A, B, POINTS, RESULT, ERRABS=ERRABS, ERRREL=ERRREL, &
 ERREST=ERREST)

! Print results

```

```

 EXACT = 61.0*ALOG(2.0) + 77.0/4.0*ALOG(7.0) - 27.0
 ERROR = ABS(RESULT-EXACT)
 WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, '/', '/', &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
!
 END
!
 REAL FUNCTION F (X)
 REAL X
 REAL ABS, ALOG
 INTRINSIC ABS, ALOG
 F = X**3*ALOG(ABS((X*X-1.0)*(X*X-2.0)))
 RETURN
 END

```

### Output

```

Computed = 52.741 Exact = 52.741
Error estimate = 5.062E-01 Error = 6.104E-04

```

---

## QDAG1D

Integrates a function with a possible internal or endpoint singularity.

### Required Arguments

*F* — User-supplied FUNCTION to be integrated. The form is  $F(X [, \dots])$ , where

#### Function Return Value

*F* — The function value. (Output)

#### Required Arguments

*X* — Independent variable. (Input)

#### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

*F* must be declared EXTERNAL in the calling program.

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. The relative values of *A* and *B* are interpreted properly. Thus if one exchanges *A* and *B*, the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of  $B - A$ . (Input)

**RESULT** — Estimate of the integral from A to B of F. (Output)

## Optional Arguments

**ERRABS** — Absolute error tolerance. See [Comment 1](#) for a discussion on the error tolerances. (Input)  
Default: `ERRABS = 0.0`.

**ERRFRAC** — A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See [Comment 1](#) for a discussion on the error tolerances. (Input)  
Default: `ERRFRAC = 0.75`.

**ERRREL** — The error tolerance relative to the value of the integral. See [Comment 1](#) for a discussion on the error tolerances. (Input)  
Default: `ERRREL = 0.0`.

**ERRPOST** — An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. This value may be computed during the evaluation of the integrand. When this optional argument is used, `FCN_DATA` must also be used as `FCN_DATA%RDATA(1)` will be used to pass the newly calculated value of `ERRPOST` back from the evaluator, F. In this case, the user should not use `FCN_DATA%RDATA(1)` for passing other data. (Input)  
Default: `ERRPOST = 0.0`.

**ERRPRIOR** — An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral. (Input)  
Default: `ERRPRIOR = 1.19e-7` for single precision and `2.22d-16` for double precision.

**MAXFCN** — The maximum number of function values to use to compute the integral. (Input)  
Default: The number of function values is not bounded.

**SINGULARITY** — The real part of the abscissa of a singularity or discontinuity in the integrand. If this option is used, `SINGULARITY_TYPE` must also be used. (Input)  
Default: It is assumed that there is no singularity in the integrand so `SINGULARITY` is not set. It is an error to set `SINGULARITY` without also setting `SINGULARITY_TYPE`.

**SINGULARITY\_TYPE** — A signed integer specifying the type of singularity which occurs in the integrand. If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not an integer, if  $\alpha$  is “large” or has the form  $\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a positive integer. Otherwise, set `SINGULARITY_TYPE` to a negative integer. (Input)  
Default: It is assumed that there is no singularity in the integrand so `SINGULARITY_TYPE` is not set. It is an error to set `SINGULARITY_TYPE` without also setting `SINGULARITY`.

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. The derived type, `s_fcn_data`, is defined as:

```
type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type
```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type. Note that if this optional argument is used then this argument must also be used in the user-supplied function. (Input/Output)

**NEVAL** — Number of function evaluations used to calculate the integral. (Output)

**ERREST** — An estimate of the upper bound of the magnitude of the difference between `RESULT` and the true value of the integral. (Output)

**ISTATUS** — A status flag indicating the error criteria which was satisfied on exit.

- `ISTATUS = -1` indicates normal termination with either the absolute or relative error tolerance criteria satisfied.
- `ISTATUS = -2` indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied.
- `ISTATUS = -3` indicates normal termination with none of the error tolerance criteria satisfied.
- `ISTATUS = any value other than the above` indicates abnormal termination due to an error condition. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAG1D(F, A, B, RESULT [, ...])`

Specific: The specific interface names are `S_QDAG1D` and `D_QDAG1D`.

## Description

`QDAG1D` is based on the JPL Library routine `SINT1`. The integral is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the  $k^{\text{th}}$  formula used all the integrand values used in the  $(k-1)^{\text{st}}$  formula, and added  $2^{k-1}$  new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to  $k = 8$ .

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including

errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

## Comments

1. The user provides the absolute error tolerance through optional argument `ERRABS`. Optional argument `ERRFRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). Optional argument `ERRREL` represents the error tolerance relative to the value of the integral. The internal value for `ERRFRAC` is bounded between .5 and 1. By default, `ERRABS` and `ERRREL` are set to 0.0 and `ERRFRAC` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.  
The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.
2. Optional arguments `SINGULARITY` and `SINGULARITY_TYPE` provide the user with a means to give the routine information about the location and type of any known singularity of the integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside  $[A, B]$ . If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not

an integer, if  $\alpha$  is “large” or has the form

$\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a positive value. Otherwise, set `SINGULARITY_TYPE` to a negative value. The meaning of “large” depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered “large”. For a singularity of the form  $x^\alpha \log x$  use the above rule, even if  $\alpha$  is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed, some experimentation may be useful.

When `SINGULARITY_TYPE` is positive, a transformation of the form  $T = TA + (X - TA)^2 / (TB - TA)$  is applied, where  $TA$  is the abscissa of the singularity and  $TB$  is the end of the interval. If  $TA$  is outside the interval,  $TB$  will be the end of the interval farthest from  $TA$ . If  $TA$  is inside the interval, the interval will immediately be subdivided at  $TA$ , and both parts will be separately integrated with  $TB$  equal to each end of the original interval, respectively. When `SINGULARITY_TYPE` is negative, a transformation of the form  $T = TA + (X - TA)^4 / (TB - TA)^3$  is applied, with  $TA$  and  $TB$  as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

### Example 1

The value of

$$\int_0^1 \ln(x) (x^{-1/2}) dx = -4$$

is estimated. Note that the optional arguments `SINGULARITY` and `SINGULARITY_TYPE` are used.

```

USE QDAG1D_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER NOUT, SINGULARITY_TYPE

REAL A, B, ERREST, F, RESULT, SINGULARITY

EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

! Set limits of integration
A = 0.0
B = 1.0

! Set singularity value and type
SINGULARITY = 0.0
SINGULARITY_TYPE = -1

```

```

 CALL QDAG1D (F, A, B, RESULT, SINGULARITY=SINGULARITY, &
 SINGULARITY_TYPE=SINGULARITY_TYPE, ERREST=ERREST)
! Print the results
 WRITE(NOUT,*)'Result = ', RESULT
 WRITE(NOUT,9999) ERREST
9999 FORMAT('Error Estimate = ', 1PE9.1)
 END

 REAL FUNCTION F (X)
 REAL X
 REAL ALOG, SQRT
 INTRINSIC ALOG, SQRT
 F = ALOG(X)/SQRT(X)
 RETURN
 END

```

## Output

```

Result = -4.0
Error Estimate = 6.0E-07

```

## Example 2

The value of

$$\int_1^2 (2x + kx) dx = 6$$

is estimated. Note that the optional argument FCN\_DATA is used to set the value of  $k = 2$  in the user-supplied function, F.

```

 USE QDAG1D_INT
 USE UMACH_INT
 USE MP_TYPES

 IMPLICIT NONE
! Declare variables
 INTEGER NOUT

 REAL A, B, ERREST, F, RESULT
 REAL, TARGET :: RDATA(1)

 TYPE (S_FCN_DATA) USER_DATA

 EXTERNAL F
! Get output unit number
 CALL UMACH (2, NOUT)
! Set limits of integration
 A = 1.0
 B = 2.0
! Set IPARAM
 RDATA(1) = 2.0
 USER_DATA%RDATA=>RDATA
 CALL QDAG1D (F, A, B, RESULT, FCN_DATA=USER_DATA, ERREST=ERREST)

```

```

! Print the results
 WRITE(NOUT,*) 'Result = ', RESULT
 WRITE(NOUT,9999) ERREST
9999 FORMAT('Error Estimate = ', 1PE9.1)
 END

 REAL FUNCTION F (X, FCN_DATA)
 USE MP_TYPES
 TYPE (S_FCN_DATA) FCN_DATA
 REAL X
 F = 2.0 * X + FCN_DATA%RDATA(1) * X
 RETURN
 END

```

## Output

```

Result = 6.0
Error Estimate = 1.2E-06

```

---

## QDAGI

Integrates a function over an infinite or semi-infinite interval.

### Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(X)$ , where

**X** — Independent variable. (Input)

**F** — The function value. (Output)

**F** must be declared EXTERNAL in the calling program.

**BOUND** — Finite bound of the integration range. (Input)  
Ignored if **INTERV** = 2.

**INTERV** — Flag indicating integration interval. (Input)

| <b>INTERV</b> | <b>Interval</b>           |
|---------------|---------------------------|
| -1            | $(-\infty, \text{BOUND})$ |
| 1             | $(\text{BOUND}, +\infty)$ |
| 2             | $(-\infty, +\infty)$      |

**RESULT** — Estimate of the integral from **A** to **B** of **F**. (Output)

## Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default: `ERRABS = 1.e-3` for single precision and `1.d-8` for double precision.

**ERRREL** — Relative accuracy desired. (Input)

Default: `ERRREL = 1.e-3` for single precision and `1.d-8` for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAGI (F, BOUND, INTERV, RESULT [, ...])`

Specific: The specific interface names are `S_QDAGI` and `D_QDAGI`.

## FORTRAN 77 Interface

Single: `CALL QDAGI (F, BOUND, INTERV, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDAGI`.

## Description

The routine `QDAGI` uses a globally adaptive scheme in an attempt to reduce the absolute error. It initially transforms an infinite or semi-infinite interval into the finite interval  $[0, 1]$ . Then, `QDAGI` uses a 21-point Gauss-Kronrod rule to estimate the integral and the error. It bisects any interval with an unacceptable error estimate and continues this process until termination. This routine is designed to handle endpoint singularities. In addition to the general strategy described in [QDAG](#), this subroutine employs an extrapolation procedure known as the  $\epsilon$ -algorithm. The routine `QDAGI` is an implementation of the subroutine `QAGI`, which is fully documented by Piessens et al. (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `Q2AGI/DQ2AGI`. The reference is

```
CALL Q2AGI (F, BOUND, INTERV, ERRABS, ERRREL, RESULT,
 ERREST, MAXSUB, NEVAL, NSUBIN, ALIST, BLIST, RLIST,
 ELIST, IORD)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)

A value of 500 is used by `QDAGI`.

**NEVAL** — Number of evaluations of `F`. (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** left endpoints. (Output)

**BLIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** right endpoints. (Output)

**RLIST** — Array of length **MAXSUB** containing approximations to the **NSUBIN** integrals over the intervals defined by **ALIST**, **BLIST**. (Output)

**ELIST** — Array of length **MAXSUB** containing the error estimates of the **NSUBIN** values in **RLIST**. (Output)

**IORD** — Array of length **MAXSUB**. (Output)

Let  $K$  be  $NSUBIN$  if  $NSUBIN \leq (MAXSUB/2 + 2)$ ,  $MAXSUB + 1 - NSUBIN$  otherwise. The first  $K$  locations contain pointers to the error estimates over the subintervals, such that  $ELIST(IORD(1))$ , ...,  $ELIST(IORD(K))$  form a decreasing sequence.

2. Informational errors

Type Code

4 1 The maximum number of subintervals allowed has been reached.

3 2 Roundoff error, preventing the requested tolerance from being achieved, has been detected.

3 3 A degradation in precision has been detected.

3 4 Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

4 5 Integral is divergent or slowly convergent.

3. If **EXACT** is the exact value, **QDAGI** attempts to find **RESULT** such that

$ABS(EXACT - RESULT) \leq MAX(ERRABS, ERRREL * ABS(EXACT))$ . To specify only a relative error, set **ERRABS** to zero. Similarly, to specify only an absolute error, set **ERRREL** to zero.

4. Since **QDAGI** makes a transformation of the original interval into the finite interval  $[0,1]$  the resulting function values can be extremely small and the resulting function might have “spikes”. In some cases **QDAGI** “overlooks” these spikes. The user can try adjusting the absolute and relative error tolerances to remedy this or, alternatively, try using IMSL routine [QDAG1D](#).

## Example

The value of

$$\int_0^{\infty} \frac{\ln(x)}{1+(10x)^2} dx = \frac{-\pi \ln(10)}{20}$$

is estimated. The values of the actual and estimated error are machine dependent. Note that we have requested an absolute error of 0 and a relative error of .001. The effect of these requests, as documented in Comment 3 above, is to ignore the absolute error requirement.

```

USE QDAGI_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER INTERV, NOUT
REAL ABS, ALOG, BOUND, ERRABS, ERREST, ERROR, &
 ERRREL, EXACT, F, PI, RESULT
INTRINSIC ABS, ALOG
EXTERNAL F

!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set limits of integration
BOUND = 0.0
INTERV = 1
!
! Set error tolerances
ERRABS = 0.0
CALL QDAGI (F, BOUND, INTERV, RESULT, ERRABS=ERRABS, &
 ERREST=ERREST)
!
! Print results
PI = CONST('PI')
EXACT = -PI*ALOG(10.)/20.
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3//' Error ', &
 'estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END

!
REAL FUNCTION F (X)
REAL X
REAL ALOG
INTRINSIC ALOG
F = ALOG(X) / (1.+(10.*X)**2)
RETURN
END

```

## Output

```

Computed = -0.362 Exact = -0.362
Error estimate = 2.652E-06 Error = 5.960E-08

```

---

## QDAWO

Integrates a function containing a sine or a cosine.

## Required Arguments

**F** — User-supplied function to be integrated. The form is  $F(x)$ , where

$x$  — Independent variable. (Input)

$F$  — The function value. (Output)

$F$  must be declared `EXTERNAL` in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)

**IWEIGH** — Type of weight function used. (Input)

| <b>IWEIGH</b> | <b>Weight</b>                  |
|---------------|--------------------------------|
| 1             | $\text{COS}(\text{OMEGA} * X)$ |
| 2             | $\text{SIN}(\text{OMEGA} * X)$ |

**OMEGA** — Parameter in the weight function. (Input)

**RESULT** — Estimate of the integral from  $A$  to  $B$  of  $F * \text{WEIGHT}$ . (Output)

## Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)  
Default:  $\text{ERRABS} = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Relative accuracy desired. (Input)  
Default:  $\text{ERRREL} = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAWO (F, A, B, IWEIGH, OMEGA, RESULT [, ...])`

Specific: The specific interface names are `S_QDAWO` and `D_QDAWO`.

## FORTRAN 77 Interface

Single: `CALL QDAWO (F, A, B, IWEIGH, OMEGA, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDAWO`.

## Description

The routine `QDAWO` uses a globally adaptive scheme in an attempt to reduce the absolute error. This routine computes integrals whose integrands have the special form  $w(x)f(x)$ , where  $w(x)$  is either  $\cos \omega x$  or  $\sin \omega x$ . Depending on the length of the subinterval in relation to the size of  $\omega$ , either a modified Clenshaw-Curtis procedure or a Gauss-Kronrod 7/15 rule is employed to approximate the integral on a subinterval. In addition to the general strategy described for the IMSL routine `QDAG`, this subroutine uses an extrapolation procedure known as the  $\epsilon$ -algorithm. The routine `QDAWO` is an implementation of the subroutine `QAWO`, which is fully documented by Piessens et al. (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `Q2AWO/DQ2AWO`. The reference is:

```
CALL Q2AWO (F, A, B, IWEIGH, OMEGA, ERRABS, ERRREL,
 RESULT, ERREST, MAXSUB, MAXCBY, NEVAL, NSUBIN, ALIST,
 BLIST, RLIST, ELIST, IORD, NNLOG, WK)
```

The additional arguments are as follows:

**MAXSUB** — Maximum number of subintervals allowed. (Input)  
A value of 390 is used by `QDAWO`.

**MAXCBY** — Upper bound on the number of Chebyshev moments which can be stored. That is, for the intervals of lengths  $ABS(B - A) * 2^{*(-L)}$ ,  $L = 0, 1, \dots, MAXCBY - 2$ ,  $MAXCBY.GE.1$ . The routine `QDAWO` uses 21. (Input)

**NEVAL** — Number of evaluations of `F`. (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length `MAXSUB` containing a list of the `NSUBIN` left endpoints. (Output)

**BLIST** — Array of length `MAXSUB` containing a list of the `NSUBIN` right endpoints. (Output)

**RLIST** — Array of length `MAXSUB` containing approximations to the `NSUBIN` integrals over the intervals defined by `ALIST`, `BLIST`. (Output)

**ELIST** — Array of length `MAXSUB` containing the error estimates of the `NSUBIN` values in `RLIST`. (Output)

**IORD** — Array of length `MAXSUB`. Let `K` be `NSUBIN` if `NSUBIN.LE.`  
`(MAXSUB/2 + 2)`, `MAXSUB + 1 - NSUBIN` otherwise. The first `K`

locations contain pointers to the error estimates over the subintervals, such that  $ELIST(IORD(1)), \dots, ELIST(IORD(K))$  form a decreasing sequence. (Output)

**NNLOG** — Array of length  $MAXSUB$  containing the subdivision levels of the subintervals, i.e.  $NNLOG(I) = L$  means that the subinterval numbered  $I$  is of length  $ABS(B - A) * (1 - L)$ . (Output)

**WK** — Array of length  $25 * MAXCBY$ . (Workspace)

2. Informational errors

Type Code

- |   |   |                                                                                                                        |
|---|---|------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The maximum number of subintervals allowed has been reached.                                                           |
| 3 | 2 | Roundoff error, preventing the requested tolerance from being achieved, has been detected.                             |
| 3 | 3 | A degradation in precision has been detected.                                                                          |
| 3 | 4 | Roundoff error in the extrapolation table, preventing the requested tolerances from being achieved, has been detected. |
| 4 | 5 | Integral is probably divergent or slowly convergent.                                                                   |

3. If **EXACT** is the exact value, **QDAWO** attempts to find **RESULT** such that  $ABS(EXACT - RESULT) \leq MAX(ERRABS, ERRREL * ABS(EXACT))$ . To specify only a relative error, set **ERRABS** to zero. Similarly, to specify only an absolute error, set **ERRREL** to zero.

### Example

The value of

$$\int_0^1 \ln(x) \sin(10\pi x) dx$$

is estimated. The values of the actual and estimated error are machine dependent. Notice that the log function is coded to protect for the singularity at zero.

```

USE QDAWO_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER IWEIGH, NOUT
REAL A, ABS, B, ERRABS, ERREST, ERROR, &
 EXACT, F, OMEGA, PI, RESULT
INTRINSIC ABS
EXTERNAL F
! Get output unit number
CALL UMACH (2, NOUT)
! Set limits of integration

```

```

A = 0.0
B = 1.0
! Weight function = sin(10.*pi*x)
IWEIGH = 2
PI = CONST('PI')
OMEGA = 10.*PI
! Set error tolerances
ERRABS = 0.0
CALL QDAWO (F, A, B, IWEIGH, OMEGA, RESULT, ERRABS=ERRABS, &
 ERREST=ERREST)
! Print results
EXACT = -0.1281316
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, /, /, &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END
!
REAL FUNCTION F (X)
REAL X
REAL ALOG
INTRINSIC ALOG
IF (X .EQ. 0.) THEN
 F = 0.0
ELSE
 F = ALOG(X)
END IF
RETURN
END

```

## Output

```

Computed = -0.128 Exact = -0.128
Error estimate = 7.504E-05 Error = 5.260E-06

```

---

## QDAWF

Computes a Fourier integral.

### Required Arguments

*F* — User-supplied FUNCTION to be integrated. The form is  
 $F(X)$ , where

*X* — Independent variable. (Input)

*F* — The function value. (Output)

*F* must be declared EXTERNAL in the calling program.

*A* — Lower limit of integration. (Input)

**IWEIGH** — Type of weight function used. (Input)

| <b>IWEIGH</b> | <b>Weight</b>                  |
|---------------|--------------------------------|
| 1             | $\text{COS}(\text{OMEGA} * X)$ |
| 2             | $\text{SIN}(\text{OMEGA} * X)$ |

**OMEGA** — Parameter in the weight function. (Input)

**RESULT** — Estimate of the integral from A to infinity of  $F * \text{WEIGHT}$ . (Output)

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default:  $\text{ERRABS} = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

Default:  $\text{ERREST} = 1.e-3$  for single precision and  $1.d-8$  for double precision.

### FORTRAN 90 Interface

Generic: `CALL QDAWF (F, A, IWEIGH, OMEGA, RESULT [, ...])`

Specific: The specific interface names are `S_QDAWF` and `D_QDAWF`.

### FORTRAN 77 Interface

Single: `CALL QDAWF (F, A, IWEIGH, OMEGA, ERRABS, RESULT, ERREST)`

Double: The double precision name is `DQDAWF`.

### Description

The routine `QDAWF` uses a globally adaptive scheme in an attempt to reduce the absolute error. This routine computes integrals whose integrands have the special form  $w(x)f(x)$ , where  $w(x)$  is either  $\cos \omega x$  or  $\sin \omega x$ . The integration interval is always semi-infinite of the form  $[A, \infty]$ . These Fourier integrals are approximated by repeated calls to the IMSL routine `QDAWO` followed by extrapolation. The routine `QDAWF` is an implementation of the subroutine `QAWF`, which is fully documented by Piessens et al. (1983).

### Comments

1. Workspace may be explicitly provided, if desired, by use of `Q2AWF/DQ2AWF`. The reference is:

```
CALL Q2AWF (F, A, IWEIGH, OMEGA, ERRABS, RESULT, ERREST,
MAXCYL, MAXSUB, MAXCBY, NEVAL, NCYCLE, RSLIST,
ERLIST, IERLST, NSUBIN, WK, IWK)
```

The additional arguments are as follows:

**MAXSUB** — Maximum number of subintervals allowed. (Input)  
A value of 365 is used by QDAWF.

**MAXCYL** — Maximum number of cycles allowed. (Input)  
MAXCYL must be at least 3. QDAWF uses 50.

**MAXCBY** — Maximum number of Chebyshev moments allowed. (Input)  
QDAWF uses 21.

**NEVAL** — Number of evaluations of F. (Output)

**NCYCLE** — Number of cycles used. (Output)

**RSLIST** — Array of length MAXCYL containing the contributions to the integral over the interval  $(A + (k - 1) * C, A + k * C)$ , for  $k = 1, \dots, NCYCLE$ . (Output)  
 $C = (2 * INT(ABS(OMEGA)) + 1) * PI / ABS(OMEGA)$ .

**ERLIST** — Array of length MAXCYL containing the error estimates for the intervals defined in RSLIST. (Output)

**IERLST** — Array of length MAXCYL containing error flags for the intervals defined in RSLIST. (Output)

**IERLST(K)**      **Meaning**

| <b>IERLST(K)</b> | <b>Meaning</b>                                                                                                                                                                                                       |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                | The maximum number of subdivisions (MAXSUB) has been achieved on the k-th cycle.                                                                                                                                     |
| 2                | Roundoff error prevents the desired accuracy from being achieved on the k-th cycle.                                                                                                                                  |
| 3                | Extremely bad integrand behavior occurs at some points of the k-th cycle.                                                                                                                                            |
| 4                | Integration procedure does not converge (to the desired accuracy) due to roundoff in the extrapolation procedure on the k-th cycle. It is assumed that the result on this interval is the best that can be obtained. |
| 5                | Integral over the k-th cycle is divergent or slowly convergent.                                                                                                                                                      |

**NSUBIN** — Number of subintervals generated. (Output)

**WK** — Work array of length  $4 * \text{MAXSUB} + 25 * \text{MAXCBY}$ .

**IWK** — Work array of length  $2 * \text{MAXSUB}$ .

2. Informational errors

| Type | Code |                                                                                                                                                                            |
|------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Bad integrand behavior occurred in one or more cycles.                                                                                                                     |
| 4    | 2    | Maximum number of cycles allowed has been reached.                                                                                                                         |
| 3    | 3    | Extrapolation table constructed for convergence acceleration of the series formed by the integral contributions of the cycles does not converge to the requested accuracy. |

3. If EXACT is the exact value, QDAWF attempts to find RESULT such that  $\text{ABS}(\text{EXACT} - \text{RESULT}) \leq \text{ERRABS}$ .

### Example

The value of

$$\int_0^{\infty} x^{-1/2} \cos(\pi x/2) dx = 1$$

is estimated. The values of the actual and estimated error are machine dependent. Notice that  $F$  is coded to protect for the singularity at zero.

```
USE QDAWF_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER IWEIGH, NOUT
REAL A, ABS, ERRABS, ERREST, ERROR, EXACT, F, &
 OMEGA, PI, RESULT
INTRINSIC ABS
EXTERNAL F
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set lower limit of integration
A = 0.0
!
! Select weight W(X) = COS(PI*X/2)
IWEIGH = 1
PI = CONST('PI')
OMEGA = PI/2.0
!
! Set error tolerance
CALL QDAWF (F, A, IWEIGH, OMEGA, RESULT, ERREST=ERREST)
!
! Print results
EXACT = 1.0
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, /, /, &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END
```

```

!
REAL FUNCTION F (X)
REAL X
REAL SQRT
INTRINSIC SQRT
IF (X .GT. 0.0) THEN
 F = 1.0/SQRT(X)
ELSE
 F = 0.0
END IF
RETURN
END

```

### Output

```

Computed = 1.000 Exact = 1.000
Error estimate = 6.267E-04 Error = 2.205E-06

```

## QDAWS

Integrates a function with algebraic-logarithmic singularities.

### Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(X)$ , where

**X** — Independent variable. (Input)

**F** — The function value. (Output)

**F** must be declared EXTERNAL in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)  
**B** must be greater than **A**

**IWEIGH** — Type of weight function used. (Input)

**IWEIGH**    **Weight**

1             $(X - A)**ALPHA * (B - X)**BETAW$

2             $(X - A)**ALPHA * (B - X)**BETAW * LOG(X - A)$

3             $(X - A)**ALPHA * (B - X)**BETAW * LOG(B - X)$

$$4 \quad (X - A)**ALPHA * (B - X)**BETAW * LOG(X - A) * LOG(B - X)$$

**ALPHA** — Parameter in the weight function. (Input)

ALPHA must be greater than  $-1.0$ .

**BETAW** — Parameter in the weight function. (Input)

BETAW must be greater than  $-1.0$ .

**RESULT** — Estimate of the integral from A to B of  $F * WEIGHT$ . (Output)

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default: ERRABS =  $1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Relative accuracy desired. (Input)

Default: ERRREL =  $1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

### FORTRAN 90 Interface

Generic:     CALL QDAWS (F, A, B, IWEIGH, ALPHA, BETAW, RESULT [, ...] )

Specific:    The specific interface names are S\_QDAWS and D\_QDAWS.

### FORTRAN 77 Interface

Single:      CALL QDAWS (F, A, B, IWEIGH, ALPHA, BETAW, ERRABS, ERRREL, RESULT,  
                  ERREST)

Double:      The double precision name is DQDAWS.

### Description

The routine QDAWS uses a globally adaptive scheme in an attempt to reduce the absolute error. This routine computes integrals whose integrands have the special form  $w(x)f(x)$ , where  $w(x)$  is a weight function described above. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas is employed. In addition to the general strategy described for the IMSL routine QDAG, this routine uses an extrapolation procedure known as the  $\epsilon$ -algorithm. The routine QDAWS is an implementation of the routine QAWS, which is fully documented by Piessens et al. (1983).

### Comments

1.     Workspace may be explicitly provided, if desired, by use of Q2AWS/DQ2AWS. The reference is

```
CALL Q2AWS (F, A, B, IWEIGH, ALPHA, BETAW, ERRABS, ERRREL,
 RESULT, ERREST, MAXSUB, NEVAL, NSUBIN, ALIST, BLIST,
 RLIST, ELIST, IORD)
```

The additional arguments are as follows:

**MAXSUB** — Maximum number of subintervals allowed. (Input)  
A value of 500 is used by QDAWS.

**NEVAL** — Number of evaluations of  $F$ . (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** left endpoints. (Output)

**BLIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** right endpoints. (Output)

**RLIST** — Array of length **MAXSUB** containing approximations to the **NSUBIN** integrals over the intervals defined by **ALIST**, **BLIST**. (Output)

**ELIST** — Array of length **MAXSUB** containing the error estimates of the **NSUBIN** values in **RLIST**. (Output)

**IORD** — Array of length **MAXSUB**. Let  $k$  be **NSUBIN** if **NSUBIN.LE**. ( $\text{MAXSUB}/2 + 2$ ),  $\text{MAXSUB} + 1 - \text{NSUBIN}$  otherwise. The first  $k$  locations contain pointers to the error estimates over the subintervals, such that  $\text{ELIST}(\text{IORD}(1)), \dots, \text{ELIST}(\text{IORD}(k))$  form a decreasing sequence. (Output)

2. Informational errors

Type Code

4 1 The maximum number of subintervals allowed has been reached.

3 2 Roundoff error, preventing the requested tolerance from being achieved, has been detected.

3 3 A degradation in precision has been detected.

3. If **EXACT** is the exact value, QDAWS attempts to find **RESULT** such that  $\text{ABS}(\text{EXACT} - \text{RESULT}) \cdot \text{LE} \cdot \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set **ERRABS** to zero. Similarly, to specify only an absolute error, set **ERRREL** to zero.

## Example

The value of

$$\int_0^1 [(1+x)(1-x)]^{1/2} x \ln(x) dx = \frac{3\ln(2)-4}{9}$$

is estimated. The values of the actual and estimated error are machine dependent.

```

USE QDAWS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IWEIGH, NOUT
REAL A, ABS, ALOG, ALPHA, B, BETAW, ERRABS, ERREST, ERROR, &
 EXACT, F, RESULT
INTRINSIC ABS, ALOG
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

! Set limits of integration
A = 0.0
B = 1.0

! Select weight
ALPHA = 1.0
BETAW = 0.5
IWEIGH = 2

! Set error tolerances
ERRABS = 0.0
CALL QDAWS (F, A, B, IWEIGH, ALPHA, BETAW, RESULT, &
 ERRABS=ERRABS, ERREST=ERREST)

! Print results
EXACT = (3.*ALOG(2.)-4.)/9.
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, /, /, &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END

!
REAL FUNCTION F (X)
REAL X
REAL SQRT
INTRINSIC SQRT
F = SQRT(1.0+X)
RETURN
END

```

### Output

```

Computed = -0.213 Exact = -0.213
Error estimate = 1.261E-08 Error = 2.980E-08

```

---

## QDAWC

Integrates a function  $f(x)/(x-c)$  in the Cauchy principal value sense.

## Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(x)$ , where

$x$  — Independent variable. (Input)

$F$  — The function value. (Output)

$F$  must be declared EXTERNAL in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)

**C** — Singular point. (Input)  
 $C$  must not equal  $A$  or  $B$ .

**RESULT** — Estimate of the integral from  $A$  to  $B$  of  $F(x)/(x - C)$ . (Output)

## Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)  
Default:  $ERRABS = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Relative accuracy desired. (Input)  
Default:  $ERRREL = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAWC (F, A, B, C, RESULT [, ...])`

Specific: The specific interface names are `S_QDAWC` and `D_QDAWC`.

## FORTRAN 77 Interface

Single: `CALL QDAWC (F, A, B, C, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDAWC`.

## Description

The routine QDAWC uses a globally adaptive scheme in an attempt to reduce the absolute error. This routine computes integrals whose integrands have the special form  $w(x)f(x)$ , where  $w(x) = 1/(x - c)$ . If  $c$  lies in the interval of integration, then the integral is interpreted as a Cauchy principal value. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas are

employed. In addition to the general strategy described for the IMSL routine [QDAG](#), this routine uses an extrapolation procedure known as the  $\epsilon$ -algorithm. The routine [QDAWC](#) is an implementation of the subroutine [QAWC](#), which is fully documented by Piessens et al. (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of [Q2AWC/DQ2AWC](#). The reference is:

```
CALL Q2AWC (F, A, B, C, ERRABS, ERRREL, RESULT, ERREST,
 MAXSUB, NEVAL, NSUBIN, ALIST, BLIST, RLIST, ELIST,
 IORD)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)  
A value of 500 is used by [QDAWC](#).

**NEVAL** — Number of evaluations of  $F$ . (Output)

**NSUBIN** — Number of subintervals generated. (Output)

**ALIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** left endpoints. (Output)

**BLIST** — Array of length **MAXSUB** containing a list of the **NSUBIN** right endpoints. (Output)

**RLIST** — Array of length **MAXSUB** containing approximations to the **NSUBIN** integrals over the intervals defined by **ALIST**, **BLIST**. (Output)

**ELIST** — Array of length **MAXSUB** containing the error estimates of the **NSUBIN** values in **RLIST**. (Output)

**IORD** — Array of length **MAXSUB**. (Output)

Let  $k$  be **NSUBIN** if  $\text{NSUBIN} \leq (\text{MAXSUB}/2 + 2)$ ,  $\text{MAXSUB} + 1 - \text{NSUBIN}$  otherwise. The first  $k$  locations contain pointers to the error estimates over the subintervals, such that  $\text{ELIST}(\text{IORD}(1))$ , ...,  $\text{ELIST}(\text{IORD}(k))$  form a decreasing sequence.

2. Informational errors

Type Code

|   |   |                                                                                            |
|---|---|--------------------------------------------------------------------------------------------|
| 4 | 1 | The maximum number of subintervals allowed has been reached.                               |
| 3 | 2 | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| 3 | 3 | A degradation in precision has been detected.                                              |

3. If EXACT is the exact value, QDAWC attempts to find RESULT such that  $\text{ABS}(\text{EXACT} - \text{RESULT}) \leq \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.

## Example

The Cauchy principal value of

$$\int_{-1}^5 \frac{1}{x(5x^3 + 6)} dx = \frac{\ln(125/631)}{18}$$

is estimated. The values of the actual and estimated error are machine dependent.

```

USE QDAWC_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
REAL A, ABS, ALOG, B, C, ERRABS, ERREST, ERROR, EXACT, &
 F, RESULT
INTRINSIC ABS, ALOG
EXTERNAL F

!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set limits of integration and C
A = -1.0
B = 5.0
C = 0.0
!
! Set error tolerances
ERRABS = 0.0
CALL QDAWC (F, A, B, C, RESULT, ERRABS=ERRABS, ERREST=ERREST)
!
! Print results
EXACT = ALOG(125./631.)/18.
ERROR = 2*ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR
99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, '/', '/', &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END
!
REAL FUNCTION F (X)
REAL X
F = 1.0/(5.*X**3+6.0)
RETURN
END

```

## Output

```

Computed = -0.090 Exact = -0.090
Error estimate = 2.022E-06 Error = 2.980E-08

```

---

# QDNG

Integrates a smooth function using a nonadaptive rule.

## Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(x)$ , where

$x$  — Independent variable. (Input)

$F$  — The function value. (Output)

$F$  must be declared EXTERNAL in the calling program.

**A** — Lower limit of integration. (Input)

**B** — Upper limit of integration. (Input)

**RESULT** — Estimate of the integral from  $A$  to  $B$  of  $F$ . (Output)

## Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default:  $ERRABS = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Relative accuracy desired. (Input)

Default:  $ERRREL = 1.e-3$  for single precision and  $1.d-8$  for double precision.

**ERREST** — Estimate of the absolute value of the error. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDNG (F, A, B, RESULT [, ...])`

Specific: The specific interface names are `S_QDNG` and `D_QDNG`.

## FORTRAN 77 Interface

Single: `CALL QDNG (F, A, B, ERRABS, ERRREL, RESULT, ERREST)`

Double: The double precision name is `DQDNG`.

## Description

The routine `QDNG` is designed to integrate smooth functions. This routine implements a nonadaptive quadrature procedure based on nested Paterson rules of order 10, 21, 43, and 87. These rules are positive quadrature rules with degree of accuracy 19, 31, 64, and 130, respectively. The routine `QDNG` applies these rules successively, estimating the error, until either the error

estimate satisfies the user-supplied constraints or the last rule is applied. The routine QDNG is based on the routine QNG by Piessens et al. (1983).

This routine is not very robust, but for certain smooth functions it can be efficient. If QDNG should not perform well, we recommend the use of the IMSL routine QDAGS.

## Comments

1. Informational error
 

| Type | Code |                                                                                              |
|------|------|----------------------------------------------------------------------------------------------|
| 4    | 1    | The maximum number of steps allowed have been taken. The integral is too difficult for QDNG. |
2. If EXACT is the exact value, QDNG attempts to find RESULT such that  $\text{ABS}(\text{EXACT} - \text{RESULT}) \leq \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.
3. This routine is designed for efficiency, not robustness. If the above error is encountered, try QDAGS.

## Example

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is estimated. The values of the actual and estimated error are machine dependent.

```

USE QDNG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
REAL A, ABS, B, ERRABS, ERREST, ERROR, EXACT, EXP, &
 F, RESULT
INTRINSIC ABS, EXP
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

! Set limits of integration
A = 0.0
B = 2.0

! Set error tolerances
ERRABS = 0.0
CALL QDNG (F, A, B, RESULT, ERRABS=ERRABS, ERREST=ERREST)

! Print results
EXACT = 1.0 + EXP(2.0)
ERROR = ABS(RESULT-EXACT)
WRITE (NOUT,99999) RESULT, EXACT, ERREST, ERROR

```

```

99999 FORMAT (' Computed =', F8.3, 13X, ' Exact =', F8.3, /, /, &
 ' Error estimate =', 1PE10.3, 6X, 'Error =', 1PE10.3)
END
!
REAL FUNCTION F (X)
REAL X
REAL EXP
INTRINSIC EXP
F = X*EXP(X)
RETURN
END

```

## Output

```

Computed = 8.389 Exact = 8.389
Error estimate = 5.000E-05 Error = 9.537E-07

```

# TWODQ

Computes a two-dimensional iterated integral.

## Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(X, Y)$ , where

**X** — First argument of F. (Input)

**Y** — Second argument of F. (Input)

**F** — The function value. (Output)

**F** must be declared EXTERNAL in the calling program.

**A** — Lower limit of outer integral. (Input)

**B** — Upper limit of outer integral. (Input)

**G** — User-supplied FUNCTION to evaluate the lower limits of the inner integral. The form is  $G(X)$ , where

**X** — Only argument of G. (Input)

**G** — The function value. (Output)

**G** must be declared EXTERNAL in the calling program.

**H** — User-supplied FUNCTION to evaluate the upper limits of the inner integral. The form is  $H(X)$ , where

**X** — Only argument of H. (Input)

**H** — The function value. (Output)

H must be declared `EXTERNAL` in the calling program.

**RESULT** — Estimate of the integral from A to B of F. (Output)

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)

Default: `ERRABS = 1.e-3` for single precision and `1.d-8` for double precision.

**ERRREL** — Relative accuracy desired. (Input)

Default: `ERRREL = 1.e-3` for single precision and `1.d-8` for double precision.

**IRULE** --- Choice of quadrature rule. (Input)

Default: `IRULE = 2`.

The Gauss-Kronrod rule is used with the following points:

| <b>IRULE</b> | <b>Points</b> |
|--------------|---------------|
| 1            | 7-15          |
| 2            | 10-21         |
| 3            | 15-31         |
| 4            | 20-41         |
| 5            | 25-51         |
| 6            | 30-61         |

If the function has a peak singularity, use `IRULE = 1`. If the function is oscillatory, use `IRULE = 6`.

**ERREST** — Estimate of the absolute value of the error. (Output)

### FORTRAN 90 Interface

Generic: `CALL TWODQ (F, A, B, G, H, RESULT [, ...])`

Specific: The specific interface names are `S_TWODQ` and `D_TWODQ`.

### FORTRAN 77 Interface

Single: `CALL TWODQ (F, A, B, G, H, ERRABS, ERRREL, IRULE, RESULT, ERREST)`

Double: The double precision name is `DTWODQ`.

### Description

The routine `TWODQ` approximates the two-dimensional iterated integral

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y) dy dx$$

with the approximation returned in `RESULT`. An estimate of the error is returned in `ERREST`. The approximation is achieved by iterated calls to `QDAG`. Thus, this algorithm will share many of the characteristics of the routine `QDAG`. As in `QDAG`, several options are available. The absolute and relative error must be specified, and in addition, the Gauss-Kronrod pair must be specified (`IRULE`). The lower-numbered rules are used for less smooth integrands while the higher-order rules are more efficient for smooth (oscillatory) integrands.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `T2ODQ/DT2ODQ`. The reference is:

```
CALL T2ODQ (F, A, B, G, H, ERRABS, ERRREL, IRULE, RESULT,
 ERREST, MAXSUB, NEVAL, NSUBIN, ALIST, BLIST, RLIST,
 ELIST, IORD, WK, IWK)
```

The additional arguments are as follows:

**MAXSUB** — Number of subintervals allowed. (Input)  
A value of 250 is used by `TWODQ`.

**NEVAL** — Number of evaluations of `F`. (Output)

**NSUBIN** — Number of subintervals generated in the outer integral. (Output)

**ALIST** — Array of length `MAXSUB` containing a list of the `NSUBIN` left endpoints for the outer integral. (Output)

**BLIST** — Array of length `MAXSUB` containing a list of the `NSUBIN` right endpoints for the outer integral. (Output)

**RLIST** — Array of length `MAXSUB` containing approximations to the `NSUBIN` integrals over the intervals defined by `ALIST`, `BLIST`, pertaining only to the outer integral. (Output)

**ELIST** — Array of length `MAXSUB` containing the error estimates of the `NSUBIN` values in `RLIST`. (Output)

**IORD** — Array of length `MAXSUB`. (Output)

Let `K` be `NSUBIN` if `NSUBIN.LE.(MAXSUB/2 + 2)`, `MAXSUB + 1 - NSUBIN` otherwise. Then the first `K` locations contain pointers to the error estimates over the corresponding subintervals, such that `ELIST(IORD(1)), ..., ELIST(IORD(K))` form a decreasing sequence.

**WK** — Work array of length `4 * MAXSUB`, needed to evaluate the inner integral.

**IWK** — Work array of length MAXSUB, needed to evaluate the inner integral.

2. Informational errors

Type Code

- |   |   |                                                                                            |
|---|---|--------------------------------------------------------------------------------------------|
| 4 | 1 | The maximum number of subintervals allowed has been reached.                               |
| 3 | 2 | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| 3 | 3 | A degradation in precision has been detected.                                              |

3. If EXACT is the exact value, TWODQ attempts to find RESULT such that

$\text{ABS}(\text{EXACT} - \text{RESULT}) \leq \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.

### Example 1

In this example, we approximate the integral

$$\int_0^1 \int_1^3 y \cos(x + y^2) dy dx$$

The value of the error estimate is machine dependent.

```

USE TWODQ_INT
USE UMACH_INT
IMPLICIT NONE
INTEGER IRULE, NOUT
REAL A, B, ERRABS, ERREST, ERRREL, F, G, H, RESULT
EXTERNAL F, G, H
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set limits of integration
A = 0.0
B = 1.0
!
! Set error tolerances
ERRABS = 0.0
ERRREL = 0.01
!
! Parameter for oscillatory function
IRULE = 6
CALL TWODQ (F, A, B, G, H, RESULT, ERRABS, ERRREL, IRULE, ERREST)
!
! Print results
WRITE (NOUT,99999) RESULT, ERREST
99999 FORMAT (' Result =', F8.3, 13X, ' Error estimate = ', 1PE9.3)
END
!
REAL FUNCTION F (X, Y)
REAL X, Y
REAL COS
INTRINSIC COS
F = Y * COS (X + Y * Y)

```

```

 RETURN
 END
!
 REAL FUNCTION G (X)
 REAL X
 G = 1.0
 RETURN
 END
!
 REAL FUNCTION H (X)
 REAL X
 H = 3.0
 RETURN
 END

```

## Output

Result = -0.514                      Error estimate = 3.065E-06

## Additional Examples

### Example 2

We modify the above example by assuming that the limits for the inner integral depend on  $x$  and, in particular, are  $g(x) = -2x$  and  $h(x) = 5x$ . The integral now becomes

$$\int_0^1 \int_{-2x}^{5x} y \cos(x + y^2) dy dx$$

The value of the error estimate is machine dependent.

```

 USE TWODQ_INT
 USE UMACH_INT
!
! Declare F, G, H
 INTEGER IRULE, NOUT
 REAL A, B, ERRABS, ERREST, ERRREL, F, G, H, RESULT
 EXTERNAL F, G, H
!
 CALL UMACH (2, NOUT)
! Set limits of integration
 A = 0.0
 B = 1.0
! Set error tolerances
 ERRABS = 0.001
 ERRREL = 0.0
! Parameter for oscillatory function
 IRULE = 6
 CALL TWODQ (F, A, B, G, H, RESULT, ERRABS, ERRREL, IRULE, ERREST)
! Print results
 WRITE (NOUT,99999) RESULT, ERREST
99999 FORMAT (' Computed =', F8.3, 13X, ' Error estimate = ', 1PE9.3)
 END
 REAL FUNCTION F (X, Y)
 REAL X, Y
!
 REAL COS

```

```

 INTRINSIC COS
!
 F = Y*COS (X+Y*Y)
 RETURN
 END
 REAL FUNCTION G (X)
 REAL X
!
 G = -2.0*X
 RETURN
 END
 REAL FUNCTION H (X)
 REAL X
!
 H = 5.0*X
 RETURN
 END

```

### Output

Computed = -0.083

Error estimate = 2.095E-06

## QDAG2D

Integrates a function of two variables with a possible internal or end point singularity.

### Required Arguments

*F* — User-supplied FUNCTION to be integrated. The form is  $F(X, Y [, \dots])$ , where

#### Function Return Value

*F* — The function value. (Output)

#### Required Arguments

*X* — Independent variable. (Input)

*Y* — Independent variable. (Input)

#### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

*F* must be declared EXTERNAL in the calling program.

*A* — Lower limit of integration for outer dimension. (Input)

**B** — Upper limit of integration. The relative values of **A** and **B** are interpreted properly. Thus if one exchanges **A** and **B**, the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of  $B - A$ . (Input)

**G** — User-supplied FUNCTION to compute the lower limit of integration for the inner dimension. The form is  $G(X [, \dots])$ , where

#### Function Return Value

**G** — The function value. (Output)

#### Required Arguments

**X** — Independent variable. (Input)

#### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

**G** must be declared EXTERNAL in the calling program.

**H** — User-supplied FUNCTION to compute the upper limit of integration for the inner dimension. The form is  $H(X [, \dots])$ , where

#### Function Return Value

**H** — The function value. (Output)

#### Required Arguments

**X** — Independent variable. (Input)

#### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

**H** must be declared EXTERNAL in the calling program.

**RESULT** — Estimate of the integral from **A** to **B** of the integral from  $G(X)$  to  $H(X)$  of  $G(X, Y)$ . (Output)

### Optional Arguments

**ERRABS** — Absolute error tolerance. See [Comment 1](#) for a discussion of the error tolerances. (Input)

Default: `ERRABS = 0.0`.

**ERRFRAC** — A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See [Comment 1](#) for a discussion of the error tolerances. (Input)  
Default: `ERRFRAC = 0.75`.

**ERRREL** — Relative error tolerance. See [Comment 1](#) for a discussion of the error tolerances. (Input)  
Default: `ERRABS = 0.0`.

**ERRPOST** — An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. This value may be computed during the evaluation of the integrand. When this optional argument is used, `FCN_DATA` must also be used as `FCN_DATA%RDATA(1)` will be used to pass the newly calculated value of `ERRPOST` back from the evaluator, `F`. In this case, the user should not use `FCN_DATA%RDATA(1)` for passing other data. (Input)  
Default: `ERRPOST = 0.0`.

**ERRPRIOR** — An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral. (Input)  
Default: `ERRPRIOR = 1.19e-7` for single precision and `2.22d-16` for double precision.

**MAXFCN** — The maximum number of function values to use to compute the integral. (Input)  
Default: The number of function values is not bounded.

**SINGULARITY** — The real part of the abscissa of a singularity or discontinuity in the innermost integrand. If this option is used, `SINGULARITY_TYPE` must also be used. (Input)  
Default: It is assumed that there is no singularity in the innermost integrand so `SINGULARITY` is not set. It is an error to set `SINGULARITY` without also setting `SINGULARITY_TYPE`.

**SINGULARITY\_TYPE** — A signed integer specifying the type of singularity which occurs in the innermost integrand. If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not an integer, if  $\alpha$  is “large” or has the form  $\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a positive integer. Otherwise, set `SINGULARITY_TYPE` to a negative integer. (Input)  
Default: It is assumed that there is no singularity in the innermost integrand so `SINGULARITY_TYPE` is not set. It is an error to set `SINGULARITY_TYPE` without also setting `SINGULARITY`.

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. The derived type, `s_fcn_data`, is defined as:

```
type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
```

```

integer, pointer, dimension(:) :: idata
end type

```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type. (Input/Output)

**NEVAL** — Number of function evaluations used to calculate the integral. (Output)

**ERREST** — An estimate of the upper bound of the magnitude of the difference between `RESULT` and the true value of the integral. (Output)

**ISTATUS** — A status flag indicating the error criteria which was satisfied on exit.

`ISTATUS = -1` indicates normal termination with either the absolute or relative error tolerance criteria satisfied.

`ISTATUS = -2` indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied.

`ISTATUS = -3` indicates normal termination with none of the error tolerance criteria satisfied.

`ISTATUS = any value other than the above` indicates abnormal termination due to an error condition. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAG2D(F, A, B, G, H, RESULT [...])`

Specific: The specific interface names are `S_QDAG2D` and `D_QDAG2D`.

## Description

`QDAG2D`, based on the JPL Library routine `SINTM`, approximates an iterated two-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y) dy dx$$

The integral over two dimensions is computed by repeated integration over one dimension. The integration over one dimension is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the  $k^{\text{th}}$  formula used all the integrand values used in the  $(k-1)^{\text{st}}$  formula, and added  $2^{k-1}$  new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to  $k = 8$ .

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

## Comments

1. The user provides the absolute error tolerance through optional argument `ERRABS`. Optional argument `ERRFRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). The internal value for `ERRFRAC` is bounded between .5 and 1. The error tolerance relative to the value of the integral is specified via optional argument `ERRREL`. By default, `ERRABS` and `ERRREL` are set to 0.0 and `ERRFRAC` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.

The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.

2. Optional arguments `SINGULARITY` and `SINGULARITY_TYPE` provide the user with a means to give the routine information about the location and type of any known singularity of the innermost integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside  $[A, B]$ . If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not an integer, if  $\alpha$  is "large" or has the form  $\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a

positive value. Otherwise, set `SINGULARITY_TYPE` to a negative value. The meaning of “large” depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered “large”. For a singularity of the form  $x^\alpha$   $\log x$  use the above rule, even if  $\alpha$  is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed, some experimentation may be useful.

When `SINGULARITY_TYPE` is positive, a transformation of the form  $T = TA + (X - TA)^2 / (TB - TA)$  is applied, where  $TA$  is the abscissa of the singularity and  $TB$  is the end of the interval. If  $TA$  is outside the interval,  $TB$  will be the end of the interval farthest from  $TA$ . If  $TA$  is inside the interval, the interval will immediately be subdivided at  $TA$ , and both parts will be separately integrated with  $TB$  equal to each end of the original interval, respectively. When `SINGULARITY_TYPE` is negative, a transformation of the form  $T = TA + (X - TA)^4 / (TB - TA)^3$  is applied, with  $TA$  and  $TB$  as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

## Example

The value of

$$\int_0^1 \int_1^3 y \cos(x + y^2) dy dx$$

is estimated.

```

USE QDAG2D_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER NOUT

REAL A, B, ERREST, F, G, H, RESULT

EXTERNAL F, G, H
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Set limits of integration
A = 0.0
B = 1.0
!
! Set singularity value and type
CALL QDAG2D (F, A, B, G, H, RESULT, ERREST=ERREST)
!
! Print the results
WRITE(NOUT,*) 'Result = ', RESULT
WRITE(NOUT,9999) ERREST
9999 FORMAT('Error Estimate = ', 1PE9.1)

```

```

END

REAL FUNCTION F (X, Y)
REAL X, Y
REAL COS
INTRINSIC COS
F = Y*COS (X+Y*Y)
RETURN
END

REAL FUNCTION G (X)
REAL X
G = 1.0
RETURN
END

REAL FUNCTION H (X)
REAL X
H = 3.0
RETURN
END

```

## Output

```

RESULT = -0.51425
Error Estimate = 5.3-06

```

---

# QDAG3D

Integrates a function of three variables with a possible internal or endpoint singularity.

## Required Arguments

*F* — User-supplied FUNCTION to be integrated. The form is  $F(X, Y, Z [, \dots])$ , where

### Function Return Value

*F* — The function value. (Output)

### Required Arguments

*X* — Independent variable. (Input)

*Y* — Independent variable. (Input)

*Z* — Independent variable. (Input)

### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

$F$  must be declared `EXTERNAL` in the calling program.

$A$  — Lower limit of integration for outer dimension. (Input)

$B$  — Upper limit of integration for outer dimension. The relative values of  $A$  and  $B$  are interpreted properly. Thus if one exchanges  $A$  and  $B$ , the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of  $B - A$ . (Input)

$G$  — User-supplied `FUNCTION` to compute the lower limit of integration for the middle dimension. The form is  $G(X [, \dots])$ , where

#### Function Return Value

$G$  — The function value. (Output)

#### Required Arguments

$X$  — Independent variable. (Input)

#### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

$G$  must be declared `EXTERNAL` in the calling program.

$H$  — User-supplied `FUNCTION` to compute the upper limit of integration for the middle dimension. The form is  $H(X [, \dots])$ , where

#### Function Return Value

$H$  — The function value. (Output)

#### Required Arguments

$X$  — Independent variable. (Input)

#### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

$H$  must be declared `EXTERNAL` in the calling program

$P$  — User-supplied `FUNCTION` to compute the lower limit of integration for the inner dimension. The form is  $P(X, Y [, \dots])$ , where

#### Function Return Value

$P$  — The function value. (Output)

### Required Arguments

$X$  — Independent variable. (Input)

$Y$  — Independent variable. (Input)

### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

$P$  must be declared `EXTERNAL` in the calling program.

$Q$  — User-supplied `FUNCTION` to compute the upper limit of integration for the inner dimension. The form is `Q(X, Y [...])`, where

### Function Return Value

$Q$  — The function value. (Output)

### Required Arguments

$X$  — Independent variable. (Input)

$Y$  — Independent variable. (Input)

### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

$Q$  must be declared `EXTERNAL` in the calling program

**RESULT** — Estimate of the integral from  $A$  to  $B$  of the integral from  $G(X)$  to  $H(X)$  of the integral from  $P(X,Y)$  to  $Q(X,Y)$  of  $F(X,Y,Z)$ . (Output)

### Optional Arguments

**ERRABS** — Absolute error tolerance. See [Comment 1](#) for a discussion of the error tolerances. (Input)

Default: `ERRABS = 0.0`.

**ERRFRAC** — A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See [Comment 1](#) for a discussion of the error tolerances. (Input)

Default: `ERRFRAC = 0.75`.

**ERRREL** — The error tolerance relative to the value of the integral. See [Comment 1](#) for a discussion of the error tolerances. (Input)

Default: `ERRREL = 0.0`.

**ERRPOST** — An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. This value may be computed during the evaluation of the integrand. When this optional argument is used, `FCN_DATA` must also be used as `FCN_DATA%RDATA(1)` will be used to pass the newly calculated value of `ERRPOST` back from the evaluator, `F`. In this case, the user should not use `FCN_DATA%RDATA(1)` for passing other data. (Input)  
 Default: `ERRPOST = 0.0`.

**ERRPRIOR** — An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral. (Input)  
 Default: `ERRPRIOR = 1.19e-7` for single precision and `2.22d-16` for double precision.

**MAXFCN** — The maximum number of function values to use to compute the integral. (Input)  
 Default: The number of function values is not bounded.

**SINGULARITY** — The real part of the abscissa of a singularity or discontinuity in the innermost integrand. If this option is used, `SINGULARITY_TYPE` must also be used. (Input)  
 Default: It is assumed that there is no singularity in the innermost integrand so `SINGULARITY` is not set. It is an error to set `SINGULARITY` without also setting `SINGULARITY_TYPE`.

**SINGULARITY\_TYPE** — A signed integer specifying the type of singularity which occurs in the innermost integrand. If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not an integer, if  $\alpha$  is “large” or has the form  $\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a positive integer. Otherwise, set `SINGULARITY_TYPE` to a negative integer. (Input)  
 Default: It is assumed that there is no singularity in the innermost integrand so `SINGULARITY_TYPE` is not set. It is an error to set `SINGULARITY_TYPE` without also setting `SINGULARITY`.

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. The derived type, `s_fcn_data`, is defined as:

```

type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type

```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type. (Input/Output)

**NEVAL** — Number of function evaluations used to calculate the integral. (Output)

**ERREST** — An estimate of the upper bound of the magnitude of the difference between `RESULT` and the true value of the integral. (Output)

**ISTATUS** — A status flag indicating the error criteria which was satisfied on exit.

`ISTATUS` = -1 indicates normal termination with either the absolute or relative error tolerance criteria satisfied.

`ISTATUS` = -2 indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied.

`ISTATUS` = -3 indicates normal termination with none of the error tolerance criteria satisfied.

`ISTATUS` = any value other than the above indicates abnormal termination due to an error condition. (Output)

## FORTRAN 90 Interface

Generic: `CALL QDAG3D(F, A, B, G, H, P, Q, RESULT [...])`

Specific: The specific interface names are `S_QDAG3D` and `D_QDAG3D`.

## Description

`QDAG3D`, based on the JPL Library routine `SINTM`, approximates an iterated three-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} \int_{p(x,y)}^{q(x,y)} f(x, y, z) dz dy dx$$

The integral over three dimensions is computed by repeated integration over one dimension. The integration over one dimension is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the  $k^{\text{th}}$  formula used all the integrand values used in the  $(k-1)^{\text{st}}$  formula, and added  $2^{k-1}$  new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to  $k = 8$ .

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

## Comments

1. The user provides the absolute error tolerance through optional argument `ERRABS`. Optional argument `ERRFRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). Optional argument `ERRREL` represents the error tolerance relative to the value of the integral. The internal value for `ERRFRAC` is bounded between .5 and 1. By default, `ERRABS` and `ERRREL` are set to 0.0 and `ERRFRAC` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.

The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.

2. Optional arguments `SINGULARITY` and `SINGULARITY_TYPE` provide the user with a means to give the routine information about the location and type of any known singularity of the innermost integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside  $[A, B]$ . If the singularity has a leading term of the form  $x^\alpha$  where  $\alpha$  is not an integer, if  $\alpha$  is “large” or has the form  $\alpha = (2n-1)/2$  where  $n$  is a nonnegative integer, or the singularity is well outside the interval, set `SINGULARITY_TYPE` to a positive value. Otherwise, set `SINGULARITY_TYPE` to a negative value. The meaning of “large” depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered “large”. For a singularity of the form  $x^\alpha \log x$  use the above rule, even if  $\alpha$  is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed,

some experimentation may be useful.

When `SINGULARITY_TYPE` is positive, a transformation of the form  $T = TA + (X - TA)^2 / (TB - TA)$  is applied, where  $TA$  is the abscissa of the singularity and  $TB$  is the end of the interval. If  $TA$  is outside the interval,  $TB$  will be the end of the interval farthest from  $TA$ . If  $TA$  is inside the interval, the interval will immediately be subdivided at  $TA$ , and both parts will be separately integrated with  $TB$  equal to each end of the original interval, respectively. When `SINGULARITY_TYPE` is negative, a transformation of the form  $T = TA + (X - TA)^4 / (TB - TA)^3$  is applied, with  $TA$  and  $TB$  as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

## Example

The value of

$$\int_0^1 \int_0^{1-x} \int_0^{1-x-y} (1.0 + x + y + 2z) dz dy dx$$

is estimated.

```

USE QDAG3D_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER NOUT
REAL A, B, ERREST, F, G, H, P, Q, RESULT
EXTERNAL F, G, H, P, Q
! Get output unit number
CALL UMACH (2, NOUT)
! Set limits of integration
A = 0.0
B = 1.0
! Set singularity value and type
CALL QDAG3D (F, A, B, G, H, P, Q, RESULT, &
ERREST=ERREST)
! Print the results
WRITE(NOUT,*) 'Result = ', RESULT
WRITE(NOUT,9999) ERREST
9999 FORMAT('Error Estimate = ', 1PE9.1)
END

REAL FUNCTION F (X, Y, Z)
REAL X, Y, Z

```

```

F = 1.0 + X + Y + 2.0*Z
RETURN
END

REAL FUNCTION G (X)
REAL X
G = 0.0
RETURN
END

REAL FUNCTION H (X)
REAL X
H = 1.0 - X
RETURN
END

REAL FUNCTION P (X, Y)
REAL X, Y
P = 0.0
RETURN
END

REAL FUNCTION Q (X, Y)
REAL X, Y
Q = 1.0 - X - Y
RETURN
END

```

## Output

```

RESULT = 0.333333
Error Estimate = 1.9E-07

```

---

## QAND

Integrates a function on a hyper-rectangle.

### Required Arguments

**F** — User-supplied FUNCTION to be integrated. The form is  $F(N, X)$ , where

**N** — The dimension of the hyper-rectangle. (Input)

**X** — The independent variable of dimension N. (Input)

**F** — The value of the integrand at X. (Output)

**F** must be declared EXTERNAL in the calling program.

**N** — The dimension of the hyper-rectangle. (Input)  
**N** must be less than or equal to 20.

**A** — Vector of length *N*. (Input)  
Lower limits of integration.

**B** — Vector of length *N*. (Input)  
Upper limits of integration.

**RESULT** — Estimate of the integral from *A* to *B* of *F*. (Output)  
The integral of *F* is approximated over the *N*-dimensional hyper-rectangle  
*A*.*LE*.*X*.*LE*.*B*.

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)  
Default: *ERRABS* = 1.e-3 for single precision and 1.d-8 for double precision.

**ERRREL** — Relative accuracy desired. (Input)  
Default: *ERRREL* = 1.e-3 for single precision and 1.d-8 for double precision.

**MAXFCN** — Approximate maximum number of function evaluations to be permitted.  
(Input)  
*MAXFCN* cannot be greater than  $256^N$  or *IMACH*(5) if *N* is greater than 3.  
Default: *MAXFCN* = 32\**N*.

**ERREST** — Estimate of the absolute value of the error. (Output)

### FORTRAN 90 Interface

Generic:     CALL *QAND* (*F*, *N*, *A*, *B*, *RESULT* [, ...])

Specific:    The specific interface names are *S\_QAND* and *D\_QAND*.

### FORTRAN 77 Interface

Single:     CALL *QAND* (*F*, *N*, *A*, *B*, *ERRABS*, *ERRREL*, *MAXFCN*, *RESULT*, *ERREST*)

Double:     The double precision name is *DQAND*.

### Description

The routine *QAND* approximates the *n*-dimensional iterated integral

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} f(x_1, \dots, x_n) dx_n \dots dx_1$$

with the approximation returned in *RESULT*. An estimate of the error is returned in *ERREST*. The approximation is achieved by iterated applications of product Gauss formulas. The integral is first estimated by a two-point tensor product formula in each direction. Then for *i* = 1, ..., *n* the routine calculates a new estimate by doubling the number of points in the *i*-th direction, but halving the

number immediately afterwards if the new estimate does not change appreciably. This process is repeated until either one complete sweep results in no increase in the number of sample points in any dimension, or the number of Gauss points in one direction exceeds 256, or the number of function evaluations needed to complete a sweep would exceed MAXFCN.

## Comments

1. Informational errors

Type Code

3 1 MAXFCN was set greater than  $256^N$ .

4 2 The maximum number of function evaluations has been reached, and convergence has not been attained.

2. If EXACT is the exact value, QAND attempts to find RESULT such that

$\text{ABS}(\text{EXACT} - \text{RESULT}) \cdot \text{LE} \cdot \text{MAX}(\text{ERRABS}, \text{ERRREL} * \text{ABS}(\text{EXACT}))$ . To specify only a relative error, set ERRABS to zero. Similarly, to specify only an absolute error, set ERRREL to zero.

## Example

In this example, we approximate the integral of

$$e^{-(x_1^2 + x_2^2 + x_3^2)}$$

on an expanding cube. The values of the error estimates are machine dependent. The exact integral over

$$\mathbf{R}^3 \text{ is } \pi^{3/2}$$

```

USE QAND_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER I, J, MAXFCN, N, NOUT
REAL A(3), B(3), CNST, ERRABS, ERREST, ERRREL, F, RESULT
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

!
N = 3
MAXFCN = 100000

! Set error tolerances
ERRABS = 0.0001
ERRREL = 0.001

!
DO 20 I=1, 6
 CNST = I/2.0

! Set limits of integration
! As CNST approaches infinity, the
! answer approaches PI**1.5
DO 10 J=1, 3

```

```

 A(J) = -CNST
 B(J) = CNST
10 CONTINUE
 CALL QAND (F, N, A, B, RESULT, ERRABS, ERRREL, MAXFCN, ERREST)
 WRITE (NOUT,99999) CNST, RESULT, ERREST
20 CONTINUE
99999 FORMAT (1X, 'For CNST = ', F4.1, ', result = ', F7.3, ' with ', &
 'error estimate ', 1PE10.3)
 END
!
 REAL FUNCTION F (N, X)
 INTEGER N
 REAL X(N)
 REAL EXP
 INTRINSIC EXP
 F = EXP(-(X(1)*X(1)+X(2)*X(2)+X(3)*X(3)))
 RETURN
 END

```

## Output

```

For CNST = 0.5, result = 0.785 with error estimate 3.934E-06
For CNST = 1.0, result = 3.332 with error estimate 2.100E-03
For CNST = 1.5, result = 5.021 with error estimate 1.192E-05
For CNST = 2.0, result = 5.491 with error estimate 2.413E-04
For CNST = 2.5, result = 5.561 with error estimate 4.232E-03
For CNST = 3.0, result = 5.568 with error estimate 2.580E-04

```

---

## QMC

Integrates a function over a hyper rectangle using a quasi-Monte Carlo method.

### Required Arguments

**FCN** — User-supplied FUNCTION to be integrated. The form is FCN (X) , where

X — The independent variable. (Input)

FCN — The value of the integrand at X. (Output)

FCN must be declared EXTERNAL in the calling program.

**A** — Vector containing lower limits of integration. (Input)

**B** — Vector containing upper limits of integration. (Input)

**RESULT** — The value of

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} f(x_1, \dots, x_n) dx_n \dots dx_1$$

is returned, where  $n$  is the dimension of  $x$ . If no value can be computed, then NaN is returned. (Output)

### Optional Arguments

**ERRABS** — Absolute accuracy desired. (Input)  
Default: 1.0e-2.

**ERRREL** — Relative accuracy desired. (Input)  
Default: 1.0e-2.

**ERREST** — Estimate of the absolute value of the error. (Output)

**MAXEVALS** — Number of evaluations allowed. (Input)  
Default: No limit.

**BASE** — The base of the Faure sequence. (Input)  
Default: The smallest prime number greater than or equal to the number of dimensions (length of  $a$  and  $b$ ).

**SKIP** — The number of points to be skipped at the beginning of the Faure sequence. (Input)  
Default:  $\lfloor \text{base}^{m/2-1} \rfloor$ , where  $m = \lfloor \log B / \log \text{base} \rfloor$  and  $B$  is the largest representable integer.

### FORTRAN 90 Interface

Generic:     CALL QMC (FCN, A, B, RESULT [, ...])

Specific:    The specific interface names are S\_QMC and D\_QMC.

### Description

Integration of functions over hyper rectangle by direct methods, such as [QAND](#), is practical only for fairly low dimensional hypercubes. This is because the amount of work required increases exponentially as the dimension increases.

An alternative to direct methods is QMC, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like

$$1/\sqrt{k}$$

where  $k$  is the number of points at which the function is evaluated.

It is possible to improve on the performance of QMC by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to a sequence of random points is a *low-discrepancy* sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence as computed by `FAURE_NEXT`, see Stat Library, Chapter 18, Random Number Generation.

## Example

This example evaluates the n-dimensional integral

$$\int_0^1 \dots \int_0^1 \sum_{i=1}^n \prod_{j=1}^i (-1)^i x_j dx_1 \dots dx_n = -\frac{1}{3} \left[ 1 - \left( -\frac{1}{2} \right)^n \right]$$

with  $n=10$ .

```

use qmc_int
implicit none
integer, parameter :: ndim=10
real(kind(1d0)) :: a(ndim)
real(kind(1d0)) :: b(ndim)
real(kind(1d0)) :: result
integer :: I
external fcn

a = 0.d0
b = 1.d0

call qmc(fcn, a, b, result)
write (*,*) 'result = ', result
end

real(kind(1d0)) function fcn(x)
 implicit none
 real(kind(1d0)), dimension(:) :: x
 integer :: i, j
 real(kind(1d0)) :: prod, sum, sign

 sign = -1.d0
 sum = 0.d0
 do i=1, size(x)
 prod = 1.d0
 prod = product(x(1:i))
 sum = sum + (sign * prod)
 sign = -sign
 end do
 fcn = sum
end function fcn

```

## Output

```
result = -0.3334789
```

---

# GQRUL

Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

## Required Arguments

*N* — Number of quadrature points. (Input)

*QX* — Array of length *N* containing quadrature points. (Output)

*QW* — Array of length *N* containing quadrature weights. (Output)

## Optional Arguments

*IWEIGH* — Index of the weight function. (Input)

Default: *IWEIGH* = 1.

| <i>IWEIGH</i> | <i>WT(X)</i>               | Interval                  | Name                 |
|---------------|----------------------------|---------------------------|----------------------|
| 1             | 1                          | (-1, +1)                  | Legendre             |
| 2             | $1/\sqrt{1-X^2}$           | (-1, +1)                  | Chebyshev 1st kind   |
| 3             | $\sqrt{1-X^2}$             | (-1, +1)                  | Chebyshev 2nd kind   |
| 4             | $e^{-X^2}$                 | ( $-\infty$ , $+\infty$ ) | Hermite              |
| 5             | $(1-X)^\alpha (1+X)^\beta$ | (-1, +1)                  | Jacobi               |
| 6             | $e^{-X} X^\alpha$          | (0, $+\infty$ )           | Generalized Laguerre |
| 7             | $1/\cosh(X)$               | ( $-\infty$ , $+\infty$ ) | COSH                 |

*ALPHA* — Parameter used in the weight function with some values of *IWEIGH*, otherwise it is ignored. (Input)

Default: *ALPHA* = 2.0.

*BETA* — Parameter used in the weight function with some values of *IWEIGH*, otherwise it is ignored. (Input)

Default: *BETA* = 2.0.

*NFIX* — Number of fixed quadrature points. (Input)

*NFIX* = 0, 1 or 2. For the usual Gauss quadrature rules, *NFIX* = 0.

Default: *NFIX* = 0.

*QXFIX* — Array of length *NFIX* (ignored if *NFIX* = 0) containing the preset quadrature point(s). (Input)

## FORTRAN 90 Interface

Generic:     CALL GQRUL (N, QX, QW [, ...])

Specific:    The specific interface names are S\_GQRUL and D\_GQRUL.

## FORTRAN 77 Interface

Single:     CALL GQRUL (N, IWEIGH, ALPHA, BETAW, NFIX, QXFIX, QX, QW)

Double:     The double precision name is DGQRUL.

## Description

The routine GQRUL produces the points and weights for the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature formulas for some of the most popular weights. In fact, it is slightly more general than this suggests because the extra one or two points that may be specified do not have to lie at the endpoints of the interval. This routine is a modification of the subroutine GAUSSQUADRULE (Golub and Welsch 1969).

In the simple case when NFIX = 0, the routine returns points in  $x = QX$  and weights in  $w = QW$  so that

$$\int_a^b f(x)w(x) dx = \sum_{i=1}^N f(x_i)w_i$$

for all functions  $f$  that are polynomials of degree less than  $2N$ .

If NFIX = 1, then one of the above  $x_i$  equals the first component of QXFIX. Similarly, if NFIX = 2, then two of the components of  $x$  will equal the first two components of QXFIX. In general, the accuracy of the above quadrature formula degrades when NFIX increases. The quadrature rule will integrate all functions  $f$  that are polynomials of degree less than  $2N - \text{NFIX}$ .

## Comments

1.    Workspace may be explicitly provided, if desired, by use of G2RUL/DG2RUL. The reference is

```
CALL G2RUL (N, IWEIGH, ALPHA, BETAW, NFIX, QXFIX, QX, QW,
 WK)
```

The additional argument is

**WK** — Work array of length N.

2.    If IWEIGH specifies the weight  $WT(x)$  and the interval  $(a, b)$ , then approximately

$$\int_a^b F(X) * WT(X) dX = \sum_{I=1}^N F(QX(I)) * QW(I)$$

3. Gaussian quadrature is always the method of choice when the function  $F(X)$  behaves like a polynomial. Gaussian quadrature is also useful on infinite intervals (with appropriate weight functions), because other techniques often fail.
4. The weight function  $1/\cosh(X)$  behaves like a polynomial near zero and like  $e^{|X|}$  far from zero.

### Example 1

In this example, we obtain the classical Gauss-Legendre quadrature formula, which is accurate for polynomials of degree less than  $2N$ , and apply this when  $N = 6$  to the function  $x^8$  on the interval  $[-1, 1]$ . This quadrature rule is accurate for polynomials of degree less than 12.

```

USE GQRUL_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=6)
INTEGER I, NOUT
REAL ANSWER, QW(N), QX(N), SUM
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Get points and weights from GQRUL
CALL GQRUL (N, QX, QW)
!
! Write results from GQRUL
WRITE (NOUT,99998) (I,QX(I),I,QW(I),I=1,N)
99998 FORMAT (6(6X,'QX(',I1,',') = ',F8.4,7X,'QW(',I1,',') = ',F8.5,/)
!
! Evaluate the integral from these
! points and weights
SUM = 0.0
DO 10 I=1, N
 SUM = SUM + QX(I)**8*QW(I)
10 CONTINUE
ANSWER = SUM
WRITE (NOUT,99999) ANSWER
99999 FORMAT (/, ' The quadrature result making use of these ', &
 'points and weights is ', 1PE10.4, '.')
END

```

### Output

```

QX(1) = -0.9325 QW(1) = 0.17132
QX(2) = -0.6612 QW(2) = 0.36076
QX(3) = -0.2386 QW(3) = 0.46791
QX(4) = 0.2386 QW(4) = 0.46791
QX(5) = 0.6612 QW(5) = 0.36076
QX(6) = 0.9325 QW(6) = 0.17132

```

The quadrature result making use of these points and weights is 2.2222E-01.

## Additional Examples

### Example 2

We modify Example 1 by requiring that both endpoints be included in the quadrature formulas and again apply the new formulas to the function  $x^8$  on the interval  $[-1, 1]$ . This quadrature rule is accurate for polynomials of degree less than 10.

```
USE GQRUL_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=6)
INTEGER I, IWEIGH, NFIX, NOUT
REAL ALPHA, ANSWER, BETAW, QW(N), QX(N), QXFIX(2), SUM
!
! Get output unit number
CALL UMACH (2, NOUT)
!
IWEIGH = 1
ALPHA = 0.0
BETAW = 0.0
NFIX = 2
QXFIX(1) = -1.0
QXFIX(2) = 1.0
!
! Get points and weights from GQRUL
CALL GQRUL (N, QX, QW, ALPHA=ALPHA, BETAW=BETAW, NFIX=NFIX, &
 QXFIX=QXFIX)
!
! Write results from GQRUL
WRITE (NOUT,99998) (I,QX(I),I,QW(I),I=1,N)
99998 FORMAT (6(6X,'QX(',I1,',') = ',F8.4,7X,'QW(',I1,',') = ',F8.5,/)
!
! Evaluate the integral from these
! points and weights
SUM = 0.0
DO 10 I=1, N
 SUM = SUM + QX(I)**8*QW(I)
10 CONTINUE
ANSWER = SUM
WRITE (NOUT,99999) ANSWER
99999 FORMAT (/, ' The quadrature result making use of these ', &
 'points and weights is ', 1PE10.4, '.')
END
```

### Output

```
QX(1) = -1.0000 QW(1) = 0.06667
QX(2) = -0.7651 QW(2) = 0.37847
QX(3) = -0.2852 QW(3) = 0.55486
QX(4) = 0.2852 QW(4) = 0.55486
QX(5) = 0.7651 QW(5) = 0.37847
QX(6) = 1.0000 QW(6) = 0.06667
```

The quadrature result making use of these points and weights is 2.2222E-01.

---

## GQRCF

Computes a Gauss, Gauss-Radau or Gauss-Lobatto quadrature rule given the recurrence coefficients for the monic polynomials orthogonal with respect to the weight function.

### Required Arguments

*N* — Number of quadrature points. (Input)

*B* — Array of length *N* containing the recurrence coefficients. (Input)  
See [Comments](#) for definitions.

*C* — Array of length *N* containing the recurrence coefficients. (Input)  
See [Comments](#) for definitions.

*QX* — Array of length *N* containing quadrature points. (Output)

*QW* — Array of length *N* containing quadrature weights. (Output)

### Optional Arguments

*NFIX* — Number of fixed quadrature points. (Input)  
*NFIX* = 0, 1 or 2. For the usual Gauss quadrature rules *NFIX* = 0.  
Default: *NFIX* = 0.

*QXFIX* — Array of length *NFIX* (ignored if *NFIX* = 0) containing the preset quadrature point(s). (Input)

### FORTRAN 90 Interface

Generic:    CALL GQRCF (N, B, C, QX, QW [, ...])

Specific:   The specific interface names are S\_GQRCF and D\_GQRCF.

### FORTRAN 77 Interface

Single:     CALL GQRCF (N, B, C, NFIX, QXFIX, QX, QW)

Double:     The double precision name is DGQRCF.

### Description

The routine GQRCF produces the points and weights for the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature formulas given the three-term recurrence relation for the orthogonal polynomials. In particular, it is assumed that the orthogonal polynomials are monic, and hence, the three-term recursion may be written as

$$p_i(x) = (x - b_i) p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i=1, \dots, N$$

where  $p_0 = 1$  and  $p_{-1} = 0$ . It is obvious from this representation that the degree of  $p_i$  is  $i$  and that  $p_i$  is monic. In order for the recurrence to give rise to a sequence of orthogonal polynomials (with respect to a nonnegative measure), it is necessary and sufficient that  $c_i > 0$ . This routine is a modification of the subroutine `GAUSSQUADRULE` (Golub and Welsch 1969). In the simple case when `NFIX = 0`, the routine returns points in  $x = QX$  and weights in  $w = QW$  so that

$$\int_a^b f(x)w(x) dx = \sum_{i=1}^N f(x_i)w_i$$

for all functions  $f$  that are polynomials of degree less than  $2N$ . Here,  $w$  is any weight function for which the above recurrence produces the orthogonal polynomials  $p_i$  on the interval  $[a, b]$  and  $w$  is normalized by

$$\int_a^b w(x) dx = c_1$$

If `NFIX = 1`, then one of the above  $x_i$  equals the first component of `QXFIX`. Similarly, if `NFIX = 2`, then two of the components of  $x$  will equal the first two components of `QXFIX`. In general, the accuracy of the above quadrature formula degrades when `NFIX` increases. The quadrature rule will integrate all functions  $f$  that are polynomials of degree less than  $2N - \text{NFIX}$ .

## Comments

1. Workspace may be explicitly provided, if desired, by use of `G2RCF/DG2RCF`. The reference is:

```
CALL G2RCF (N, B, C, NFIX, QXFIX, QX, QW, WK)
```

The additional argument is:

**WK** — Work array of length `N`.

2. Informational error

Type Code

4 1 No convergence in 100 iterations.

3. The recurrence coefficients `B(I)` and `C(I)` define the monic polynomials via the relation  $P(I) = (x - B(I + 1)) * P(I - 1) - C(I + 1) * P(I - 2)$ . `C(1)` contains the zero-th moment

$$\int WT(X) dX$$

of the weight function. Each element of `C` must be greater than zero.

4. If  $WT(X)$  is the weight specified by the coefficients and the interval is  $(a, b)$ , then approximately

$$\int_a^b F(X) * WT(X) dX = \sum_{I=1}^N F(QX(I)) * QW(I)$$

5. Gaussian quadrature is always the method of choice when the function  $F(x)$  behaves like a polynomial. Gaussian quadrature is also useful on infinite intervals (with appropriate weight functions) because other techniques often fail.

### Example

We compute the Gauss quadrature rule (with  $N = 6$ ) for the Chebyshev weight,  $(1 + x^2)^{-1/2}$ , from the recurrence coefficients. These coefficients are obtained by a call to the IMSL routine [RECCF](#).

```

USE GQRCF_INT
USE UMACH_INT
USE RECCF_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=6)
INTEGER I, NFIX, NOUT
REAL B(N), C(N), QW(N), QX(N), QXFIX(2)
! Get output unit number
CALL UMACH (2, NOUT)
! Recursion coefficients will come from
! routine RECCF.
! The call to RECCF finds recurrence
! coefficients for Chebyshev
! polynomials of the 1st kind.
CALL RECCF (N, B, C)
!
! The call to GQRCF will compute the
! quadrature rule from the recurrence
! coefficients determined above.
CALL GQRCF (N, B, C, QX, QW)
WRITE (NOUT, 99999) (I, QX(I), I, QW(I), I=1, N)
99999 FORMAT (6(6X, 'QX(', I1, ') = ', F8.4, 7X, 'QW(', I1, ') = ', F8.5, /))
!
END

```

### Output

```

QX(1) = -0.9325 QW(1) = 0.17132
QX(2) = -0.6612 QW(2) = 0.36076
QX(3) = -0.2386 QW(3) = 0.46791
QX(4) = 0.2386 QW(4) = 0.46791
QX(5) = 0.6612 QW(5) = 0.36076
QX(6) = 0.9325 QW(6) = 0.17132

```

---

# RECCF

Computes recurrence coefficients for various monic polynomials.

## Required Arguments

*N* — Number of recurrence coefficients. (Input)

*B* — Array of length *N* containing recurrence coefficients. (Output)

*C* — Array of length *N* containing recurrence coefficients. (Output)

## Optional Arguments

*IWEIGH* — Index of the weight function. (Input)

Default: *IWEIGH* = 1.

| <b>IWEIGH</b> | <b>WT(<i>X</i>)</b>        | <b>Interval</b>           | <b>Name</b>          |
|---------------|----------------------------|---------------------------|----------------------|
| 1             | 1                          | (-1, +1)                  | Legendre             |
| 2             | $1/\sqrt{1-X^2}$           | (-1, +1)                  | Chebyshev 1st kind   |
| 3             | $\sqrt{1-X^2}$             | (-1, +1)                  | Chebyshev 2nd kind   |
| 4             | $e^{-X^2}$                 | ( $-\infty$ , $+\infty$ ) | Hermite              |
| 5             | $(1-X)^\alpha (1+X)^\beta$ | (-1, +1)                  | Jacobi               |
| 6             | $e^{-X} X^\alpha$          | (0, $+\infty$ )           | Generalized Laguerre |
| 7             | $1/\cosh(X)$               | ( $-\infty$ , $+\infty$ ) | COSH                 |

*ALPHA* — Parameter used in the weight function with some values of *IWEIGH*, otherwise it is ignored. (Input)

Default: *ALPHA*=1.0.

*BETAW* — Parameter used in the weight function with some values of *IWEIGH*, otherwise it is ignored. (Input)

Default: *BETAW*=1.0.

## FORTRAN 90 Interface

Generic:    CALL RECCF (N, B, C [, ...])

Specific:   The specific interface names are S\_RECCF and D\_RECCF.

## FORTRAN 77 Interface

Single:      CALL RECCF (N, IWEIGH, ALPHA, BETAW, B, C)

Double:     The double precision name is DRECCF.

## Description

The routine RECCF produces the recurrence coefficients for the orthogonal polynomials for some of the most important weights. It is assumed that the orthogonal polynomials are monic; hence, the three-term recursion may be written as

$$p_i(x) = (x - b_i) p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i=1, \dots, N$$

where  $p_0 = 1$  and  $p_{-1} = 0$ . It is obvious from this representation that the degree of  $p_i$  is  $i$  and that  $p_i$  is monic. In order for the recurrence to give rise to a sequence of orthogonal polynomials (with respect to a nonnegative measure), it is necessary and sufficient that  $c_i > 0$ .

## Comments

The recurrence coefficients  $B(I)$  and  $C(I)$  define the monic polynomials via the relation

$P(I) = (X - B(I + 1)) * P(I - 1) - C(I + 1) * P(I - 2)$ . The zero-th moment

$$\left( \int WT(X) dX \right)$$

of the weight function is returned in  $C(1)$ .

## Example

Here, we obtain the well-known recurrence relations for the first six *monic* Legendre polynomials, Chebyshev polynomials of the first kind, and Laguerre polynomials.

```
USE RECCF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=6)
INTEGER I, IWEIGH, NOUT
REAL ALPHA, B(N), C(N), BETAW
! Get output unit number
CALL UMACH (2, NOUT)
!
CALL RECCF (N, B, C)
WRITE (NOUT,99996)
WRITE (NOUT,99999) (I,B(I),I,C(I),I=1,N)
!
IWEIGH = 2
CALL RECCF (N, B, C, IWEIGH=IWEIGH)
WRITE (NOUT,99997)
WRITE (NOUT,99999) (I,B(I),I,C(I),I=1,N)
!
```

```

IWEIGH = 6
ALPHA = 0.0
BETAW = 0.0
CALL RECCF (N, B, C, IWEIGH=IWEIGH, ALPHA=ALPHA)
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I,B(I),I,C(I),I=1,N)
!
99996 FORMAT (1X, 'Legendre')
99997 FORMAT (/, 1X, 'Chebyshev, first kind')
99998 FORMAT (/, 1X, 'Laguerre')
99999 FORMAT (6(6X,'B(',I1,') = ',F8.4,7X,'C(',I1,') = ',F8.5,/))
END

```

## Output

### Legendre

|        |        |        |         |
|--------|--------|--------|---------|
| B(1) = | 0.0000 | C(1) = | 2.00000 |
| B(2) = | 0.0000 | C(2) = | 0.33333 |
| B(3) = | 0.0000 | C(3) = | 0.26667 |
| B(4) = | 0.0000 | C(4) = | 0.25714 |
| B(5) = | 0.0000 | C(5) = | 0.25397 |
| B(6) = | 0.0000 | C(6) = | 0.25253 |

### Chebyshev, first kind

|        |        |        |         |
|--------|--------|--------|---------|
| B(1) = | 0.0000 | C(1) = | 3.14159 |
| B(2) = | 0.0000 | C(2) = | 0.50000 |
| B(3) = | 0.0000 | C(3) = | 0.25000 |
| B(4) = | 0.0000 | C(4) = | 0.25000 |
| B(5) = | 0.0000 | C(5) = | 0.25000 |
| B(6) = | 0.0000 | C(6) = | 0.25000 |

### Laguerre

|        |         |        |          |
|--------|---------|--------|----------|
| B(1) = | 1.0000  | C(1) = | 1.00000  |
| B(2) = | 3.0000  | C(2) = | 1.00000  |
| B(3) = | 5.0000  | C(3) = | 4.00000  |
| B(4) = | 7.0000  | C(4) = | 9.00000  |
| B(5) = | 9.0000  | C(5) = | 16.00000 |
| B(6) = | 11.0000 | C(6) = | 25.00000 |

---

## RECQR

Computes recurrence coefficients for monic polynomials given a quadrature rule.

### Required Arguments

***QX*** — Array of length *N* containing the quadrature points. (Input)

***QW*** — Array of length *N* containing the quadrature weights. (Input)

***B*** — Array of length *NTERM* containing recurrence coefficients. (Output)

***C*** — Array of length *NTERM* containing recurrence coefficients. (Output)

## Optional Arguments

$N$  — Number of quadrature points. (Input)  
Default:  $N = \text{size}(QX, 1)$ .

$NTERM$  — Number of recurrence coefficients. (Input)  
 $NTERM$  must be less than or equal to  $N$ .  
Default:  $NTERM = \text{size}(B, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL RECQR (QX, QW, B, C [, ...])`

Specific: The specific interface names are `S_RECQR` and `D_RECQR`.

## FORTRAN 77 Interface

Single: `CALL RECQR (N, QX, QW, NTERM, B, C)`

Double: The double precision name is `DRECQR`.

## Description

The routine `RECQR` produces the recurrence coefficients for the orthogonal polynomials given the points and weights for the Gauss quadrature formula. It is assumed that the orthogonal polynomials are monic; hence the three-term recursion may be written

$$p_i(x) = (x - b_i) p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i=1, \dots, N$$

where  $p_0 = 1$  and  $p_{-1} = 0$ . It is obvious from this representation that the degree of  $p_i$  is  $i$  and that  $p_i$  is monic. In order for the recurrence to give rise to a sequence of orthogonal polynomials (with respect to a nonnegative measure), it is necessary and sufficient that  $c_i > 0$ .

This routine is an inverse routine to `GQRCE`. Given the recurrence coefficients, the routine `GQRCE` produces the corresponding Gauss quadrature formula, whereas the routine `RECQR` produces the recurrence coefficients given the quadrature formula.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `R2CQR/DR2CQR`. The reference is:

```
CALL R2CQR (N, QX, QW, NTERM, B, C, WK)
```

The additional argument is:

**WK** — Work array of length  $2 * N$ .

2. The recurrence coefficients  $B(I)$  and  $C(I)$  define the monic polynomials via the relation  $P(I) = (X - B(I + 1)) * P(I - 1) - C(I + 1) * P(I - 2)$ . The zero-th moment

$$\left(\int wT(X)dX\right)$$

of the weight function is returned in C(1).

### Example

To illustrate the use of RECQR, we will input a simple choice of recurrence coefficients, call GQRCF for the quadrature formula, put this information into RECQR, and recover the recurrence coefficients.

```

USE RECQR_INT
USE UMACH_INT
USE GQRCF_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=5)
INTEGER I, J, NFIX, NOUT, NTERM
REAL B(N), C(N), FLOAT, QW(N), QX(N), QXFIX(2)
INTRINSIC FLOAT

! Get output unit number
CALL UMACH (2, NOUT)
NFIX = 0

! Set arrays B and C of recurrence
! coefficients
DO 10 J=1, N
 B(J) = FLOAT(J)
 C(J) = FLOAT(J)/2.0
10 CONTINUE
WRITE (NOUT,99995)
99995 FORMAT (1X, 'Original recurrence coefficients')
WRITE (NOUT,99996) (I,B(I),I,C(I),I=1,N)
99996 FORMAT (5(6X,'B(',I1,') = ',F8.4,7X,'C(',I1,') = ',F8.5,/))
!
! The call to GQRCF will compute the
! quadrature rule from the recurrence
! coefficients given above.
!
CALL GQRCF (N, B, C, QX, QW)
WRITE (NOUT,99997)
99997 FORMAT (/, 1X, 'Quadrature rule from the recurrence coefficients' &
)
WRITE (NOUT,99998) (I,QX(I),I,QW(I),I=1,N)
99998 FORMAT (5(6X,'QX(',I1,') = ',F8.4,7X,'QW(',I1,') = ',F8.5,/))
!
! Call RECQR to recover the original
! recurrence coefficients
!
NTERM = N
CALL RECQR (QX, QW, B, C)
WRITE (NOUT,99999)
99999 FORMAT (/, 1X, 'Recurrence coefficients determined by RECQR')
WRITE (NOUT,99996) (I,B(I),I,C(I),I=1,N)
!

```

END

## Output

Original recurrence coefficients

|        |        |        |         |
|--------|--------|--------|---------|
| B(1) = | 1.0000 | C(1) = | 0.50000 |
| B(2) = | 2.0000 | C(2) = | 1.00000 |
| B(3) = | 3.0000 | C(3) = | 1.50000 |
| B(4) = | 4.0000 | C(4) = | 2.00000 |
| B(5) = | 5.0000 | C(5) = | 2.50000 |

Quadrature rule from the recurrence coefficients

|         |        |         |         |
|---------|--------|---------|---------|
| QX(1) = | 0.1525 | QW(1) = | 0.25328 |
| QX(2) = | 1.4237 | QW(2) = | 0.17172 |
| QX(3) = | 2.7211 | QW(3) = | 0.06698 |
| QX(4) = | 4.2856 | QW(4) = | 0.00790 |
| QX(5) = | 6.4171 | QW(5) = | 0.00012 |

Recurrence coefficients determined by RECQR

|        |        |        |         |
|--------|--------|--------|---------|
| B(1) = | 1.0000 | C(1) = | 0.50000 |
| B(2) = | 2.0000 | C(2) = | 1.00000 |
| B(3) = | 3.0000 | C(3) = | 1.50000 |
| B(4) = | 4.0000 | C(4) = | 2.00000 |
| B(5) = | 5.0000 | C(5) = | 2.50000 |

---

## FQRUL

Computes a Fejér quadrature rule with various classical weight functions.

### Required Arguments

*N* — Number of quadrature points. (Input)

*A* — Lower limit of integration. (Input)

*B* — Upper limit of integration. (Input)  
B must be greater than A.

*QX* — Array of length *N* containing quadrature points. (Output)

*QW* — Array of length *N* containing quadrature weights. (Output)

### Optional Arguments

*IWEIGH* — Index of the weight function. (Input)  
Default: *IWEIGH* = 1.

| <i>IWEIGH</i> | <i>WT(x)</i>           |
|---------------|------------------------|
| 1             | 1                      |
| 2             | $1/(x - \text{ALPHA})$ |

- 3  $(B - X)^{\alpha} (X - A)^{\beta}$
- 4  $(B - X)^{\alpha} (X - A)^{\beta} \log(X - A)$
- 5  $(B - X)^{\alpha} (X - A)^{\beta} \log(B - X)$

**ALPHA** — Parameter used in the weight function (except if `IWEIGH = 1`, it is ignored).

(Input)

If `IWEIGH = 2`, then it must satisfy `A .LT. ALPHA .LT. B`. If `IWEIGH = 3, 4, or 5`, then

**ALPHA** must be greater than  $-1$ .

Default: `ALPHA = 0.0`.

**BETAW** — Parameter used in the weight function (ignored if `IWEIGH = 1` or `2`). (Input)

**BETAW** must be greater than  $-1.0$ .

Default: `BETAW = 0.0`.

### FORTRAN 90 Interface

Generic: `CALL FQRUL (N, A, B, QX, QW [, ...])`

Specific: The specific interface names are `S_FQRUL` and `D_FQRUL`.

### FORTRAN 77 Interface

Single: `CALL FQRUL (N, A, B, IWEIGH, ALPHA, BETAW, QX, QW)`

Double: The double precision name is `DFQRUL`.

### Description

The routine `FQRUL` produces the weights and points for the Fejér quadrature rule. Since this computation is based on a quarter-wave cosine transform, the computations are most efficient when  $N$ , the number of points, is a product of small primes. These quadrature formulas may be an intermediate step in a more complicated situation, see for instance Gautschi and Milovanovic (1985).

The Fejér quadrature rules are based on polynomial interpolation. First, choose classical abscissas (in our case, the Gauss points for the Chebyshev weight function  $(1 - x^2)^{-1/2}$ ), then derive the quadrature rule for a different weight. In order to keep the presentation simple, we will describe the case where the interval of integration is  $[-1, 1]$  even though `FQRUL` allows rescaling to an arbitrary interval  $[a, b]$ .

We are looking for quadrature rules of the form

$$Q(f) := \sum_{j=1}^N w_j f(x_j)$$

where the

$$\{x_j\}_{j=1}^N$$

are the zeros of the  $N$ -th Chebyshev polynomial (of the first kind)  $T_N(x) = \cos(N \arccos x)$ . The weights in the quadrature rule  $Q$  are chosen so that, for all polynomials  $p$  of degree less than  $N$ ,

$$Q(p) = \sum_{j=1}^N w_j p(x_j) = \int_{-1}^1 p(x) w(x) dx$$

for some weight function  $w$ . In `FQRUL`, the user has the option of choosing  $w$  from five families of functions with various algebraic and logarithmic endpoint singularities.

These Fejér rules are important because they can be computed using specialized `FFT` quarter-wave transform routines. This means that rules with a large number of abscissas may be computed efficiently. If we insert  $T_l$  for  $p$  in the above formula, we obtain

$$Q(T_l) = \sum_{j=1}^N w_j T_l(x_j) = \int_{-1}^1 T_l(x) w(x) dx$$

for  $l = 0, \dots, N - 1$ . This is a system of linear equations for the unknown weights  $w_j$  that can be simplified by noting that

$$x_j = \cos \frac{(2j-1)\pi}{2N} \quad j = 1, \dots, N$$

and hence,

$$\begin{aligned} \int_{-1}^1 T_l(x) w(x) dx &= \sum_{j=1}^N w_j T_l(x_j) \\ &= \sum_{j=1}^N w_j \cos \frac{l(2j-1)\pi}{2N} \end{aligned}$$

The last expression is the cosine quarter-wave forward transform for the sequence

$$\{w_j\}_{j=1}^N$$

that is implemented in [Chapter 6, Transforms](#) under the name `QCOSF`. More importantly, `QCOSF` has an inverse `QCOSE`. It follows that if the integrals on the left in the last expression can be computed, then the Fejér rule can be derived efficiently for highly composite integers  $N$  utilizing `QCOSE`. For more information on this topic, consult Davis and Rabinowitz (1984, pages 84–86) and Gautschi (1968, page 259).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2RUL/DF2RUL`. The reference is:

```
CALL F2RUL (N, A, B, IWEIGH, ALPHA, BETAW, QX, QW, WK)
```

The additional argument is:

**WK** — Work array of length  $3 * N + 15$ .

2. If **IWEIGH** specifies the weight  $WT(X)$  and the interval  $(A, B)$ , then approximately

$$\int_A^B F(X) * WT(X) dX = \sum_{I=1}^N F(QX(I)) * QW(I)$$

3. The routine **FQRUL** uses an *fft*, so it is most efficient when **N** is the product of small primes.

### Example

Here, we obtain the Fejér quadrature rules using 10, 100, and 200 points. With these rules, we get successively better approximations to the integral

$$\int_0^1 x \sin(41\pi x^2) dx = \frac{1}{41\pi}$$

```
USE FQRUL_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER NMAX
PARAMETER (NMAX=200)
INTEGER I, K, N, NOUT
REAL A, ANSWER, B, F, QW(NMAX), &
 QX(NMAX), SIN, SUM, X, PI, ERROR
INTRINSIC SIN, ABS
!
F(X) = X*SIN(41.0*PI*X**2)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
PI = CONST('PI')
DO 20 K=1, 3
 IF (K .EQ. 1) N = 10
 IF (K .EQ. 2) N = 100
 IF (K .EQ. 3) N = 200
 A = 0.0
 B = 1.0
!
! Get points and weights from FQRUL
CALL FQRUL (N, A, B, QX, QW)
!
! Evaluate the integral from these
! points and weights
SUM = 0.0
```

```

 DO 10 I=1, N
 SUM = SUM + F(QX(I))*QW(I)
10 CONTINUE
 ANSWER = SUM
 ERROR = ABS(ANSWER - 1.0/(41.0*PI))
 WRITE (NOUT,99999) N, ANSWER, ERROR
20 CONTINUE
!
99999 FORMAT (/, 1X, 'When N = ', I3, ', the quadrature result making ' &
, 'use of these points ', /, ' and weights is ', 1PE11.4, &
', with error ', 1PE9.2, '.')
END

```

## Output

When N = 10, the quadrature result making use of these points and weights is -1.6523E-01, with error 1.73E-01.

When N = 100, the quadrature result making use of these points and weights is 7.7637E-03, with error 2.79E-08.

When N = 200, the quadrature result making use of these points and weights is 7.7636E-03, with error 1.40E-08.

---

## DERIV

This function computes the first, second or third derivative of a user-supplied function.

### Function Return Value

**DERIV** — Estimate of the first (KORDER = 1), second (KORDER = 2) or third (KORDER = 3) derivative of FCN at X. (Output)

### Required Arguments

**FCN** — User-supplied FUNCTION whose derivative at X will be computed. The form is FCN(X), where

X — Independent variable. (Input)

FCN — The function value. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Point at which the derivative is to be evaluated. (Input)

### Optional Arguments

**KORDER** — Order of the derivative desired (1, 2 or 3). (Input)  
Default: KORDER = 1.

**BGSTEP** — Beginning value used to compute the size of the interval used in computing the derivative. (Input)

The interval used is the closed interval  $(x - 4 * BGSTEP, x + 4 * BGSTEP)$ . **BGSTEP** must be positive.

Default: **BGSTEP** = .01.

**TOL** — Relative error desired in the derivative estimate. (Input)

Default: **TOL** = 1.e-2 for single precision and 1.d-4 for double precision.

## **FORTRAN 90 Interface**

Generic:     **DERIV** (FCN, X [, ...])

Specific:    The specific interface names are **S\_DERIV** and **D\_DERIV**.

## **FORTRAN 77 Interface**

Single:     **DERIV** (FCN, KORDER, X, **BGSTEP**, **TOL**)

Double:     The double precision function name is **DDERIV**.

## **Description**

**DERIV** produces an estimate to the first, second, or third derivative of a function. The estimate originates from first computing a spline interpolant to the input function using values within the interval  $(x - 4.0 * BGSTEP, x + 4.0 * BGSTEP)$ , then differentiating the spline at  $x$ .

## **Comments**

1. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                  |
|---|---|------------------------------------------------------------------------------------------------------------------|
| 3 | 2 | Roundoff error became dominant before estimates converged.<br>Increase precision and/or increase <b>BGSTEP</b> . |
|---|---|------------------------------------------------------------------------------------------------------------------|

|   |   |                                                                                                                                                                                                                 |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | Unable to achieve desired tolerance in derivative estimation.<br>Increase precision, increase <b>TOL</b> and/or change <b>BGSTEP</b> . If this error continues, the function may not have a derivative at $x$ . |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

2. Convergence is assumed when

$$\frac{2}{3} |D2 - D1| < TOL$$

for two successive derivative estimates **D1** and **D2**.

- The initial step size, `BGSTEP`, must be chosen small enough that `FCN` is defined and reasonably smooth in the interval  $(x - 4 * \text{BGSTEP}, x + 4 * \text{BGSTEP})$ , yet large enough to avoid roundoff problems.

### Example 1

In this example, we obtain the approximate first derivative of the function

$$f(x) = -2 \sin(3x/2)$$

at the point  $x = 2$ .

```

USE DERIV_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER KORDER, NCOUNT, NOUT
REAL BGSTEP, DERV, TOL, X
EXTERNAL FCN
! Get output unit number
CALL UMACH (2, NOUT)
!
X = 2.0
BGSTEP = 0.2
NCOUNT = 1
DERV = DERIV(FCN,X, BGSTEP=BGSTEP)
WRITE (NOUT,99999) DERV
99999 FORMAT (/, 1X, 'First derivative of FCN is ', 1PE10.3)
END
!
REAL FUNCTION FCN (X)
REAL X
REAL SIN
INTRINSIC SIN
FCN = -2.0*SIN(1.5*X)
RETURN
END

```

### Output

First derivative of FCN is 2.970E+00

### Additional Example

#### Example 2

In this example, we attempt to approximate in single precision the third derivative of the function

$$f(x) = 2x^4 + 3x$$

at the point  $x = 0.75$ . Although the function is well-behaved near  $x = 0.75$ , finding derivatives is often computationally difficult on 32-bit machines. The difficulty is overcome in double precision.

```

USE IMSL_LIBRARIES

```

```

 IMPLICIT NONE
 INTEGER KORDER, NOUT
 REAL BGSTEP, DERV, X, TOL
 DOUBLE PRECISION DBGSTE, DDERV, DFCN, DTOL, DX
 EXTERNAL DFCN, FCN
!
 CALL UMACH (2, NOUT) Get output unit number
!
! Turn off stopping due to error
! condition
 CALL ERSET (0, -1, 0)
!
 X = 0.75
 BGSTEP = 0.1
 KORDER = 3
!
! In single precision, on a 32-bit
! machine, the following attempt
! produces an error message
 DERV = DERIV(FCN, X, KORDER, BGSTEP, TOL)
!
! In double precision, we get good
! results
 DX = 0.75D0
 DBGSTE = 0.1D0
 DTOL = 0.01D0
 KORDER = 3
 DDERV = DERIV(DFCN, DX, KORDER, DBGSTE, DTOL)
 WRITE (NOUT, 99999) DDERV
99999 FORMAT (/, 1X, 'The third derivative of DFCN is ', 1PD10.4)
 END
!
 REAL FUNCTION FCN (X)
 REAL X
 FCN = 2.0*X**4 + 3.0*X
 RETURN
 END
!
 DOUBLE PRECISION FUNCTION DFCN (X)
 DOUBLE PRECISION X
 DFCN = 2.0D0*X**4 + 3.0D0*X
 RETURN
 END

```

## Output

```

*** FATAL ERROR 1 from DERIV. Unable to achieve desired tolerance.
*** Increase precision, increase TOL = 1.000000E-02 and/or change
*** BGSTEP = 1.000000E-01. If this error continues the function
*** may not have a derivative at X = 7.500000E-01

```

The third derivative of DFCN is 3.6000D+01

# Chapter 5: Differential Equations

---

## Routines

|              |                                                               |                             |      |
|--------------|---------------------------------------------------------------|-----------------------------|------|
| <b>5.1</b>   | <b>First-Order Ordinary Differential Equations</b>            |                             |      |
| 5.1.1        | Solution of the Initial-Value Problem for ODEs                |                             |      |
|              | Runge-Kutta method.....                                       | <a href="#">IVPRK</a>       | 1003 |
|              | Runge-Kutta method, various orders.....                       | <a href="#">IVMRK</a>       | 1011 |
|              | Adams or Gear method .....                                    | <a href="#">IVPAG</a>       | 1021 |
| 5.1.2        | Solution of the Boundary-Value Problem for ODEs               |                             |      |
|              | Finite-difference method.....                                 | <a href="#">BVPFD</a>       | 1037 |
|              | Multiple-shooting method.....                                 | <a href="#">BVPMS</a>       | 1050 |
| 5.1.3        | Solution of the Differential-Algebraic Systems                |                             |      |
|              | Solves a first order differential-algebraic system            |                             |      |
|              | of equations .....                                            | <a href="#">DAESL</a>       | 1057 |
| <b>5.1.4</b> | <b>First-and-Second-Order Ordinary Differential Equations</b> |                             |      |
| 5.1.5        | Solution of the Initial-Value Problem for ODEs                |                             |      |
|              | Solves an initial-value problem for a system of ODEs          |                             |      |
|              | using a variable order Adams method .....                     | <a href="#">IVOAM</a>       | 1072 |
| <b>5.2</b>   | <b>Partial Differential Equations</b>                         |                             |      |
| 5.2.1        | Solution of Systems of PDEs in One Dimension                  |                             |      |
|              | Method of lines with Variable Griddings .....                 | <a href="#">PDE_1D_MG</a>   | 1081 |
|              | Method of lines with a Hermite cubic basis .....              | <a href="#">MMOLCH</a>      | 1115 |
|              | Solves a generalized Feynman-Kac equation on a                |                             |      |
|              | finite interval using Hermite quintic splines.....            | <a href="#">FEYNMAN_KAC</a> | 1128 |
|              | Computes the value of a Hermite quintic spline or the         |                             |      |
|              | value of one of its derivatives .....                         | <a href="#">HQSVAL</a>      | 1185 |
| 5.2.2        | Solution of a PDE in Two and Three Dimensions                 |                             |      |
|              | Two-dimensional fast Poisson solver .....                     | <a href="#">FPS2H</a>       | 1188 |
|              | Three-dimensional fast Poisson solver.....                    | <a href="#">FPS3H</a>       | 1194 |
| <b>5.3.</b>  | <b>Sturm-Liouville Problems</b>                               |                             |      |
|              | Eigenvalues, eigenfunctions,                                  |                             |      |
|              | and spectral density functions .....                          | <a href="#">SLEIG</a>       | 1201 |
|              | Indices of eigenvalues .....                                  | <a href="#">SLCNT</a>       | 1213 |

---

## Usage Notes

A *differential equation* is an equation involving one or more dependent variables (called  $y_i$  or  $u_i$ ), their derivatives, and one or more independent variables (called  $t$ ,  $x$ , and  $y$ ). Users will typically need to relabel their own model variables so that they correspond to the variables used in the solvers described here. A differential equation with one independent variable is called an *ordinary differential equation* (ODE). A system of equations involving derivatives in one independent variable and other dependent variables is called a *differential-algebraic system*. A differential equation with more than one independent variable is called a *partial differential equation* (PDE).

The *order* of a differential equation is the highest order of any of the derivatives in the equation. Some of the routines in this chapter require the user to reduce higher-order problems to systems of first-order differential equations.

### Ordinary Differential Equations

It is convenient to use the vector notation below. We denote the number of equations as the value  $N$ . The problem statement is abbreviated by writing it as a *system* of first-order ODEs

$$y(t) = [y_1(t), \dots, y_N(t)]^T, \quad f(t, y) = [f_1(t, y), \dots, f_N(t, y)]^T$$

The problem becomes

$$y' = \frac{dy(t)}{dt} = f(t, y)$$

with initial values  $y(t_0)$ . Values of  $y(t)$  for  $t > t_0$  or  $t < t_0$  are required. The routines [IVPRK](#), [IVMRK](#), and [IVPAG](#), solve the IVP for systems of ODEs of the form  $y' = f(t, y)$  with  $y(t = t_0)$  specified.

Here,  $f$  is a user supplied function that must be evaluated at any set of values  $(t, y_1, \dots, y_N)$ ;

$i = 1, \dots, N$ . The routines [IVPAG](#), and [DAESL](#), will also solve implicit systems of the form  $Ay' = f(t, y)$  where  $A$  is a user supplied matrix. For [IVPAG](#), the matrix  $A$  must be nonsingular.

The system  $y' = f(t, y)$  is said to be *stiff* if some of the eigenvalues of the Jacobian matrix  $\{\partial f_i / \partial y_j\}$  have large, negative real parts. This is often the case for differential equations representing the behavior of physical systems such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reaction in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*). This definition of stiffness, based on the eigenvalues of the Jacobian matrix, is not satisfactory. Users typically identify stiff systems by the fact that numerical differential equation solvers such as [IVPRK](#), are inefficient, or else they fail. The most common inefficiency is that a large number of evaluations of the functions  $f_i$  are required. In such cases, use routine [IVPAG](#), or [DAESL](#). For more about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

In the *boundary value problem* (BVP) for ODEs, constraints on the dependent variables are given at the endpoints of the interval of interest,  $[a, b]$ . The routines [BVPFD](#) and [BVPMS](#) solve the BVP for systems of the form  $y'(t) = f(t, y)$ , subject to the conditions

$$h_i(y_1(a), \dots, y_N(a), y_1(b), \dots, y_N(b)) = 0 \quad i = 1, \dots, N$$

Here,  $f$  and  $h = [h_1, \dots, h_N]^T$  are user-supplied functions.

[IVOAM](#) solves systems of ordinary differential equations of order one, order two, or mixed order one and two.

## Differential-algebraic Equations

Frequently, it is not possible or not convenient to express the model of a dynamical system as a set of ODEs. Rather, an implicit equation is available in the form

$$g_i(t, y, \dots, y_N, y'_1, \dots, y'_N) = 0 \quad i = 1, \dots, N$$

The  $g_i$  are user-supplied functions. The system is abbreviated as

$$g(t, y, y') = [g_1(t, y, y'), \dots, g_N(t, y, y')]^T = 0$$

With initial value  $y(t_0)$ . Any system of ODEs can be trivially written as a differential-algebraic system by defining

$$g(t, y, y') = f(t, y) - y'$$

The routine [DAESL](#) solves differential-algebraic systems of index 1 or index 0. For a definition of *index* of a differential-algebraic system, see (Brenan et al. 1989). Also, see Gear and Petzold (1984) for an outline of the computing methods used.

## Partial Differential Equations

The routine [MMOLCH](#) solves the IVP problem for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i \left( x, t, u_1, \dots, u_N, \frac{\partial u_1}{\partial x}, \dots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \dots, \frac{\partial^2 u_N}{\partial x^2} \right)$$

subject to the boundary conditions

$$\alpha_1^{(i)} u_i(a) + \beta_1^{(i)} \frac{\partial u_i}{\partial x}(a) = \gamma_1^{(i)}(t)$$

$$\alpha_2^{(i)} u_i(b) + \beta_2^{(i)} \frac{\partial u_i}{\partial x}(b) = \gamma_2^{(i)}(t)$$

and subject to the initial conditions

$$u_i(x, t = t_0) = g_i(x)$$

for  $i = 1, \dots, N$ . Here,  $f_i$ ,  $g_i$ ,  $\alpha_j^{(i)}$ ,  $\beta_j^{(i)}$  and  $\gamma_j^{(i)}(t)$  are user-supplied,  $j = 1, 2$ .

The routines [FPS2H](#) and [FPS3H](#) solve Laplace's, Poisson's, or Helmholtz's equation in two or three dimensions. [FPS2H](#) uses a fast Poisson method to solve a PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = f(x, y)$$

over a rectangle, subject to boundary conditions on each of the four sides. The scalar constant  $c$  and the function  $f$  are user specified. `FPS3H` solves the three-dimensional analogue of this problem.

NOTE: Users wishing to solve more general PDE's, in more general 2-d and 3-d regions, are referred to Visual Numerics' affiliated product PDE2D ([www.vni.com/pde2d](http://www.vni.com/pde2d)).

## Summary

The following table summarizes the types of problems handled by the routines in this chapter. With the exception of `FPS2H` and `FPS3H`, the routines can handle more than one differential equation.

| Problem                                      | Consideration                                                                                                                                                                                                                | Routine               |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| $Ay' = f(t, y)$<br>$y(t_0) = y_0$            | $A$ is a general, symmetric positive definite, band or symmetric positive definite band matrix.                                                                                                                              | <a href="#">IVPAG</a> |
|                                              | Stiff or expensive to evaluate $f(t, y)$ , banded Jacobian or finely spaced output needed.                                                                                                                                   | <a href="#">IVPAG</a> |
| $y' = f(t, y)$ ,<br>$y(t_0) = y_0$           | High accuracy needed and not stiff. (Uses Adams methods)                                                                                                                                                                     | <a href="#">IVPAG</a> |
|                                              | Moderate accuracy needed and not stiff.                                                                                                                                                                                      | <a href="#">IVPRK</a> |
| $y' = f(t, y)$<br>$h(y(a), y(b)) = 0$        | BVP solver using finite differences                                                                                                                                                                                          | <a href="#">BVPFD</a> |
|                                              | BVP solver using multiple shooting                                                                                                                                                                                           | <a href="#">BVPMS</a> |
| $g(t, y, y') = 0$<br>$y(t_0), y'(t_0)$ given | Stiff, differential-algebraic solver for systems of index 1 or 0.<br><b>Note:</b> <code>DAESL</code> uses the user-supplied $y'(t_0)$ only as an initial guess to help it find the correct initial $y'(t_0)$ to get started. | <a href="#">DAESL</a> |

| Problem                                                                                                                                                           | Consideration                                  | Routine |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|---------|
| $u_t = f(x, t, u, u_x, u_{xx})$<br>$\alpha_1 u(a) + \beta_1 u_x(a) = \gamma_1(t)$<br>$\alpha_2 u(b) + \beta_2 u_x(b) = \gamma_2(t)$                               | Method of lines using cubic Hermites and ODEs. | MMOLCH  |
| $u_{xx} + u_{yy} + cu = f(x, y)$ on a rectangle, given $u$ or $u_n$ on each edge.                                                                                 | Fast Poisson solver                            | FPS2H   |
| $u_{xx} + u_{yy} + u_{zz} + cu = f(x, y, z)$ on a box, given $u$ or $u_n$ on each face.                                                                           | Fast Poisson solver                            | FPS3H   |
| $-(pu')' + qu = \lambda ru,$<br>$\alpha_1 u(a) - \alpha_2 (pu'(a))$<br>$= \lambda (\alpha_1' u(a) - \alpha_2' (pu'(a)))$<br>$\beta_1 u(b) + \beta_2 (pu'(b)) = 0$ | Sturm-Liouville problems                       | SLEIG   |

---

## IVPRK

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

### Required Arguments

**IDO** — Flag indicating the state of the computation. (Input/Output)

**IDO State**

- 1 Initial entry
- 2 Normal re-entry
- 3 Final call to release workspace
- 4 Return because of interrupt 1
- 5 Return because of interrupt 2 with step accepted
- 6 Return because of interrupt 2 with step rejected

Normally, the initial call is made with **IDO** = 1. The routine then sets **IDO** = 2, and this value is used for all but the last call that is made with **IDO** = 3. This final call is used to release workspace, which was automatically allocated by the initial call with **IDO** = 1.

No integration is performed on this final call. See Comment 3 for a description of the other interrupts.

**FCN** — User-supplied subroutine to evaluate functions. The usage is

CALL FCN(N, T, Y, YPRIME), where

N — Number of equations. (Input)

T — Independent variable,  $t$ . (Input)

Y — Array of size N containing the dependent variable values,  $y$ . (Input)

YPRIME — Array of size N containing the values of the vector  $y'$  evaluated at ( $t$ ,  $y$ ). (Output)

FCN must be declared EXTERNAL in the calling program.

**T** — Independent variable. (Input/Output)

On input, T contains the initial value. On output, T is replaced by TEND unless error conditions have occurred. See IDO for details.

**TEND** — Value of  $t$  where the solution is required. (Input)

The value TEND may be less than the initial value of  $t$ .

**Y** — Array of size NEQ of dependent variables. (Input/Output)

On input, Y contains the initial values. On output, Y contains the approximate solution.

## Optional Arguments

**NEQ** — Number of differential equations. (Input)

Default: NEQ = size (Y,1).

**TOL** — Tolerance for error control. (Input)

An attempt is made to control the norm of the local error such that the global error is proportional to TOL.

Default: TOL = machine precision.

**PARAM** — A floating-point array of size 50 containing optional parameters. (Input/ Output)

If a parameter is zero, then a default value is used. These default values are given below. Parameters that concern values of step size are applied in the direction of integration. The following parameters may be set by the user:

|   | <b>PARAM</b> | <b>Meaning</b>                                                                                                                                             |
|---|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | HINIT        | Initial value of the step size. Default: $10.0 * \text{MAX}(\text{AMACH}(1), \text{AMACH}(4) * \text{MAX}(\text{ABS}(\text{TEND}), \text{ABS}(\text{T})))$ |
| 2 | HMIN         | Minimum value of the step size. Default: 0.0                                                                                                               |
| 3 | HMAX         | Maximum value of the step size. Default: 2.0                                                                                                               |

|       |        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4     | MXSTEP | Maximum number of steps allowed. Default: 500                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 5     | MXFCN  | Maximum number of function evaluations allowed. Default: No enforced limit.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 6     |        | Not used.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 7     | INTRP1 | If nonzero, then return with IDO = 4 before each step. See Comment 3. Default: 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 8     | INTRP2 | If nonzero, then return with IDO = 5 after every successful step and with IDO = 6 after every unsuccessful step. See Comment 3. Default: 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 9     | SCALE  | A measure of the scale of the problem, such as an approximation to the average value of a norm of the Jacobian matrix along the solution. Default: 1.0                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 10    | INORM  | Switch determining error norm. In the following, $e_i$ is the absolute value of an estimate of the error in $y_i(t)$ .<br>Default: 0.0 – $\min(\text{absolute error}, \text{relative error}) = \max(e_i/w_i)$ ; $i = 1, \dots, NEQ$ , where $w_i = \max( y_i(t) , 1.0)$ .<br>1 – absolute error = $\max(e_i), i = 1 \dots, NEQ$ .<br>2 – $\max(e_i/w_i), i = 1 \dots, NEQ$ where $w_i = \max( y_i(t) , \text{FLOOR})$ , and FLOOR is PARAM(11).<br>3 – Scaled Euclidean norm defined as<br>where $w_i = \max( y_i(t) , 1.0)$ . Other definitions of YMAX can be specified by the user, as explained in Comment 1. |
| 11    | FLOOR  | Used in the norm computation associated with parameter INORM. Default: 1.0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 12-30 |        | Not used.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

The following entries in PARAM are set by the program.

|       | PARAM  | Meaning                              |
|-------|--------|--------------------------------------|
| 31    | HTRIAL | Current trial step size.             |
| 32    | HMINC  | Computed minimum step size allowed.  |
| 33    | HMAXC  | Computed maximum step size allowed.  |
| 34    | NSTEP  | Number of steps taken.               |
| 35    | NFCN   | Number of function evaluations used. |
| 36-50 |        | Not used.                            |

## FORTRAN 90 Interface

Generic: CALL IVPRK (IDO, FCN, T, TEND, Y [, ...])

Specific: The specific interface names are S\_IVPRK and D\_IVPRK.

## FORTRAN 77 Interface

Single:      CALL IVPK (IDO, NEQ, FCN, T, TEND, TOL, PARAM, Y)

Double:     The double precision name is DIVPRK.

## Description

Routine IVPK finds an approximation to the solution of a system of first-order differential equations of the form  $y_0 = f(t, y)$  with given initial data. The routine attempts to keep the global error proportional to a user-specified tolerance. This routine is efficient for nonstiff systems where the derivative evaluations are not expensive.

The routine IVPK is based on a code designed by Hull, Enright and Jackson (1976, 1977). It uses Runge-Kutta formulas of order five and six developed by J. H. Verner.

## Comments

1.   Workspace may be explicitly provided, if desired, by use of I2PRK/DI2PRK. The reference is:

```
CALL I2PRK (IDO, NEQ, FCN, T, TEND, TOL, PARAM, Y, VNORM,
 WK)
```

The additional arguments are as follows:

$$YMAX = \sqrt{\sum_{i=1}^{NEQ} e_i^2 / w_i^2}$$

**VNORM** — A Fortran subroutine to compute the norm of the error.

(Input)

The routine may be provided by the user, or the IMSL routine I3PRK/DI3PRK may be used. In either case, the name must be declared in a Fortran EXTERNAL statement. If usage of the IMSL routine is intended, then the name I3PRK/DI3PRK should be used. The usage of the error norm routine is CALL VNORM (N, V, Y, YMAX, ENORM), where

| Arg   | Definition                                                                  |
|-------|-----------------------------------------------------------------------------|
| N     | Number of equations. (Input).                                               |
| V     | Array of size N containing the vector whose norm is to be computed. (Input) |
| Y     | Array of size N containing the values of the dependent variable. (Input)    |
| YMAX  | Array of size N containing the maximum values of $ y(t) $ . (Input).        |
| ENORM | Norm of the vector V. (Output).                                             |

VNORM must be declared EXTERNAL in the calling program.

**WK** — Work array of size  $10N$  using the working precision. The contents of **WK** must not be changed from the first call with `IDO = 1` until after the final call with `IDO = 3`.

2. Informational errors

Type Code

- |   |   |                                                                                 |
|---|---|---------------------------------------------------------------------------------|
| 4 | 1 | Cannot satisfy error condition. The value of <code>TOL</code> may be too small. |
| 4 | 2 | Too many function evaluations needed.                                           |
| 4 | 3 | Too many steps needed. The problem may be stiff.                                |

3. If `PARAM(7)` is nonzero, the subroutine returns with `IDO = 4` and will resume calculation at the point of interruption if re-entered with `IDO = 4`. If `PARAM(8)` is nonzero, the subroutine will interrupt the calculations immediately after it decides whether or not to accept the result of the most recent trial step. The values used are `IDO = 5` if the routine plans to accept, or `IDO = 6` if it plans to reject the step. The values of `IDO` may be changed by the user (by changing `IDO` from 6 to 5) in order to force acceptance of a step that would otherwise be rejected. Some parameters the user might want to examine after return from an interrupt are `IDO`, `HTRIAL`, `NSTEP`, `NFCN`, `T`, and `Y`. The array `Y` contains the newly computed trial value for  $y(t)$ , accepted or not.

### Example 1

Consider a predator-prey problem with rabbits and foxes. Let  $r$  be the density of rabbits and let  $f$  be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. Mathematically,

$$r' = 2r$$

$$f' = -f$$

The rate at which the rabbits are eaten by the foxes is  $2rf$ , and the rate at which the foxes increase, because they are eating the rabbits, is  $rf$ . So, the model to be solved is

$$r' = 2r - 2rf$$

$$f' = -f + rf$$

The initial conditions are  $r(0) = 1$  and  $f(0) = 3$  over the interval  $0 \leq t \leq 10$ .

In the program `Y(1) = r` and `Y(2) = f`. Note that the parameter vector `PARAM` is first set to zero with IMSL routine `SSET` ([Chapter 9, Basic Matrix/Vector Operations](#)). Then, absolute error control is selected by setting `PARAM(10) = 1.0`.

The last call to `IVPRK` with `IDO = 3` deallocates IMSL workspace allocated on the first call to `IVPRK`. It is not necessary to release the workspace in this example because the program ends after solving a single problem. The call to release workspace is made as a model of what would be needed if the program included further calls to IMSL routines.

```

USE IVPRK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER MXPARM, N
PARAMETER (MXPARM=50, N=2)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IDO, ISTEP, NOUT
REAL PARAM(MXPARM), T, TEND, TOL, Y(N)
!
! SPECIFICATIONS FOR SUBROUTINES
EXTERNAL FCN
!
CALL UMACH (2, NOUT)
!
! Set initial conditions
T = 0.0
Y(1) = 1.0
Y(2) = 3.0
!
! Set error tolerance
TOL = 0.0005
!
! Set PARAM to default
PARAM = 0.E0
!
! Select absolute error control
PARAM(10) = 1.0
!
! Print header
WRITE (NOUT,99999)
IDO = 1
ISTEP = 0
10 CONTINUE
ISTEP = ISTEP + 1
TEND = ISTEP
CALL IVPRK (IDO, FCN, T, TEND, Y, TOL=TOL, PARAM=PARAM)
IF (ISTEP .LE. 10) THEN
WRITE (NOUT,'(I6,3F12.3)') ISTEP, T, Y
!
! Final call to release workspace
IF (ISTEP .EQ. 10) IDO = 3
GO TO 10
END IF
99999 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
END
SUBROUTINE FCN (N, T, Y, YPRIME)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER N
REAL T, Y(N), YPRIME(N)
!
YPRIME(1) = 2.0*Y(1) - 2.0*Y(1)*Y(2)
YPRIME(2) = -Y(2) + Y(1)*Y(2)
RETURN
END

```

## Output

| ISTEP | Time  | Y1    | Y2    |
|-------|-------|-------|-------|
| 1     | 1.000 | 0.078 | 1.465 |
| 2     | 2.000 | 0.085 | 0.578 |

|    |        |       |       |
|----|--------|-------|-------|
| 3  | 3.000  | 0.292 | 0.250 |
| 4  | 4.000  | 1.449 | 0.187 |
| 5  | 5.000  | 4.046 | 1.444 |
| 6  | 6.000  | 0.176 | 2.256 |
| 7  | 7.000  | 0.066 | 0.908 |
| 8  | 8.000  | 0.148 | 0.367 |
| 9  | 9.000  | 0.655 | 0.188 |
| 10 | 10.000 | 3.157 | 0.352 |

## Additional Examples

### Example 2

This is a mildly stiff problem (F2) from the test set of Enright and Pryce (1987). It is included here because it illustrates the inefficiency of requiring more function evaluations with a nonstiff solver, for a requested accuracy, than would be required using a stiff solver. Also, see [IVPAG Example 2](#), where the problem is solved using a BDF method. The number of function evaluations may vary, depending on the accuracy and other arithmetic characteristics of the computer. The test problem has  $n = 2$  equations:

$$\begin{aligned}
 y_1' &= -y_1 - y_1 y_2 + k_1 y_2 \\
 y_2' &= -k_2 y_2 + k_3 (1 - y_2) y_1 \\
 y_1(0) &= 1 \\
 y_2(0) &= 0 \\
 k_1 &= 294 \\
 k_2 &= 3 \\
 k_3 &= 0.01020408 \\
 tend &= 240
 \end{aligned}$$

```

USE IVPBK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER MXPARAM, N
PARAMETER (MXPARAM=50, N=2)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IDO, ISTEP, NOUT
REAL PARAM(MXPARAM), T, TEND, TOL, Y(N)
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCN
!
CALL UMACH (2, NOUT)
! Set initial conditions
T = 0.0
Y(1) = 1.0
Y(2) = 0.0
! Set error tolerance
TOL = 0.001
! Set PARAM to default

```

```

PARAM = 0.0E0
! Select absolute error control
PARAM(10) = 1.0
! Print header
WRITE (NOUT,99998)
IDO = 1
ISTEP = 0
10 CONTINUE
ISTEP = ISTEP + 24
TEND = ISTEP
CALL IVPRK (IDO, FCN, T, TEND, Y, TOL=TOL, PARAM=PARAM)
IF (ISTEP .LE. 240) THEN
 WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
! Final call to release workspace
 IF (ISTEP .EQ. 240) IDO = 3
 GO TO 10
END IF
! Show number of function calls.
WRITE (NOUT,99999) PARAM(35)
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
99999 FORMAT (4X, 'Number of fcn calls with IVPRK =', F6.0)
END
SUBROUTINE FCN (N, T, Y, YPRIME)
! SPECIFICATIONS FOR ARGUMENTS
INTEGER N
REAL T, Y(N), YPRIME(N)
! SPECIFICATIONS FOR DATA VARIABLES
REAL AK1, AK2, AK3
!
DATA AK1, AK2, AK3/294.0E0, 3.0E0, 0.01020408E0/
!
YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
YPRIME(2) = -AK2*Y(2) + AK3*(1.0E0-Y(2))*Y(1)
RETURN
END

```

## Output

| ISTEP | Time    | Y1    | Y2    |
|-------|---------|-------|-------|
| 1     | 24.000  | 0.688 | 0.002 |
| 2     | 48.000  | 0.634 | 0.002 |
| 3     | 72.000  | 0.589 | 0.002 |
| 4     | 96.000  | 0.549 | 0.002 |
| 5     | 120.000 | 0.514 | 0.002 |
| 6     | 144.000 | 0.484 | 0.002 |
| 7     | 168.000 | 0.457 | 0.002 |
| 8     | 192.000 | 0.433 | 0.001 |
| 9     | 216.000 | 0.411 | 0.001 |
| 10    | 240.000 | 0.391 | 0.001 |

Number of fcn calls with IVPRK = 2153.

---

## IVMRK

Solves an initial-value problem  $y' = f(t, y)$  for ordinary differential equations using Runge-Kutta pairs of various orders.

### Required Arguments

**IDO** — Flag indicating the state of the computation. (Input/Output)

| <b>IDO</b> | <b>State</b>                                           |
|------------|--------------------------------------------------------|
| 1          | Initial entry                                          |
| 2          | Normal re-entry                                        |
| 3          | Final call to release workspace                        |
| 4          | Return after a step                                    |
| 5          | Return for function evaluation (reverse communication) |

Normally, the initial call is made with `IDO = 1`. The routine then sets `IDO = 2`, and this value is used for all but the last call that is made with `IDO = 3`. This final call is used to release workspace, which was automatically allocated by the initial call with `IDO = 1`.

**FCN** — User-supplied subroutine to evaluate functions. The usage is

`CALL FCN (N, T, Y, YPRIME)`, where

**N** — Number of equations. (Input)

**T** — Independent variable. (Input)

**Y** — Array of size **N** containing the dependent variable values,  $y$ . (Input)

**YPRIME** — Array of size **N** containing the values of the vector  $y'$  evaluated at  $(t, y)$ . (Output)

**FCN** must be declared `EXTERNAL` in the calling program.

**T** — Independent variable. (Input/Output)

On input, **T** contains the initial value. On output, **T** is replaced by **TEND** unless error conditions have occurred.

**TEND** — Value of  $t$  where the solution is required. (Input)

The value of **TEND** may be less than the initial value of  $t$ .

**Y** — Array of size **N** of dependent variables. (Input/Output)

On input, **Y** contains the initial values. On output, **Y** contains the approximate solution.

**YPRIME** — Array of size **N** containing the values of the vector  $y'$  evaluated at  $(t, y)$ .

(Output)

### Optional Arguments

**N** — Number of differential equations. (Input)

Default: `N = size (Y,1)`.

## FORTRAN 90 Interface

Generic:     `CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME [, ...])`

Specific:    The specific interface names are `S_IVMRK` and `D_IVMRK`.

## FORTRAN 77 Interface

Single:     `CALL IVMRK (IDO, N, FCN, T, TEND, Y, YPRIME)`

Double:     The double precision name is `DIVMRK`.

## Description

Routine `IVMRK` finds an approximation to the solution of a system of first-order differential equations of the form  $y' = f(t, y)$  with given initial data. Relative local error is controlled according to a user-supplied tolerance. For added efficiency, three Runge-Kutta formula pairs, of orders 3, 5, and 8, are available.

Optionally, the values of the vector  $y'$  can be passed to `IVMRK` by reverse communication, avoiding the user-supplied subroutine `FCN`. Reverse communication is especially useful in applications that have complicated algorithmic requirement for the evaluations of  $f(t, y)$ . Another option allows assessment of the global error in the integration.

The routine `IVMRK` is based on the codes contained in `RKSUITE`, developed by R. W. Brankin, I. Gladwell, and L. F. Shampine (1991).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of `I2MRK/DI2MRK`. The reference is:

```
CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES,
PARAM, YMAX, RMSEERR, WORK, IWORK)
```

The additional arguments are as follows:

**TOL** — Tolerance for error control. (Input)

**THRES** — Array of size `N`. (Input)

`THRES(I)` is a threshold for solution component `Y(I)`. It is chosen so that the value of `Y(L)` is not important when `Y(L)` is smaller in magnitude than `THRES(L)`. `THRES(L)` must be greater than or equal to `sqrt(amach(4))`.

**PARAM** — A floating-point array of size 50 containing optional parameters. (Input/Output)

If a parameter is zero, then a default value is used. These default values are given below. The following parameters must be set by the user:

| <b>PARAM</b> | <b>Definition</b>                                                                                                                                                                                                                                                               |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 HINIT      | Initial value of the step size. Must be chosen such that $0.01 \geq \text{HINIT} \geq 10.0 \text{ amach}(4)$ . Default: automatic selection of stepsize                                                                                                                         |
| 2 METHOD     | 1 - use the (2, 3) pair<br>2 - use the (4, 5) pair<br>3 - use the (7, 8) pair.<br>Default: METHOD = 1 if $1.e-2 \geq \text{tol} > 1.e-4$<br>METHOD = 2 if $1.e-4 \geq \text{tol} > 1.e-6$<br>METHOD = 3 if $1.e-6 \geq \text{tol}$                                              |
| 3 ERREST     | ERREST = 1 attempts to assess the true error, the difference between the numerical solution and the true solution. The cost of this is roughly twice the cost of the integration itself with METHOD = 2 or METHOD = 3, and three times with METHOD = 1.<br>Default: ERREST = 0. |
| 4 INTRP      | If nonzero, then return the IDO = 4 before each step. See Comment 3.<br>Default: 0.                                                                                                                                                                                             |
| 5 RCSTAT     | If nonzero, then reverse communication is used to get derivative information. See Comment 4.<br>Default: 0.                                                                                                                                                                     |
| 6 - 30       | Not used.                                                                                                                                                                                                                                                                       |

The following entries are set by the program:

|           |                                                                                                     |
|-----------|-----------------------------------------------------------------------------------------------------|
| 31 HTRIAL | Current trial step size.                                                                            |
| 32 NSTEP  | Number of steps taken.                                                                              |
| 33 NFCN   | Number of function evaluations.                                                                     |
| 34 ERRMAX | The maximum approximate weighted true error taken over all solution components and all steps from T |
| 35 TERRMX | First value of the independent variable where an                                                    |

**YMAX** — Array of size N, where YMAX(L) is the largest value of ABS (Y (L) ) computed at any step in the integration so far.

**RMSERR** — Array of size N where RMSERR(L) approximates the RMS average of the true error of the numerical solution for the L-th solution component,  $L = 1, \dots, N$ . The average is taken over all steps from T through the current integration point. RMSERR is accessed and set only if PARAM(3) = 1.

**WORK** — Floating point work array of size 39N using the working precision. The contents of WORK must not be changed from the first call with IDO = 1 until after the final call with IDO = 3.

**IWORK** — Length of array work. (Input)

2. Informational errors

Type Code

4 1 It does not appear possible to achieve the accuracy specified by `TOL` and `THRES(*)` using the current precision and `METHOD`. A larger value for `METHOD`, if possible, will permit greater accuracy with this precision. The integration must be restarted.

4 2 The global error assessment may not be reliable beyond the current integration point `T`. This may occur because either too little or too much accuracy has been requested or because  $f(t, y)$  is not smooth enough for values of  $t$  just past `TEND` and current values of the solution  $y$ . This return does not mean that you cannot integrate past `TEND`, rather that you cannot do it with `PARAM(3) = 1`.

3 If `PARAM(4)` is nonzero, the subroutine returns with `IDO = 4` and will resume calculation at the point of interruption if re-entered with `IDO = 4`. Some parameters the user might want to examine are `IDO`, `HTRIAL`, `NSTEP`, `NFCN`, `T`, and `Y`. The array `Y` contains the newly computed trial value for  $y(t)$ , accepted or not.

4 If `PARAM(5)` is nonzero, the subroutine will return with `IDO = 5`. At this time, evaluate the derivatives at `T`, place the result in `YPRIME`, and call `IVMRK` again. The dummy function `I40RK/DI40RK` may be used in place of `FCN`.

### Example 1

This example integrates the small system (A.2.B2) from the test set of Enright and Pryce (1987):

$$\begin{aligned}y_1' &= -y_1 + y_2 \\y_2' &= y_1 - 2y_2 + y_3 \\y_3' &= y_2 - y_3 \\y_1(0) &= 2 \\y_2(0) &= 0 \\y_3(0) &= 1\end{aligned}$$

```
USE IVMRK_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER N

PARAMETER (N=3)
! Specifications for local variables
INTEGER IDO
REAL T, TEND, Y(N), YPRIME(N)
```

```

EXTERNAL FCN
! Set initial conditions
T = 0.0
TEND = 20.0
Y(1) = 2.0
Y(2) = 0.0
Y(3) = 1.0
IDO = 1
CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME)
!
! Final call to release workspace
IDO = 3
CALL IVMRK (IDO, FCN, T, TEND, Y, YPRIME)
!
CALL WRRRN ('Y', Y)
END
!
SUBROUTINE FCN (N, T, Y, YPRIME)
! Specifications for arguments
INTEGER N
REAL T, Y(*), YPRIME(*)
!
YPRIME(1) = -Y(1) + Y(2)
YPRIME(2) = Y(1) - 2.0*Y(2) + Y(3)
YPRIME(3) = Y(2) - Y(3)
RETURN
END

```

## Output

```

 Y
1 1.000
2 1.000
3 1.000

```

## Additional Examples

### Example 2

This problem is the same mildly stiff problem (A.1.F2) from the test set of Enright and Pryce as Example 2 for [IVPRK](#).

$$\begin{aligned}
 y_1' &= -y_1 - y_1 y_2 + k_1 y_2 \\
 y_2' &= -k_2 y_2 + k_3 (1 - y_2) y_1 \\
 y_1(0) &= 1 \\
 y_2(0) &= 0 \\
 k_1 &= 294 \\
 k_2 &= 3 \\
 k_3 &= 0.01020408 \\
 tend &= 240
 \end{aligned}$$

Although not a stiff solver, one notes the greater efficiency of IVMRK over IVPK, in terms of derivative evaluations. Reverse communication is also used in this example. Users will find this feature particularly helpful if their derivative evaluation scheme is difficult to isolate in a separate subroutine.

```

 USE I2MRK_INT
 USE UMACH_INT
 USE AMACH_INT

 IMPLICIT NONE
 INTEGER N

 PARAMETER (N=2)
!
! Specifications for local variables
 INTEGER IDO, ISTEP, LWORK, NOUT
 REAL PARAM(50), PREC, RMSERR(N), T, TEND, THRES(N), TOL, &
 WORK(1000), Y(N), YMAX(N), YPRIME(N)
 REAL AK1, AK2, AK3
 SAVE AK1, AK2, AK3
!
! Specifications for intrinsics
 INTRINSIC SQRT
 REAL SQRT
!
! Specifications for subroutines
 EXTERNAL I40RK
!
! Specifications for functions
!
 DATA AK1, AK2, AK3/294.0, 3.0, 0.01020408/
!
 CALL UMACH (2, NOUT)
!
! Set initial conditions
 T = 0.0
 Y(1) = 1.0
 Y(2) = 0.0
!
! Set tolerance for error control,
! threshold vector and parameter
! vector
 TOL = .001
 PREC = AMACH(4)
 THRES = SQRT (PREC)
 PARAM = 0.0E0
 LWORK = 1000
!
! Turn on derivative evaluation by
! reverse communication
 PARAM(5) = 1
 IDO = 1
 ISTEP = 24
!
! Print header
 WRITE (NOUT,99998)
10 CONTINUE
 TEND = ISTEP
 CALL I2MRK (IDO, N, I40RK, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
 YMAX, RMSERR, WORK, LWORK)
 IF (IDO .EQ. 5) THEN
!
! Evaluate derivatives
!

```

```

 YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
 YPRIME(2) = -AK2*Y(2) + AK3*(1.0-Y(2))*Y(1)
 GO TO 10
 ELSE IF (ISTEP .LE. 240) THEN
!
! Integrate to 10 equally spaced points
!
 WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
 IF (ISTEP .EQ. 240) IDO = 3
 ISTEP = ISTEP + 24
 GO TO 10
 END IF
!
! Show number of derivative evaluations
!
 WRITE (NOUT,99999) PARAM(33)
99998 FORMAT (3X, 'ISTEP', 5X, 'TIME', 9X, 'Y1', 10X, 'Y2')
99999 FORMAT (/, 4X, 'NUMBER OF DERIVATIVE EVALUATIONS WITH IVMRK =', &
 F6.0)
 END

! DUMMY FUNCTION TO TAKE THE PLACE OF DERIVATIVE EVALUATOR
 SUBROUTINE I40RK (N, T, Y, YPRIME)
 INTEGER N
 REAL T, Y(*), YPRIME(*)
 RETURN
 END

```

## Output

| ISTEP | TIME    | Y1    | Y2    |
|-------|---------|-------|-------|
| 1     | 24.000  | 0.688 | 0.002 |
| 2     | 48.000  | 0.634 | 0.002 |
| 3     | 72.000  | 0.589 | 0.002 |
| 4     | 96.000  | 0.549 | 0.002 |
| 5     | 120.000 | 0.514 | 0.002 |
| 6     | 144.000 | 0.484 | 0.002 |
| 7     | 168.000 | 0.457 | 0.002 |
| 8     | 192.000 | 0.433 | 0.001 |
| 9     | 216.000 | 0.411 | 0.001 |
| 10    | 240.000 | 0.391 | 0.001 |

NUMBER OF DERIVATIVE EVALUATIONS WITH IVMRK = 1375.

## Example 3

This example demonstrates how exceptions may be handled. The problem is from Enright and Pryce (A.2.F1), and has discontinuities. We choose this problem to force a failure in the global error estimation scheme, which requires some smoothness in  $y$ . We also request an initial relative error tolerance which happens to be unsuitably small in this precision.

If the integration fails because of problems in global error assessment, the assessment option is turned off, and the integration is restarted. If the integration fails because the requested accuracy is not achievable, the tolerance is increased, and global error assessment is requested. The reason error assessment is turned on is that prior assessment failures may have been due more in part to an overly stringent tolerance than lack of smoothness in the derivatives.

When the integration is successful, the example prints the final relative error tolerance, and indicates whether or not global error estimation was possible.

$$y_1' = y_2$$

$$y_2' = \begin{cases} 2ay_2 - (\pi^2 + a^2)y_1 + 1, & [x] \text{ even} \\ 2ay_2 - (\pi^2 + a^2)y_1 - 1, & [x] \text{ odd} \end{cases}$$

$$y_1(0) = 0$$

$$y_2(0) = 0$$

$$a = 0.1$$

$$[x] = \text{largest integer} \leq x$$

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)

!
! Specifications for local variables
INTEGER IDO, LWORK, NOUT
REAL PARAM(50), PREC, RMSERR(N), T, TEND, THRES(N), TOL, &
WORK(100), Y(N), YMAX(N), YPRIME(N)

!
! Specifications for intrinsics
INTRINSIC SQRT
REAL SQRT

!
! Specifications for subroutines

!
! Specifications for functions
EXTERNAL FCN

!
!
CALL UMACH (2, NOUT)
! Turn off stopping for FATAL errors
CALL ERSET (4, -1, 0)
! Initialize input, turn on global
! error assessment

LWORK = 100
PREC = AMACH(4)
TOL = SQRT(PREC)
PARAM = 0.0E01
THRES = TOL
TEND = 20.0E0
PARAM(3) = 1

!
10 CONTINUE
! Set initial values

T = 0.0E0
Y(1) = 0.0E0

```

```

Y(2) = 0.0E0
IDO = 1
CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
 YMAX, RMSERR, WORK, LWORK)
IF (IERCD() .EQ. 32) THEN
!
! Unable to achieve requested
! accuracy, so increase tolerance.
! Activate global error assessment
!
! TOL = 10.0*TOL
! PARAM(3) = 1
! WRITE (NOUT,99995) TOL
! GO TO 10
ELSE IF (IERCD() .EQ. 34) THEN
!
! Global error assessment has failed,
! cannot continue from this point,
! so restart integration
!
! WRITE (NOUT,99996)
! PARAM(3) = 0
! GO TO 10
END IF
!
! Final call to release workspace
!
! IDO = 3
! CALL I2MRK (IDO, N, FCN, T, TEND, Y, YPRIME, TOL, THRES, PARAM,&
! YMAX, RMSERR, WORK, LWORK)
!
! Summarize status
!
! WRITE (NOUT,99997) TOL
! IF (PARAM(3) .EQ. 1) THEN
! WRITE (NOUT,99998)
! ELSE
! WRITE (NOUT,99999)
! END IF
! CALL WRRRN ('Y', Y)
!
99995 FORMAT (/, 'CHANGING TOLERANCE TO ', E9.3, ' AND RESTARTING ...'&
 , //, 'ALSO (RE)ENABLING GLOBAL ERROR ASSESSMENT', /)
99996 FORMAT (/, 'DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...'&
 , /)
99997 FORMAT (/, 72('-'), //, 'SOLUTION OBTAINED WITH TOLERANCE = ',&
 E9.3)
99998 FORMAT ('GLOBAL ERROR ASSESSMENT IS AVAILABLE')
99999 FORMAT ('GLOBAL ERROR ASSESSMENT IS NOT AVAILABLE')
!
END
!
SUBROUTINE FCN (N, T, Y, YPRIME)
USE CONST_INT
!
! Specifications for arguments
!
INTEGER N
REAL T, Y(*), YPRIME(*)
!
! Specifications for local variables
!
REAL A
REAL PI
LOGICAL FIRST
SAVE FIRST, PI
!
! Specifications for intrinsics

```

```

 INTRINSIC INT, MOD
 INTEGER INT, MOD
!
! Specifications for functions
!
 DATA FIRST/.TRUE./
!
 IF (FIRST) THEN
 PI = CONST('PI')
 FIRST = .FALSE.
 END IF
!
 A = 0.1E0
 YPRIME(1) = Y(2)
 IF (MOD(INT(T),2) .EQ. 0) THEN
 YPRIME(2) = 2.0E0*A*Y(2) - (PI*PI+A*A)*Y(1) + 1.0E0
 ELSE
 YPRIME(2) = 2.0E0*A*Y(2) - (PI*PI+A*A)*Y(1) - 1.0E0
 END IF
 RETURN
 END

```

## Output

```

*** FATAL ERROR 34 from i2mrk. The global error assessment may not
*** be reliable for T past 9.994749E-01. The integration is
*** being terminated.

```

DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...

```

*** FATAL ERROR 32 from i2mrk. In order to satisfy the error
*** requirement I6MRK would have to use a step size of
*** 3.647129E- 06 at TNOW = 9.999932E-01. This is too small
*** for the current precision.

```

CHANGING TOLERANCE TO 0.345E-02 AND RESTARTING ...  
ALSO (RE)ENABLING GLOBAL ERROR ASSESSMENT

```

*** FATAL ERROR 34 from i2mrk. The global error assessment may
*** not be reliable for T past 9.986024E-01. The integration
*** is being terminated.

```

DISABLING GLOBAL ERROR ASSESSMENT AND RESTARTING ...

-----  
SOLUTION OBTAINED WITH TOLERANCE = 0.345E-02  
GLOBAL ERROR ASSESSMENT IS NOT AVAILABLE

```

 Y
1 -12.30

```

---

## IVPAG



Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.

### Required Arguments

**IDO** — Flag indicating the state of the computation. (Input/Output)

**IDO State**

- 1 Initial entry
- 2 Normal re-entry
- 3 Final call to release workspace
- 4 Return because of interrupt 1
- 5 Return because of interrupt 2 with step accepted
- 6 Return because of interrupt 2 with step rejected
- 7 Return for new value of matrix  $A$ .

Normally, the initial call is made with  $IDO = 1$ . The routine then sets  $IDO = 2$ , and this value is then used for all but the last call that is made with  $IDO = 3$ . This final call is only used to release workspace, which was automatically allocated by the initial call with  $IDO = 1$ . See Comment 5 for a description of the interrupts.

When  $IDO = 7$ , the matrix  $A$  at  $t$  must be recomputed and IVPAG/DIVPAG called again. No other argument (including  $IDO$ ) should be changed. This value of  $IDO$  is returned only if  $PARAM(19) = 2$ .

**FCN** — User-supplied subroutine to evaluate functions. The usage is

CALL FCN (N, T, Y, YPRIME), where

- N — Number of equations. (Input)
- T — Independent variable,  $t$ . (Input)
- Y — Array of size N containing the dependent variable values,  $y$ . (Input)

YPRIME – Array of size N containing the values of the vector  $y'$  evaluated at ( $t$ ,  $y$ ). (Output)  
See Comment 3.

FCN must be declared EXTERNAL in the calling program.

**FCNJ** — User-supplied subroutine to compute the Jacobian. The usage is

CALL FCNJ (N, T, Y, DYPDY) where

N – Number of equations. (Input)

T – Independent variable,  $t$ . (Input)

Y – Array of size N containing the dependent variable values,  $y(t)$ . (Input)

DYPDY – An array, with data structure and type determined by PARAM(14) = MTYPE, containing the required partial derivatives  $\partial f_i / \partial y_j$ . (Output)

These derivatives are to be evaluated at the current values of ( $t$ ,  $y$ ). When the Jacobian is dense, MTYPE = 0 or = 2, the leading dimension of DYPDY has the value N. When the Jacobian matrix is banded, MTYPE = 1, and the leading dimension of DYPDY has the value  $2 * NLC + NUC + 1$ . If the matrix is banded positive definite symmetric, MTYPE = 3, and the leading dimension of DYPDY has the value NUC + 1.

FCNJ must be declared EXTERNAL in the calling program. If PARAM(19) = IATYPE is nonzero, then FCNJ should compute the Jacobian of the righthand side of the equation  $Ay' = f(t, y)$ . The subroutine FCNJ is used only if PARAM(13) = MITER = 1.

**T** — Independent variable,  $t$ . (Input/Output)

On input, T contains the initial independent variable value. On output, T is replaced by TEND unless error or other normal conditions arise. See IDO for details.

**TEND** — Value of  $t = tend$  where the solution is required. (Input)

The value  $tend$  may be less than the initial value of  $t$ .

**Y** — Array of size NEQ of dependent variables,  $y(t)$ . (Input/Output)

On input, Y contains the initial values,  $y(t_0)$ . On output, Y contains the approximate solution,  $y(t)$ .

## Optional Arguments

**NEQ** — Number of differential equations. (Input)

Default: NEQ = size (Y,1)

**A** — Matrix structure used when the system is implicit. (Input)

The matrix A is referenced only if PARAM(19) = IATYPE is nonzero. Its data structure is determined by PARAM(14) = MTYPE. The matrix A must be nonsingular and MITER must be 1 or 2. See Comment 3.

**TOL** — Tolerance for error control. (Input)

An attempt is made to control the norm of the local error such that the global error is

proportional to TOL.  
 Default: TOL = .001

**PARAM** — A *floating-point* array of size 50 containing optional parameters. (Input/Output)  
 If a parameter is zero, then the default value is used. These default values are given below. Parameters that concern values of the step size are applied in the direction of integration. The following parameters may be set by the user:

|    | <b>PARAM</b> | <b>Meaning</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | HINIT        | Initial value of the step size H. Always nonnegative.<br>Default: $0.001 tend - t_0 $ .                                                                                                                                                                                                                                                                                                                                                                                                   |
| 2  | HMIN         | Minimum value of the step size H. Default: 0.0.                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 3  | HMAX         | Maximum value of the step size H. Default: No limit, beyond the machine scale, is imposed on the step size.                                                                                                                                                                                                                                                                                                                                                                               |
| 4  | MXSTEP       | Maximum number of steps allowed. Default: 500.                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 5  | MXFCN        | Maximum number of function evaluations allowed. Default: No enforced limit.                                                                                                                                                                                                                                                                                                                                                                                                               |
| 6  | MAXORD       | Maximum order of the method. Default: If Adams-Moulton method is used, then 12. If Gear's or BDF method is used, then 5. The defaults are the maximum values allowed.                                                                                                                                                                                                                                                                                                                     |
| 7  | INTRP1       | If this value is set nonzero, the subroutine will return before every step with IDO = 4. See Comment 5. Default: 0.                                                                                                                                                                                                                                                                                                                                                                       |
| 8  | INTRP2       | If this value is nonzero, the subroutine will return after every successful step with IDO = 5 and return with IDO = 6 after every unsuccessful step. See Comment 5. Default: 0                                                                                                                                                                                                                                                                                                            |
| 9  | SCALE        | A measure of the scale of the problem, such as an approximation to the average value of a norm of the Jacobian along the solution. Default: 1.0                                                                                                                                                                                                                                                                                                                                           |
| 10 | INORM        | Switch determining error norm. In the following, $e_i$ is the absolute value of an estimate of the error in $y_i(t)$ .<br>Default: 0.<br>0 — $\min(\text{absolute error, relative error}) = \max(e_i/w_i); i = 1, \dots, N$ , where $w_i = \max( y_i(t) , 1.0)$ .<br>1 — absolute error = $\max(e_i), i = 1 \dots, NEQ$ .<br>2 — $\max(e_i / w_i), i = 1 \dots, N$ where $w_i = \max( y_i(t) , \text{FLOOR})$ , and FLOOR is the value PARAM(11).<br>3 — Scaled Euclidean norm defined as |

$$YMAX = \sqrt{\sum_{i=1}^{NEQ} e_i^2 / w_i^2}$$

where  $w_i = \max(|y_i(t)|, 1.0)$ . Other definitions of YMAX can be specified by the user, as explained in Comment 1.

|    |       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 11 | FLOOR | Used in the norm computation associated the parameter INORM. Default: 1.0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 12 | METH  | Integration method indicator.<br>1 = METH selects the Adams-Moulton method.<br>2 = METH selects Gear's BDF method.<br>Default: 1.                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 13 | MITER | Nonlinear solver method indicator.<br><i>Note:</i> If the problem is stiff and a chord or modified Newton method is most efficient, use MITER = 1 or = 2.<br>0 = MITER selects functional iteration. The value IATYPE must be set to zero with this option.<br>1 = MITER selects a chord method with a user-provided Jacobian.<br>2 = MITER selects a chord method with a divided-difference Jacobian.<br>3 = MITER selects a chord method with the Jacobian replaced by a diagonal matrix based on a directional derivative. The value IATYPE must be set to zero with this option.<br>Default: 0. |
| 14 | MTYPE | Matrix type for A (if used) and the Jacobian (if MITER = 1 or = 2). When both are used, A and the Jacobian must be of the same type.<br>0 = MTYPE selects full matrices.<br>1 = MTYPE selects banded matrices.<br>2 = MTYPE selects symmetric positive definite matrices.<br>3 = MTYPE selects banded symmetric positive definite matrices.<br>Default: 0.                                                                                                                                                                                                                                          |
| 15 | NLC   | Number of lower codiagonals, used if MTYPE = 1.<br>Default: 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 16 | NUC   | Number of upper codiagonals, used if MTYPE = 1 or MTYPE = 3.<br>Default: 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 17 |       | Not used.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 18 | EPSJ  | Relative tolerance used in computing divided difference Jacobians.<br>Default: SQRT(AMACH(4)) .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

|       |        |                                                                                                                                                                                                                               |
|-------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 19    | IATYPE | Type of the matrix $A$ .<br>$0 = \text{IATYPE}$ implies $A$ is not used (the system is explicit).<br>$1 = \text{IATYPE}$ if $A$ is a constant matrix.<br>$2 = \text{IATYPE}$ if $A$ depends on $t$ .<br>Default: 0.           |
| 20    | LDA    | Leading dimension of array $A$ exactly as specified in the dimension statement in the calling program. Used if IATYPE is not zero.<br>Default:<br>N if MTYPE = 0 or = 2<br>NUC + NLC + 1 if MTYPE = 1<br>NUC + 1 if MTYPE = 3 |
| 21–30 |        | Not used.                                                                                                                                                                                                                     |

The following entries in the array `PARAM` are set by the program:

|       | <b>PARAM</b> | <b>Meaning</b>                       |
|-------|--------------|--------------------------------------|
| 31    | HTRIAL       | Current trial step size.             |
| 32    | HMINC        | Computed minimum step size.          |
| 33    | HMAXC        | Computed maximum step size.          |
| 34    | NSTEP        | Number of steps taken.               |
| 35    | NFCN         | Number of function evaluations used. |
| 36    | NJE          | Number of Jacobian evaluations.      |
| 37–50 |              | Not used.                            |

### **FORTRAN 90 Interface**

Generic: `CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y [, ...])`

Specific: The specific interface names are `S_IVPAG` and `D_IVPAG`.

### **FORTRAN 77 Interface**

Single: `CALL IVPAG (IDO, NEQ, FCN, FCNJ, A, T, TEND, TOL, PARAM, Y)`

Double: The double precision name is `DIVPAG`.

### **Description**

The routine `IVPAG` solves a system of first-order ordinary differential equations of the form  $y' = f(t, y)$  or  $Ay' = f(t, y)$  with initial conditions where  $A$  is a square nonsingular matrix of order  $N$ . Two classes of implicit linear multistep methods are available. The first is the implicit Adams-Moulton method (up to order twelve); the second uses the backward differentiation formulas BDF

(up to order five). The BDF method is often called Gear's stiff method. In both cases, because basic formulas are implicit, a system of nonlinear equations must be solved at each step. The derivative matrix in this system has the form  $L = A + \eta J$  where  $\eta$  is a small number computed by IVPAG and  $J$  is the Jacobian. When it is used, this matrix is computed in the user-supplied routine FCNJ or else it is approximated by divided differences as a default. Using defaults,  $A$  is the identity matrix. The data structure for the matrix  $L$  may be identified to be real general, real banded, symmetric positive definite, or banded symmetric positive definite. The default structure for  $L$  is real general.

## Comments

1. Workspace and a user-supplied error norm subroutine may be explicitly provided, if desired, by use of I2PAG/DI2PAG. The reference is:

```
CALL I2PAG (IDO, NEQ, FCN, FCNJ, A, T, TEND, TOL, PARAM,
 Y, YTEMP, YMAX, ERROR, SAVE1, SAVE2, PW, IPVT, VNORM)
```

None of the additional array arguments should be changed from the first call with  $IDO = 1$  until after the final call with  $IDO = 3$ . The additional arguments are as follows:

**YTEMP** — Array of size NMETH. (Workspace)

**YMAX** — Array of size NEQ containing the maximum  $Y$ -values computed so far.  
(Output)

**ERROR** — Array of size NEQ containing error estimates for each component of  $Y$ .  
(Output)

**SAVE1** — Array of size NEQ. (Workspace)

**SAVE2** — Array of size NEQ. (Workspace)

**PW** — Array of size NPW. (Workspace)

**IPVT** — Array of size NEQ. (Workspace)

**VNORM** — A Fortran subroutine to compute the norm of the error. (Input)

The routine may be provided by the user, or the IMSL routine I3PRK/DI3PRK may be used. In either case, the name must be declared in a Fortran EXTERNAL statement. If usage of the IMSL routine is intended, then the name I3PRK/DI3PRK should be specified. The usage of the error norm routine is  
CALL VNORM (NEQ, V, Y, YMAX, ENORM) where

| Arg | Definition                                                                  |
|-----|-----------------------------------------------------------------------------|
| NEQ | Number of equations. (Input).                                               |
| V   | Array of size N containing the vector whose norm is to be computed. (Input) |

| Arg   | Definition                                                               |
|-------|--------------------------------------------------------------------------|
| Y     | Array of size N containing the values of the dependent variable. (Input) |
| YMAX  | Array of size N containing the maximum values of $ y(t) $ . (Input).     |
| ENORM | Norm of the vector V. (Output).                                          |

VNORM must be declared EXTERNAL in the calling program.

## 2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                                                                                                                                                                                                  |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | After some initial success, the integration was halted by repeated error-test failures.                                                                                                                                                                                                                                                                                          |
| 4 | 2 | The maximum number of function evaluations have been used.                                                                                                                                                                                                                                                                                                                       |
| 4 | 3 | The maximum number of steps allowed have been used. The problem may be stiff.                                                                                                                                                                                                                                                                                                    |
| 4 | 4 | On the next step $T + H$ will equal $T$ . Either TOL is too small, or the problem is stiff.<br>Note: If the Adams-Moulton method is the one used in the integration, then users can switch to the BDF methods. If the BDF methods are being used, then these comments are gratuitous and indicate that the problem is too stiff for this combination of method and value of TOL. |
| 4 | 5 | After some initial success, the integration was halted by a test on TOL.                                                                                                                                                                                                                                                                                                         |
| 4 | 6 | Integration was halted after failing to pass the error test even after dividing the initial step size by a factor of $1.0E + 10$ . The value TOL may be too small.                                                                                                                                                                                                               |
| 4 | 7 | Integration was halted after failing to achieve corrector convergence even after dividing the initial step size by a factor of $1.0E + 10$ . The value TOL may be too small.                                                                                                                                                                                                     |
| 4 | 8 | IATYPE is nonzero and the input matrix A multiplying $y'$ is singular.                                                                                                                                                                                                                                                                                                           |

3. Both explicit systems, of the form  $y' = f(t, y)$ , and implicit systems,  $Ay' = f(t, y)$ , can be solved. If the system is explicit, then  $PARAM(19) = 0$ ; and the matrix A is not referenced. If the system is implicit, then  $PARAM(14)$  determines the data structure of the array A. If  $PARAM(19) = 1$ , then A is assumed to be a constant matrix. The value of A used on the first call (with  $IDO = 1$ ) is saved until after a call with  $IDO = 3$ . The value of A must not be changed between these calls.  
If  $PARAM(19) = 2$ , then the matrix is assumed to be a function of  $t$ .

4. If MTYPE is greater than zero, then MITER must equal 1 or 2.

5. If `PARAM(7)` is nonzero, the subroutine returns with `IDO= 4` and will resume calculation at the point of interruption if re-entered with `IDO = 4`. If `PARAM(8)` is nonzero, the subroutine will interrupt immediately after decides to accept the result of the most recent trial step. The value `IDO = 5` is returned if the routine plans to accept, or `IDO = 6` if it plans to reject. The value `IDO` may be changed by the user (by changing `IDO` from 6 to 5) to force acceptance of a step that would otherwise be rejected. Relevant parameters to observe after return from an interrupt are `IDO`, `HTRIAL`, `NSTEP`, `NFCN`, `NJE`, `T` and `Y`. The array `Y` contains the newly computed trial value  $y(t)$ .

### Example 1

Euler's equation for the motion of a rigid body not subject to external forces is

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

Its solution is, in terms of Jacobi elliptic functions,  $y_1(t) = \text{sn}(t; k)$ ,  $y_2(t) = \text{cn}(t; k)$ ,  $y_3(t) = \text{dn}(t; k)$  where  $k^2 = 0.51$ . The Adams-Moulton method of `IVPAG` is used to solve this system, since this is the default. All parameters are set to defaults.

The last call to `IVPAG` with `IDO = 3` releases IMSL workspace that was reserved on the first call to `IVPAG`. It is not necessary to release the workspace in this example because the program ends after solving a single problem. The call to release workspace is made as a model of what would be needed if the program included further calls to IMSL routines.

Because `PARAM(13) = MITER = 0`, functional iteration is used and so subroutine `FCNJ` is never called. It is included only because the calling sequence for `IVPAG` requires it.

```

USE IVPAG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N, NPARAM
PARAMETER (N=3, NPARAM=50)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IDO, IEND, NOUT
REAL A(1,1), T, TEND, TOL, Y(N)
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCN, FCNJ
! Initialize
!
IDO = 1
T = 0.0
Y(1) = 0.0
Y(2) = 1.0
Y(3) = 1.0
TOL = 1.0E-6
! Write title
CALL UMACH (2, NOUT)

```

```

 WRITE (NOUT,99998)
!
! Integrate ODE
 IEND = 0
10 CONTINUE
 IEND = IEND + 1
 TEND = IEND
!
! The array a(*,*) is not used.
 CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL)
 IF (IEND .LE. 10) THEN
 WRITE (NOUT,99999) T, Y
!
! Finish up
 IF (IEND .EQ. 10) IDO = 3
 GO TO 10
 END IF
99998 FORMAT (11X, 'T', 14X, 'Y(1)', 11X, 'Y(2)', 11X, 'Y(3)')
99999 FORMAT (4F15.5)
 END
!
 SUBROUTINE FCN (N, X, Y, YPRIME)
!
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL X, Y(N), YPRIME(N)
!
 YPRIME(1) = Y(2)*Y(3)
 YPRIME(2) = -Y(1)*Y(3)
 YPRIME(3) = -0.51*Y(1)*Y(2)
 RETURN
 END
!
 SUBROUTINE FCNJ (N, X, Y, DYDPDY)
!
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL X, Y(N), DYDPDY(N,*)
!
! This subroutine is never called
 RETURN
 END

```

## Output

| T        | Y(1)     | Y(2)     | Y(3)    |
|----------|----------|----------|---------|
| 1.00000  | 0.80220  | 0.59705  | 0.81963 |
| 2.00000  | 0.99537  | -0.09615 | 0.70336 |
| 3.00000  | 0.64141  | -0.76720 | 0.88892 |
| 4.00000  | -0.26961 | -0.96296 | 0.98129 |
| 5.00000  | -0.91173 | -0.41079 | 0.75899 |
| 6.00000  | -0.95751 | 0.28841  | 0.72967 |
| 7.00000  | -0.42877 | 0.90342  | 0.95197 |
| 8.00000  | 0.51092  | 0.85963  | 0.93106 |
| 9.00000  | 0.97567  | 0.21926  | 0.71730 |
| 10.00000 | 0.87790  | -0.47884 | 0.77906 |

## Additional Examples

### Example 2

The BDF method of IVPAG is used to solve Example 2 of IVPRK. We set PARAM(12) = 2 to designate the BDF method. A chord or modified Newton method, with the Jacobian computed by divided differences, is used to solve the nonlinear equations. Thus, we set PARAM(13) = 2. The number of evaluations of  $y'$  is printed after the last output point, showing the efficiency gained when using a stiff solver compared to using IVPRK on this problem. The number of evaluations may vary, depending on the accuracy and other arithmetic characteristics of the computer.

```
USE IVPAG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER MXPARAM, N
PARAMETER (MXPARAM=50, N=2)
!
! SPECIFICATIONS FOR PARAMETERS
INTEGER MABSE, MBDF, MSOLVE
PARAMETER (MABSE=1, MBDF=2, MSOLVE=2)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IDO, ISTEP, NOUT
REAL A(1,1), PARAM(MXPARAM), T, TEND, TOL, Y(N)
!
! SPECIFICATIONS FOR SUBROUTINES
!
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCN, FCNJ
!
CALL UMACH (2, NOUT)
!
! Set initial conditions
T = 0.0
Y(1) = 1.0
Y(2) = 0.0
!
! Set error tolerance
TOL = 0.001
!
! Set PARAM to defaults
PARAM = 0.0E0
!
PARAM(10) = MABSE
!
! Select BDF method
PARAM(12) = MBDF
!
! Select chord method and
! a divided difference Jacobian.
PARAM(13) = MSOLVE
!
! Print header
WRITE (NOUT,99998)
IDO = 1
ISTEP = 0
10 CONTINUE
ISTEP = ISTEP + 24
TEND = ISTEP
!
! The array a(*,*) is not used.
CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL, &
PARAM=PARAM)
IF (ISTEP .LE. 240) THEN
```

```

 WRITE (NOUT,'(I6,3F12.3)') ISTEP/24, T, Y
! Final call to release workspace
 IF (ISTEP .EQ. 240) IDO = 3
 GO TO 10
 END IF
! Show number of function calls.
 WRITE (NOUT,99999) PARAM(35)
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2')
99999 FORMAT (4X, 'Number of fcn calls with IVPAG =', F6.0)
 END
 SUBROUTINE FCN (N, T, Y, YPRIME)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL T, Y(N), YPRIME(N)
! SPECIFICATIONS FOR SAVE VARIABLES
 REAL AK1, AK2, AK3
 SAVE AK1, AK2, AK3
!
 DATA AK1, AK2, AK3/294.0E0, 3.0E0, 0.01020408E0/
!
 YPRIME(1) = -Y(1) - Y(1)*Y(2) + AK1*Y(2)
 YPRIME(2) = -AK2*Y(2) + AK3*(1.0E0-Y(2))*Y(1)
 RETURN
 END
 SUBROUTINE FCNJ (N, T, Y, DYDPY)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL T, Y(N), DYDPY(N,*)
!
 RETURN
 END

```

## Output

| ISTEP | Time    | Y1    | Y2    |
|-------|---------|-------|-------|
| 1     | 24.000  | 0.689 | 0.002 |
| 2     | 48.000  | 0.636 | 0.002 |
| 3     | 72.000  | 0.590 | 0.002 |
| 4     | 96.000  | 0.550 | 0.002 |
| 5     | 120.000 | 0.515 | 0.002 |
| 6     | 144.000 | 0.485 | 0.002 |
| 7     | 168.000 | 0.458 | 0.002 |
| 8     | 192.000 | 0.434 | 0.001 |
| 9     | 216.000 | 0.412 | 0.001 |
| 10    | 240.000 | 0.392 | 0.001 |

Number of fcn calls with IVPAG = 73.

## Example 3

The BDF method of IVPAG is used to solve the so-called Robertson problem:

$$\begin{aligned}
y_1' &= -c_1 y_1 + c_2 y_2 y_3 & y_1(0) &= 1 \\
y_2' &= -y_1' - y_3' & y_2(0) &= 0 \\
y_3' &= c_3 y_2^2 & y_3(0) &= 0 \\
c_1 &= 0.04, c_2 = 10^4, c_3 = 3 \times 10^7 & 0 \leq t \leq 10
\end{aligned}$$

Output is obtained after each unit of the independent variable. A user-provided subroutine for the Jacobian matrix is used. An absolute error tolerance of  $10^{-5}$  is required.

```

USE IVPAG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER MXPARM, N
PARAMETER (MXPARM=50, N=3)
! SPECIFICATIONS FOR PARAMETERS
INTEGER MABSE, MBDF, MSOLVE
PARAMETER (MABSE=1, MBDF=2, MSOLVE=1)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IDO, ISTEP, NOUT
REAL A(1,1), PARAM(MXPARM), T, TEND, TOL, Y(N)
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCN, FCNJ
!
CALL UMACH (2, NOUT)
! Set initial conditions
T = 0.0
Y(1) = 1.0
Y(2) = 0.0
Y(3) = 0.0
! Set error tolerance
TOL = 1.0E-5
! Set PARAM to defaults
PARAM = 0.0E0
!
! Select absolute error control
PARAM(10) = MABSE
! Select BDF method
PARAM(12) = MBDF
! Select chord method and
! a user-provided Jacobian.
PARAM(13) = MSOLVE
! Print header
WRITE (NOUT,99998)
IDO = 1
ISTEP = 0
10 CONTINUE
ISTEP = ISTEP + 1
TEND = ISTEP
!
! The array a(*,*) is not used.
CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y, TOL=TOL, PARAM=PARAM)

```

```

 IF (ISTEP .LE. 10) THEN
 WRITE (NOUT,'(I6,F12.2,3F13.5)') ISTEP, T, Y
! Final call to release workspace
 IF (ISTEP .EQ. 10) IDO = 3
 GO TO 10
 END IF
99998 FORMAT (4X, 'ISTEP', 5X, 'Time', 9X, 'Y1', 11X, 'Y2', 11X, &
 'Y3')
 END
 SUBROUTINE FCN (N, T, Y, YPRIME)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL T, Y(N), YPRIME(N)
! SPECIFICATIONS FOR SAVE VARIABLES
 REAL C1, C2, C3
 SAVE C1, C2, C3
!
 DATA C1, C2, C3/0.04E0, 1.0E4, 3.0E7/
!
 YPRIME(1) = -C1*Y(1) + C2*Y(2)*Y(3)
 YPRIME(3) = C3*Y(2)**2
 YPRIME(2) = -YPRIME(1) - YPRIME(3)
 RETURN
 END
 SUBROUTINE FCNJ (N, T, Y, DYDPY)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL T, Y(N), DYDPY(N,*)
! SPECIFICATIONS FOR SAVE VARIABLES
 REAL C1, C2, C3
 SAVE C1, C2, C3
! SPECIFICATIONS FOR SUBROUTINES
 EXTERNAL SSET
!
 DATA C1, C2, C3/0.04E0, 1.0E4, 3.0E7/
! Clear array to zero
 CALL SSET (N**2, 0.0, DYDPY, 1)
! Compute partials
 DYDPY(1,1) = -C1
 DYDPY(1,2) = C2*Y(3)
 DYDPY(1,3) = C2*Y(2)
 DYDPY(3,2) = 2.0*C3*Y(2)
 DYDPY(2,1) = -DYDPY(1,1)
 DYDPY(2,2) = -DYDPY(1,2) - DYDPY(3,2)
 DYDPY(2,3) = -DYDPY(1,3)
 RETURN
 END

```

## Output

| ISTEP | Time | Y1      | Y2      | Y3      |
|-------|------|---------|---------|---------|
| 1     | 1.00 | 0.96647 | 0.00003 | 0.03350 |
| 2     | 2.00 | 0.94164 | 0.00003 | 0.05834 |
| 3     | 3.00 | 0.92191 | 0.00002 | 0.07806 |
| 4     | 4.00 | 0.90555 | 0.00002 | 0.09443 |
| 5     | 5.00 | 0.89153 | 0.00002 | 0.10845 |
| 6     | 6.00 | 0.87928 | 0.00002 | 0.12070 |

|    |       |         |         |         |
|----|-------|---------|---------|---------|
| 7  | 7.00  | 0.86838 | 0.00002 | 0.13160 |
| 8  | 8.00  | 0.85855 | 0.00002 | 0.14143 |
| 9  | 9.00  | 0.84959 | 0.00002 | 0.15039 |
| 10 | 10.00 | 0.84136 | 0.00002 | 0.15862 |

#### Example 4

Solve the partial differential equation

$$e^{-t} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with the initial condition

$$u(t=0, x) = \sin x$$

and the boundary conditions

$$u(t, x=0) = u(t, x=\pi) = 0$$

on the square  $[0, 1] \times [0, \pi]$ , using the method of lines with a piecewise-linear Galerkin discretization. The exact solution is  $u(t, x) = \exp(1 - e^t) \sin x$ . The interval  $[0, \pi]$  is divided into equal intervals by choosing breakpoints  $x_k = k\pi/(N+1)$  for  $k = 0, \dots, N+1$ . The unknown function  $u(t, x)$  is approximated by

$$\sum_{k=1}^N c_k(t) \phi_k(x)$$

where  $\phi_k(x)$  is the piecewiselinear function that equals 1 at  $x_k$  and is zero at all of the other breakpoints. We approximate the partial differential equation by a system of  $N$  ordinary differential equations,  $A dc/dt = Rc$  where  $A$  and  $R$  are matrices of order  $N$ . The matrix  $A$  is given by

$$A_{ij} = \begin{cases} e^{-t} 2h/3 & \text{if } i = j \\ e^{-t} \int_0^\pi \phi_i(x) \phi_j(x) dx = e^{-t} h/6 & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

where  $h = 1/(N+1)$  is the mesh spacing. The matrix  $R$  is given by

$$R_{ij} = \begin{cases} -2/h & \text{if } i = j \\ \int_0^\pi \phi_i''(x) \phi_j(x) dx = -\int_0^\pi \phi_i'(x) \phi_j'(x) dx = 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The integrals involving

$$\phi_i''$$

are assigned the values of the integrals on the right-hand side, by using the boundary values and integration by parts. Because this system may be stiff, Gear's BDF method is used.

In the following program, the array  $Y(1:N)$  corresponds to the vector of coefficients,  $c$ . Note that  $Y$  contains  $N + 2$  elements;  $Y(0)$  and  $Y(N + 1)$  are used to store the boundary values. The matrix  $A$  depends on  $t$  so we set  $PARAM(19) = 2$  and evaluate  $A$  when  $IVPAG$  returns with  $IDO = 7$ . The subroutine  $FCN$  computes the vector  $Rc$ , and the subroutine  $FCNJ$  computes  $R$ . The matrices  $A$  and  $R$  are stored as band-symmetric positive-definite structures having one upper co-diagonal.

```

 USE IVPAG_INT
 USE CONST_INT
 USE WRRRN_INT
 USE SSET_INT

 IMPLICIT NONE
 INTEGER LDA, N, NPARAM, NUC
 PARAMETER (N=9, NPARAM=50, NUC=1, LDA=NUC+1)
! SPECIFICATIONS FOR PARAMETERS
 INTEGER NSTEP
 PARAMETER (NSTEP=4)
! SPECIFICATIONS FOR LOCAL VARIABLES
 INTEGER I, IATYPE, IDO, IMETH, INORM, ISTEP, MITER, MTYPE
 REAL A(LDA,N), C, HINIT, PARAM(NPARAM), PI, T, TEND, TMAX, &
 TOL, XPOINT(0:N+1), Y(0:N+1)
 CHARACTER TITLE*10
! SPECIFICATIONS FOR COMMON /COMHX/
 COMMON /COMHX/ HX
 REAL HX
! SPECIFICATIONS FOR INTRINSICS
 INTRINSIC EXP, REAL, SIN
 REAL EXP, REAL, SIN
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
 EXTERNAL FCN, FCNJ
! Initialize PARAM
 HINIT = 1.0E-3
 INORM = 1
 IMETH = 2
 MITER = 1
 MTYPE = 3
 IATYPE = 2
 PARAM = 0.0E0
 PARAM(1) = HINIT
 PARAM(10) = INORM921

 PARAM(12) = IMETH
 PARAM(13) = MITER
 PARAM(14) = MTYPE
 PARAM(16) = NUC
 PARAM(19) = IATYPE
! Initialize other arguments
 PI = CONST('PI')
 HX = PI/REAL(N+1)
 CALL SSET (N-1, HX/6., A(1:,2), LDA)

```

```

CALL SSET (N, 2.*HX/3., A(2:,1), LDA)
DO 10 I=0, N + 1
 XPOINT(I) = I*HX
 Y(I) = SIN(XPOINT(I))
10 CONTINUE
TOL = 1.0E-6
T = 0.0
TMAX = 1.0
!
! Integrate ODE
IDO = 1
ISTEP = 0
20 CONTINUE
ISTEP = ISTEP + 1
TEND = TMAX*REAL(ISTEP)/REAL(NSTEP)
30 CALL IVPAG (IDO, FCN, FCNJ, T, TEND, Y(1:), NEQ=N, A=A, &
 TOL=TOL, PARAM=PARAM)
!
! Set matrix A
IF (IDO .EQ. 7) THEN
 C = EXP(-T)
 CALL SSET (N-1, C*HX/6., A(1:,2), LDA)
 CALL SSET (N, 2.*C*HX/3., A(2:,1), LDA)
 GO TO 30
END IF
IF (ISTEP .LE. NSTEP) THEN
!
! Print solution
WRITE (TITLE, '(A,F5.3,A)') 'U(T=', T, ')'
CALL WRRRN (TITLE, Y, 1, N+2, 1)
!
! Final call to release workspace
IF (ISTEP .EQ. NSTEP) IDO = 3
GO TO 20
END IF
END
!
SUBROUTINE FCN (N, T, Y, YPRIME)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER N
REAL T, Y(*), YPRIME(N)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I
!
! SPECIFICATIONS FOR COMMON /COMHX/
COMMON /COMHX/ HX
REAL HX
!
! SPECIFICATIONS FOR SUBROUTINES
EXTERNAL SSCAL
!
YPRIME(1) = -2.0*Y(1) + Y(2)
DO 10 I=2, N - 1
 YPRIME(I) = -2.0*Y(I) + Y(I-1) + Y(I+1)
10 CONTINUE
YPRIME(N) = -2.0*Y(N) + Y(N-1)
CALL SSCAL (N, 1.0/HX, YPRIME, 1)
RETURN
END
!
SUBROUTINE FCNJ (N, T, Y, DYPDY)
!
! SPECIFICATIONS FOR ARGUMENTS

```

```

 INTEGER N
 REAL T, Y(*), DYDPDY(2,*)
!
! SPECIFICATIONS FOR COMMON /COMHX/
 COMMON /COMHX/ HX
 REAL HX
!
! SPECIFICATIONS FOR SUBROUTINES
 EXTERNAL SSET
!
 CALL SSET (N-1, 1.0/HX, DYDPDY(1,2), 2)
 CALL SSET (N, -2.0/HX, DYDPDY(2,1), 2)
 RETURN
 END

```

## Output

```

 U(T=0.250)
 1 2 3 4 5 6 7 8
0.0000 0.2321 0.4414 0.6076 0.7142 0.7510 0.7142 0.6076

 9 10 11
0.4414 0.2321 0.0000

 U(T=0.500)
 1 2 3 4 5 6 7 8
0.0000 0.1607 0.3056 0.4206 0.4945 0.5199 0.4945 0.4206

 9 10 11
0.3056 0.1607 0.0000

 U(T=0.750)
 1 2 3 4 5 6 7 8
0.0000 0.1002 0.1906 0.2623 0.3084 0.3243 0.3084 0.2623

 9 10 11
0.1906 0.1002 0.0000

 U(T=1.000)
 1 2 3 4 5 6 7 8
0.0000 0.0546 0.1039 0.1431 0.1682 0.1768 0.1682 0.1431

 9 10 11
0.1039 0.0546 0.0000

```

---

## BVPFD

Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite difference method with deferred corrections.

## Required Arguments

**FCNEQN** — User-supplied subroutine to evaluate derivatives. The usage is

CALL FCNEQN (N, T, Y, P, DYDT), where

N — Number of differential equations. (Input)

T — Independent variable,  $t$ . (Input)

Y — Array of size N containing the dependent variable values,  $y(t)$ . (Input)

P — Continuation parameter,  $p$ . (Input)

See Comment 3.

DYDT — Array of size N containing the derivatives  $y'(t)$ . (Output)

The name FCNEQN must be declared EXTERNAL in the calling program.

**FCNJAC** — User-supplied subroutine to evaluate the Jacobian. The usage is

CALL FCNJAC (N, T, Y, P, DYPDY), where

N — Number of differential equations. (Input)

T — Independent variable,  $t$ . (Input)

Y — Array of size N containing the dependent variable values. (Input)

P — Continuation parameter,  $p$ . (Input)

See Comments 3.

DYPDY — N by N array containing the partial derivatives  $a_{i,j} = \partial f_i / \partial y_j$  evaluated at  $(t, y)$ . The values  $a_{i,j}$  are returned in DYPDY(i, j). (Output)

The name FCNJAC must be declared EXTERNAL in the calling program.

**FCNBC** — User-supplied subroutine to evaluate the boundary conditions. The usage is

CALL FCNBC (N, YLEFT, YRIGHT, P, H), where

N — Number of differential equations. (Input)

YLEFT — Array of size N containing the values of the dependent variable at the left endpoint. (Input)

YRIGHT — Array of size N containing the values of the dependent variable at the right endpoint. (Input)

P — Continuation parameter,  $p$ . (Input)

See Comment 3.

H — Array of size N containing the boundary condition residuals. (Output)

The boundary conditions are defined by  $h_i = 0$ ; for  $i = 1, \dots, N$ . The left endpoint conditions must be defined first, then, the conditions involving both endpoints, and finally the right endpoint conditions.

The name `FCNBC` must be declared `EXTERNAL` in the calling program.

**FCNPEQ** — User-supplied subroutine to evaluate the derivative of  $y'$  with respect to the parameter  $p$ . The usage is

`CALL FCNPEQ (N, T, Y, P, DYDPD), where`

`N` — Number of differential equations. (Input)

`T` — Dependent variable,  $t$ . (Input)

`Y` — Array of size `N` containing the dependent variable values. (Input)

`P` — Continuation parameter,  $p$ . (Input)

See Comment 3.

`DYDPD` — Array of size `N` containing the derivative of  $y'$  evaluated at  $(t, y)$ . (Output)

The name `FCNPEQ` must be declared `EXTERNAL` in the calling program.

**FCNPBC** — User-supplied subroutine to evaluate the derivative of the boundary conditions with respect to the parameter  $p$ . The usage is

`CALL FCNPBC (N, YLEFT, YRIGHT, P, H), where`

`N` — Number of differential equations. (Input)

`YLEFT` — Array of size `N` containing the values of the dependent variable at the left endpoint. (Input)

`YRIGHT` — Array of size `N` containing the values of the dependent variable at the right endpoint. (Input)

`P` — Continuation parameter,  $p$ . (Input)

See Comment 3.

`H` — Array of size `N` containing the derivative of  $f_i$  with respect to  $p$ . (Output)

The name `FCNPBC` must be declared `EXTERNAL` in the calling program.

**NLEFT** — Number of initial conditions. (Input)

The value `NLEFT` must be greater than or equal to zero and less than `N`.

**NCUPBC** — Number of coupled boundary conditions. (Input)

The value `NLEFT + NCUPBC` must be greater than zero and less than or equal to `N`.

**TLEFT** — The left endpoint. (Input)

**TRIGHT** — The right endpoint. (Input)

**PISTEP** — Initial increment size for  $p$ . (Input)

If this value is zero, continuation will not be used in this problem. The routines `FCNPEQ` and `FCNPBC` will not be called.

**TOL** — Relative error control parameter. (Input)  
 The computations stop when  $\text{ABS}(\text{ERROR}(J, I))/\text{MAX}(\text{ABS}(Y(J, I)), 1.0) \cdot \text{LT} \cdot \text{TOL}$  for all  $J = 1, \dots, N$  and  $I = 1, \dots, \text{NGRID}$ . Here,  $\text{ERROR}(J, I)$  is the estimated error in  $Y(J, I)$ .

**TINIT** — Array of size `NINIT` containing the initial grid points. (Input)

**YINIT** — Array of size `N` by `NINIT` containing an initial guess for the values of  $Y$  at the points in `TINIT`. (Input)

**LINEAR** — Logical `.TRUE.` if the differential equations and the boundary conditions are linear. (Input)

**MXGRID** — Maximum number of grid points allowed. (Input)

**NFINAL** — Number of final grid points, including the endpoints. (Output)

**TFINAL** — Array of size `MXGRID` containing the final grid points. (Output)  
 Only the first `NFINAL` points are significant.

**YFINAL** — Array of size `N` by `MXGRID` containing the values of  $Y$  at the points in `TFINAL`. (Output)

**ERREST** — Array of size `N`. (Output)  
 $\text{ERREST}(J)$  is the estimated error in  $Y(J)$ .

### Optional Arguments

**N** — Number of differential equations. (Input)  
 Default: `N = size (YINIT,1)`.

**NINIT** — Number of initial grid points, including the endpoints. (Input)  
 It must be at least 4.  
 Default: `NINIT = size (TINIT,1)`.

**LDYINI** — Leading dimension of `YINIT` exactly as specified in the dimension statement of the calling program. (Input)  
 Default: `LDYINI = size (YINIT,1)`.

**PRINT** — Logical `.TRUE.` if intermediate output is to be printed. (Input)  
 Default: `PRINT = .FALSE.`

**LDYFIN** — Leading dimension of `YFINAL` exactly as specified in the dimension statement of the calling program. (Input)  
 Default: `LDYFIN = size (YFINAL,1)`.

## FORTRAN 90 Interface

Generic:    CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, NLEFT, NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT, YINIT, LINEAR, MXGRID, NFINAL, TFINAL, YFINAL, ERREST [, ...])

Specific:   The specific interface names are S\_BVPFD and D\_BVPFD.

## FORTRAN 77 Interface

Single:     CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, N, NLEFT, NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, NINIT, TINIT, YINIT, LDYINI, LINEAR, PRINT, MXGRID, NFINAL, TFINAL, YFINAL, LDYFIN, ERREST)

Double:     The double precision name is DBVPFD.

## Description

The routine BVPFD is based on the subprogram PASVA3 by M. Lentini and V. Pereyra (see Pereyra 1978). The basic discretization is the trapezoidal rule over a nonuniform mesh. This mesh is chosen adaptively, to make the local error approximately the same size everywhere. Higher-order discretizations are obtained by deferred corrections. Global error estimates are produced to control the computation. The resulting nonlinear algebraic system is solved by Newton's method with step control. The linearized system of equations is solved by a special form of Gauss elimination that preserves the sparseness.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of B2PFD/DB2PFD. The reference is:

```
CALL B2PFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, N,
 NLEFT, NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, NINIT,
 TINIT, YINIT, LDYINI, LINEAR, PRINT, MXGRID, NFINAL,
 TFINAL, YFINAL, LDYFIN, ERREST, RWORK, IWORK)
```

The additional arguments are as follows:

**RWORK** — Floating-point work array of size  $N(3N * MXGRID + 4N + 1) + MXGRID * (7N + 2)$ .

**IWORK** — Integer work array of size  $2N * MXGRID + N + MXGRID$ .

2.    Informational errors

| Type | Code |                                                               |
|------|------|---------------------------------------------------------------|
| 4    | 1    | More than MXGRID grid points are needed to solve the problem. |
| 4    | 2    | Newton's method diverged.                                     |
| 3    | 3    | Newton's method reached roundoff error level.                 |

- If the value of `PISTEP` is greater than zero, then the routine `BVPFD` assumes that the user has embedded the problem into a one-parameter family of problems:

$$y' = y'(t, y, p)$$

$$h(y_{left}, y_{right}, p) = 0$$

such that for  $p = 0$  the problem is simple. For  $p = 1$ , the original problem is recovered. The routine `BVPFD` automatically attempts to increment from  $p = 0$  to  $p = 1$ . The value `PISTEP` is the beginning increment used in this continuation. The increment will usually be changed by routine `BVPFD`, but an arbitrary minimum of 0.01 is imposed.

- The vectors `TINIT` and `TFINAL` may be the same.
- The arrays `YINIT` and `YFINAL` may be the same.

### Example 1

This example solves the third-order linear equation

$$y''' - 2y'' + y' - y = \sin t$$

subject to the boundary conditions  $y(0) = y(2\pi)$  and  $y'(0) = y'(2\pi) = 1$ . (Its solution is  $y = \sin t$ .)

To use `BVPFD`, the problem is reduced to a system of first-order equations by defining

$y_1 = y$ ,  $y_2 = y'$  and  $y_3 = y''$ . The resulting system is

$$\begin{aligned} y_1' &= y_2 & y_2(0) - 1 &= 0 \\ y_2' &= y_3 & y_1(0) - y_1(2\pi) &= 0 \\ y_3' &= 2y_3 - y_2 + y_1 + \sin t & y_2(2\pi) - 1 &= 0 \end{aligned}$$

Note that there is one boundary condition at the left endpoint  $t = 0$  and one boundary condition coupling the left and right endpoints. The final boundary condition is at the right endpoint. The total number of boundary conditions must be the same as the number of equations (in this case 3).

Note that since the parameter  $p$  is not used in the call to `BVPFD`, the routines `FCNPEQ` and `FCNPBC` are not needed. Therefore, in the call to `BVPFD`, `FCNEQN` and `FCNBC` were used in place of `FCNPEQ` and `FCNPBC`.

```

USE BVPFD_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE

!
! SPECIFICATIONS FOR PARAMETERS
INTEGER LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
PARAMETER (MXGRID=45, NEQNS=3, NINIT=10, LDYFIN=NEQNS, &
 LDYINI=NEQNS)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, J, NCUPBC, NFINAL, NLEFT, NOUT
REAL ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TINIT(NINIT), &

```

```

 TLEFT, TOL, TRIGHT, YFINAL(LDYFIN,MXGRID), &
 YINIT(LDYINI,NINIT)
LOGICAL LINEAR, PRINT
!
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC FLOAT
REAL FLOAT
!
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCNBC, FCNEQN, FCNJAC
!
! Set parameters
NLEFT = 1
NCUPBC = 1
TOL = .001
TLEFT = 0.0
TRIGHT = CONST('PI')
TRIGHT = 2.0*TRIGHT
PISTEP = 0.0
PRINT = .FALSE.
LINEAR = .TRUE.
!
! Define TINIT
DO 10 I=1, NINIT
TINIT(I) = TLEFT + (I-1)*(TRIGHT-TLEFT)/FLOAT(NINIT-1)
10 CONTINUE
!
! Set YINIT to zero
YINIT = 0.0E0
!
! Solve problem
CALL BVPPD (FCNEQN, FCNJAC, FCNBC, FCNEQN, FCNBC, NLEFT, &
 NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT, &
 YINIT, LINEAR, MXGRID, NFINAL, &
 TFINAL, YFINAL, ERREST)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)
WRITE (NOUT,99998) (I,TFINAL(I), (YFINAL(J,I),J=1,NEQNS),I=1, &
 NFINAL)
WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, 'I', 7X, 'T', 14X, 'Y1', 13X, 'Y2', 13X, 'Y3')
99998 FORMAT (I5, 1P4E15.6)
99999 FORMAT (' Error estimates', 4X, 1P3E15.6)
END
SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDX)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYDX(NEQNS)
!
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC SIN
REAL SIN
!
! Define PDE
DYDX(1) = Y(2)
DYDX(2) = Y(3)
DYDX(3) = 2.0*Y(3) - Y(2) + Y(1) + SIN(T)
RETURN
END
SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)

```

```

! Define d(DYDX)/dY
 DYPDY(1,1) = 0.0
 DYPDY(1,2) = 1.0
 DYPDY(1,3) = 0.0
 DYPDY(2,1) = 0.0
 DYPDY(2,2) = 0.0
 DYPDY(2,3) = 1.0
 DYPDY(3,1) = 1.0
 DYPDY(3,2) = -1.0
 DYPDY(3,3) = 2.0
 RETURN
 END
 SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER NEQNS
 REAL P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
! Define boundary conditions
 F(1) = YLEFT(2) - 1.0
 F(2) = YLEFT(1) - YRIGHT(1)
 F(3) = YRIGHT(2) - 1.0
 RETURN
 END

```

## Output

| I               | T            | Y1            | Y2            | Y3            |
|-----------------|--------------|---------------|---------------|---------------|
| 1               | 0.000000E+00 | -1.123191E-04 | 1.000000E+00  | 6.242319E-05  |
| 2               | 3.490659E-01 | 3.419107E-01  | 9.397087E-01  | -3.419580E-01 |
| 3               | 6.981317E-01 | 6.426908E-01  | 7.660918E-01  | -6.427230E-01 |
| 4               | 1.396263E+00 | 9.847531E-01  | 1.737333E-01  | -9.847453E-01 |
| 5               | 2.094395E+00 | 8.660529E-01  | -4.998747E-01 | -8.660057E-01 |
| 6               | 2.792527E+00 | 3.421830E-01  | -9.395474E-01 | -3.420648E-01 |
| 7               | 3.490659E+00 | -3.417234E-01 | -9.396111E-01 | 3.418948E-01  |
| 8               | 4.188790E+00 | -8.656880E-01 | -5.000588E-01 | 8.658733E-01  |
| 9               | 4.886922E+00 | -9.845794E-01 | 1.734571E-01  | 9.847518E-01  |
| 10              | 5.585054E+00 | -6.427721E-01 | 7.658258E-01  | 6.429526E-01  |
| 11              | 5.934120E+00 | -3.420819E-01 | 9.395434E-01  | 3.423986E-01  |
| 12              | 6.283185E+00 | -1.123186E-04 | 1.000000E+00  | 6.743190E-04  |
| Error estimates |              | 2.840430E-04  | 1.792939E-04  | 5.588399E-04  |

## Additional Examples

### Example 2

In this example, the following nonlinear problem is solved:

$$y'' - y^3 + (1 + \sin^2 t) \sin t = 0$$

with  $y(0) = y(\pi) = 0$ . Its solution is  $y = \sin t$ . As in Example 1, this equation is reduced to a system of first-order differential equations by defining  $y_1 = y$  and  $y_2 = y'$ . The resulting system is

$$y_1' = y_2 \quad y_1(0) = 0$$

$$y_2' = y_1^3 - (1 + \sin^2 t) \sin t \quad y_1(\pi) = 0$$

In this problem, there is one boundary condition at the left endpoint and one at the right endpoint; there are no coupled boundary conditions.

Note that since the parameter  $p$  is not used, in the call to BVFPD the routines FCNPEQ and FCNPBC are not needed. Therefore, in the call to BVFPD, FCNEQN and FCNBC were used in place of FCNPEQ and FCNPBC.

```

USE BVFPD_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE

!
! SPECIFICATIONS FOR PARAMETERS
INTEGER LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
PARAMETER (MXGRID=45, NEQNS=2, NINIT=12, LDYFIN=NEQNS, &
 LDYINI=NEQNS)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, J, NCUPBC, NFINAL, NLEFT, NOUT
REAL ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TINIT(NINIT), &
 TLEFT, TOL, TRIGHT, YFINAL(LDYFIN,MXGRID), &
 YINIT(LDYINI,NINIT)
LOGICAL LINEAR, PRINT
!
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC FLOAT
REAL FLOAT
!
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCNBC, FCNEQN, FCNJAC
!
! Set parameters
NLEFT = 1
NCUPBC = 0
TOL = .001
TLEFT = 0.0
TRIGHT = CONST('PI')
PISTEP = 0.0
PRINT = .FALSE.
LINEAR = .FALSE.
!
! Define TINIT and YINIT
DO 10 I=1, NINIT
 TINIT(I) = TLEFT + (I-1)*(TRIGHT-TLEFT)/FLOAT(NINIT-1)
 YINIT(1,I) = 0.4*(TINIT(I)-TLEFT)*(TRIGHT-TINIT(I))
 YINIT(2,I) = 0.4*(TLEFT-TINIT(I)+TRIGHT-TINIT(I))
10 CONTINUE
!
! Solve problem
CALL BVFPD (FCNEQN, FCNJAC, FCNBC, FCNEQN, FCNBC, NLEFT, &
 NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, TINIT, &
 YINIT, LINEAR, MXGRID, NFINAL, &
 TFINAL, YFINAL, ERREST)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)
WRITE (NOUT,99998) (I,TFINAL(I), (YFINAL(J,I), J=1,NEQNS), I=1, &

```

```

 NFINAL)
WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, 'I', 7X, 'T', 14X, 'Y1', 13X, 'Y2')
99998 FORMAT (I5, 1P3E15.6)
99999 FORMAT (' Error estimates', 4X, 1P2E15.6)
END
SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDT)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYDT(NEQNS)
!
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC SIN
REAL SIN
!
! Define PDE
DYDT(1) = Y(2)
DYDT(2) = Y(1)**3 - SIN(T)*(1.0+SIN(T)**2)
RETURN
END
SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!
! Define d(DYDT)/dY
DYPDY(1,1) = 0.0
DYPDY(1,2) = 1.0
DYPDY(2,1) = 3.0*Y(1)**2
DYPDY(2,2) = 0.0
RETURN
END
SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!
! Define boundary conditions
F(1) = YLEFT(1)
F(2) = YRIGHT(1)
RETURN
END

```

## Output

| I               | T            | Y1           | Y2            |
|-----------------|--------------|--------------|---------------|
| 1               | 0.000000E+00 | 0.000000E+00 | 9.999277E-01  |
| 2               | 2.855994E-01 | 2.817682E-01 | 9.594315E-01  |
| 3               | 5.711987E-01 | 5.406458E-01 | 8.412407E-01  |
| 4               | 8.567980E-01 | 7.557380E-01 | 6.548904E-01  |
| 5               | 1.142397E+00 | 9.096186E-01 | 4.154530E-01  |
| 6               | 1.427997E+00 | 9.898143E-01 | 1.423307E-01  |
| 7               | 1.713596E+00 | 9.898143E-01 | -1.423307E-01 |
| 8               | 1.999195E+00 | 9.096185E-01 | -4.154530E-01 |
| 9               | 2.284795E+00 | 7.557380E-01 | -6.548903E-01 |
| 10              | 2.570394E+00 | 5.406460E-01 | -8.412405E-01 |
| 11              | 2.855994E+00 | 2.817683E-01 | -9.594313E-01 |
| 12              | 3.141593E+00 | 0.000000E+00 | -9.999274E-01 |
| Error estimates |              | 3.906105E-05 | 7.124186E-05  |

### Example 3

In this example, the following nonlinear problem is solved:

$$y'' - y^3 = \frac{40}{9} \left( t - \frac{1}{2} \right)^{2/3} - \left( t - \frac{1}{2} \right)^8$$

with  $y(0) = y(1) = \pi/2$ . As in the previous examples, this equation is reduced to a system of first-order differential equations by defining  $y_1 = y$  and  $y_2 = y'$ . The resulting system is

$$\begin{aligned} y_1' &= y_2 & y_1(0) &= \pi/2 \\ y_2' &= y_1^3 - \frac{40}{9} \left( t - \frac{1}{2} \right)^{2/3} + \left( t - \frac{1}{2} \right)^8 & y_1(1) &= \pi/2 \end{aligned}$$

The problem is embedded in a family of problems by introducing the parameter  $p$  and by changing the second differential equation to

$$y_2' = py_1^3 + \frac{40}{9} \left( t - \frac{1}{2} \right)^{2/3} - \left( t - \frac{1}{2} \right)^8$$

At  $p = 0$ , the problem is linear; and at  $p = 1$ , the original problem is recovered. The derivatives  $\partial y' / \partial p$  must now be specified in the subroutine FCNPEQ. The derivatives  $\partial f / \partial p$  are zero in FCNPBC.

```

 USE BVFPD_INT
 USE UMACH_INT

 IMPLICIT NONE

! SPECIFICATIONS FOR PARAMETERS
 INTEGER LDYFIN, LDYINI, MXGRID, NEQNS, NINIT
 PARAMETER (MXGRID=45, NEQNS=2, NINIT=5, LDYFIN=NEQNS, &
 LDYINI=NEQNS)

! SPECIFICATIONS FOR LOCAL VARIABLES
 INTEGER NCUPBC, NFINAL, NLEFT, NOUT
 REAL ERREST(NEQNS), PISTEP, TFINAL(MXGRID), TLEFT, TOL, &
 XRIGHT, YFINAL(LDYFIN,MXGRID)
 LOGICAL LINEAR, PRINT

! SPECIFICATIONS FOR SAVE VARIABLES
 INTEGER I, J
 REAL TINIT(NINIT), YINIT(LDYINI,NINIT)
 SAVE I, J, TINIT, YINIT

! SPECIFICATIONS FOR FUNCTIONS
 EXTERNAL FCNBC, FCNEQN, FCNJAC, FCNPBC, FCNPEQ

!
 DATA TINIT/0.0, 0.4, 0.5, 0.6, 1.0/
 DATA ((YINIT(I,J),J=1,NINIT),I=1,NEQNS)/0.15749, 0.00215, 0.0, &
 0.00215, 0.15749, -0.83995, -0.05745, 0.0, 0.05745, 0.83995/

! Set parameters
 NLEFT = 1
 NCUPBC = 0
 TOL = .001
 TLEFT = 0.0

```

```

XRIGHT = 1.0
PISTEP = 0.1
PRINT = .FALSE.
LINEAR = .FALSE.
!
CALL BVFPD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, NLEFT, &
 NCUPBC, TLEFT, XRIGHT, PISTEP, TOL, TINIT, &
 YINIT, LINEAR, MXGRID, NFINAL,TFINAL, YFINAL, ERREST)
!
 Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)
WRITE (NOUT,99998) (I,TFINAL(I), (YFINAL(J,I),J=1,NEQNS),I=1, &
 NFINAL)
WRITE (NOUT,99999) (ERREST(J),J=1,NEQNS)
99997 FORMAT (4X, 'I', 7X, 'T', 14X, 'Y1', 13X, 'Y2')
99998 FORMAT (I5, 1P3E15.6)
99999 FORMAT (' Error estimates', 4X, 1P2E15.6)
END
SUBROUTINE FCNEQN (NEQNS, T, Y, P, DYDT)
!
 SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYDT(NEQNS)
!
 Define PDE
DYDT(1) = Y(2)
DYDT(2) = P*Y(1)**3 + 40./9.*((T-0.5)**2)**(1./3.) - (T-0.5)**8
RETURN
END
SUBROUTINE FCNJAC (NEQNS, T, Y, P, DYPDY)
!
 SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
!
 Define d(DYDT)/dY
DYPDY(1,1) = 0.0
DYPDY(1,2) = 1.0
DYPDY(2,1) = P*3.*Y(1)**2
DYPDY(2,2) = 0.0
RETURN
END
SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
USE CONST_INT
!
 SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
!
 SPECIFICATIONS FOR LOCAL VARIABLES
REAL PI
!
 Define boundary conditions
PI = CONST('PI')
F(1) = YLEFT(1) - PI/2.0
F(2) = YRIGHT(1) - PI/2.0
RETURN
END
SUBROUTINE FCNPEQ (NEQNS, T, Y, P, DYPDP)
!
 SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL T, P, Y(NEQNS), DYPDP(NEQNS)
!
 Define d(DYDT)/dP

```

```

DYPDP(1) = 0.0
DYPDP(2) = Y(1)**3
RETURN
END
SUBROUTINE FCNPBC (NEQNS, YLEFT, YRIGHT, P, DFDP)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER NEQNS
 REAL P, YLEFT(NEQNS), YRIGHT(NEQNS), DFDP(NEQNS)
! SPECIFICATIONS FOR SUBROUTINES
 EXTERNAL SSET
! Define dF/dP
 CALL SSET (NEQNS, 0.0, DFDP, 1)
 RETURN
 END

```

## Output

| I               | T            | Y1           | Y2            |
|-----------------|--------------|--------------|---------------|
| 1               | 0.000000E+00 | 1.570796E+00 | -1.949336E+00 |
| 2               | 4.444445E-02 | 1.490495E+00 | -1.669567E+00 |
| 3               | 8.888889E-02 | 1.421951E+00 | -1.419465E+00 |
| 4               | 1.333333E-01 | 1.363953E+00 | -1.194307E+00 |
| 5               | 2.000000E-01 | 1.294526E+00 | -8.958461E-01 |
| 6               | 2.666667E-01 | 1.243628E+00 | -6.373191E-01 |
| 7               | 3.333334E-01 | 1.208785E+00 | -4.135206E-01 |
| 8               | 4.000000E-01 | 1.187783E+00 | -2.219351E-01 |
| 9               | 4.250000E-01 | 1.183038E+00 | -1.584200E-01 |
| 10              | 4.500000E-01 | 1.179822E+00 | -9.973146E-02 |
| 11              | 4.625000E-01 | 1.178748E+00 | -7.233893E-02 |
| 12              | 4.750000E-01 | 1.178007E+00 | -4.638248E-02 |
| 13              | 4.812500E-01 | 1.177756E+00 | -3.399763E-02 |
| 14              | 4.875000E-01 | 1.177582E+00 | -2.205547E-02 |
| 15              | 4.937500E-01 | 1.177480E+00 | -1.061177E-02 |
| 16              | 5.000000E-01 | 1.177447E+00 | -1.479182E-07 |
| 17              | 5.062500E-01 | 1.177480E+00 | 1.061153E-02  |
| 18              | 5.125000E-01 | 1.177582E+00 | 2.205518E-02  |
| 19              | 5.187500E-01 | 1.177756E+00 | 3.399727E-02  |
| 20              | 5.250000E-01 | 1.178007E+00 | 4.638219E-02  |
| 21              | 5.375000E-01 | 1.178748E+00 | 7.233876E-02  |
| 22              | 5.500000E-01 | 1.179822E+00 | 9.973124E-02  |
| 23              | 5.750000E-01 | 1.183038E+00 | 1.584199E-01  |
| 24              | 6.000000E-01 | 1.187783E+00 | 2.219350E-01  |
| 25              | 6.666667E-01 | 1.208786E+00 | 4.135205E-01  |
| 26              | 7.333333E-01 | 1.243628E+00 | 6.373190E-01  |
| 27              | 8.000000E-01 | 1.294526E+00 | 8.958461E-01  |
| 28              | 8.666667E-01 | 1.363953E+00 | 1.194307E+00  |
| 29              | 9.111111E-01 | 1.421951E+00 | 1.419465E+00  |
| 30              | 9.555556E-01 | 1.490495E+00 | 1.669566E+00  |
| 31              | 1.000000E+00 | 1.570796E+00 | 1.949336E+00  |
| Error estimates |              | 3.448358E-06 | 5.549869E-05  |

---

# BVPMS



Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.

## Required Arguments

**FCNEQN** — User-supplied subroutine to evaluate derivatives. The usage is  
CALL FCNEQN (NEQNS, T, Y, P, DYDT), where

NEQNS – Number of equations. (Input)

T – Independent variable,  $t$ . (Input)

Y – Array of length NEQNS containing the dependent variable. (Input)

P – Continuation parameter used in solving highly nonlinear problems. (Input)  
See Comment 4.

DYDT – Array of length NEQNS containing  $y'$  at T. (Output)

The name FCNEQN must be declared EXTERNAL in the calling program.

**FCNJAC** — User-supplied subroutine to evaluate the Jacobian. The usage is  
CALL FCNJAC (NEQNS, T, Y, P, DYPDY), where

NEQNS – Number of equations. (Input)

T – Independent variable. (Input)

Y – Array of length NEQNS containing the dependent variable. (Input)

P – Continuation parameter used in solving highly nonlinear problems. (Input)  
See Comment 4.

DYPDY – Array of size NEQNS by NEQNS containing the Jacobian. (Output)

The entry DYPDY( $i, j$ ) contains the partial derivative  $\partial f_i / \partial y_j$  evaluated at  $(t, y)$ .

The name FCNJAC must be declared EXTERNAL in the calling program.

**FCNBC** — User-supplied subroutine to evaluate the boundary conditions. The usage is  
CALL FCNBC (NEQNS, YLEFT, YRIGHT, P, H), where

NEQNS – Number of equations. (Input)

YLEFT – Array of length NEQNS containing the values of Y at TLEFT. (Input)

YRIGHT – Array of length NEQNS containing the values of Y at TRIGHT. (Input)

**P** — Continuation parameter used in solving highly nonlinear problems. (Input)  
See Comment 4.

**H** — Array of length **NEQNS** containing the boundary function values. (Output)  
The computed solution satisfies (within **BTOL**) the conditions  $h_i = 0$ ,  $i = 1$ ,  
..., **NEQNS**.

The name **FCNBC** must be declared **EXTERNAL** in the calling program.

**TLEFT** — The left endpoint. (Input)

**TRIGHT** — The right endpoint. (Input)

**NMAX** — Maximum number of shooting points to be allowed. (Input)  
If **NINIT** is nonzero, then **NMAX** must equal **NINIT**. It must be at least 2.

**NFINAL** — Number of final shooting points, including the endpoints. (Output)

**TFINAL** — Vector of length **NMAX** containing the final shooting points. (Output)  
Only the first **NFINAL** points are significant.

**YFINAL** — Array of size **NEQNS** by **NMAX** containing the values of **Y** at the points in **TFINAL**.  
(Output)

## Optional Arguments

**NEQNS** — Number of differential equations. (Input)

**DTOL** — Differential equation error tolerance. (Input)  
An attempt is made to control the local error in such a way that the global error is  
proportional to **DTOL**.  
Default: **DTOL** = 1.0e-4.

**BTOL** — Boundary condition error tolerance. (Input)  
The computed solution satisfies the boundary conditions, within **BTOL** tolerance.  
Default: **BTOL** = 1.0e-4.

**MAXIT** — Maximum number of Newton iterations allowed. (Input)  
Iteration stops if convergence is achieved sooner. Suggested values are **MAXIT** = 2 for  
linear problems and **MAXIT** = 9 for nonlinear problems.  
Default: **MAXIT** = 9.

**NINIT** — Number of shooting points supplied by the user. (Input)  
It may be 0. A suggested value for the number of shooting points is 10.  
Default: **NINIT** = 0.

**TINIT** — Vector of length **NINIT** containing the shooting points supplied by the user.  
(Input)  
If **NINIT** = 0, then **TINIT** is not referenced and the routine chooses all of the shooting  
points. This automatic selection of shooting points may be expensive and should only

be used for linear problems. If `NINIT` is nonzero, then the points must be an increasing sequence with `TINIT(1) = TLEFT` and `TINIT(NINIT) = TRIGHT`. By default, `TINIT` is not used.

**YINIT** — Array of size `NEQNS` by `NINIT` containing an initial guess for the values of `Y` at the points in `TINIT`. (Input)  
`YINIT` is not referenced if `NINIT = 0`. By default, `YINIT` is not used.

**LDYINI** — Leading dimension of `YINIT` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDYINI = size (YINIT ,1)`.

**LDYFIN** — Leading dimension of `YFINAL` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDYFIN = size (YFINAL,1)`.

## FORTRAN 90 Interface

Generic:     `CALL BVPMS (FCNEQN, FCNJAC, FCNBC, TLEFT, TRIGHT, NMAX, NFINAL, TFINAL, YFINAL [, ...])`

Specific:    The specific interface names are `S_BVPMS` and `D_BVPMS`.

## FORTRAN 77 Interface

Single:     `CALL BVPMS (FCNEQN, FCNJAC, FCNBC, NEQNS, TLEFT, TRIGHT, DTOL, BTOL, MAXIT, NINIT, TINIT, YINIT, LDYINI, NMAX, NFINAL, TFINAL, YFINAL, LDYFIN)`

Double:     The double precision name is `DBVPMS`.

## Description

Define  $N = \text{NEQNS}$ ,  $M = \text{NFINAL}$ ,  $t_a = \text{TLEFT}$  and  $t_b = \text{TRIGHT}$ . The routine `BVPMS` uses a multiple-shooting technique to solve the differential equation system  $y' = f(t, y)$  with boundary conditions of the form

$$h_k(y_1(t_a), \dots, y_N(t_a), y_1(t_b), \dots, y_N(t_b)) = 0 \quad \text{for } k = 1, \dots, N$$

A modified version of `IVPRK` is used to compute the initial-value problem at each “shot.” If there are  $M$  shooting points (including the endpoints  $t_a$  and  $t_b$ ), then a system of  $NM$  simultaneous nonlinear equations must be solved. Newton’s method is used to solve this system, which has a Jacobian matrix with a “periodic band” structure. Evaluation of the  $NM$  functions and the  $NM \times NM$  (almost banded) Jacobian for one iteration of Newton’s method is accomplished in one pass from  $t_a$  to  $t_b$  of the modified `IVPRK`, operating on a system of  $N(N + 1)$  differential equations. For most problems, the total amount of work should not be highly dependent on  $M$ . Multiple shooting avoids many of the serious ill-conditioning problems that plague simple shooting methods. For more details on the algorithm, see Sewell (1982).

The boundary functions should be scaled so that all components  $h_k$  are of comparable magnitude since the absolute error in each is controlled.

## Comments

1. Workspace may be explicitly provided, if desired, by use of B2PMS/DB2PMS. The reference is:

```
CALL B2PMS (FCNEQN, FCNJAC, FCNBC, NEQNS, TLEFT, TRIGHT,
 DTOL, BTOL, MAXIT, NINIT, TINIT, YINIT, LDYINI, NMAX,
 NFINAL, TFINAL, YFINAL, LDYFIN, WORK, IWK)
```

The additional arguments are as follows:

**WORK** — Work array of length  $NEQNS * (NEQNS + 1) (NMAX + 12) + NEQNS + 30$ .

**IWK** — Work array of length  $NEQNS$ .

2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                                                                                                                                                        |
|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | 5 | Convergence has been achieved; but to get acceptably accurate approximations to $y(t)$ , it is often necessary to start an initial-value solver, for example <a href="#">IVPRK</a> , at the nearest $TFINAL(i)$ point to $t$ with $t \geq TFINAL(i)$ . The vectors $YFINAL(j, i), j = 1, \dots, NEQNS$ are used as the initial values. |
| 4 | 1 | The initial-value integrator failed. Relax the tolerance $DTOL$ or see Comment 3.                                                                                                                                                                                                                                                      |
| 4 | 2 | More than $NMAX$ shooting points are needed for stability.                                                                                                                                                                                                                                                                             |
| 4 | 3 | Newton's iteration did not converge in $MAXIT$ iterations. If the problem is linear, do an extra iteration. If this error still occurs, check that the routine <code>FCNJAC</code> is giving the correct derivatives. If this does not fix the problem, see Comment 3.                                                                 |
| 4 | 4 | Linear-equation solver failed. The problem may not have a unique solution, or the problem may be highly nonlinear. In the latter case, see Comment 3.                                                                                                                                                                                  |

3. Many linear problems will be successfully solved using program-selected shooting points. Nonlinear problems may require user effort and input data. If the routine fails, then increase  $NMAX$  or parameterize the problem. With many shooting points the program essentially uses a finite-difference method, which has less trouble with nonlinearities than shooting methods. After a certain point, however, increasing the number of points will no longer help convergence. To parameterize the problem, see Comment 4.
4. If the problem to be solved is highly nonlinear, then to obtain convergence it may be necessary to embed the problem into a one-parameter family of boundary value

problems,  $y' = f(t, y, p)$ ,  $h(y(t_a, t_b, p)) = 0$  such that for  $p = 0$ , the problem is simple, e.g., linear; and for  $p = 1$ , the stated problem is solved. The routine BVPMS/DBVPMS automatically moves the parameter from  $p = 0$  toward  $p = 1$ .

5. This routine is not recommended for stiff systems of differential equations.

### Example

The differential equations that model an elastic beam are (see Washizu 1968, pages 142–143):

$$\begin{aligned} \mathbf{M}_{xx} - \frac{\mathbf{N}\mathbf{M}}{\mathbf{EI}} + \mathbf{L}(x) &= 0 \\ \mathbf{EI}\mathbf{W}_{xx} + \mathbf{M} &= 0 \\ \mathbf{EA}_0(\mathbf{U}_x + \mathbf{W}_x^2/2) - \mathbf{N} &= 0 \\ \mathbf{N}_x &= 0 \end{aligned}$$

where  $\mathbf{U}$  is the axial displacement,  $\mathbf{W}$  is the transverse displacement,  $\mathbf{N}$  is the axial force,  $\mathbf{M}$  is the bending moment,  $\mathbf{E}$  is the elastic modulus,  $\mathbf{I}$  is the moment of inertia,  $\mathbf{A}_0$  is the cross-sectional area, and  $\mathbf{L}(x)$  is the transverse load.

Assume we have a clamped cylindrical beam of radius 0.1in, a length of 10in, and an elastic modulus  $\mathbf{E} = 10.6 \times 10^6$  lb/in<sup>2</sup>. Then,  $\mathbf{I} = 0.784 \times 10^{-4}$ , and  $\mathbf{A}_0 = \pi 10^{-2}$  in<sup>2</sup>, and the boundary conditions are  $\mathbf{U} = \mathbf{W} = \mathbf{W}_x = 0$  at each end. If we let  $y_1 = \mathbf{U}$ ,  $y_2 = \mathbf{N}/\mathbf{EA}_0$ ,  $y_3 = \mathbf{W}$ ,  $y_4 = \mathbf{W}_x$ ,  $y_5 = \mathbf{M}/\mathbf{EI}$ , and  $y_6 = \mathbf{M}_x/\mathbf{EI}$ , then the above nonlinear equations can be written as a system of six first-order equations.

$$\begin{aligned} y_1' &= y_2 - \frac{y_4^2}{2} \\ y_2' &= 0 \\ y_3' &= y_4 \\ y_4' &= -y_5 \\ y_5' &= y_6 \\ y_6' &= \frac{\mathbf{A}_0 y_2 y_5}{\mathbf{I}} - \frac{\mathbf{L}(x)}{\mathbf{EI}} \end{aligned}$$

The boundary conditions are  $y_1 = y_3 = y_4 = 0$  at  $x = 0$  and at  $x = 10$ . The loading function is  $\mathbf{L}(x) = -2$ , if  $3 \leq x \leq 7$ , and is zero elsewhere.

The material parameters,  $\mathbf{A}_0 = \mathbf{A0}$ ,  $\mathbf{I} = \mathbf{AI}$ , and  $\mathbf{E}$ , are passed to the evaluation subprograms using the common block PARAM.

```
USE BVPMS_INT
USE UMACH_INT

IMPLICIT NONE
```

```

INTEGER LDY, NEQNS, NMAX
PARAMETER (NEQNS=6, NMAX=21, LDY=NEQNS)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, MAXIT, NFINAL, NINIT, NOUT
REAL TOL, X(NMAX), XLEFT, XRIGHT, Y(LDY,NMAX)
!
! SPECIFICATIONS FOR COMMON /PARAM/
COMMON /PARAM/ A0, A1, E
REAL A0, A1, E
!
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC REAL
REAL REAL
!
! SPECIFICATIONS FOR SUBROUTINES
EXTERNAL FCNBC, FCNEQN, FCNJAC
!
! Set material parameters
A0 = 3.14E-2
A1 = 0.784E-4
E = 10.6E6
!
! Set parameters for BVPMS
XLEFT = 0.0
XRIGHT = 10.0
MAXIT = 19
NINIT = NMAX
Y = 0.0E0
!
! Define the shooting points
DO 10 I=1, NINIT
 X(I) = XLEFT + REAL(I-1)/REAL(NINIT-1)*(XRIGHT-XLEFT)
10 CONTINUE
!
! Solve problem
CALL BVPMS (FCNEQN, FCNJAC, FCNBC, XLEFT, XRIGHT, NMAX, NFINAL, &
 X, Y, MAXIT=MAXIT, NINIT=NINIT, TINIT=X, YINIT=Y)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT, '(26X,A/12X,A,10X,A,7X,A)') 'Displacement', &
 'X', 'Axial', 'Transvers'// &
 'e'
WRITE (NOUT, '(F15.1,1P2E15.3)') (X(I),Y(1,I),Y(3,I),I=1,NFINAL)
END
SUBROUTINE FCNEQN (NEQNS, X, Y, P, DYDX)
!
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL X, P, Y(NEQNS), DYDX(NEQNS)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
REAL FORCE
!
! SPECIFICATIONS FOR COMMON /PARAM/
COMMON /PARAM/ A0, A1, E
REAL A0, A1, E
!
! Define derivatives
FORCE = 0.0
IF (X.GT.3.0 .AND. X.LT.7.0) FORCE = -2.0
DYDX(1) = Y(2) - P*0.5*Y(4)**2
DYDX(2) = 0.0
DYDX(3) = Y(4)
DYDX(4) = -Y(5)
DYDX(5) = Y(6)
DYDX(6) = P*A0*Y(2)*Y(5)/A1 - FORCE/E/A1
RETURN

```

```

END
SUBROUTINE FCNBC (NEQNS, YLEFT, YRIGHT, P, F)
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL P, YLEFT(NEQNS), YRIGHT(NEQNS), F(NEQNS)
! SPECIFICATIONS FOR COMMON /PARAM/
COMMON /PARAM/ A0, A1, E
REAL A0, A1, E
! Define boundary conditions
F(1) = YLEFT(1)
F(2) = YLEFT(3)
F(3) = YLEFT(4)
F(4) = YRIGHT(1)
F(5) = YRIGHT(3)
F(6) = YRIGHT(4)
RETURN
END
SUBROUTINE FCNJAC (NEQNS, X, Y, P, DYPDY)
! SPECIFICATIONS FOR ARGUMENTS
INTEGER NEQNS
REAL X, P, Y(NEQNS), DYPDY(NEQNS,NEQNS)
! SPECIFICATIONS FOR COMMON /PARAM/
COMMON /PARAM/ A0, A1, E
REAL A0, A1, E
! SPECIFICATIONS FOR SUBROUTINES
! Define partials, d(DYDX)/dY
DYPDY = 0.0E0
DYPDY(1,2) = 1.0
DYPDY(1,4) = -P*Y(4)
DYPDY(3,4) = 1.0
DYPDY(4,5) = -1.0
DYPDY(5,6) = 1.0
DYPDY(6,2) = P*Y(5)*A0/A1
DYPDY(6,5) = P*Y(2)*A0/A1
RETURN
END

```

## Output

| X    | Displacement |            |
|------|--------------|------------|
|      | Axial        | Transverse |
| 0.0  | 1.631E-11    | -8.677E-10 |
| 5.0  | 1.914E-05    | -1.273E-03 |
| 10.0 | 2.839E-05    | -4.697E-03 |
| 15.0 | 2.461E-05    | -9.688E-03 |
| 20.0 | 1.008E-05    | -1.567E-02 |
| 25.0 | -9.550E-06   | -2.206E-02 |
| 30.0 | -2.721E-05   | -2.830E-02 |
| 35.0 | -3.644E-05   | -3.382E-02 |
| 40.0 | -3.379E-05   | -3.811E-02 |
| 45.0 | -2.016E-05   | -4.083E-02 |
| 50.0 | -4.414E-08   | -4.176E-02 |
| 55.0 | 2.006E-05    | -4.082E-02 |
| 60.0 | 3.366E-05    | -3.810E-02 |
| 65.0 | 3.627E-05    | -3.380E-02 |

|       |            |            |
|-------|------------|------------|
| 70.0  | 2.702E-05  | -2.828E-02 |
| 75.0  | 9.378E-06  | -2.205E-02 |
| 80.0  | -1.021E-05 | -1.565E-02 |
| 85.0  | -2.468E-05 | -9.679E-03 |
| 90.0  | -2.842E-05 | -4.692E-03 |
| 95.0  | -1.914E-05 | -1.271E-03 |
| 100.0 | 0.000E+00  | 0.000E+00  |

---

## DAESL



Solves a first order differential-algebraic system of equations,  $g(t, y, y') = 0$ , with optional additional constraints and user-defined linear system solver.

---

**Note:** DAESL replaces deprecated routine [DASPG](#).

---

### Required Arguments

**T** — Independent variable,  $t$ . (Input/Output)

Set **T** to the starting value  $t_0$  at the first step. On output, **T** is set to the value to which the integration has advanced. Normally, this new value is **TEND**.

**TEND** — Final value of the independent variable. (Input)

Update this value when re-entering after output with **IDO** = 2.

**IDO** — Flag indicating the state of the computation. (Input/Output)

**IDO**    **State**

- 1      Initial entry
- 2      Normal re-entry after obtaining output
- 3      Release workspace, last call

The user sets **IDO** = 1 on the first call at **T** =  $t_0$ . The routine then sets **IDO** = 2, and this value is used for all but the last entry, which is made with **IDO** = 3.

**Y** — Array of size **NEQ** containing the dependent variable values,  $y$ . (Input/Output)

On input, **Y** must contain initial values. On output, **Y** contains the computed solution at **TEND**.

**YPRIME** — Array of size **NEQ** containing derivative values,  $y'$ . (Input/Output)

This array must contain initial values, but they need not be such that  $g(t, y, y') = 0$  at  $t = t_0$ . See the description of parameter **IYPR** for more information.

**GCN** — User-supplied subroutine to evaluate  $g(t, y, y')$ , and any constraints. Also partial derivative evaluations and optionally linear solving steps occur here. The equations  $g(t, y, y') = 0$  consist of  $NEQ$  differential-algebraic equations of the form.

$$F_i(t, y_1, \dots, y_{NEQ}, y'_1, \dots, y'_{NEQ}) \equiv F_i(t, y, y') = 0, \quad i = 1, \dots, NEQ$$

The routine **GCN** is also used to evaluate the  $NCON$  additional algebraic constraints

$$G_i(t, y_1, \dots, y_{NEQ}) \equiv G_i(t, y) = 0, \quad i = 1, \dots, NCON, \quad NCON \geq 0$$

The usage is `CALL GCN (T, Y, YPRIME, DELTA, D, LDD, IRES [, ...])` where

### Required Arguments

**T** — Integration variable  $t$ . (Input)

**Y** — Array of  $NEQ$  dependent variables,  $y$ . (Input)

**YPRIME** — Array of  $NEQ$  derivative values,  $y'$ . (Input)

**DELTA** — Output array of length  $\text{MAX}(NEQ, NCON)$  containing residuals. See parameter **IRES** for definition. (Input/Output)

**D** — Output array dimensioned  $D(LDD, NEQ)$ , containing partial derivatives. See parameter **IRES** for definition. (Input/Output)

**LDD** — Leading dimension of **D**. (Input)

**IRES** — Flag indicating what is to be calculated in the user routine, **GCN**. (Input/Output)

Note: **IRES** is input only, except when **IRES** = 6. It is input/output when **IRES** = 6. For a detailed description see the table below.

The code calls **GCN** with **IRES** = 0, 1, 2, 3, 4, 5, 6, or 7, defined as follows:

| <b>IRES Value</b> | <b>Explanation</b>                                                                                                                                                                                                                                                        |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0                 | Do initializations, if any are required.                                                                                                                                                                                                                                  |
| 1                 | Compute $\text{DELTA}(i) = F_i(t, y, y')$ , the $i$ -th residual, for $i=1, \dots, NEQ$ .                                                                                                                                                                                 |
| 2                 | (Required only if <b>IUJAC</b> =1 and <b>MATSTR</b> = 0 or 1).<br>Compute $D(i, j) = \frac{\partial F_i(t, y, y')}{\partial y_j}$ , the partial derivative matrix. These are derivatives of $F_i$ with respect to $y_j$ , for $i=1, \dots, NEQ$ and $j = 1, \dots, NEQ$ . |

| IRES Value | Explanation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3          | <p>(Required only if IUJAC = 1 and MATSTR = 0 or 1).</p> <p>Compute <math>D(i, j) = \frac{\partial F_i(t, y, y')}{\partial y'_j}</math>, the partial derivative of <math>F_i</math> with respect to <math>y'_j</math>, for <math>i = 1, \dots, \text{NEQ}</math> and <math>j = 1, \dots, \text{NEQ}</math>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 4          | <p>(Required only if IYPR = 2).</p> <p>Compute <math>\text{DELTA}(i) = \frac{\partial F_i(t, y, y')}{\partial t}</math>, the partial derivative of <math>F_i</math> with respect to <math>t</math>, for <math>i = 1, \dots, \text{NEQ}</math>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 5          | <p>(Required only if NCON &gt; 0).</p> <p>Compute <math>\text{DELTA}(i) = G_i(t, y)</math>, the <math>i</math>-th residual in the additional constraints, for <math>i = 1, \dots, \text{NCON}</math>, and <math>D(i, j) = \frac{\partial G_i(t, y)}{\partial y_j}</math>, the partial derivative of <math>G_i</math> with respect to <math>y_j</math> for <math>i = 1, \dots, \text{NCON}</math> and <math>j = 1, \dots, \text{NEQ}</math>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 6          | <p>(Required only if ISOLVE = 1.)</p> <p>If MATSTR = 2, the user must compute the matrix <math>A = \frac{\partial F}{\partial y} + cj \frac{\partial F}{\partial y'}</math>, where <math>cj = \text{DELTA}(1)</math>, and save this matrix in any user-defined format. This is for later use when IRES = 7. The matrix may also be factored in this step, if desired. The array D is not referenced if MATSTR = 2.</p> <p>If MATSTR = 0 or 1, the A matrix will already be defined and passed to GCN in the array D, which will be in full matrix format if MATSTR = 0, and band matrix format, if MATSTR = 1. The user may factor D in this step, if desired.</p> <p><b>Note:</b> For MATSTR = 0, 1, or 2, the user must set IRES = 0 to signal that A is nonsingular. If A is nearly singular, leave IRES = 6. This results in using a smaller step-size internally.</p> |
| 7          | <p>(Required only if ISOLVE = 1.) The user must solve <math>Ax = b</math>, where <math>b</math> is passed to GCN in the vector DELTA, and <math>x</math> is returned in DELTA. If MATSTR = 2, A is the matrix which was computed and saved at the call with IRES = 6; if MATSTR = 0 or 1, A is passed to GCN in the array D. In either case, the A matrix will remain factored if the user factored it when IRES = 6.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

### Optional Arguments

`FCN_DATA` — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. For a description of this argument see [FCN\\_DATA](#) below. (Input/Output)

`GCN` must be declared `EXTERNAL` in the calling program.

## Optional Arguments

*NEQ* — Number of dependent variables, and number of differential/algebraic equations, not counting any additional constraints. (Input)  
Default: `NEQ = size (Y)`.

*NCON* — Number of additional constraints. (Input)  
Default: `NCON = 0`.

*IUJAC* — Jacobian calculation option. (Input)

| Value | Description                                                                                                                                                                   |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | Calculates using finite difference approximations.                                                                                                                            |
| 1     | User supplies the Jacobian matrices of partial derivatives of $F_i, i = 1, \dots, NEQ$ , in the subroutine <code>GCN</code> , when <code>IRES = 2</code> and <code>3</code> . |

Default: `IUJAC = 0` for `MATSTR = 0` or `1`.  
`IUJAC = 1` for `MATSTR = 2`.

*IYPR* — Initial  $y'$  calculation method. (Input)

| Value | Description                                                                                                                                                                                                                                               |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | The initial input values of <code>YPRIME</code> are already consistent with the input values of <code>Y</code> . That is $g(t, y, y') = 0$ at $t = t_0$ . Any constraints must be satisfied at $t = t_0$ .                                                |
| 1     | Consistent values of <code>YPRIME</code> are calculated by Petzold's original <code>DASSL</code> algorithm.                                                                                                                                               |
| 2     | Consistent values of <code>YPRIME</code> are calculated using a new algorithm [Hanson and Krogh, 2008], which is generally more robust but requires that <code>IUJAC = 1</code> and <code>ISOLVE = 0</code> , and additional derivatives corresponding to |

| Value | Description                             |
|-------|-----------------------------------------|
|       | $IRES = 4$ are to be calculated in GCN. |

Default:  $IYPR = 1$ .

**MATSTR** — Parameter specifying the Jacobian matrix structure (Input)

Set to:

| Value | Description                                                                                                                                                                                                                                                            |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | The Jacobian matrices (whether $IUJAC = 0$ or 1) are to be stored in full storage mode.                                                                                                                                                                                |
| 1     | The Jacobian matrices are to be stored in band storage mode. In this case, if $IUJAC = 1$ , the partial derivative matrices have their entries for row $i$ and column $j$ , stored as array elements $D(i - j + MU + 1, j)$ . This occurs when $IRES = 2$ or 3 in GCN. |
| 2     | A user-defined matrix structure is used (see the documentation for $IRES = 6$ or 7 for more details). If $MATSTR = 2$ , $ISOLVE$ and $IUJAC$ are set to 1 internally.                                                                                                  |

Default:  $MATSTR = 0$ .

**ISOLVE** — Solve method. (Input)

| Value | Description                                                                               |
|-------|-------------------------------------------------------------------------------------------|
| 0     | DAESL solves the linear systems.                                                          |
| 1     | The user wishes to solve the linear system in routine GCN. See parameter GCN for details. |

Default:  $ISOLVE = 0$  for  $MATSTR = 0$  or 1.

$ISOLVE = 1$  for  $MATSTR = 2$ .

**ML** — Number of non-zero diagonals below the main diagonal in the Jacobian matrices when band storage mode is used. (Input)

$ML$  is ignored if  $MATSTR \neq 1$ .

Default:  $ML = NEQ - 1$ .

**MU** — Number of non-zero diagonals above the main diagonal in the Jacobian matrices when band storage mode is used. (Input)

$MU$  is ignored if  $MATSTR \neq 1$ .

Default:  $MU = NEQ - 1$ .

**RTOL** — Relative error tolerance for solver. (Input)

The program attempts to maintain a local error in  $Y(i)$  less than

$RTOL * |Y(i)| + ATOL(i)$ .

Default:  $RTOL = \sqrt{\mathcal{E}}$ , where  $\mathcal{E}$  is machine precision.

**ATOL** — Array of size `NEQ` containing absolute error tolerances. (Input)

See description of `RTOL`.

Default: `ATOL(i) = 0`.

**H0** — Initial stepsize used by the solver. (Input)

If `H0 = 0`, the routine defines the initial stepsize.

Default: `H0 = 0`.

**HMAX** — Maximum stepsize used by the solver. (Input)

If `HMAX=0`, the routine defines the maximum stepsize.

Default: `HMAX = 0`.

**MAXORD** — Maximum order of the backward difference formulas used. (Input).

$1 \leq \text{MAXORD} \leq 5$ .

Default: `MAXORD = 5`.

**MAXSTEPS** — Maximum number of steps taken from `T` to `TEND`. (Input).

Default: `MAXSTEPS = 500`.

**TSTOP** — Integration limit point. (Input)

For efficiency reasons, the code sometimes integrates past `TEND` and interpolates a solution at `TEND`. If a value for `TSTOP` is specified, the code will never integrate past `T=TSTOP`.

Default: No `TSTOP` value is specified.

**FMAG** — Order-of-magnitude estimate. (Input)

`FMAG` is used as an order-of-magnitude estimate of the magnitude of the functions  $F_i$  (see description of `GCN`), for convergence testing, if `IYPR=2`. `FMAG` is ignored if `IYPR=0` or `1`.

Default: `FMAG = 1`.

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied subroutine. (Input/Output)

The derived type, `s_fcn_data`, is defined as:

```
type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type
```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type.

Note that if this optional argument is present then `FCN_DATA` must also be defined as an optional argument in the user-supplied subroutine.

## FORTRAN 90 Interface

Generic:     CALL DAESL (T, TEND, IDO, Y, YPRIME, GCN [, ...])

Specific:    The specific interface names are S\_DAESL and D\_DAESL.

## Description

Routine DAESL finds an approximation to the solution of a system of differential-algebraic equations  $g(t, y, y') = 0$  with given initial data for  $y$  and  $y'$ . The routine uses BDF formulas, which are appropriate for stiff systems. DAESL is based on the code DASSL designed by Linda Petzold [1982], and has been modified by Hanson and Krogh [2008] *Solving Constrained Differential-Algebraic Systems Using Projections* to allow the inclusion of additional constraints, including conservation principles, after each time step. The modified code also provides a more robust algorithm to calculate initial  $y'$  values consistent with the given initial  $y$  values. This occurs when the initial  $y'$  are not known.

A differential-algebraic system of equations is said to have “index 0” if the Jacobian matrix of partial derivatives of the  $F_i$  with respect to the  $y'_j$  is nonsingular. Thus it is possible to solve for all the initial values of  $y'_j$  and put the system in the form of a standard ODE system. If it is possible to reduce the system to a system of index 0 by taking first derivatives of some of the equations, the system has index 1, otherwise the index is greater than 1. See Brenan [1989] for a definition of index. DAESL can generally only solve systems of index 0 or 1; other systems will usually have to be reduced to such a form through differentiation.

## Examples

### Example 1 – Method of Lines PDE Problem

This example solves the partial differential equation  $U_t = U_{xx} + U$ , with initial condition  $U(x, 0) = 1 + x$ , and boundary conditions  $U(0, t) = e^t$ ,  $U(1, t) = 2e^t$  which has exact solution  $U(x, t) = (1 + x)e^t$ . If we approximate the  $U_{xx}$  term using finite differences, where  $x_i = (i - 1)h$ , and  $h = 1/(n - 1)$ , we get:

$$U(x_1, t) = e^t$$

$$U'(x_i, t) = [U(x_{i+1}, t) - 2U(x_i, t) + U(x_{i-1}, t)]/h^2 + U(x_i, t), \quad i = 2, \dots, n - 1$$

$$U(x_n, t) = 2e^t$$

If  $Y_i(t) = U(x_i, t)$ , the first and last equations are algebraic and the others are differential equations, so this is a system of differential-algebraic equations. The system has index=1, since it could be transformed into an ODE system by differentiating the first and last equations. Note that the Jacobian matrices are banded (tridiagonal), with  $ML = MU = 1$ . We use this and specify the option

for dealing with banded matrices in DAESL. The parameter  $h$  and the number of equations is passed to the evaluation routine, GCN, with the optional argument USER\_DATA.

[Link to example source \(daesl\\_ex1.f90\)](#)

```

USE DAESL_INT
USE MP_TYPES
IMPLICIT NONE

!
! NEQ = Number of equations
INTEGER, PARAMETER :: NEQ=101
REAL T, Y(NEQ), YPRIME(NEQ), TEND, X, TRUE, HX, ERRMAX
INTEGER NOUT, IDO, I, NSTEPS
REAL, TARGET :: RPARAM(1)
INTEGER, TARGET :: IPARAM(1)
TYPE (S_FCN_DATA) USER_DATA
EXTERNAL GCN

!
! Pass NEQ, HX to GCN
HX = 1.0 / (NEQ-1)
IPARAM(1) = NEQ
RPARAM(1) = HX
USER_DATA%RDATA=>RPARAM
USER_DATA%IDATA=>IPARAM

!
! Initial values for y, initial guesses for y'
DO I = 1, NEQ
 X = (I-1) * HX
 Y(I) = 1 + X
END DO
YPRIME = 0.0

NSTEPS = 10

!
! Always set IDO=1 on first call
IDO = 1
DO I = 1, NSTEPS

!
! Output solution at T=0.1,0.2,...,1.0
 T = 0.1 * (I-1)
 TEND = 0.1 * I

!
! Set IDO = 3 on last call
 IF (I == NSTEPS) IDO = 3

!
! User-supplied Jacobian matrix (IUJAC=1)
! Banded Jacobian (MATSTR=1)
 CALL DAESL (T, TEND, IDO, Y, YPRIME, GCN, IYPR=1, IUJAC=1, &
 MATSTR=1, ML=1, MU=1, RTOL=1.0E-4, FCN_DATA=USER_DATA)
END DO

ERRMAX = 0.0
DO I = 1, NEQ
 X = (I-1) * HX
 TRUE = (1+X) * EXP(T)
 ERRMAX = MAX(ERRMAX, ABS(Y(I) - TRUE))
END DO

```

```

CALL UMACH(2, NOUT)
WRITE (NOUT, *) ' Max Error at T=1 is ', ERRMAX
END

SUBROUTINE GCN (T, Y, YPRIME, DELTA, D, LDD, IRES, FCN_DATA)
USE MP_TYPES
IMPLICIT NONE

REAL T, Y(*), YPRIME(*), DELTA(*), D(LDD,*), HX
INTEGER IRES, LDD, I, J, NEQ, MU
TYPE (S_FCN_DATA), OPTIONAL, INTENT(INOUT) :: FCN_DATA

NEQ = FCN_DATA%IDATA(1)
HX = FCN_DATA%RDATA(1)
MU = 1

SELECT CASE (IRES)

!
! F_I defined here
CASE(1)
 DELTA(1) = Y(1) - EXP(T)
 DO I = 2, NEQ-1
 DELTA(I) = -YPRIME(I) + (Y(I+1) - 2.0 * Y(I) + Y(I-1)) &
 / HX**2 + Y(I)
 END DO
 DELTA(NEQ) = Y(NEQ) - 2.0 * EXP(T)

!
! D(I-J+MU+1,J) = D(F_I)/D(Y_J)
! in band storage mode
CASE(2)
 D(MU+1,1) = 1.0
 DO I = 2, NEQ-1
 J = I-1
 D(I-J+MU+1, J) = 1.0 / HX**2
 J = I
 D(I-J+MU+1, J) = -2.0 / HX**2 + 1.0
 J = I+1
 D(I-J+MU+1, J) = 1.0 / HX**2
 END DO
 D(MU+1, NEQ) = 1.0

!
! D(I-J+MU+1,J) = D(F_I)/D(YPRIME_J)
CASE(3)
 DO I = 2, NEQ-1
 D(MU+1, I) = -1.0
 END DO

END SELECT
END

```

## Output

```
Max Error at T=1 is 5.6743621E-5
```

## Example 2 – Pendulum Problem

The first-order equations of motion of a point-mass  $m$  suspended on a massless wire of length  $L$  under the influence of gravity,  $mg$ , and wire tension,  $\lambda$ , in Cartesian coordinates  $(p,q)$  are

$$\begin{aligned} p' &= u \\ q' &= v \\ mu' &= -p\lambda \\ mv' &= -q\lambda - mg \\ p^2 + q^2 - L^2 &= 0 \end{aligned}$$

The problem above has an index number equal to 3, thus it cannot be solved with `DAESL` directly. Unfortunately, the fact that the index is greater than 1 is not obvious, but an attempt to solve it will generally produce an error message stating the corrector equation did not converge, or if `IYPR=2` an error message stating that the index appears to be greater than 1 should be issued. The user then differentiates the last equation, which after replacing  $p'$  by  $u$  and  $q'$  by  $v$ , gives  $pu+qv=0$ . This system still has `index=2` (again not obvious, the user discovers this by unsuccessfully trying to solve the new system) and the last equation must be differentiated again, to finally (after appropriate substitutions) give the equation of total energy balance:

$$m(u^2 + v^2) - mgq - L^2\lambda = 0$$

With initial conditions and appropriate definitions of the dependent variables, the system becomes:

$$p(0) = L, q(0) = u(0) = v(0) = \lambda(0) = 0$$

$$y_1 = p$$

$$y_2 = q$$

$$y_3 = u$$

$$y_4 = v$$

$$y_5 = \lambda$$

$$F_1 = y_3 - y_1' = 0$$

$$F_2 = y_4 - y_2' = 0$$

$$F_3 = -y_1y_5 - my_3' = 0$$

$$F_4 = -y_2y_5 - mg - my_4' = 0$$

$$F_5 = m(y_3^2 + y_4^2) - mgy_2 - L^2y_5 = 0$$

The initial conditions correspond to the pendulum starting in a horizontal position.

Since we have replaced the original constraint,  $G_1 = p^2 + q^2 - L^2 = 0$ , which requires that the pendulum length be  $L$ , by differentiating it twice, this constraint is no longer explicitly enforced, and if we try to solve the above system alone (ie, with `NCON=0`), the pendulum length drifts

substantially from  $L$  at larger times. DAESL therefore allows the user to add additional constraints, to be re-enforced after each time step, so we add this original constraint, as well as the intermediate constraint  $G_2 = pu + qv = 0$ . Using these two supplementary constraints, (NCON=2), the pendulum length is constant.

[Link to example source \(daesl\\_ex2.f90\)](#)

```

USE DAESL_INT
USE MP_TYPES
IMPLICIT NONE

! NEQ = Number of equations
! NCON = Number of extra constraints
INTEGER, PARAMETER :: NEQ=5, NCON = 2
REAL, PARAMETER :: MASS=1.0, LENGTH=1.1, GRAVITY=9.806650

REAL T, Y(NEQ), YPRIME(NEQ), TEND, ATOL(NEQ), TOL, LEN
INTEGER NOUT, IDO, I, NSTEPS
REAL, TARGET :: RPARAM(3)
TYPE (S_FCN_DATA) USER_DATA
EXTERNAL GCN

! Pass Mass, Pendulum length and G as parameters
RPARAM(1) = MASS
RPARAM(2) = LENGTH
RPARAM(3) = GRAVITY
USER_DATA%RDATA=>RPARAM

! Initial values for y, guesses for initial y'
Y = 0.0
Y(1) = LENGTH

YPRIME = 0.0
TOL = 1.0E-5
ATOL = TOL

CALL UMACH(2, NOUT)
WRITE (NOUT, 5)

NSTEPS = 5

! Always set IDO=1 on first call
IDO = 1
DO I = 1, NSTEPS
! Output solution at T=10,20,30,40,50
T = 10.0 * (I-1)
TEND = 10.0 * I

! Set IDO = 3 on last call
IF (I.EQ.NSTEPS) IDO = 3

! User-supplied Jacobian matrix (IUJAC=1)
! Use new algorithm to get compatible y'
CALL DAESL (T, TEND, IDO, Y, YPRIME, GCN, NCON=NCON, RTOL=TOL, &
ATOL=ATOL, IYPR=2, IUJAC=1, MAXSTEPS=50000, &
FCN_DATA=USER_DATA)

```

```

! LEN = pendulum length (should be constant)
 LEN = SQRT(Y(1)**2 + Y(2)**2)
 WRITE (NOUT, 10) T, Y(1), Y(2), LEN
 END DO

5 FORMAT (8X, 'T', 14X, 'Y(1)', 11X, 'Y(2)', 11X, 'Length', /)
10 FORMAT (4F15.7)
 END

SUBROUTINE GCN (T, Y, YPRIME, DELTA, D, LDD, IRES, FCN_DATA)
USE MP_TYPES
IMPLICIT NONE

! Simple swinging pendulum problem
REAL T, Y(*), YPRIME(*), DELTA(*), D(LDD,*), MASS, &
 LENGTH, GRAVITY, MG, LSQ
INTEGER IRES, LDD
TYPE (S_FCN_DATA), OPTIONAL, INTENT(INOUT) :: FCN_DATA

MASS = FCN_DATA%RDATA(1)
LENGTH = FCN_DATA%RDATA(2)
GRAVITY = FCN_DATA%RDATA(3)
MG = MASS * GRAVITY
LSQ = LENGTH**2

SELECT CASE (IRES)

! F_I defined here
CASE(1)
 DELTA(1) = Y(3) - YPRIME(1)
 DELTA(2) = Y(4) - YPRIME(2)
 DELTA(3) = -Y(1) * Y(5) - MASS * YPRIME(3)
 DELTA(4) = -Y(2) * Y(5) - MASS * YPRIME(4) - MG
 DELTA(5) = MASS * (Y(3)**2 + Y(4)**2) - MG * Y(2) - LSQ * Y(5)

! D(I,J) = D(F_I)/D(Y_J)
CASE(2)
 D(1, 3) = 1.0
 D(2, 4) = 1.0
 D(3, 1) = -Y(5)
 D(3, 5) = -Y(1)
 D(4, 2) = -Y(5)
 D(4, 5) = -Y(2)
 D(5, 2) = -MG
 D(5, 3) = MASS * 2.0 * Y(3)
 D(5, 4) = MASS * 2.0 * Y(4)
 D(5, 5) = -LSQ

! D(I,J) = D(F_I)/D(YPRIME_J)
CASE(3)
 D(1, 1) = -1.0
 D(2, 2) = -1.0
 D(3, 3) = -MASS
 D(4, 4) = -MASS

```

```

! DELTA(I) = D(F_I)/DT
CASE (4)
 DELTA(1:5) = 0.0

! DELTA(I) = G_I
! D(I,J) = D(G_I)/D(Y_J)
CASE (5)
 DELTA(1) = Y(1)**2 + Y(2)**2 - LSQ
 DELTA(2) = Y(1) * Y(3) + Y(2) * Y(4)
 D(1, 1) = 2.0 * Y(1)
 D(1, 2) = 2.0 * Y(2)
 D(1, 3) = 0.0
 D(1, 4) = 0.0
 D(1, 5) = 0.0
 D(2, 1) = Y(3)
 D(2, 2) = Y(4)
 D(2, 3) = Y(1)
 D(2, 4) = Y(2)
 D(2, 5) = 0.0

END SELECT
END

```

## Output

| T          | Y(1)      | Y(2)       | Length    |
|------------|-----------|------------|-----------|
| 10.0000000 | 1.0998126 | -0.0203017 | 1.0999999 |
| 20.0000000 | 1.0970103 | -0.0810476 | 1.1000000 |
| 30.0000000 | 1.0850314 | -0.1808525 | 1.1000004 |
| 40.0000000 | 1.0535675 | -0.3162208 | 1.1000000 |
| 50.0000000 | 0.9896186 | -0.4802662 | 1.1000003 |

## Example 3 – User Solves Linear System

Consider the system of ordinary differential equations,  $y' = By$ , where  $B$  is the bi-diagonal matrix with  $(-1, -1/2, -1/3, \dots, -1/(n-1), 0)$  on the main diagonal and with 1's along the first sub-diagonal. The initial condition is  $y(0) = (1, 0, 0, \dots, 0)^T$ , and since  $y'(0) = By(0) = (-1, 1, 0, \dots, 0)^T$ ,  $y'(0)$  is also known for this problem.

Since  $B^T v = 0$ , where  $v_i = 1/(i-1)!$ ,  $v$  is an eigenvector of  $B^T$  corresponding to the eigenvalue 0. Thus

$$0 = v^T (y' - By) = v^T y' - (B^T v)^T y = v^T y' = (v^T y)'$$

so  $v^T y(t)$  is constant. Since it has the value  $v^T y(0) = v_1 = 1$  at  $t = 0$ , the constraint  $v^T y(t) = 1$  is satisfied for all  $t$ . This constraint is imposed in this example.

This example also illustrates how the user can solve his/her own linear systems (`MATSTR=2`). Normally, when `IRES=6`, the matrix

$$A = \frac{\partial g}{\partial y} + c_j \frac{\partial g}{\partial y'}$$

is computed, saved and possibly factored, using a sparse matrix factorization routine of the user's choice. Then when `IRES=7`, the system  $Ax = \text{DELTA}$  is solved, using the matrix  $B$  saved and factored earlier, and the solution is returned in `DELTA`. In this case,  $B$  is just a bidiagonal matrix, so there is no need to save or factor  $A$  when `IRES = 6`, since a bi-diagonal system can be solved directly using forward substitution, when `IRES = 7`.

[Link to example source \(daesl\\_ex3.f90\)](#)

```

USE DAESL_INT
USE MP_TYPES
IMPLICIT NONE

! NEQ = Number of equations
INTEGER, PARAMETER :: NEQ=100
REAL T, Y(NEQ), YPRIME(NEQ), TEND, ATOL(NEQ), CON
INTEGER NOUT, IDO, I, NSTEPS
REAL, TARGET :: RPARAM(NEQ)
INTEGER, TARGET :: IPARAM(1)
TYPE (S_FCN_DATA) USER_DATA
EXTERNAL GCN

! Pass NEQ and A^T eigenvector V to GCN
IPARAM(1) = NEQ
RPARAM(1) = 1.0
DO I = 2, NEQ
 RPARAM(I) = RPARAM(I-1) / FLOAT(I-1)
END DO
USER_DATA%RDATA=>RPARAM
USER_DATA%IDATA=>IPARAM

! Initial values for y, y'
Y = 0.0
Y(1) = 1.0

YPRIME = 0.0
YPRIME(1) = -1.0
YPRIME(2) = 1.0

ATOL = 1.0E-4
NSTEPS = 10

! Always set IDO=1 on first call
IDO = 1
DO I = 1, NSTEPS
! Output solution at T=1,2,...,10
 T = I-1
 TEND = I
! Set IDO = 3 on last call
 IF (I == NSTEPS) IDO = 3
! User-defined Jacobian matrix structure (MATSTR=2)
 CALL DAESL (T, TEND, IDO, Y, YPRIME, GCN, IYPR=0, MATSTR=2, &
 NCON=1, ATOL=ATOL, FCN_DATA=USER_DATA)
END DO

! Check if solution satisfies constraint

```

```

CON = 0.0
DO I = 1, NEQ
 CON = CON + RPARAM(I) * Y(I)
END DO

CALL UMACH(2, NOUT)
WRITE (NOUT, *) ' V dot Y =', CON
END

SUBROUTINE GCN (T, Y, YPRIME, DELTA, D, LDD, IRES, FCN_DATA)
USE MP_TYPES
IMPLICIT NONE

REAL T, Y(*), YPRIME(*), DELTA(*), D(LDD,*), CON, CJ
INTEGER IRES, LDD, I, NEQ
SAVE CJ
TYPE (S_FCN_DATA), OPTIONAL, INTENT(INOUT) :: FCN_DATA

NEQ = FCN_DATA%IDATA(1)
SELECT CASE (IRES)

! F_I defined here
CASE(1)
 DELTA(1) = YPRIME(1) + Y(1)
 DO I = 2, NEQ-1
 DELTA(I) = YPRIME(I) - Y(I-1) + Y(I) / FLOAT(I)
 END DO
 DELTA(NEQ) = YPRIME(NEQ) - Y(NEQ-1)

! Constraint is V dot Y = 1
CASE(5)
 CON = -1.0
 DO I = 1, NEQ
 CON = CON + FCN_DATA%RDATA(I) * Y(I)
 D(1,I) = FCN_DATA%RDATA(I)
 END DO
 DELTA(1) = CON

! Normally, compute matrix A = dF/dY + CJ*dF/dY'
! = -B + CJ*I here. Only CJ needs to be saved
! in this case, however, since B is bidiagonal,
! so A*x=DELTA can be solved (IRES=7) without
! saving or factoring B.
CASE(6)
 CJ = DELTA(1)
! If CJ > 0 not close to zero, A is nonsingular,
! so set IRES = 0.
 IF (CJ >= 1.0E-4) IRES = 0

! Solve A*x=DELTA and return x in DELTA.
CASE(7)
 DELTA(1) = DELTA(1) / (1.0 + CJ)
 DO I = 2, NEQ-1
 DELTA(I) = (DELTA(I) + DELTA(I-1)) / (1.0 / FLOAT(I) + CJ)
 END DO
 DELTA(NEQ) = (DELTA(NEQ) + DELTA(NEQ-1)) / CJ

```

```
END SELECT
END
```

## Output

V dot Y = 1.0

---

# DASPG

---

**Deprecated Routine:** DASPG is a deprecated routine and has been replaced with [DAESL](#). Click [here](#) to view the DASPG documentation.

---

---

# IWOAM

Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method.

## Required Arguments

*IDO* — Flag indicating the state of the computation. (Input/Output)

### *IDO* State

- 1 Initial entry input value.
- 2 Normal re-entry input value. On output, if *IDO* = 2 then the integration is finished. If the integrator is called with a new value for *TEND*, the integration continues. If the integrator is called with *TEND* unchanged, an error message is issued.
- 3 Input value to use on final call to release workspace.
- >3 Output value that indicates that a fatal error has occurred.

The initial call is made with *IDO* = 1. The routine then sets *IDO* = 2, and this value is used for all but the last call that is made with *IDO* = 3. This final call is only used to release workspace which was automatically allocated by the initial call with *IDO* = 1.

*FCN* — User-supplied subroutine to evaluate functions.

The usage is `CALL FCN (IDO, T, Y, HIDRVS [, ...])`, where

## Required Arguments

*IDO* — Flag indicating the state of the computation. (Input)

This flag corresponds to the *IDO* argument described above. If *FCN* has complicated subexpressions, which depend only weakly or not at all on *Y*

then these subexpressions need only be computed when `IDO = 1` and their values then reused when `IDO = 2`.

**T** — Independent variable,  $t$ . (Input)

**Y** — Array of length  $k$  containing the dependent variable values,  $y$ , and first derivatives, if any.  $k$  will be the sum of the orders of the equations in the system of equations to solve. (Input)

**HIDRVS** — Array of length  $n = \text{NEQ}$ , where  $n$  is the number of equations in the system to solve, containing the values of the highest order derivatives evaluated at  $(t, y)$ . (Output)

`IWOAM` uses `size(HIDRVS)` to set the default value of `NEQ` unless the optional argument `NEQ` is used.

### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. (Input/Output)

For a detailed description of this argument see [FCN\\_DATA](#) below.

`FCN` must be declared `EXTERNAL` in the calling program.

**T** — Independent variable,  $t$ . (Input/Output)

On input, **T** contains the initial independent variable value. On output, **T** is replaced by `TEND` unless error conditions arise. See `IDO` for details. (Input/Output)

**TEND** — Value of  $t = \text{tend}$  where the solution is required. (Input)

**Y** — Array of length  $k$  containing the dependent variables,  $y(t)$ , and first derivatives, if any. (Input/Output)

$k$  will be the sum of the orders of the equations in the system of equations to solve. On input, **Y** contains the initial values,  $y(t_0)$  and  $y'(t_0)$  (if needed). On output, **Y** contains the approximate solution,  $y(t)$ . For example, for a system of first order equations,  $Y(i)$  is the  $i$ -th dependent variable. For a system of second order equations,  $Y(2i-1)$  is the  $i$ -th dependent variable and  $Y(2i)$  is the derivative of the  $i$ -th dependent variable. For systems of equations in which one or more equations is of order 2, optional argument `KORDER` must be used to denote the order of each equation so that the derivatives in **Y** can be identified. By default it is assumed that all equations are of order 1 and **Y** contains only dependent variables.

**HIDRVS** — Array of length  $n = \text{NEQ}$ , where  $n$  is the number of equations in the system to solve, containing the highest order derivatives at the point **Y**. (Output)

`IWOAM` uses `size(HIDRVS)` to set the default value of `NEQ` unless the optional argument `NEQ` is used.

## Optional Arguments

**NEQ** — Number of differential equations in the system of equations to solve. (Input)

Default:  $NEQ = \text{size}(\text{HIDRVS})$ .

**KORDER** — An array of length  $NEQ$  specifying the orders of the equations in the system of equations to solve. The elements of  $KORDER$  can be 1 or 2.  $KORDER$  must be used with argument  $Y$  to define systems of mixed or higher order. (Input)

Default:  $KORDER = (1,1,1,\dots,1)$ .

**EQNERR** — An array of length  $NEQ$  specifying the error tolerance for each equation. (Input)

Let  $e(i)$  be the error tolerance for equation  $i$ . Then

| Value      | Explanation                                                                                                                                                                                                                                                                            |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $e(i) > 0$ | Implies an absolute error tolerance of $e(i)$ is to be used for equation $i$ .                                                                                                                                                                                                         |
| $e(i) = 0$ | implies no error checking is to be performed for equation $i$ .                                                                                                                                                                                                                        |
| $e(i) < 0$ | Implies a relative error test is to be performed for equation $i$ . In this case, the base error tolerance used will be $ e(i) $ and the relative error factor used will be $(15/16 *  e(i) )$ . Thus the actual absolute error tolerance used will be $ e(i)  * ((15/16 *  e(i) ))$ . |

Default: An absolute error tolerance of 1.E-5 is used for single precision and 1.D-10 for double precision for all equations.

**HINC** — Factor used for increasing the stepsize. (Input)

One should set  $HINC$  such that  $9/8 \leq HINC \leq 4$ .

Default:  $HINC = 2.0$ .

**HDEC** — Factor used for decreasing the stepsize. (Input)

One should set  $HDEC$  such that  $1/4 \leq HDEC \leq 7/8$ .

Default:  $HDEC = 0.5$ .

**HMIN** — Absolute value of the minimum stepsize permitted. (Input)

Default:  $HMIN = 10.0/\text{amach}(2)$  for single precision and  $10.0/\text{dmach}(2)$  for double precision.

**HMAX** — Absolute value of the maximum stepsize permitted. (Input)

Default:  $HMAX = \text{amach}(2)$  for single precision and  $\text{dmach}(2)$  for double precision.

**FCN\_DATA** — A derived type,  $s\_fcn\_data$ , which may be used to pass additional information to/from the user-supplied subroutine. (Input/Output)

The derived type,  $s\_fcn\_data$ , is defined as:

```

type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type

```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type.

Note that if this optional argument is present then `FCN_DATA` must also be defined as an optional argument in the user-supplied subroutine.

## Fortran 90 Interface

Generic: `CALL IVOAM (IDO, FCN, T, TEND, Y, HIDRVS [, ...])`

Specific: The specific interface names are `S_IVOAM` and `D_IVOAM`.

## Description

Routine `IVOAM` is based on the JPL Library routine `SIVA`. `IVOAM` uses a variable order Adams method to solve the initial value problem

$$\left. \begin{aligned} \frac{dy_i}{dt} &= f_i(t, y_1, y_2, \dots, y_{NEQ}) \\ y_i(t_0) &= \eta_i \end{aligned} \right\}, \quad i = 1, 2, \dots, NEQ$$

or more generally

$$z_i^{(d_i)} = f_i(t, y), \quad y(t_0) = \eta_0, \quad i = 1, 2, \dots, NEQ,$$

where  $y$  is the vector

$$\left( z_1, z_1', \dots, z_1^{(d_1-1)}, z_2, \dots, z_{NEQ}^{(d_{NEQ}-1)} \right),$$

$z_i^{(k)}$  is the  $k^{th}$  derivative of  $z_i$  with respect to  $t$ ,  $d_i$  is the order of the  $i^{th}$  differential equation, and  $\eta$  is a vector with the same dimension as  $y$ .

Note that the systems of equations solved by `IVOAM` can be of order one, order two, or mixed order one and two.

## Comments

Informational errors

Type                      Code

- 3            1 The requested error tolerance is too small.
- 3            2 The stepsize has been reduced too rapidly. The integrator is going to do a restart.

### Example 1

In this example a system of two equations of order two is solved.

$$Y_1'' = -Y_1 / \left( (Y_1^2 + Y_2^2)^{\frac{3}{2}} \right)$$

$$Y_2'' = -Y_2 / \left( (Y_1^2 + Y_2^2)^{\frac{3}{2}} \right)$$

The initial conditions are

$$Y_1(0) = 1.0, \quad Y_1'(0) = 0.0, \quad Y_2(0) = 0.0, \quad Y_2'(0) = 1.0$$

Since the system is of order two, optional argument `KORDER` must be used to specify the orders of the equations. Also, because the system is of order two, `Y(1)` contains the first dependent variable, `Y(2)` contains the derivative of the first dependent variable, `Y(3)` contains the second dependent variable, and `Y(4)` contains the derivative of the second dependent variable.

```

USE IVOAM_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER IDO, IEND, NOUT, KORDER(2)
REAL T, TEND, Y(4), HIDRVS(2), DELTA

EXTERNAL FCN

!
! Initialize
IDO = 1
T = 0.0
Y(1) = 1.0
Y(2) = 0.0
Y(3) = 0.0
Y(4) = 1.0
KORDER = 2

!
! Write title
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)

!
! Integrate ODE
IEND = 0
DELTA = CONST('PI')
DELTA = 2.0*DELTA
DO
 IEND = IEND + 1
 TEND = T + DELTA

```

```

IF(TEND .GT. 20.0) TEND = 20.0
CALL IVOAM (IDO, FCN, T, TEND, Y, HIDRVS, KORDER=KORDER)
IF (IEND .LE. 4) THEN
 WRITE (NOUT,99998) T, Y(1), Y(2), HIDRVS(1)
 WRITE (NOUT,99999) Y(3), Y(4), HIDRVS(2)
!
 Finish up
 IF (IEND .EQ. 4) IDO = 3
 CYCLE
END IF
EXIT
END DO
99997 FORMAT (11X, 'T', 12X, 'Y1/Y2', 9X, 'Y1P/Y2P', 7X, 'Y1PP/Y2PP')
99998 FORMAT (4F15.4)
99999 FORMAT (15X, 3F15.4)
END

SUBROUTINE FCN (IDO, T, Y, HIDRVS)
INTEGER IDO
REAL T, Y(*), HIDRVS(*)
REAL TP

TP = Y(1)*Y(1) + Y(3)*Y(3)
TP = 1.0E0/(TP*SQRT(TP))
HIDRVS(1) = -Y(1)*TP
HIDRVS(2) = -Y(3)*TP
RETURN
END

```

## Output

| T       | Y1/Y2  | Y1P/Y2P | Y1PP/Y2PP |
|---------|--------|---------|-----------|
| 6.2832  | 1.0000 | -0.0000 | -1.0000   |
|         | 0.0000 | 1.0000  | 0.0000    |
| 12.5664 | 1.0000 | -0.0000 | -1.0000   |
|         | 0.0000 | 1.0000  | -0.0000   |
| 18.8496 | 1.0000 | -0.0000 | -1.0000   |
|         | 0.0000 | 1.0000  | -0.0000   |
| 20.0000 | 0.4081 | -0.9129 | -0.4081   |
|         | 0.9129 | 0.4081  | -0.9129   |

## Example 2

This contrived example illustrates how to use `IVOAM` to solve a system of equations of mixed order.

The height,  $y(t)$ , of an object of mass  $m$  above the surface of the Earth can be modelled using Newton's second law as:

$$my'' = -mg - ky'$$

or

$$y'' = -g - (k/m)y' \quad (1)$$

where  $-mg$  is the downward force of gravity and  $-ky'$  is the force due to air resistance, in a direction opposing the velocity. If the object is a meteor, the mass,  $m$ , and air resistance,  $k$ , will decrease as the meteor burns up in the atmosphere. The mass is proportional to  $r^3$  ( $r$ =radius) and the air resistance, presumably dependent on the surface area, may be assumed to be proportional to  $r^2$ , so that  $k/m = k_0/r$ . The rate at which the meteor's radius decreases as it burns up may depend on  $r$ , on the velocity  $y'$ , and, since the density of the atmosphere depends on  $y$ , on  $y$  itself. However, we will construct a very simple model where the rate is just proportional to the square of the velocity,

$$r' = -c_0 (y')^2 \quad (2)$$

We solve (1) and (2), with  $k_0 = 0.005$ ,  $c_0 = 10^{-8}$ ,  $g = 9.8$  and initial conditions  $y(0) = 100,000$  meters,  $y'(0) = -1000$  meters/second,  $r(0) = 1$  meter.

```

USE IVOAM_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IDO, IEND, NOUT, KORDER(2)
REAL T, TEND, Y(3), HIDRVS(2), DELTA, EQNERR(2)
EXTERNAL FCN
!
! Initialize
IDO = 1
T = 0.0
Y(1) = 100000.0
Y(2) = -1000.0
Y(3) = 1.0
KORDER(1) = 2
KORDER(2) = 1
EQNERR = .003
!
! Write title
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)
!
! Integrate ODE
IEND = 0
DELTA = 10.0
DO
 IEND = IEND + 1
 TEND = T + DELTA
 IF(TEND .GT. 50.0) TEND = 50.0
 CALL IVOAM (IDO, FCN, T, TEND, Y, HIDRVS, &
 KORDER=KORDER, EQNERR=EQNERR)
 IF (IEND .LE. 5) THEN
 WRITE (NOUT,99998) T, Y(1), Y(2), HIDRVS(1)
 WRITE (NOUT,99999) Y(3), HIDRVS(2)
!
! Finish up
 IF (IEND .EQ. 5) IDO = 3
 CYCLE
 END IF
END DO
END DO
99997 FORMAT (11X, 'T', 10X, 'Y1/Y2', 11X, 'Y1P', 11X, 'Y1PP/Y2PP')
99998 FORMAT (4F15.4)
99999 FORMAT (2(15X, F15.4))

```

```

END

SUBROUTINE FCN (IDO, T, Y, HIDRVS)
INTEGER IDO
REAL T, Y(*), HIDRVS(*)

HIDRVS(1) = -9.8 - .005/Y(3)*Y(2)
HIDRVS(2) = -1.0E-8 * Y(2)*Y(2)
RETURN
END

```

## Output

| T       | Y1/Y2                | Y1P        | Y1PP/Y2PP          |
|---------|----------------------|------------|--------------------|
| 10.0000 | 89773.0391<br>0.8954 | -1044.0096 | -3.9701<br>-0.0109 |
| 20.0000 | 79150.9844<br>0.7826 | -1078.6334 | -2.9083<br>-0.0116 |
| 30.0000 | 68240.9453<br>0.6635 | -1101.0380 | -1.5031<br>-0.0121 |
| 40.0000 | 57184.9062<br>0.5413 | -1106.9635 | 0.4253<br>-0.0121  |
| 50.0000 | 46178.1367<br>0.4201 | -1089.8292 | 3.1700<br>-0.0119  |

---

## Introduction to Subroutine PDE\_1D\_MG

The section describes an algorithm and a corresponding integrator subroutine PDE\_1D\_MG for solving a system of partial differential equations

$$u_t \equiv \frac{\partial u}{\partial t} = f(u, x, t), \quad x_L < x < x_R, t > t_0$$

### Equation 1

This software is a one-dimensional solver. It requires initial and boundary conditions in addition to values of  $u_t$ . The integration method is noteworthy due to the maintenance of grid lines in the space variable,  $x$ . Details for choosing new grid lines are given in Blom and Zegeling, (1994). The class of problems solved with PDE\_1D\_MG is expressed by equations:

$$\sum_{k=1}^{NPDE} C_{j,k}(x, t, u, u_x) \frac{\partial u^k}{\partial t} = x^{-m} \frac{\partial}{\partial x} (x^m R_j(x, t, u, u_x)) - Q_j(x, t, u, u_x),$$
$$j = 1, \dots, NPDE, \quad x_L < x < x_R, \quad t > t_0, \quad m \in \{0, 1, 2\}$$

### Equation 2

The vector

$$u \equiv [u', \dots, u^{NPDE}]^T$$

is the solution. The integer value  $NPDE \geq 1$  is the number of differential equations. The functions  $R_j$  and  $Q_j$  can be regarded, in special cases, as flux and source terms. The functions

$$u, C_{j,k}, R_j \text{ and } Q_j$$

are expected to be continuous. Allowed values

$$m=0, m-1, \text{ and } m=2$$

are for problems in Cartesian, cylindrical or polar, and spherical coordinates. In the two cases  $m > 0$ , the interval

$$x_L, x_R$$

must not contain  $x = 0$  as an interior point.

The boundary conditions have the master equation form

$$\beta_j(x, t) R_j(x, t, u, u_x) = \gamma_j(x, t, u, u_x),$$

at  $x = x_L$  and  $x = x_R$ ,  $j = 1, \dots, NPDE$

### Equation 3

In the boundary conditions the

$\beta_j$  and  $\gamma_j$

are continuous functions of their arguments. In the two cases  $m > 0$  and an endpoint occurs at 0, the finite value of the solution at  $x = 0$  must be ensured. This requires the specification of the solution at  $x = 0$ , or implies that

$$R_j \Big|_{x=x_L} = 0$$

or

$$R_j \Big|_{x=x_R} = 0$$

The initial values satisfy

$$u(x, t_0) = u_0(x), \quad x \in [x_L, x_R],$$

where  $u_0$  is a piece-wise continuous vector function of  $x$  with *NPDE* components.

The user must pose the problem so that mathematical definitions are known for the functions

$$C_{k,j}, R_j, Q_j, \beta_j, \gamma_j \quad \text{and} \quad u_0$$

These functions are provided to the routine `PDE_1D_MG` in the form of three subroutines. Optionally, this information can be provided by *reverse communication*. These forms of the interface are explained below and illustrated with examples. Users may turn directly to the examples if they are comfortable with the description of the algorithm.

---

## PDE\_1D\_MG



Invokes a module, with the statement `USE PDE_1D_MG`, near the second line of the program unit. The integrator is provided with single or double precision arithmetic, and a generic named interface is provided. We do not recommend using 32-bit floating point arithmetic here. The routine is called within the following loop, and is entered with each value of `IDO`. The loop continues until a value of `IDO` results in an exit.

```
IDO=1
DO
 CASE(IDO == 1) {Do required initialization steps}
 CASE(IDO == 2) {Save solution, update T0 and TOUT }
 IF{Finished with integration} IDO=3
 CASE(IDO == 3) EXIT {Normal}
 CASE(IDO == 4) EXIT {Due to errors}
```

```

CASE(IDO == 5) {Evaluate initial data}
CASE(IDO == 6) {Evaluate differential equations}
CASE(IDO == 7) {Evaluate boundary conditions}
CASE(IDO == 8) {Prepare to solve banded system}
CASE(IDO == 9) {Solve banded system}

CALL PDE_1D_MG (T0, TOUT, IDO, U, &
initial_conditions, &
pde_system_definition, &
boundary_conditions, IOPT)
END DO

```

The arguments to PDE\_1D\_MG are *required* or *optional*.

## Required Arguments

### T0—(Input/Output)

This is the value of the independent variable  $t$  where the integration of  $u_t$  begins. It is set to the value TOUT on return.

### TOUT—(Input)

This is the value of the independent variable  $t$  where the integration of  $u_t$  ends. Note: Values of  $T0 < TOUT$  imply integration in the forward direction, while values of  $T0 > TOUT$  imply integration in the backward direction. Either direction is permitted.

### IDO—(Input/Output)

This is an integer flag that directs program control and user action. Its value is used for initialization, termination, and for directing user response during reverse communication:

### IDO—(Input/Output)

This is an integer flag that directs program control and user action. Its value is used for initialization, termination, and for directing user response during reverse communication:

**IDO=1** This value is assigned by the user for the start of a new problem. Internally it causes allocated storage to be reallocated, conforming to the problem size. Various initialization steps are performed.

**IDO=2** This value is assigned by the routine when the integrator has successfully reached the end point, TOUT.

**IDO=3** This value is assigned by the user at the end of a problem. The routine is called by the user with this value. Internally it causes termination steps to be performed.

**IDO=4** This value is assigned by the integrator when a type FATAL or TERMINAL error condition has occurred, and error processing is set **NOT** to **STOP** for these

types of errors. It is not necessary to make a final call to the integrator with **IDO=3** in this case.

Values of **IDO = 5,6,7,8,9** are reserved for applications that provide problem information or linear algebra computations using reverse communication. When problem information is provided using reverse communication, the differential equations, boundary conditions and initial data must all be given. The absence of optional subroutine names in the calling sequence directs the routine to use reverse communication. In the module `PDE_1D_MG_INT`, scalars and arrays for evaluating results are named below. The names are preceded by the prefix “s\_pde\_1d\_mg\_” or “d\_pde\_1d\_mg\_”, depending on the precision. We use the prefix “?\_pde\_1d\_mg\_”, for the appropriate choice.

**IDO=5** This value is assigned by the integrator, requesting data for the initial conditions. Following this evaluation the integrator is re-entered.

(Optional) Update the grid of values in array locations  $U(NPDE + 1, j) j = 2, \dots, N$ . This grid is returned to the user equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values for all components of the system at the grid of values  $U(NPDE + 1, j) j = 1, \dots, N$ . If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

**IDO=6** This value is assigned by the integrator, requesting data for the differential equations. Following this evaluation the integrator is re-entered. Evaluate the terms of the system of Equation 2. A default value of  $m = 0$  is assumed, but this can be changed to one of the other choices  $m = 1$  or  $2$ . Use the optional argument `IOPT(:)` for that purpose. Put the values in the arrays as indicated<sup>1</sup>.

$$x \equiv ?\_pde\_1d\_mg\_x$$

$$t \equiv ?\_pde\_1d\_mg\_t$$

$$u^j \equiv ?\_pde\_1d\_mg\_u(j)$$

$$\frac{\partial u^j}{\partial x} = u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j)$$

$$?\_pde\_1d\_mg\_c(j,k) := C_{j,k}(x,t,u,u_x)$$

$$?\_pde\_1d\_mg\_r(j) := r_j(x,t,u,u_x)$$

$$?\_pde\_1d\_mg\_q(j) := q_j(x,t,u,u_x)$$

$$j, k = 1, \dots, NPDE$$

---

<sup>1</sup> The assign-to equality,  $a := b$ , used here and below, is read “the expression  $b$  is evaluated and then assigned to the location  $a$ .”

If any of the functions cannot be evaluated, set `pde_1d_mg_ires=3`.  
Otherwise do not change its value.

**IDO=7** This value is assigned by the integrator, requesting data for the boundary conditions, as expressed in Equation 3. Following the evaluation the integrator is re-entered.

$$\begin{aligned}
 x &\equiv ?\_pde\_1d\_mg\_x \\
 t &\equiv ?\_pde\_1d\_mg\_t \\
 u^j &\equiv ?\_pde\_1d\_mg\_u(j) \\
 \frac{\partial u^j}{\partial x} &= u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j) \\
 ?\_pde\_1d\_mg\_beta(j) &:= \beta_j(x, t, u, u_x) \\
 ?\_pde\_1d\_mg\_gamma(j) &:= \gamma_j(x, t, u, u_x) \\
 j &= 1, \dots, NPDE
 \end{aligned}$$

The value  $x \in \{x_L, x_R\}$ , and the logical flag `pde_1d_mg_LEFT=.TRUE.` for  $x = x_L$ . It has the value `pde_1d_mg_LEFT=.FALSE.` for  $x = x_R$ . If any of the functions cannot be evaluated, set `pde_1d_mg_ires=3`. Otherwise do not change its value.

**IDO=8** This value is assigned by the integrator, requesting the calling program to prepare for solving a banded linear system of algebraic equations. This value will occur only when the option for “reverse communication solving” is set in the array `IOPT(:)`, with option `PDE_1D_MG_REV_COMM_FACTOR_SOLVE`. The matrix data for this system is in *Band Storage Mode*, described in the section: Reference Material for the IMSL Fortran Numerical Libraries.

|                                   |                                                                                                                     |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>PDE_1D_MG_IBAND</code>      | Half band-width of linear system                                                                                    |
| <code>PDE_1D_MG_LDA</code>        | The value $3 * PDE\_1D\_MG\_IBAND + 1$ , with $NEQ = (NPDE + 1)N$                                                   |
| <code>?_PDE_1D_MG_A</code>        | Array of size <code>PDE_1D_MG_LDA</code> by <code>NEQ</code> holding the problem matrix in <i>Band Storage Mode</i> |
| <code>PDE_1D_MG_PANIC_FLAG</code> | Integer set to a non-zero value only if the linear system is detected as singular                                   |

**IDO=9** This value is assigned by the integrator, requesting the calling program to solve a linear system with the matrix defined as noted with **IDO=8**.

|                      |                                                                       |
|----------------------|-----------------------------------------------------------------------|
| ?_PDE_1D_MG_RHS      | Array of size NEQ holding the linear system problem right-hand side   |
| PDE_1D_MG_PANIC_FLAG | Integer set to a non-zero value only if the linear system is singular |
| ?_PDE_1D_MG_SOL      | Array of size NEQ to receive the solution, after the solving step     |

$U(1:NPDE+1, 1:N)$ —(Input/Output)

This assumed-shape array specifies *Input* information about the problem size and boundaries. The dimension of the problem is obtained from  $NPDE + 1 = size(U,1)$ . The number of grid points is obtained by  $N = size(U,2)$ . Limits for the variable  $x$  are assigned as input in array locations,  $U(NPDE + 1, 1) = x_L$ ,  $U(NPDE + 1, N) = x_R$ . It is not required to define  $U(NPDE + 1, j)$ ,  $j=2, \dots, N-1$ . At completion, the array  $U(1:NPDE, 1:N)$  contains the approximate solution value  $U_i(x_j(TOUT), TOUT)$  in location  $U(I, J)$ . The grid value  $x_j(TOUT)$  is in location  $U(NPDE+1, J)$ . Normally the grid values are equally spaced as the integration starts. Variable spaced grid values can be provided by defining them as *Output* from the subroutine `initial_conditions` or during reverse communication, `IDO=5`.

## Optional Arguments

`initial_conditions`—(Input)

The name of an external subroutine, written by the user, when using forward communication. If this argument is not used, then reverse communication is used to provide the problem information. The routine gives the initial values for the system at the starting independent variable value  $T_0$ . This routine can also provide a non-uniform grid at the initial value.

```
SUBROUTINE initial_conditions (NPDE,N,U)
 Integer NPDE,N
 REAL(kind(T0)) U(:,)
END SUBROUTINE
```

(Optional) Update the grid of values in array locations

$U(NPDE + 1, j)$ ,  $j = 2, \dots, N - 1$ . This grid is input equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values  $U(:, j)$ ,  $j = 1, \dots, N$  for all components of the system at the grid of values  $U(NPDE + 1, j)$ ,  $j = 1, \dots, N$ . If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

`pde_system_definition`—(Input)

The name of an external subroutine, written by the user, when using forward communication. It gives the differential equation, as expressed in Equation 2.

```

SUBROUTINE pde_system_definition&
 (t, x, NPDE, u, dudx, c, q, r, IRES)
 Integer NPDE, IRES
 REAL(kind(T0)) t, x, u(:), dudx(:)
 REAL(kind(T0)) c(:,:), q(:), r(:)
END SUBROUTINE

```

Evaluate the terms of the system of . A default value of  $m=0$  is assumed, but this can be changed to one of the other choices  $m=1$  or  $2$ . Use the optional argument `IOPT(:)` for that purpose. Put the values in the arrays as indicated.

$$\begin{aligned}
 u^j &\equiv u(j) \\
 \frac{\partial u^j}{\partial x} &= u_x^j \equiv dudx(j) \\
 c(j,k) &:= C_{j,k}(x,t,u,u_x) \\
 r(j) &:= r_j(x,t,u,u_x) \\
 q(j) &:= q_j(x,t,u,u_x) \\
 j,k &= 1, \dots, NPDE
 \end{aligned}$$

If any of the functions cannot be evaluated, set `IRES=3`. Otherwise do not change its value.

boundary\_conditions—(Input)

The name of an external subroutine, written by the user when using forward communication. It gives the boundary conditions, as expressed in Equation 2.

```

SUBROUTINE BOUNDARY_CONDITIONS(T,BETA,GAMMA,U,DUDX,NPDE,LEFT,IRES)
 real(kind(ld0)),intent(in)::t
 real(kind(ld0)),intent(out),dimension(:)::BETA, GAMMA
 real(kind(ld0)),intent(in),dimension(:)::U,DUDX
 integer,intent(in)::NPDE
 logical,intent(in)::LEFT
 integer,intent(out)::IRES
 END SUBROUTINE

```

$$u^j \equiv u(j)$$

$$\frac{\partial u^j}{\partial x} = u_x^j \equiv dudx(j)$$

$$beta(j) := \beta_j(x, t, u, u_x)$$

$$gamma(j) := \gamma_j(x, t, u, u_x)$$

$$j = 1, \dots, NPDE$$

The value  $x \in \{x_L, x_R\}$ , and the logical flag `LEFT=.TRUE.` for  $x = x_L$ . The flag has the value `LEFT=.FALSE.` for  $x = x_R$ .

**IOPT—(Input)**

Derived type array `s_options` or `d_options`, used for passing optional data to `PDE_1D_MG`. See the section [Optional Data](#) in the Introduction for an explanation of the derived type and its use. It is necessary to invoke a module, with the statement `USE ERROR_OPTION_PACKET`, near the second line of the program unit. Examples 2-8 use this optional argument. The choices are as follows:

| Packaged Options for PDE_1D_MG |                                              |              |
|--------------------------------|----------------------------------------------|--------------|
| Option Prefix = ?              | Option Name                                  | Option Value |
| <code>s_, d_</code>            | <code>PDE_1D_MG_CART_COORDINATES</code>      | 1            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_CYL_COORDINATES</code>       | 2            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_SPH_COORDINATES</code>       | 3            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_TIME_SMOOTHING</code>        | 4            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_SPATIAL_SMOOTHING</code>     | 5            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_MONITOR_REGULARIZING</code>  | 6            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_RELATIVE_TOLERANCE</code>    | 7            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_ABSOLUTE_TOLERANCE</code>    | 8            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_MAX_BDF_ORDER</code>         | 9            |
| <code>s_, d_</code>            | <code>PDE_1D_MG_REV_COMM_FACTOR_SOLVE</code> | 10           |
| <code>s_, d_</code>            | <code>PDE_1D_MG_NO_NULLIFY_STACK</code>      | 11           |

`IOPT(IO) = PDE_1D_MG_CART_COORDINATES`  
Use the value  $m=0$  in Equation 2. This is the default.

`IOPT(IO) = PDE_1D_MG_CYL_COORDINATES`  
Use the value  $m=1$  in Equation 2. The default value is  $m=0$ .

`IOPT(IO) = PDE_1D_MG_SPH_COORDINATES`  
Use the value  $m=2$  in Equation 2. The default value is  $m=0$ .

IOPT(IO) =  
   ?\_OPTIONS(PDE\_1D\_MG\_TIME\_SMOOTHING,TAU)  
 This option resets the value of the parameter  $\tau \geq 0$ , described above.  
 The default value is  $\tau = 0$ .

IOPT(IO) =  
   ?\_OPTIONS(PDE\_1D\_MG\_SPATIAL\_SMOOTHING,KAP)  
 This option resets the value of the parameter  $\kappa \geq 0$ , described above.  
 The default value is  $\kappa = 2$ .

IOPT(IO) =  
   ?\_OPTIONS(PDE\_1D\_MG\_MONITOR\_REGULARIZING,ALPH)  
 This option resets the value of the parameter  $\alpha \geq 0$ , described above.  
 The default value is  $\alpha = 0.01$ .

IOPT(IO) = ?\_OPTIONS  
   (PDE\_1D\_MG\_RELATIVE\_TOLERANCE,RTOL)  
 This option resets the value of the relative accuracy parameter used in DASPG.  
 The default value is  $RTOL=1E-2$  for single precision and  $RTOL=1D-4$  for double  
 precision.

IOPT(IO) = ?\_OPTIONS  
   (PDE\_1D\_MG\_ABSOLUTE\_TOLERANCE,ATOL)  
 This option resets the value of the absolute accuracy parameter used in DASPG. The  
 default value is  $ATOL=1E-2$  for single precision and  
 $ATOL=1D-4$  for double precision.

IOPT(IO) = PDE\_1D\_MG\_MAX\_BDF\_ORDER  
 IOPT(IO+1) = MAXBDF  
 Reset the maximum order for the BDF formulas used in DASPG. The default value is  
 $MAXBDF=2$ . The new value can be any integer between 1 and 5. Some problems will  
 benefit by making this change. We used the default value due to the fact that DASPG  
 may cycle on its selection of order and step-size with orders higher than value 2.

IOPT(IO) = PDE\_1D\_MG\_REV\_COMM\_FACTOR\_SOLVE  
 The calling program unit will solve the banded linear systems required in the stiff  
 differential-algebraic equation integrator. Values of **IDO=8, 9** will occur only when  
 this optional value is used.

IOPT(IO) = PDE\_1D\_MG\_NO\_NULLIFY\_STACK  
 To maintain an efficient interface, the routine PDE\_1D\_MG collapses the subroutine call  
 stack with `CALL_E1PSH("NULLIFY_STACK")`. This implies that the overhead of  
 maintaining the stack will be eliminated, which may be important with reverse  
 communication. It does not eliminate error processing. However, precise information  
 of which routines have errors will not be displayed. To see the full call chain, this  
 option should be used. Following completion of the integration, stacking is turned  
 back on with `CALL_E1POP("NULLIFY_STACK")`.

## FORTRAN 90 Interface

Generic: `CALL PDE_1D_MG (T0, TOUT, IDO, [, ...])`

Specific: The specific interface names are `S_PDE_1D_MG` and `D_PDE_1D_MG`.

## Description

The equation

$$u_t = f(u, x, t), \quad x_L < x < x_R, \quad t > t_0,$$

is approximated at  $N$  time-dependent grid values

$$x_L = x_0 < \dots < x_i(t) < x_{i+1}(t) < \dots < x_N = x_R.$$

Using the total differential

$$\frac{du}{dt} = u_t + u_x \frac{dx}{dt}$$

transforms the differential equation to

$$\frac{du}{dt} - u_x \frac{dx}{dt} = u_t = f(u, x, t)$$

Using central divided differences for the factor  $u_x$  leads to the system of ordinary differential equations in implicit form

$$\frac{dU_i}{dt} - \frac{(U_{i+1} - U_{i-1})}{(x_{i+1} - x_{i-1})} \frac{dx_i}{dt} = F_i, \quad t > t_0, \quad i = 1, \dots, N$$

The terms  $U_i$ ,  $F_i$  respectively represent the approximate solution to the partial differential equation and the value of  $f(u, x, t)$  at the point  $(x, t) = (x_i(t), t)$ . The truncation error is second-order in the space variable,  $x$ . The above ordinary differential equations are underdetermined, so additional equations are added for the variation of the time-dependent grid points. It is necessary to discuss these equations, since they contain parameters that can be adjusted by the user. Often it will be necessary to modify these parameters to solve a difficult problem. For this purpose the following quantities are defined<sup>2</sup>:

$$\begin{aligned} \Delta x_i &= x_{i+1} - x_i, \quad n_i = (\Delta x_i)^{-1} \\ \mu_i &= n_i - \kappa(\kappa + 1)(n_{i+1} - 2n_i + n_{i-1}), \quad 0 \leq i \leq N \\ n_{-1} &\equiv n_0, \quad n_{N+1} \equiv n_N \end{aligned}$$

---

<sup>2</sup> The three-tiered equal sign, used here and below, is read “ $a \equiv b$  or  $a$  and  $b$  are exactly the same object or value.”

The values  $n_i$  are the so-called point concentration of the grid, and  $\kappa \geq 0$  denotes a spatial smoothing parameter. Now the grid points are defined implicitly so that

$$\frac{\mu_{i-1} + \tau \frac{d\mu_{i-1}}{dt}}{M_{i-1}} = \frac{\mu_i + \tau \frac{d\mu_i}{dt}}{M_i}, 1 \leq i \leq N$$

where  $\tau \geq 1$  is a time-smoothing parameter. Choosing  $\tau$  very large results in a fixed grid. Increasing the value of  $\tau$  from its default avoids the error condition where grid lines cross. The divisors are

$$M_i^2 = \alpha + NPDE^{-1} \sum_{j=1}^{NPDE} \frac{(U_{i+1}^j - U_i^j)^2}{(\Delta x_i)^2}$$

The value  $\kappa$  determines the level of clustering or spatial smoothing of the grid points. Decreasing  $\kappa$  from its default decrease the amount of spatial smoothing. The parameters  $M_i$  approximate arc length and help determine the shape of the grid or  $x_i$ -distribution. The parameter  $\tau$  prevents the grid movement from adjusting immediately to new values of the  $M_i$ , thereby avoiding oscillations in the grid that cause large relative errors. This is important when applied to solutions with steep gradients.

The discrete form of the differential equation and the smoothing equations are combined to yield the implicit system of differential equations.  $\kappa$

$$A(Y) \frac{dY}{dt} = L(Y),$$

$$Y = \left[ U_1^1, \dots, U_1^{NPDE}, x_1, \dots, U_j^1, \dots, U_j^{NPDE}, x_j, \dots \right]^T$$

This is frequently a stiff differential-algebraic system. It is solved using the integrator `DASPG` and its subroutines, including `D2SPG`. These are documented in this chapter. Note that `DASPG` is restricted to use within `PDE_1D_MG` until the routine exits with the flag `IDO = 3`. If `DASPG` is needed during the evaluations of the differential equations or boundary conditions, use of a second processor and inter-process communication is required. The only options for `DASPG` set by `PDE_1D_MG` are the Maximum BDF Order, and the absolute and relative error values, `ATOL` and `RTOL`. Users may set other options using the Options Manager. This is described in routine `DASPG` and generally in Chapter 11 of this manual.

## Remarks on the Examples

Due to its importance and the complexity of its interface, this subroutine is presented with several examples. Many of the program features are exercised. The problems complete without any change to the optional arguments, except where these changes are required to describe or to solve the problem.

In many applications the solution to a PDE is used as an auxiliary variable, perhaps as part of a larger design or simulation process. The truncation error of the approximate solution is commensurate with piece-wise linear interpolation on the grid of values, at each output point. To

show that the solution is reasonable, a graphical display is revealing and helpful. We have not provided graphical output as part of our documentation, but users may already have the Visual Numerics, Inc. product, PV-WAVE, not included with Fortran Numerical Library. Examples 1-8 write results in files "PDE\_ex0?.out" that can be visualized with PV-WAVE. We provide a script of commands, "pde\_1d\_mg\_plot.pro", for viewing the solutions (see example below). The grid of values and each consecutive solution component is displayed in separate plotting windows. The script and data files written by examples 1-8 on a SUN-SPARC system are in the directory for Fortran Numerical Library examples. When inside PV-WAVE, execute the command line "pde\_1d\_mg\_plot,filename='PDE\_ex0?.out'" to view the output of a particular example.

### Code for PV-WAVE Plotting (Examples Directory)

```

PRO PDE_1d_mg_plot, FILENAME = filename, PAUSE = pause
;
 if keyword_set(FILENAME) then file = filename else file = "res.dat"
 if keyword_set(PAUSE) then twait = pause else twait = .1
;
 Define floating point variables that will be read
; from the first line of the data file.
 xl = 0D0
 xr = 0D0
 t0 = 0D0
 tlast = 0D0
;
 Open the data file and read in the problem parameters.
 openr, lun, filename, /get_lun
 readf, lun, npde, np, nt, xl, xr, t0, tlast
;
 Define the arrays for the solutions and grid.
 u = dblarr(nt, npde, np)
 g = dblarr(nt, np)
 times = dblarr(nt)
;
 Define a temporary array for reading in the data.
 tmp = dblarr(np)
 t_tmp = 0D0
;
 Read in the data.
 for i = 0, nt-1 do begin ; For each step in time
 readf, lun, t_tmp
 times(i) = t_tmp

 for k = 0, npde-1 do begin ; For each PDE:
 rmf, lun, tmp
 u(i,k,*) = tmp ; Read in the components.
 end

 rmf, lun, tmp
 g(i,*) = tmp ; Read in the grid.
 end
;
 Close the data file and free the unit.
 close, lun
 free_lun, lun
;

```

```

; We now have all of the solutions and grids.
;
; Delete any window that is currently open.
while (!d.window NE -1) do WDELETE
;
; Open two windows for plotting the solutions
; and grid.
window, 0, xsize = 550, ysize = 420
window, 1, xsize = 550, ysize = 420
;
; Plot the grid.
wset, 0
plot, [xl, xr], [t0, tlast], /nodata, ystyle = 1, $
 title = "Grid Points", xtitle = "X", ytitle = "Time"
for i = 0, np-1 do begin
 oplot, g(*, i), times, psym = -1
end
;
; Plot the solution(s):
wset, 1
for k = 0, npde-1 do begin
 umin = min(u(*,k,*))
 umax = max(u(*,k,*))
 for i = 0, nt-1 do begin
 title = strcompress("U_"+string(k+1), /remove_all)+ $
 " at time "+string(times(i))
 plot, g(i, *), u(i,k,*), ystyle = 1, $
 title = title, xtitle = "X", $
 ytitle = strcompress("U_"+string(k+1), /remove_all), $
 xr = [xl, xr], yr = [umin, umax], $
 psym = -4
 wait, twait
 end
end
end
end

```

## Example 1 - Electrodynamics Model

This example is from Blom and Zegeling (1994). The system is

$$u_t = \varepsilon p u_{xx} - g(u - v)$$

$$v_t = p v_{xx} + g(u - v),$$

$$\text{where } g(z) = \exp(\eta z / 3) - \exp(-2\eta z / 3)$$

$$0 \leq x \leq 1, 0 \leq t \leq 4$$

$$u_x = 0 \text{ and } v = 0 \text{ at } x = 0$$

$$u = 1 \text{ and } v_x = 0 \text{ at } x = 1$$

$$\varepsilon = 0.143, p = 0.1743, \eta = 17.19$$

We make the connection between the model problem statement and the example:

$$C = I_2$$

$$m = 0, R_1 = \varepsilon p u_x, R_2 = p v_x$$

$$Q_1 = g(u - v), Q_2 = -Q_1$$

The boundary conditions are

$$\beta_1 = 1, \beta_2 = 0, \gamma_1 = 0, \gamma_2 = v, \text{ at } x = x_L = 0$$

$$\beta_1 = 0, \beta_2 = 1, \gamma_1 = u - 1, \gamma_2 = 0, \text{ at } x = x_R = 1$$

### Rationale: Example 1

This is a non-linear problem with sharply changing conditions near  $t=0$ . The default settings of integration parameters allow the problem to be solved. The use of `PDE_1D_MG` with forward communication requires three subroutines provided by the user to describe the initial conditions, differential equations, and boundary conditions.

```

program PDE_EX1
! Electrodynamics Model:
 USE PDE_1d_mg_int
 IMPLICIT NONE

 INTEGER, PARAMETER :: NPDE=2, N=51, NFRAMES=5
 INTEGER I, IDO

! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), T0, TOUT
 real(kind(1d0)) :: ZERO=0D0, ONE=1D0, &
 DELTA_T=10D0, TEND=4D0
 EXTERNAL IC_01, PDE_01, BC_01

! Start loop to integrate and write solution values.
 IDO=1
 DO
 SELECT CASE (IDO)

! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=1D-3
 U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
 OPEN(FILE='PDE_ex01.out',UNIT=7)
 WRITE(7, "(3I5, 4F10.5)") NPDE, N, NFRAMES,&
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)

 WRITE(7, "(F10.5)") TOUT
 DO I=1,NPDE+1
 WRITE(7, "(4E15.5)") U(I,:)
 END DO
 T0=TOUT;TOUT=TOUT*DELTA_T

```

```

 IF(T0 >= TEND) IDO=3
 TOUT=MIN(TOUT, TEND)

! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

 END SELECT

! Forward communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U,&
 initial_conditions= IC_01,&
 PDE_system_definition= PDE_01,&
 boundary_conditions= BC_01)

 END DO
 END

 SUBROUTINE IC_01(NPDE, NPTS, U)
! This is the initial data for Example 1.
 IMPLICIT NONE
 INTEGER NPDE, NPTS
 REAL(KIND(1D0)) U(NPDE+1,NPTS)
 U(1,:)=1D0;U(2,:)=0D0
 END SUBROUTINE

 SUBROUTINE PDE_01(T, X, NPDE, U, DUDX, C, Q, R, IRES)
! This is the differential equation for Example 1.
 IMPLICIT NONE
 INTEGER NPDE, IRES
 REAL(KIND(1D0)) T, X, U(NPDE), DUDX(NPDE),&
 C(NPDE,NPDE), Q(NPDE), R(NPDE)
 REAL(KIND(1D0)) :: EPS=0.143D0, P=0.1743D0,&
 ETA=17.19D0, Z, TWO=2D0, THREE=3D0

 C=0D0;C(1,1)=1D0;C(2,2)=1D0
 R=P*DUDX;R(1)=R(1)*EPS
 Z=ETA*(U(1)-U(2))/THREE
 Q(1)=EXP(Z)-EXP(-TWO*Z)
 Q(2)=-Q(1)

 END SUBROUTINE

 SUBROUTINE BC_01(T, BTA, GAMA, U, DUDX, NPDE, LEFT, IRES)
! These are the boundary conditions for Example 1.
 IMPLICIT NONE
 INTEGER NPDE, IRES
 LOGICAL LEFT
 REAL(KIND(1D0)) T, BTA(NPDE), GAMA(NPDE),&
 U(NPDE), DUDX(NPDE)

 IF(LEFT) THEN
 BTA(1)=1D0;BTA(2)=0D0
 GAMA(1)=0D0;GAMA(2)=U(2)
 ELSE

```

```

 BTA (1) = 0D0; BTA (2) = 1D0
 GAMA (1) = U (1) - 1D0; GAMA (2) = 0D0
 END IF
END SUBROUTINE

```

## Additional Examples

### Example 2 - Inviscid Flow on a Plate

This example is a first order system from Pennington and Berzins, (1994). The equations are

$$\begin{aligned}
 u_t &= -v_x \\
 uu_t &= -vu_x + w_{xx} \\
 w &= u_x, \text{ implying that } uu_t = -vu_x + u_{xx} \\
 u(0, t) &= v(0, t) = 0, u(\infty, t) = u(x_R, t) = 1, t \geq 0 \\
 u(x, 0) &= 1, v(x, 0) = 0, x \geq 0
 \end{aligned}$$

Following elimination of  $w$ , there remain  $NPDE = 2$  differential equations. The variable  $t$  is not time, but a second space variable. The integration goes from  $t = 0$  to  $t = 5$ . It is necessary to truncate the variable  $x$  at a finite value, say  $x_{max} = x_R = 25$ . In terms of the integrator, the system is defined by letting  $m = 0$  and

$$C = \{C_{jk}\} = \begin{bmatrix} 1 & 0 \\ u & 0 \end{bmatrix}, R = \begin{bmatrix} -v \\ u_x \end{bmatrix}, Q = \begin{bmatrix} 0 \\ vu_x \end{bmatrix}$$

The boundary conditions are satisfied by

$$\begin{aligned}
 \beta = 0, \gamma &= \begin{bmatrix} u - \exp(-20t) \\ v \end{bmatrix}, \text{ at } x = x_L \\
 \beta = 0, \gamma &= \begin{bmatrix} u - 1 \\ v_x \end{bmatrix}, \text{ at } x = x_R
 \end{aligned}$$

We use  $N = 10 + 51 = 61$  grid points and output the solution at steps of  $\Delta t = 0.1$ .

### Rationale: Example 2

This is a non-linear boundary layer problem with sharply changing conditions near  $t = 0$ . The problem statement was modified so that boundary conditions are continuous near  $t = 0$ . Without this change the underlying integration software, `DASPG`, cannot solve the problem. The continuous blending function  $u - \exp(-20t)$  is arbitrary and artfully chosen. This is a mathematical change to the problem, required because of the stated discontinuity at  $t = 0$ . Reverse communication is used for the problem data. No additional user-written subroutines are required when using reverse communication. We also have chosen 10 of the initial grid points to be concentrated near  $x_L = 0$ , anticipating rapid change in the solution near that point. Optional changes are made to use a pure absolute error tolerance and non-zero time-smoothing.

```

 program PDE_1D_MG_EX02
! Inviscid Flow Over a Plate
 USE PDE_1d_mg_int
 USE ERROR_OPTION_PACKET
 IMPLICIT NONE

 INTEGER, PARAMETER :: NPDE=2, N1=10, N2=51, N=N1+N2
 INTEGER I, IDO, NFRAMES
! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), T0, TOUT, DX1, DX2, DIF
 real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-1, &
 TEND=5D0, XMAX=25D0
 real(kind(1d0)) :: U0=1D0, U1=0D0, TDELTA=1D-1, TOL=1D-2
 TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)
! Define values that determine limits and options.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=ZERO;U(NPDE+1,N)=XMAX
 OPEN(FILE='PDE_ex02.out',UNIT=7)
 NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
 DX1=XMAX/N2;DX2=DX1/N1
 IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
 IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,TOL)
 IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)

! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1,NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF

! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

! Define initial data values.
 CASE (5)
 U(:NPDE, :)=ZERO;U(1, :)=ONE
 DO I=1,N1

```

```

 U(NPDE+1,I)=(I-1)*DX2
 END DO
 DO I=N1+1,N
 U(NPDE+1,I)=(I-N1)*DX1
 END DO
 WRITE(7,"(F10.5)")T0
 DO I=1,NPDE+1
 WRITE(7,"(4E15.5)")U(I,:)
 END DO

! Define differential equations.
 CASE (6)
 D_PDE_1D_MG_C=ZERO
 D_PDE_1D_MG_C(1,1)=ONE
 D_PDE_1D_MG_C(2,1)=D_PDE_1D_MG_U(1)

 D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_U(2)
 D_PDE_1D_MG_R(2)= D_PDE_1D_MG_DUDX(1)

 D_PDE_1D_MG_Q(1)= ZERO
 D_PDE_1D_MG_Q(2)= &
 D_PDE_1D_MG_U(2)*D_PDE_1D_MG_DUDX(1)
! Define boundary conditions.
 CASE (7)
 D_PDE_1D_MG_BETA=ZERO
 IF(PDE_1D_MG_LEFT) THEN
 DIF=EXP(-20D0*D_PDE_1D_MG_T)
! Blend the left boundary value down to zero.
 D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-DIF,D_PDE_1D_MG_U(2)/)
 ELSE
 D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-
ONE,D_PDE_1D_MG_DUDX(2)/)
 END IF
 END SELECT

! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
END DO
end program

```

### Example 3 - Population Dynamics

This example is from Pennington and Berzins (1994). The system is

$$u_t = -u_x - I(t)u, \quad x_L = 0 \leq x \leq a = x_R, t \geq 0$$

$$I(t) = \int_0^a u(x, t) dx$$

$$u(x, 0) = \frac{\exp(-x)}{2 - \exp(-a)}$$

$$u(0, t) = g \left( \int_0^a b(x, I(t)) u(x, t) dx, t \right), \text{ where}$$

$$b(x, y) = \frac{xy \exp(-x)}{(y+1)^2}, \text{ and}$$

$$g(z, t) = \frac{4z(2 - 2\exp(-a) + \exp(-t))^2}{(1 - \exp(-a))(1 - (1 + 2a)\exp(-2a)(1 - \exp(-a) + \exp(-t)))}$$

This is a notable problem because it involves the unknown

$$u(x, t) = \frac{\exp(-x)}{1 - \exp(-a) + \exp(-t)}$$

across the entire domain. The software can solve the problem by introducing two dependent algebraic equations:

$$v_1(t) = \int_0^a u(x, t) dx,$$

$$v_2(t) = \int_0^a x \exp(-x) u(x, t) dx$$

This leads to the modified system

$$u_t = -u_x - v_1 u, \quad 0 \leq x \leq a, t \geq 0$$

$$u(0, t) = \frac{g(1, t) v_1 v_2}{(v_1 + 1)^2}$$

In the interface to the evaluation of the differential equation and boundary conditions, it is necessary to evaluate the integrals, which are computed with the values of  $u(x, t)$  on the grid. The integrals are approximated using the trapezoid rule, commensurate with the truncation error in the integrator.

### Rationale: Example 3

This is a non-linear integro-differential problem involving non-local conditions for the differential equation and boundary conditions. Access to evaluation of these conditions is provided using reverse communication. It is not possible to solve this problem with forward communication, given the current subroutine interface. Optional changes are made to use an absolute error tolerance and non-zero time-smoothing. The time-smoothing value  $\tau = 1$  prevents grid lines from crossing.

```

 program PDE_1D_MG_EX03
! Population Dynamics Model.
 USE PDE_1d_mg_int
 USE ERROR_OPTION_PACKET
 IMPLICIT NONE
 INTEGER, PARAMETER :: NPDE=1, N=101
 INTEGER IDO, I, NFRAMES
! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), MID(N-1), T0, TOUT, V_1, V_2
 real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0, &
 TWO=2D0, FOUR=4D0, DELTA_T=1D-1, TEND=5D0, A=5D0
 TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)
! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=ZERO; U(NPDE+1,N)=A
 OPEN(FILE='PDE_ex03.out',UNIT=7)
 NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
 IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
 IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
 IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D0)
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF
! All completed. Solver is shut down.

```

```

 CASE (3)
 CLOSE(UNIT=7)
 EXIT
! Define initial data values.
 CASE (5)
 U(1,:) = EXP(-U(2,:)) / (TWO-EXP(-A))
 WRITE(7, "(F10.5)") T0
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I,:)
 END DO
! Define differential equations.
 CASE (6)
 D_PDE_1D_MG_C(1,1) = ONE
 D_PDE_1D_MG_R(1) = -D_PDE_1D_MG_U(1)
! Evaluate the approximate integral, for this t.
 V_1 = HALF * SUM((U(1,1:N-1) + U(1,2:N)) * &
 (U(2,2:N) - U(2,1:N-1)))
 D_PDE_1D_MG_Q(1) = V_1 * D_PDE_1D_MG_U(1)
! Define boundary conditions.
 CASE (7)
 IF(PDE_1D_MG_LEFT) THEN
! Evaluate the approximate integral, for this t.
! A second integral is needed at the edge.
 V_1 = HALF * SUM((U(1,1:N-1) + U(1,2:N)) * &
 (U(2,2:N) - U(2,1:N-1)))
 MID = HALF * (U(2,2:N) + U(2,1:N-1))
 V_2 = HALF * SUM(MID * EXP(-MID) * &
 (U(1,1:N-1) + U(1,2:N)) * (U(2,2:N) - U(2,1:N-1)))
 D_PDE_1D_MG_BETA = ZERO
 ELSE
 D_PDE_1D_MG_BETA = ZERO
 D_PDE_1D_MG_GAMMA = D_PDE_1D_MG_DUDX(1)
 END IF
 END SELECT
! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
 END DO
CONTAINS
 FUNCTION G(z,t)
 IMPLICIT NONE
 REAL(KIND(1d0)) Z, T, G
 G = FOUR * Z * (TWO - TWO * EXP(-A) + EXP(-T)) ** 2
 G = G / ((ONE - EXP(-A)) * (ONE - (ONE + TWO * A) * &
 EXP(-TWO * A)) * (1 - EXP(-A) + EXP(-T)))
 END FUNCTION
end program

```

## Example 4 - A Model in Cylindrical Coordinates

This example is from Blom and Zegeling (1994). The system models a reactor-diffusion problem:

$$T_z = r^{-1} \frac{\partial(\beta r T_r)}{\partial r} + \gamma \exp\left(\frac{T}{1 + \varepsilon T}\right)$$
$$T_r(0, z) = 0, T(1, z) = 0, z > 0$$
$$T(r, 0) = 0, 0 \leq r < 1$$
$$\beta = 10^{-4}, \gamma = 1, \varepsilon = 0.1$$

The axial direction  $z$  is treated as a time coordinate. The radius  $r$  is treated as the single space variable.

### Rationale: Example 4

This is a non-linear problem in cylindrical coordinates. Our example illustrates assigning  $m=1$  in Equation 2. We provide an optional argument that resets this value from its default,  $m=0$ .

Reverse communication is used to interface with the problem data.

```
program PDE_1D_MG_EX04
! Reactor-Diffusion problem in cylindrical coordinates.
 USE pde_1d_mg_int
 USE error_option_packet
 IMPLICIT NONE
 INTEGER, PARAMETER :: NPDE=1, N=41
 INTEGER IDO, I, NFRAMES
! Define array space for the solution.
 real(kind(1d0)) T(NPDE+1,N), Z0, ZOUT
 real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_Z=1D-1, &
 ZEND=1D0, ZMAX=1D0, BTA=1D-4, GAMA=1D0, EPS=1D-1
 TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)
! Define values that determine limits.
 CASE (1)
 Z0=ZERO
 ZOUT=DELTA_Z
 T(NPDE+1,1)=ZERO;T(NPDE+1,N)=ZMAX
 OPEN(FILE='PDE_ex04.out',UNIT=7)
 NFRAMES=NINT((ZEND+DELTA_Z)/DELTA_Z)
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 T(NPDE+1,1), T(NPDE+1,N), Z0, ZEND
 IOPT(1)=PDE_1D_MG_CYL_COORDINATES
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 IF(Z0 <= ZEND) THEN
 WRITE(7, "(F10.5)") ZOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") T(I, :)
 END DO
 END IF
 END SELECT
 END DO
```

```

 ZOUT=MIN(ZOUT+DELTA_Z,ZEND)
 IF(Z0 == ZEND)IDO=3
 END IF
! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT
! Define initial data values.
 CASE (5)
 T(1,:)=ZERO
 WRITE(7,"(F10.5)")Z0
 DO I=1,NPDE+1
 WRITE(7,"(4E15.5)")T(I,:)
 END DO
! Define differential equations.
 CASE (6)
 D_PDE_1D_MG_C(1,1)=ONE
 D_PDE_1D_MG_R(1)=BTA*D_PDE_1D_MG_DUDX(1)
 D_PDE_1D_MG_Q(1)= -GAMA*EXP(D_PDE_1D_MG_U(1)/&
 (ONE+EPS*D_PDE_1D_MG_U(1)))
! Define boundary conditions.
 CASE (7)
 IF(PDE_1D_MG_LEFT) THEN
 D_PDE_1D_MG_BETA=ONE; D_PDE_1D_MG_GAMMA=ZERO
 ELSE
 D_PDE_1D_MG_BETA=ZERO; D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)
 END IF
 END SELECT
! Reverse communication is used for the problem data.
! The optional derived type changes the internal model
! to use cylindrical coordinates.
 CALL PDE_1D_MG (Z0, ZOUT, IDO, T, IOPT=IOPT)
 END DO
 end program

```

## Example 5 - A Flame Propagation Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating mass density  $u(x, t)$  and temperature  $v(x, t)$ :

$$\begin{aligned}
u_t &= u_{xx} - uf(v) \\
v_t &= v_{xx} + uf(v), \\
\text{where } f(z) &= \gamma \exp(-\beta/z), \beta = 4, \gamma = 3.52 \times 10^6 \\
0 \leq x \leq 1, 0 \leq t \leq 0.006 \\
u(x, 0) &= 1, v(x, 0) = 0.2 \\
u_x = v_x &= 0, x = 0 \\
u_x = 0, v &= b(t), x = 1, \text{ where} \\
b(t) &= 1.2, \text{ for } t \geq 2 \times 10^{-4}, \text{ and} \\
&= 0.2 + 5 \times 10^3 t, \text{ for } 0 \leq t \leq 2 \times 10^{-4}
\end{aligned}$$

### Rationale: Example 5

This is a non-linear problem. The example shows the model steps for replacing the banded solver in the software with one of the user's choice. Reverse communication is used for the interface to the problem data and the linear solver. Following the computation of the matrix factorization in DL2CRB, we declare the system to be singular when the reciprocal of the condition number is smaller than the working precision. This choice is not suitable for all problems. Attention must be given to detecting a singularity when this option is used.

```

program PDE_1D_MG_EX05
! Flame propagation model
 USE pde_1d_mg_int
 USE ERROR_OPTION_PACKET
 USE Numerical_Libraries, ONLY :&
 dl2crb, dlfsrb
 IMPLICIT NONE

 INTEGER, PARAMETER :: NPDE=2, N=40, NEQ=(NPDE+1)*N
 INTEGER I, IDO, NFRAMES, IPVT(NEQ)

! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), T0, TOUT
! Define work space for the banded solver.
 real(kind(1d0)) WORK(NEQ), RCOND
 real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-4,&
 TEND=6D-3, XMAX=1D0, BTA=4D0, GAMA=3.52D6
 TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)

! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
 OPEN(FILE='PDE_ex05.out',UNIT=7)

```

```

 NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
 IOPT(1)=PDE_1D_MG_REV_COMM_FACTOR_SOLVE
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF

! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

! Define initial data values.
 CASE (5)
 U(1, :)=ONE; U(2, :)=2D-1
 WRITE(7, "(F10.5)") T0
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO

! Define differential equations.
 CASE (6)
 D_PDE_1D_MG_C=ZERO
 D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

 D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX

 D_PDE_1D_MG_Q(1)= D_PDE_1D_MG_U(1)*F(D_PDE_1D_MG_U(2))
 D_PDE_1D_MG_Q(2)= -D_PDE_1D_MG_Q(1)

! Define boundary conditions.
 CASE (7)
 IF(PDE_1D_MG_LEFT) THEN
 D_PDE_1D_MG_BETA=ZERO; D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
 ELSE
 D_PDE_1D_MG_BETA(1)=ONE
 D_PDE_1D_MG_GAMMA(1)=ZERO
 D_PDE_1D_MG_BETA(2)=ZERO
 IF(D_PDE_1D_MG_T >= 2D-4) THEN
 D_PDE_1D_MG_GAMMA(2)=12D-1
 ELSE
 D_PDE_1D_MG_GAMMA(2)=2D-1+5D3*D_PDE_1D_MG_T
 END IF
 D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_GAMMA(2) - &
 D_PDE_1D_MG_U(2)
 END IF

 CASE (8)

! Factor the banded matrix. This is the same solver used

```

```

! internally but that is not required. A user can substitute
! one of their own.
 call dl2crb (neq, d_pde_1d_mg_a, pde_1d_mg_lda, &
 pde_1d_mg_iband, pde_1d_mg_iband, d_pde_1d_mg_a, &
 pde_1d_mg_lda, ipvt, rcond, work)
 IF(rcond <= EPSILON(ONE)) pde_1d_mg_panic_flag = 1
 CASE(9)
! Solve using the factored banded matrix.
 call dlfsrb(neq, d_pde_1d_mg_a, pde_1d_mg_lda, &
 pde_1d_mg_iband, pde_1d_mg_iband, ipvt, &
 d_pde_1d_mg_rhs, 1, d_pde_1d_mg_sol)
 END SELECT

! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
 END DO
CONTAINS
 FUNCTION F(Z)
 IMPLICIT NONE
 REAL(KIND(1D0)) Z, F
 F=GAMA*EXP(-BTA/Z)
 END FUNCTION
end program

```

## Example 6 - A 'Hot Spot' Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating the temperature  $u(x, t)$ , of a reactant in a chemical system. The formula for  $h(z)$  is equivalent to their example.

$$u_t = u_{xx} + h(u),$$

$$\text{where } h(z) = \frac{R}{a\delta}(1 + a - z) \exp(-\delta(1/z - 1)),$$

$$a = 1, \delta = 20, R = 5$$

$$0 \leq x \leq 1, 0 \leq t \leq 0.29$$

$$u(x, 0) = 1$$

$$u_x = 0, x = 0$$

$$u = 1, x = 1$$

### Rationale: Example 6

This is a non-linear problem. The output shows a case where a rapidly changing front, or hot-spot, develops after a considerable way into the integration. This causes rapid change to the grid. An option sets the maximum order BDF formula from its default value of 2 to the theoretical stable maximum value of 5.

```

 USE pde_1d_mg_int
 USE error_option_packet
 IMPLICIT NONE

```

```

INTEGER, PARAMETER :: NPDE=1, N=80
INTEGER I, IDO, NFRAMES

! Define array space for the solution.
real(kind(1d0)) U(NPDE+1,N), T0, TOUT
real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-2, &
 TEND=29D-2, XMAX=1D0, A=1D0, DELTA=2D1, R=5D0
TYPE(D_OPTIONS) IOPT(2)
! Start loop to integrate and record solution values.
IDO=1
DO
 SELECT CASE (IDO)

! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
 OPEN(FILE='PDE_ex06.out',UNIT=7)
 NFRAMES=(TEND+DELTA_T)/DELTA_T
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
 IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
 IOPT(2)=5
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF
! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

! Define initial data values.
 CASE (5)
 U(1, :)=ONE
 WRITE(7, "(F10.5)") T0
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
! Define differential equations.
 CASE (6)
 D_PDE_1D_MG_C=ONE
 D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX
 D_PDE_1D_MG_Q= - H(D_PDE_1D_MG_U(1))

```

```

! Define boundary conditions.
 CASE (7)
 IF (PDE_1D_MG_LEFT) THEN
 D_PDE_1D_MG_BETA=ZERO
 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
 ELSE
 D_PDE_1D_MG_BETA=ZERO
 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)-ONE
 END IF
 END SELECT

! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
 END DO
CONTAINS
 FUNCTION H(Z)
 real(kind(ld0)) Z, H
 H=(R/(A*DELTA))* (ONE+A-Z)*EXP(-DELTA*(ONE/Z-ONE))
 END FUNCTION
end program

```

## Example 7 - Traveling Waves

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating the interaction of two waves,  $u(x, t)$  and  $v(x, t)$  moving in opposite directions. The waves meet and reduce in amplitude, due to the non-linear terms in the equation. Then they separate and travel onward, with reduced amplitude.

$$\begin{aligned}
 u_t &= -u_x - 100uv, \\
 v_t &= v_x - 100uv, \\
 -0.5 &\leq x \leq 0.5, 0 \leq t \leq 0.5 \\
 u(x, 0) &= 0.5(1 + \cos(10\pi x)), x \in [-0.3, -0.1], \text{ and} \\
 &= 0, \text{ otherwise,} \\
 v(x, 0) &= 0.5(1 + \cos(10\pi x)), x \in [0.1, 0.3], \text{ and} \\
 &= 0, \text{ otherwise,} \\
 u = v = 0 & \text{ at both ends, } t \geq 0
 \end{aligned}$$

### Rationale: Example 7

This is a non-linear system of first order equations.

```

 program PDE_1D_MG_EX07
! Traveling Waves
 USE pde_1d_mg_int
 USE error_option_packet
 IMPLICIT NONE

```

```

 INTEGER, PARAMETER :: NPDE=2, N=50
 INTEGER I, IDO, NFRAMES

! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), TEMP(N), T0, TOUT
 real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, &
 ONE=1D0, DELTA_T=5D-2, TEND=5D-1, PI
 TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)

! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=-HALF; U(NPDE+1,N)=HALF
 OPEN(FILE='PDE_ex07.out',UNIT=7)
 NFRAMES=(TEND+DELTA_T)/DELTA_T
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
 IOPT(1)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)
 IOPT(2)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
 IOPT(3)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-3)
 IOPT(4)=PDE_1D_MG_MAX_BDF_ORDER
 IOPT(5)=3

! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF

! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

! Define initial data values.
 CASE (5)
 TEMP=U(3, :)
 U(1, :)=PULSE(TEMP); U(2, :)=U(1, :)
 WHERE (TEMP < -3D-1 .or. TEMP > -1D-1) U(1, :)=ZERO
 WHERE (TEMP < 1D-1 .or. TEMP > 3D-1) U(2, :)=ZERO
 WRITE(7, "(F10.5)") T0
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO

```

```

! Define differential equations.
CASE (6)
 D_PDE_1D_MG_C=ZERO
 D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

 D_PDE_1D_MG_R=D_PDE_1D_MG_U
 D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_R(1)

 D_PDE_1D_MG_Q(1)= 100D0*D_PDE_1D_MG_U(1)*D_PDE_1D_MG_U(2)
 D_PDE_1D_MG_Q(2)= D_PDE_1D_MG_Q(1)

! Define boundary conditions.
CASE (7)
 D_PDE_1D_MG_BETA=ZERO;D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U

 END SELECT

! Reverse communication is used for the problem data.
CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
END DO
CONTAINS
FUNCTION PULSE(Z)
 real(kind(1d0)) Z(:), PULSE(SIZE(Z))
 PI=ACOS(-ONE)
 PULSE=HALF*(ONE+COS(10D0*PI*Z))
END FUNCTION
end program

```

## Example 8 - Black-Scholes

The value of a European “call option,”  $c(s, t)$ , with exercise price  $e$  and expiration date  $T$ , satisfies the “asset-or-nothing payoff”  $c(s, T) = s, s \geq e; = 0, s < e$ . Prior to expiration  $c(s, t)$  is estimated by the Black-Scholes differential equation

$$c_t + \frac{\sigma^2}{2} s^2 c_{ss} + rsc_s - rc \equiv c_t + \frac{\sigma^2}{2} (s^2 c_s)_s + (r - \sigma^2) sc_s - rc = 0$$

The parameters in the model are the risk-free interest rate,  $r$ , and the stock volatility,  $\sigma$ . The boundary conditions are  $c(0, t) = 0$  and  $c_s(s, t) \approx 1, s \rightarrow \infty$ . This development is described in Wilmott, *et al.* (1995), pages 41-57. There are explicit solutions for this equation based on the Normal Curve of Probability. The normal curve, and the solution itself, can be efficiently computed with the IMSL function ANORDF, IMSL (1994), page 186. With numerical integration the equation itself or the payoff can be readily changed to include other formulas,  $c(s, T)$ , and corresponding boundary conditions. We use

$$e = 100, r = 0.08, T - t = 0.25, \sigma^2 = 0.04, s_L = 0, \text{ and } s_R = 150$$

## Rationale: Example 8

This is a linear problem but with initial conditions that are discontinuous. It is necessary to use a positive time-smoothing value to prevent grid lines from crossing. We have used an absolute tolerance of  $10^{-3}$ . In \$US, this is one-tenth of a cent.

```
program PDE_1D_MG_EX08
! Black-Scholes call price
 USE pde_1d_mg_int
 USE error_option_packet
 IMPLICIT NONE

 INTEGER, PARAMETER :: NPDE=1, N=100
 INTEGER I, IDO, NFRAMES

! Define array space for the solution.
 real(kind(1d0)) U(NPDE+1,N), T0, TOUT, SIGSQ, XVAL
 real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0, &
 DELTA_T=25D-3, TEND=25D-2, XMAX=150, SIGMA=2D-1, &
 R=8D-2, E=100D0
 TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
 IDO=1
 DO
 SELECT CASE (IDO)

! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=DELTA_T
 U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
 OPEN(FILE='PDE_ex08.out',UNIT=7)
 NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
 WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES, &
 U(NPDE+1,1), U(NPDE+1,N), T0, TEND
 SIGSQ=SIGMA**2
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
 IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
 IOPT(2)=5
 IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,5D-3)
 IOPT(4)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
 IOPT(5)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT
 IF(T0 <= TEND) THEN
 WRITE(7, "(F10.5)") TOUT
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I, :)
 END DO
 TOUT=MIN(TOUT+DELTA_T, TEND)
 IF(T0 == TEND) IDO=3
 END IF
```

```

! All completed. Solver is shut down.
 CASE (3)
 CLOSE(UNIT=7)
 EXIT

! Define initial data values.
 CASE (5)
 U(1,:) = MAX(U(NPDE+1,:) - E, ZERO) ! Vanilla European Call
 U(1,:) = U(NPDE+1,:) ! Asset-or-nothing Call
 WHERE(U(1,:) <= E) U(1,:) = ZERO ! on these two lines
 WRITE(7, "(F10.5)") T0
 DO I=1, NPDE+1
 WRITE(7, "(4E15.5)") U(I,:)
 END DO

! Define differential equations.
 CASE (6)
 XVAL = D_PDE_1D_MG_X
 D_PDE_1D_MG_C = ONE
 D_PDE_1D_MG_R = D_PDE_1D_MG_DUDX * XVAL ** 2 * SIGSQ * HALF
 D_PDE_1D_MG_Q = -(R - SIGSQ) * XVAL * D_PDE_1D_MG_DUDX + R * D_PDE_1D_MG_U

! Define boundary conditions.
 CASE (7)
 IF (PDE_1D_MG_LEFT) THEN
 D_PDE_1D_MG_BETA = ZERO
 D_PDE_1D_MG_GAMMA = D_PDE_1D_MG_U
 ELSE

 D_PDE_1D_MG_BETA = ZERO
 D_PDE_1D_MG_GAMMA = D_PDE_1D_MG_DUDX(1) - ONE
 END IF
 END SELECT

! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
 END DO

 end program

```

## Example 9 - Electrodynamics, Parameters Studied with MPI




---

For a detailed description of MPI Requirements see “[Dense Matrix Parallelism Using MPI](#)” in Chapter 10 of this manual.

---

This example, described above in Example 1, is from Blom and Zegeling (1994). The system parameters  $\varepsilon$ ,  $p$ , and  $\eta$ , are varied, using uniform random numbers. The intervals studied are  $0.1 \leq \varepsilon \leq 0.2$ ,  $0.1 \leq p \leq 0.2$ , and  $10 \leq \eta \leq 20$ . Using  $N = 21$  grid values and other program options, the elapsed time, parameter values, and the value  $v(x, t)|_{x=1, t=4}$  are sent to the root node. This information is written on a file. The final summary includes the minimum value of

$$v(x,t)|_{x=1,t=4}$$

and the maximum and average time per integration, per node.

### Rationale: Example 9

This is a non-linear simulation problem. Using at least two integrating processors and MPI allows more values of the parameters to be studied in a given time than with a single processor. This code is valuable as a study guide when an application needs to estimate timing and other output parameters. The simulation time is controlled at the root node. An integration is started, after receiving results, within the first `SIM_TIME` seconds. The elapsed time will be longer than `SIM_TIME` by the slowest processor's time for its last integration.

```

program PDE_1D_MG_EX09
! Electrodynamics Model, parameter study.
 USE PDE_1d_mg_int
 USE MPI_SETUP_INT
 USE RAND_INT
 USE SHOW_INT
 IMPLICIT NONE
 INCLUDE "mpif.h"
 INTEGER, PARAMETER :: NPDE=2, N=21
 INTEGER I, IDO, IERROR, CONTINUE, STATUS(MPI_STATUS_SIZE)
 INTEGER, ALLOCATABLE :: COUNTS(:)
! Define array space for the solution.
 real(kind(1d0)) :: U(NPDE+1,N), T0, TOUT
 real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=10D0, TEND=4D0
! SIM_TIME is the number of seconds to run the simulation.
 real(kind(1d0)) :: EPS, P, ETA, Z, TWO=2D0, THREE=3D0, SIM_TIME=60D0
 real(kind(1d0)) :: TIMES, TIMEE, TIMEL, TIME, TIME_SIM, V_MIN, &
 DATA(5)
 real(kind(1d0)), ALLOCATABLE :: AV_TIME(:), MAX_TIME(:)
 TYPE(D_OPTIONS) IOPT(4), SHOW_IOPT(2)
 TYPE(S_OPTIONS) SHOW_INTOPT(2)
 MP_NPROCS=MP_SETUP(1)
 MPI_NODE_PRIORITY=(/ (I-1, I=1, MP_NPROCS) /)
! If NP_NPROCS=1, the program stops. Change
! MPI_ROOT_WORKS=.TRUE. if MP_NPROCS=1.
 MPI_ROOT_WORKS=.FALSE.
 IF(.NOT. MPI_ROOT_WORKS .and. MP_NPROCS == 1) STOP
 ALLOCATE(AV_TIME(MP_NPROCS), MAX_TIME(MP_NPROCS), COUNTS(MP_NPROCS))
! Get time start for simulation timing.
 TIME=MPI_WTIME()
 IF(MP_RANK == 0) OPEN(FILE='PDE_ex09.out',UNIT=7)
SIMULATE: DO
! Pick random parameter values.
 EPS=1D-1*(ONE+rand(EPS))
 P=1D-1*(ONE+rand(P))
 ETA=10D0*(ONE+rand(ETA))
! Start loop to integrate and communicate solution times.
 IDO=1
! Get time start for each new problem.
 DO

```

```

 IF(.NOT. MPI_ROOT_WORKS .and. MP_RANK == 0) EXIT
 SELECT CASE (IDO)
! Define values that determine limits.
 CASE (1)
 T0=ZERO
 TOUT=1D-3
 U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
 IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
 IOPT(2)=5
 IOPT(3)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,1D-2)
 IOPT(4)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)

 TIMES=MPI_WTIME()
! Update to the next output point.
! Write solution and check for final point.
 CASE (2)
 T0=TOUT;TOUT=TOUT*DELTA_T
 IF(T0 >= TEND) IDO=3
 TOUT=MIN(TOUT, TEND)
! All completed. Solver is shut down.
 CASE (3)
 TIMEE=MPI_WTIME()
 EXIT
! Define initial data values.
 CASE (5)
 U(1,:)=1D0;U(2,:)=0D0
! Define differential equations.
 CASE (6)
D_PDE_1D_MG_C=0D0;D_PDE_1D_MG_C(1,1)=1D0;D_PDE_1D_MG_C(2,2)=1D0
 D_PDE_1D_MG_R=P*D_PDE_1D_MG_DUDX
D_PDE_1D_MG_R(1)=D_PDE_1D_MG_R(1)*EPS
 Z=ETA*(D_PDE_1D_MG_U(1)-D_PDE_1D_MG_U(2))/THREE
 D_PDE_1D_MG_Q(1)=EXP(Z)-EXP(-TWO*Z)
 D_PDE_1D_MG_Q(2)=-D_PDE_1D_MG_Q(1)
! Define boundary conditions.
 CASE (7)
 IF(PDE_1D_MG_LEFT) THEN
 D_PDE_1D_MG_BETA(1)=1D0;D_PDE_1D_MG_BETA(2)=0D0
D_PDE_1D_MG_GAMMA(1)=0D0;D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_U(2)
 ELSE
 D_PDE_1D_MG_BETA(1)=0D0;D_PDE_1D_MG_BETA(2)=1D0
 D_PDE_1D_MG_GAMMA(1)=D_PDE_1D_MG_U(1)- &
 1D0;D_PDE_1D_MG_GAMMA(2)=0D0
 END IF
 END SELECT
! Reverse communication is used for the problem data.
 CALL PDE_1D_MG (T0, TOUT, IDO, U)
 END DO
 TIMEL=TIMEE-TIMES
 DATA=(/EPS, P, ETA, U(2,N), TIMEL/)
 IF(MP_RANK > 0) THEN
! Send parameters and time to the root.
 CALL MPI_SEND(DATA, 5, MPI_DOUBLE_PRECISION,0, MP_RANK, &
 MP_LIBRARY_WORLD, IERROR)
! Receive back a "go/stop" flag.

```

```

 CALL MPI_RECV(CONTINUE, 1, MPI_INTEGER, 0, MPI_ANY_TAG, &
 MP_LIBRARY_WORLD, STATUS, IERROR)
! If root notes that time is up, it sends node a quit flag.
 IF(CONTINUE == 0) EXIT SIMULATE
 ELSE
! If root is working, record its result and then stand ready
! for other nodes to send.
 IF(MPI_ROOT_WORKS) WRITE(7,*) MP_RANK, DATA
! If all nodes have reported, then quit.
 IF(COUNT(MPI_NODE_PRIORITY >= 0) == 0) EXIT SIMULATE
! See if time is up. Some nodes still must report.
 IF(MPI_WTIME()-TIME >= SIM_TIME) THEN
 CONTINUE=0
 ELSE
 CONTINUE=1
 END IF
! Root receives simulation data and finds which node sent it.
 IF(MP_NPROCS > 1) THEN
 CALL MPI_RECV(DATA, 5, MPI_DOUBLE_PRECISION, &
 MPI_ANY_SOURCE, MPI_ANY_TAG, MP_LIBRARY_WORLD, &
 STATUS, IERROR)
 WRITE(7,*) STATUS(MPI_SOURCE), DATA
! If time at the root has elapsed, nodes receive signal to stop.
! Send the reporting node the "go/stop" flag.
! Mark if a node has been stopped.
 CALL MPI_SEND(CONTINUE, 1, MPI_INTEGER, &
 STATUS(MPI_SOURCE), &0, MP_LIBRARY_WORLD, IERROR)
 IF (CONTINUE == 0) MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1) &
 == MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1)-1
 END IF
 IF (CONTINUE == 0) MPI_NODE_PRIORITY(1)=-1
 END IF
END DO SIMULATE
IF(MP_RANK == 0) THEN
 ENDFILE(UNIT=7);REWIND(UNIT=7)
! Read the data. Find extremes and averages.
 MAX_TIME=ZERO;AV_TIME=ZERO;COUNTS=0;V_MIN=HUGE(ONE)
 DO
 READ(7,*, END=10) I, DATA
 COUNTS(I+1)=COUNTS(I+1)+1
 AV_TIME(I+1)=AV_TIME(I+1)+DATA(5)
 IF(MAX_TIME(I+1) < DATA(5)) MAX_TIME(I+1)=DATA(5)
 V_MIN=MIN(V_MIN, DATA(4))
 END DO
10 CONTINUE
 CLOSE(UNIT=7)
! Set printing Index to match node numbering.
 SHOW_IOPT(1)= SHOW_STARTING_INDEX_IS
 SHOW_IOPT(2)=0
 SHOW_INTOPT(1)=SHOW_STARTING_INDEX_IS
 SHOW_INTOPT(2)=0
 CALL SHOW(MAX_TIME,"Maximum Integration Time, per
process:",IOPT=SHOW_IOPT)
 AV_TIME=AV_TIME/MAX(1,COUNTS)
 CALL SHOW(AV_TIME,"Average Integration Time, per
process:",IOPT=SHOW_IOPT)

```

```

 CALL SHOW(COUNTS,"Number of Integrations",IOPT=SHOW_INTOPT)
 WRITE(*,"(1x,A,F6.3)") "Minimum value for v(x,t), at x=1,t=4:
",V_MIN
 END IF
 MP_NPROCS=MP_SETUP("Final")
 end program

```

---

## MMOLCH



Solves a system of partial differential equations of the form  $u_t = f(x, t, u, u_x, u_{xx})$  using the method of lines. The solution is represented with cubic Hermite polynomials.

---

**Note:** .MMOLCH replaces deprecated routine [MOLCH](#).

---

### Required Arguments

**IDO** — Flag indicating the state of the computation. (Input/Output)

**IDO State**

- 1 Initial entry
- 2 Normal reentry
- 3 Final call, release workspace

Normally, the initial call is made with `IDO = 1`. The routine then sets `IDO = 2`, and this value is then used for all but the last call that is made with `IDO = 3`.

**FCNUT** — User-supplied subroutine to evaluate the function  $u_t$ . The usage is `CALL FCNUT (NPDES, X, T, U, UX, UXX, UT [, ...])` where

### Required Arguments

- NPDES** — Number of equations. (Input)
- X** — Space variable,  $x$ . (Input)
- T** — Time variable,  $t$ . (Input)
- U** — Array of length `NPDES` containing the dependent variable values,  $u$ . (Input)
- UX** — Array of length `NPDES` containing the first derivatives  $u_x$ . (Input)
- UXX** — Array of length `NPDES` containing the second derivative  $u_{xx}$ . (Input)

UT — Array of length NPDES containing the computed derivatives,  $u_t$ .  
(Output)

### Optional Arguments

FCN\_DATA — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

FCNUT must be declared EXTERNAL in the calling program.

**FCNBC** — User-supplied subroutine to evaluate the boundary conditions. The boundary conditions accepted by MMOLCH are  $\alpha_k u_k + \beta_k u_x = \gamma_k(t)$ . Users must supply the values  $\alpha_k$  and  $\beta_k$ , and functions  $\gamma_k(t)$ . The usage is `CALL FCNBC (NPDES, X, T, ALPHA, BETA, GAMMA [, ...])`, where

### Required Arguments

NPDES — Number of equations. (Input)

X — Space variable,  $x$ . This value directs which boundary condition to compute. (Input)

T — Time variable,  $t$ . (Input)

ALPHA — Array of length NPDES containing the  $\alpha_k$  values. (Output)

BETA — Array of length NPDES containing the  $\beta_k$  values. (Output)

GAMMA — Array of length NPDES containing the values of  $\gamma_k(t)$ . (Output)

### Optional Arguments

FCN\_DATA — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

FCNBC must be declared EXTERNAL in the calling program.

**T** — Independent variable,  $t$ . (Input/Output)

On input, T supplies the initial time,  $t_0$ . On output, T is set to the value to which the integration has been updated. Normally, this new value is TEND.

**TEND** — Value of  $t = tend$  at which the solution is desired. (Input)

**XBREAK** — Array of length NX containing the break points for the cubic Hermite splines used in the  $x$  discretization. (Input)

The points in the array XBREAK must be strictly increasing. The values XBREAK(1) and XBREAK(NX) are the endpoints of the interval.

**Y** — Array of size *NPDES* by *NX* containing the solution. (Input/Output)

The array *Y* contains the solution as  $Y(k, i) = u_k(x, t)$  at  $x = \text{XBREAK}(i)$ . On input, *Y* contains the initial values. On output, *Y* contains the computed solution.

The user can optionally supply the derivative values,  $u_x(x, t_0)$ . The user allocates twice the space for *Y* to pass this information. The optional derivative information is input as

$$Y(k, i + NX) = \frac{\partial u_k}{\partial x}(x, t_0)$$

at  $x = \text{XBREAK}(i)$ . The array *Y* contains the optional derivative values as output:

$$Y(k, i + NX) = \frac{\partial u_k}{\partial x}(x, \text{tend})$$

at  $x = \text{XBREAK}(i)$ . To signal that this information is provided, set *INPDER* = 1.

## Optional Arguments

**NPDES** — Number of differential equations. (Input)

Default: *NPDES* = size(*Y*,1).

**NX** — Number of mesh points or lines. (Input)

Default: *NX* = size(*XBREAK*,1).

**TOL** — Differential equation error tolerance. (Input)

An attempt is made to control the local error in such a way that the global error is proportional to *TOL*.

Default: *TOL* = 100. \* machine precision.

**HINIT** — Initial step size in the *t* integration. (Input)

This value must be nonnegative. If *HINIT* is zero, an initial step size of  $0.001|\text{tend} - t_0|$  will be arbitrarily used. The step will be applied in the direction of integration.

Default: *HINIT* = 0.0.

**INPDER** — Set *INPDER* = 1 if the user is supplying the derivative values,  $u_x(x, t_0)$ , in the array *Y*. (Input)

Default: *INPDER* = 0.

**FCN\_DATA** — A derived type, *s\_fcn\_data*, which may be used to pass additional information to/from the user-supplied function. (Input/Output)

The derived type, *s\_fcn\_data*, is defined as:

```
type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type
```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type.

## FORTRAN 90 Interface

Generic: `CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y [, ...])`

Specific: The specific interface names are `S_MMOLCH` and `D_MMOLCH`.

## Description

Let  $M = \text{NPDES}$ ,  $N = \text{NX}$  and  $x_i = \text{XBREAK}(I)$ . The routine `MMOLCH` uses the method of lines to solve the partial differential equation system

$$\frac{\partial u_k}{\partial t} = f_k \left( x, t, u_1, \dots, u_M, \frac{\partial u_1}{\partial x}, \dots, \frac{\partial u_M}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \dots, \frac{\partial^2 u_M}{\partial x^2} \right)$$

with the initial conditions

$$u_k = u_k(x) \quad \text{at } t = t_0$$

and the boundary conditions

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k(t) \text{ at } x = x_1 \text{ and at } x = x_N$$

for  $k = 1, \dots, M$ .

Cubic Hermite polynomials are used in the  $x$  variable approximation so that the trial solution is expanded in the series

$$\hat{u}_k(x, t) = \sum_{i=1}^N (a_{i,k}(t) \phi_i(x) + b_{i,k}(t) \psi_i(x))$$

where  $\phi_i(x)$  and  $\psi_i(x)$  are the standard basis functions for the cubic Hermite polynomials with the knots  $x_1 < x_2 < \dots < x_N$ . These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\begin{aligned} \phi_i(x_l) &= \delta_{il} & \psi_i(x_l) &= 0 \\ \frac{d\phi_i}{dx}(x_l) &= 0 & \frac{d\psi_i}{dx}(x_l) &= \delta_{il} \end{aligned}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equations at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3-\sqrt{3}}{6}(x_{j+1} - x_j)$$

$$p_{2j} = x_j + \frac{3+\sqrt{3}}{6}(x_{j+1} + x_j)$$

for  $j = 1, \dots, N$ . The collocation approximation to the differential equation is

$$\sum_{i=1}^N \frac{da_{i,k}}{dt} \phi_i(p_j) + \frac{db_{i,k}}{dt} \psi_i(p_j) = f_k(p_j, t, \hat{u}_1(p_j), \dots, \hat{u}_M(p_j), \dots, (\hat{u}_1)_{xx}(p_j), \dots, (\hat{u}_M)_{xx}(p_j))$$

for  $k = 1, \dots, M$  and  $j = 1, \dots, 2(N-1)$ .

This is a system of  $2M(N-1)$  ordinary differential equations in  $2MN$  unknown coefficient

functions,  $a_{i,k}$  and  $b_{i,k}$ . This system can be written in the matrix-vector form as  $A \frac{dc}{dt} = F(t, c)$

with  $c(t_0) = c_0$  where  $c$  is a vector of coefficients of length  $2MN$  and  $c_0$  holds the initial values of the coefficients. The last  $2M$  equations are obtained from the boundary conditions.

If  $\alpha_k = \beta_k = 0$ , it is assumed that no boundary condition is desired for the  $k$ -th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

The input/output array  $Y$  contains the values of the  $a_{i,k}$ . The initial values of the  $b_{i,k}$  are obtained by using the IMSL cubic spline routine `CSINT` (see [Chapter 3, Interpolation and Approximation](#)) to construct functions

$$\hat{u}_k(x, t_0)$$

such that

$$\hat{u}_k(x_i, t_0) = a_{i,k}$$

The IMSL routine `CSDER`, (see [Chapter 3, Interpolation and Approximation](#)), is used to approximate the values

$$\frac{d\hat{u}_k}{dx}(x_i, t_0) \equiv b_{i,k}$$

If `INPDER = 1`, the user should provide the initial values of  $b_{i,k}$ .

The order of matrix  $A$  is  $2MN$  and its maximum bandwidth is  $6M-1$ . The band structure of the Jacobian of  $F$  with respect to  $c$  is the same as the band structure of  $A$ . This system is solved using a modified version of `IVPAG`. Numerical Jacobians are used exclusively. The algorithm is unchanged. Gear's BDF method is used as the default because the system is typically stiff. For more details, see Sewell (1982).

We now present three examples of PDEs that illustrate how users can interface their problems with IMSL PDE solving software. The examples are small and not indicative of the complexities that most practitioners will face in their applications. A set of seven sample application problems, some of them with more than one equation, is given in Sincovec and Madsen (1975). Two further examples are given in Madsen and Sincovec (1979).

## Comments

Informational errors

Type Code

|   |   |                                                                                                                                                                         |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | After some initial success, the integration was halted by repeated error test failures.                                                                                 |
| 4 | 2 | On the next step, $x + h$ will equal $x$ . Either <code>TOL</code> is too small or the problem is stiff.                                                                |
| 4 | 3 | After some initial success, the integration was halted by a test on <code>TOL</code> .                                                                                  |
| 4 | 4 | Integration was halted after failing to pass the error test even after reducing the step size by a factor of $1.0E + 10$ . <code>TOL</code> may be too small.           |
| 4 | 5 | Integration was halted after failing to achieve corrector convergence even after reducing the step size by a factor of $1.0E + 10$ . <code>TOL</code> may be too small. |

## Example 1

The normalized linear diffusion PDE,  $u_t = u_{xx}$ ,  $0 \leq x \leq 1$ ,  $t > 0$ , is solved. The initial values are  $u(x, 0) = u_0 = 1$ . There is a “zero-flux” boundary condition at  $x = 1$ , namely  $u_x(1, t) = 0$ , ( $t > 0$ ). The boundary value of  $u(0, t)$  is abruptly changed from  $u_0$  to the value 0, for  $t > 0$ .

When the boundary conditions are discontinuous, or incompatible with the initial conditions such as in this example, it may be important to use double precision.

```

USE MMOLCH_INT
USE WRRRN_INT
IMPLICIT NONE

INTEGER, PARAMETER :: NPDES=1, NX=8

INTEGER :: I, IDO, J, NSTEP
REAL :: HINIT, T, TEND, TOL
REAL :: XBREAK(NX), Y(NPDES,NX), UO
CHARACTER :: TITLE*19

EXTERNAL FCNBC, FCNUT
! SET BREAKPOINTS AND INITIAL CONDITIONS
UO = 1.0
DO I=1,NX
 XBREAK(I) = FLOAT(I-1)/FLOAT(NX-1)
 Y(1,I) = UO

```

```

END DO
!
! SET PARAMETERS FOR MMOLCH
TOL = 10.e-4
HINIT = 0.01*TOL
T = 0.0
IDO = 1
NSTEP = 10
DO J=1,NSTEP
 TEND = FLOAT(J)/FLOAT(NSTEP)
!
! SOLVE THE PROBLEM
 CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, TOL=TOL, &
 HINIT=HINIT)
!
! PRINT RESULTS
 WRITE (TITLE,'(A,F4.2)') 'Solution at T =', TEND
 CALL WRRRN (TITLE, Y)
END DO
!
! LAST CALL, TO RELEASE WORKSPACE
IDO = 3
CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, TOL=TOL, &
 HINIT=HINIT)
STOP
END

SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)

INTEGER NPDES
REAL X, T, U(*), UX(*), UXX(*), UT(*)
!
! DEFINE THE PDE
UT(1) = UXX(1)
RETURN
END

SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAM)

INTEGER NPDES
REAL X, T, ALPHA(*), BTA(*), GAM(*)
!
! DEFINE THE BOUNDARY CONDITIONS
IF (X .EQ. 0.0) THEN
!
! THESE ARE FOR X=0
 ALPHA(1) = 1.0
 BTA(1) = 0.0
 GAM(1) = 0.0
!
! THESE ARE FOR X=1
ELSE
 ALPHA(1) = 0.0
 BTA(1) = 1.0
 GAM(1) = 0.0
END IF
RETURN
END

```

## Output

```

 Solution at T =0.10
 1 2 3 4 5 6 7 8
0.0000 0.2507 0.4771 0.6617 0.7972 0.8857 0.9341 0.9493

```

```

 Solution at T =0.20
 1 2 3 4 5 6 7 8
0.0000 0.1762 0.3424 0.4893 0.6100 0.6992 0.7538 0.7721

 Solution at T =0.30
 1 2 3 4 5 6 7 8
0.0000 0.1356 0.2642 0.3793 0.4751 0.5471 0.5916 0.6067

 Solution at T =0.40
 1 2 3 4 5 6 7 8
0.0000 0.1057 0.2060 0.2960 0.3711 0.4276 0.4626 0.4745

 Solution at T =0.50
 1 2 3 4 5 6 7 8
0.0000 0.0825 0.1610 0.2313 0.2900 0.3341 0.3616 0.3708

 Solution at T =0.60
 1 2 3 4 5 6 7 8
0.0000 0.0645 0.1258 0.1808 0.2267 0.2612 0.2826 0.2899

 Solution at T =0.70
 1 2 3 4 5 6 7 8
0.0000 0.0504 0.0983 0.1413 0.1772 0.2041 0.2209 0.2266

 Solution at T =0.80
 1 2 3 4 5 6 7 8
0.0000 0.0394 0.0769 0.1105 0.1385 0.1597 0.1728 0.1772

 Solution at T =0.90
 1 2 3 4 5 6 7 8
0.0000 0.0309 0.0602 0.0865 0.1084 0.1249 0.1352 0.1387

 Solution at T =1.00
 1 2 3 4 5 6 7 8
0.0000 0.0242 0.0471 0.0677 0.0849 0.0979 0.1059 0.1086

```

## Additional Examples

### Example 2

In this example, using MMOLCH, we solve the linear normalized diffusion PDE  $u_t = u_{xx}$  but with an optional usage that provides values of the derivatives,  $u_x$ , of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is  $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$ ,  $n > 1$ . The boundary conditions are “zero flux” conditions  $u_x(0, t) = u_x(1, t) = 0$  for  $t > 0$ .

```

USE MMOLCH_INT
USE CONST_INT
USE WRRRN_INT
USE PGOPT_INT
IMPLICIT NONE

INTEGER, PARAMETER :: NPDES=1, NX=10

```

```

INTEGER :: I, IDO, J, NSTEP, N, IPAGE
REAL :: HINIT, T, TEND, TOL, XBREAK(NX)
REAL :: Y(NPDES,2*NX), PI, ARG1
CHARACTER :: TITLE*36

EXTERNAL FCNBC, FCNUT
REAL FLOAT

N = 5
PI = CONST('pi')
DO I=1,NX
 XBREAK(I) = FLOAT(I-1)/FLOAT(NX-1)
 ARG1 = (2.*N-1)*PI
!
! SET FUNCTION VALUES
 Y(1,I) = 1. + COS(ARG1*XBREAK(I))
!
! SET FIRST DERIVATIVE VALUES
 Y(1,I+NX) = -ARG1*SIN(ARG1*XBREAK(I))
END DO
!
! SET PARAMETERS FOR MMOLCH
TOL = 10.0e-4
HINIT = 0.01*TOL
!
! OUTPUT AT STEPS OF 0.001
TEND = 0.
T = 0.0
IDO = 1
NSTEP = 10
DO J=1,NSTEP
 TEND = TEND + 0.001
!
! SOLVE THE PROBLEM
 CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, NPDES=NPDES, &
 NX=NX, HINIT=HINIT, TOL=TOL, INPDER=1)
!
! PRINT RESULTS
 IPAGE = 70
 CALL PGOPT(-1, IPAGE)
 WRITE (TITLE,'(A,F5.3)') 'Solution and derivatives at T =', T
 CALL WRRRN (TITLE, Y)
END DO
!
! LAST CALL, TO RELEASE WORKSPACE
IDO = 3
CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, NPDES=NPDES, &
 NX=NX, HINIT=HINIT, TOL=TOL, INPDER=1)
END

SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)

INTEGER NPDES
REAL X, T, U(*), UX(*), UXX(*), UT(*)
!
! DEFINE THE PDE
UT(1) = UXX(1)
RETURN
END

SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAM)

INTEGER NPDES
REAL X, T, ALPHA(*), BTA(*), GAM(*)
!
! DEFINE THE BOUNDARY CONDITIONS

```

```

ALPHA(1) = 0.0
BTA(1) = 1.0
GAM(1) = 0.0
RETURN
END

```

## Output

```

 Solution and derivatives at T =0.001
 1 2 3 4 5 6 7 8 9
1.482 0.518 1.482 0.518 1.482 0.518 1.482 0.518 1.482

 10 11 12 13 14 15 16 17 18
0.518 0.000 0.000 0.000 0.000 -0.000 0.000 -0.000 0.000

 19 20
-0.000 -0.000

```

```

 Solution and derivatives at T =0.002
 1 2 3 4 5 6 7 8 9
1.235 0.765 1.235 0.765 1.235 0.765 1.235 0.765 1.235

 10 11 12 13 14 15 16 17 18
0.765 0.000 0.000 0.000 0.000 -0.000 0.000 -0.000 0.000

 19 20
-0.000 0.000

```

```

 Solution and derivatives at T =0.003
 1 2 3 4 5 6 7 8 9
1.114 0.886 1.114 0.886 1.114 0.886 1.114 0.886 1.114

 10 11 12 13 14 15 16 17 18
0.886 0.000 0.000 0.000 0.000 -0.000 0.000 -0.000 0.000

 19 20
-0.000 -0.000

```

```

 Solution and derivatives at T =0.004
 1 2 3 4 5 6 7 8 9
1.055 0.945 1.055 0.945 1.055 0.945 1.055 0.945 1.055

 10 11 12 13 14 15 16 17 18
0.945 0.000 0.000 0.000 0.000 -0.000 -0.000 0.000 0.000

 19 20
-0.000 -0.000

```

```

 Solution and derivatives at T =0.005
 1 2 3 4 5 6 7 8 9
1.027 0.973 1.027 0.973 1.027 0.973 1.027 0.973 1.027

 10 11 12 13 14 15 16 17 18
0.973 0.000 -0.000 0.000 0.000 0.000 -0.000 -0.000 0.000

```

19 20  
-0.000 -0.000

Solution and derivatives at T =0.006

|       |       |       |       |        |       |       |        |       |
|-------|-------|-------|-------|--------|-------|-------|--------|-------|
| 1     | 2     | 3     | 4     | 5      | 6     | 7     | 8      | 9     |
| 1.013 | 0.987 | 1.013 | 0.987 | 1.013  | 0.987 | 1.013 | 0.987  | 1.013 |
| 10    | 11    | 12    | 13    | 14     | 15    | 16    | 17     | 18    |
| 0.987 | 0.000 | 0.000 | 0.000 | -0.000 | 0.000 | 0.000 | -0.000 | 0.000 |

19 20  
-0.000 -0.000

Solution and derivatives at T =0.007

|       |       |       |       |        |       |       |        |        |
|-------|-------|-------|-------|--------|-------|-------|--------|--------|
| 1     | 2     | 3     | 4     | 5      | 6     | 7     | 8      | 9      |
| 1.006 | 0.994 | 1.006 | 0.994 | 1.006  | 0.994 | 1.006 | 0.994  | 1.006  |
| 10    | 11    | 12    | 13    | 14     | 15    | 16    | 17     | 18     |
| 0.994 | 0.000 | 0.000 | 0.000 | -0.000 | 0.000 | 0.000 | -0.000 | -0.000 |

19 20  
-0.000 -0.000

Solution and derivatives at T =0.008

|       |       |       |       |        |        |       |       |        |
|-------|-------|-------|-------|--------|--------|-------|-------|--------|
| 1     | 2     | 3     | 4     | 5      | 6      | 7     | 8     | 9      |
| 1.003 | 0.997 | 1.003 | 0.997 | 1.003  | 0.997  | 1.003 | 0.997 | 1.003  |
| 10    | 11    | 12    | 13    | 14     | 15     | 16    | 17    | 18     |
| 0.997 | 0.000 | 0.000 | 0.000 | -0.000 | -0.000 | 0.000 | 0.000 | -0.000 |

19 20  
-0.000 -0.000

Solution and derivatives at T =0.009

|       |       |       |       |        |        |        |       |        |
|-------|-------|-------|-------|--------|--------|--------|-------|--------|
| 1     | 2     | 3     | 4     | 5      | 6      | 7      | 8     | 9      |
| 1.002 | 0.998 | 1.002 | 0.998 | 1.002  | 0.998  | 1.002  | 0.998 | 1.002  |
| 10    | 11    | 12    | 13    | 14     | 15     | 16     | 17    | 18     |
| 0.998 | 0.000 | 0.000 | 0.000 | -0.000 | -0.000 | -0.000 | 0.000 | -0.000 |

19 20  
-0.000 0.000

Solution and derivatives at T =0.010

|       |       |       |       |        |        |        |       |        |
|-------|-------|-------|-------|--------|--------|--------|-------|--------|
| 1     | 2     | 3     | 4     | 5      | 6      | 7      | 8     | 9      |
| 1.001 | 0.999 | 1.001 | 0.999 | 1.001  | 0.999  | 1.001  | 0.999 | 1.001  |
| 10    | 11    | 12    | 13    | 14     | 15     | 16     | 17    | 18     |
| 0.999 | 0.000 | 0.000 | 0.000 | -0.000 | -0.000 | -0.000 | 0.000 | -0.000 |

19 20  
-0.000 -0.000

### Example 3

In this example, we consider the linear normalized hyperbolic PDE,  $u_{tt} = u_{xx}$ , the “vibrating string” equation. This naturally leads to a system of first order PDEs. Define a new dependent variable  $u_t = v$ . Then,  $v_t = u_{xx}$  is the second equation in the system. We take as initial data  $u(x, 0) = \sin(\pi x)$  and  $u_t(x, 0) = v(x, 0) = 0$ . The ends of the string are fixed so  $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$ . The exact solution to this problem is  $u(x, t) = \sin(\pi x) \cos(\pi t)$ . Residuals are computed at the output values of  $t$  for  $0 < t \leq 2$ . Output is obtained at 200 steps in increments of 0.01.

Even though the sample code `MMOLCH` gives satisfactory results for this PDE, users should be aware that for *nonlinear problems*, “shocks” can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pages 488-490, for an introductory discussion of shocks in hyperbolic systems.

```
USE MMOLCH_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE

INTEGER, PARAMETER :: NPDES=2, NX=10
INTEGER :: I, IDO, J, NOUT, NSTEP
REAL :: HINIT, T, TEND, TOL, XBREAK(NX)
REAL :: Y(NPDES,2*NX), PI, ERROR, ERRU
CHARACTER :: TITLE*36

EXTERNAL FCNBC, FCNUT
REAL FLOAT
CALL UMACH (2,NOUT)

! SET BREAKPOINTS AND INITIAL CONDITIONS
PI = CONST('pi')
DO I=1,NX
 XBREAK(I) = FLOAT(I-1)/FLOAT(NX-1)
! SET FUNCTION VALUES
 Y(1,I) = SIN(PI*XBREAK(I))
 Y(2,I) = 0.
! SET FIRST DERIVATIVE VALUES
 Y(1,I+NX) = PI*COS(PI*XBREAK(I))
 Y(2,I+NX) = 0.
END DO

! SET PARAMETERS FOR MMOLCH
TOL = 10.0e-4
HINIT = 0.01*TOL

! OUTPUT AT STEPS OF 0.01
TEND = 0.
T = 0.0
IDO = 1
NSTEP = 200
DO J=1,NSTEP
 TEND = TEND + 0.01

! SOLVE THE PROBLEM
 CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, &
 HINIT=HINIT, TOL=TOL, INPDER=1)

! COMPUTE MAXIMUM ERROR
```

```

 ERRU = 0.0
 DO I=1,NX
 ERROR = Y(1,I) - SIN(PI*XBREAK(I))*COS(PI*TEND)
 ERRU = AMAX1(ERRU,ABS(ERROR))
 END DO
 END DO
!
! PRINT ERROR
 WRITE (NOUT, *) ' Maximum error in u(x,t): ', ERRU
! LAST CALL, TO RELEASE WORKSPACE
 IDO = 3
 CALL MMOLCH (IDO, FCNUT, FCNBC, T, TEND, XBREAK, Y, &
 HINIT=HINIT, TOL=TOL, INPDER=1)
 END

 SUBROUTINE FCNUT (NPDES, X, T, U, UX, UXX, UT)

 INTEGER NPDES
 REAL X, T, U(*), UX(*), UXX(*), UT(*)
! DEFINE THE PDEs
 UT(1) = U(2)
 UT(2) = UXX(1)
 RETURN
 END

 SUBROUTINE FCNBC (NPDES, X, T, ALPHA, BTA, GAM)

 INTEGER NPDES
 REAL X, T, ALPHA(*), BTA(*), GAM(*)
! DEFINE THE BOUNDARY CONDITIONS
 ALPHA(1) = 1.0
 BTA(1) = 0.0
 GAM(1) = 0.0
 ALPHA(2) = 1.0
 BTA(2) = 0.0
 GAM(2) = 0.0
 RETURN
 END

```

## Output

Maximum error in  $u(x,t)$ : 5.49525E-3

---

# MOLCH

---

**Deprecated Routine:** MOLCH is a deprecated routine and has been replaced with [MMOLCH](#).

Click [here](#) to view the MOLCH documentation.

---

---

# FEYNMAN\_KAC



Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided. The solution is represented by a series of  $C^2$  Hermite quintic splines.

## Required Arguments

**XGRID** — Rank-1 array containing the set of breakpoints that define the end points for the Hermite quintic splines. (Input)

Let  $m = \text{size}(\text{XGRID})$ . The points in XGRID must be in strictly increasing order, and  $m \geq 2$ .

**TGRID** — Rank-1 array containing the set of time points (in time-remaining units) at which an approximate solution is computed. (Input)

Let  $n = \text{size}(\text{TGRID})$ . The points in TGRID must be strictly positive and in strictly increasing order and  $n \geq 1$ .

**NLBC** — The number of left boundary conditions. (Input)

$1 \leq \text{NLBC} \leq 3$ .

**NRBC** — The number of right boundary conditions. (Input)

$1 \leq \text{NRBC} \leq 3$ .

**FKCOEF** — User-supplied FUNCTION to evaluate the coefficients  $\sigma, \sigma', \mu$  and  $\kappa$  of the Feynman-Kac PDE. The usage is FKCOEF (X, TX, IFLAG [, ...]), where

## Function Return Value

**FKCOEF** — Value of the coefficient function. Which value is computed depends on the input value for IFLAG, see description of IFLAG.

### Required Arguments

***X*** — Point in the  $x$ -space at which the coefficient is to be evaluated. (Input)

***TX*** — Time point at which the coefficient is to be evaluated. (Input)

***IFLAG*** — Flag related to the coefficient that has to be computed (Input/Output).

On entry, ***IFLAG*** indicates which coefficient is to be computed. The following table shows which value has to be returned by ***FKCOEF*** for all possible values of ***IFLAG***:

| <b><i>IFLAG</i></b> | <b>Computed coefficient</b>                         |
|---------------------|-----------------------------------------------------|
| 1                   | $\sigma' = \frac{\partial \sigma(x,t)}{\partial x}$ |
| 2                   | $\sigma$                                            |
| 3                   | $\mu$                                               |
| 4                   | $\kappa$                                            |

One indicates when a coefficient does not depend on  $t$  by setting ***IFLAG*** = 0 after the coefficient is defined. If there is time dependence, the value of ***IFLAG*** should not be changed. This action will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each  $t$  value.

### Optional Arguments

***FCN\_DATA*** — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

***FKCOEF*** must be declared `EXTERNAL` in the calling program.

***FKINITCOND*** — User-supplied `FUNCTION` to evaluate the initial condition function  $p(x)$  in the Feynman-Kac PDE. The usage is `FKINITCOND (X[, ...])`, where

### Function Return Value

***FKINITCOND*** — Value of the initial condition function  $p(x)$ .

### Required Arguments

***X*** — Point in the  $x$ -space at which the initial condition is to be evaluated. (Input)

### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

**FKINITCOND** must be declared `EXTERNAL` in the calling program.

**FKBC** — User-supplied subroutine to evaluate the coefficients for the left and right boundary conditions the Feynman-Kac PDE must satisfy. There are `NLBC` conditions specified at the left end,  $x_{\min}$ , and `NRBC` conditions at the right end,  $x_{\max}$ . The boundary conditions can be vectors of dimension 1, 2 or 3 and are defined by

$$a(x,t)f + b(x,t)f_x + c(x,t)f_{xx} = d(x,t), \quad x = x_{\min} \text{ or } x = x_{\max}$$

The usage is `FKBC (TX, IFLAG, BCCOEF5 [,...])` where

### Required Arguments

**TX** — Time point at which the coefficients are to be evaluated. (Input)

**IFLAG** — Flag related to the boundary conditions that have to be computed (Input/Output).

On input, `IFLAG` indicates whether the coefficients for the left or right boundary conditions have to be computed:

| IFLAG | Computed boundary conditions |
|-------|------------------------------|
| 1     | Left end, $x = x_{\min}$     |
| 2     | Right end, $x = x_{\max}$    |

If there is no time dependence for one of the boundaries then set `IFLAG = 0` after the array `BCCOEF5` is defined for either end point. This can avoid unneeded continued computation of the finite element matrices.

**BCCOEF5** — Array of size  $3 \times 4$  containing the coefficients of the left or right boundary conditions in its first `NLBC` or `NRBC` rows, respectively. (Output)

The coefficients for  $x_{\min}$  are stored row-wise according to the following matrix-scheme:

$$\begin{pmatrix} a_1(x_{\min}, t), b_1(x_{\min}, t), c_1(x_{\min}, t), d_1(x_{\min}, t) \\ \vdots \\ a_{\text{NLBC}}(x_{\min}, t), b_{\text{NLBC}}(x_{\min}, t), c_{\text{NLBC}}(x_{\min}, t), d_{\text{NLBC}}(x_{\min}, t) \end{pmatrix}$$

The coefficients for  $x_{\max}$  are stored similarly.

## Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

**FKBC** must be declared `EXTERNAL` in the calling program.

**Y** — Array of size  $(3*m)$  by  $(n+1)$  containing the coefficients of the Hermite representation of the approximate solution for the Feynman-Kac PDE at time points (in time-remaining units)  $0, \text{TGRID}(1), \dots, \text{TGRID}(n)$ . (Output)

For  $t = \text{TGRID}(j), j = 1, \dots, n$ , the coefficients are stored in columns  $1, \dots, n$  of array **Y** and the approximate solution is given by

$$f(x, t) = \sum_{i=1}^{3*m} Y(i, j) \beta_i(x)$$

The coefficients of the representation for the initial data are given in column 0 of array **Y** and are defined by

$$p(x) = \sum_{i=1}^{3*m} Y(i, 0) \beta_i(x)$$

The starting coefficients  $Y(i, 0), i = 1, \dots, m$  are estimated using least-squares.

After the integrations, use  $Y(:, 0)$  and  $Y(:, j)$  as input argument **COEFFS** to function **HQSVAL** to obtain an array of values for  $f(x, t)$  or its partials  $f_x, f_{xx}, f_{xxx}$  at time points  $t=0$  and  $t=\text{TGRID}(j), j=1, \dots, n$ , respectively.

The expressions for the basis functions  $\beta_i(x)$  are represented piece-wise and can be found in Hanson, R. (2008) “[Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements](#)”.

**YPRIME** — Array of size  $(3*m)$  by  $(n+1)$  containing the first derivatives of the coefficients of the Hermite representation of the approximate solution for the Feynman-Kac PDE at time points (in time-remaining units)  $0, \text{TGRID}(1), \dots, \text{TGRID}(n)$ . (Output)

For  $t = 0$  and  $t = \text{TGRID}(j), j = 1, \dots, n$ , the derivatives of the coefficients are stored in column 0 and columns 1 to  $n$  of array **YPRIME**, respectively. The columns in **YPRIME** represent

$$f_t(x, \bar{t}) = \sum_{i=1}^{3*m} YPRIME(i, 0) \beta_i(x) \quad \text{for } \bar{t} = \text{TGRID}(j), j = 1, \dots, n,$$

and

$$f_t(x, \bar{t}) = \sum_{i=1}^{3*m} YPRIME(i, 0) \beta_i(x) \quad \text{for } \bar{t} = 0.$$

After the integrations, use `YPRIME(:, j)` as input argument `COEFFS` to function `HQSVAL` to obtain an array of values for the partials  $f_t, f_{tx}, f_{txx}, f_{txxx}$  at time points  $t = \text{TGRID}(j), j = 1, \dots, n$ , and `YPRIME(:, 0)` for the partials at  $t = 0$ .

## Optional Arguments

**FKINIT** — User-supplied subroutine that allows for adjustment of initial data or as an opportunity for output during the integration steps.

The usage is `CALL FKINIT (XGRID, TGRID, TX, YPRIME, Y, ATOL, RTOL[, ...])` where

### Required Arguments

**XGRID** — Array of size  $m$  containing the set of breakpoints that define the end points for the Hermite quintic splines. (Input)

**TGRID** — Array of size  $n$  containing the set of time points (in time-remaining units) at which an approximate solution is computed. (Input)

**TX** — Time point for the evaluation. (Input)  
Possible values are 0 (the initial or “terminal” time point) and all values in array `TGRID`.

**YPRIME** — Array of length  $3*m$  containing the derivatives of the Hermite quintic spline coefficients at time point `TX`. (Input)

**Y** — Array of length  $3*m$  containing the Hermite quintic spline coefficients at time point `TX`. (Input/Output)  
For the initial time point `TX=0` this array can be used to reset the Hermite quintic spline coefficients to user defined values. For all other values of `TX` array `Y` is an input array.

**ATOL** — Array of length  $3*m$  containing absolute error tolerances used in the integration routine that determines the Hermite quintic spline coefficients and its derivatives. (Input/Output)

**RTOL** — Array of length  $3*m$  containing relative error tolerances used in the integration routine that determines the Hermite quintic spline coefficients and its derivatives. (Input/Output)

### Optional Arguments

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

`FKINIT` must be declared `EXTERNAL` in the calling program.

**FKFORCE** — User-supplied subroutine that computes local contributions

$$\varphi_i^i := \int_{x_i}^{x_{i+1}} \phi(f, x, t) \hat{\beta}(x) dx \quad \text{and} \quad \frac{\partial \varphi_i^i}{\partial y} := \int_{x_i}^{x_{i+1}} \frac{\partial \phi(f, x, t)}{\partial f} \hat{\beta}(x)^T \hat{\beta}(x) dx$$

The usage is CALL FKFORCE (I, T, WIDTH, Y, XLOCAL, QW, U, PHI, DPHI [, , ...]) where

### Required Arguments

- I — Index related to the integration interval (XGRID(I), XGRID(I+1)). (Input)
- T — Time point at which the local contributions are computed. (Input)
- WIDTH — Width of the integration interval I, WIDTH= XGRID(I+1) - XGRID(I). (Input)
- Y — Array of length 3\*m containing the coefficients of the Hermite quintic spline representing the solution of the Feynman-Kac PDE at time point T. (Input)  
For each

$$x \in [x_i, x_{i+1}], h_i = x_{i+1} - x_i, z = (x - x_i) / h_i, i = 1, \dots, m-1,$$

the approximate solution is locally defined by

$$f(x, t) = f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f_i' b_1(z) - h_i f_{i+1}' b_1(1-z) + h_i^2 f_i'' b_2(z) + h_i^2 f_{i+1}'' b_2(1-z).$$

Here, the functions  $b_0(z), b_1(z), b_2(z)$  are basis polynomials of order 5 and

$$f_i := f(x_i, t), f_i' := f_x(x_i, t), f_i'' := f_{xx}(x_i, t)$$

The values

$$y_{3i-2} = f_i, y_{3i-1} = f_i', y_{3i} = f_i'', i = 1, \dots, m,$$

are stored as successive triplets in array Y.

- XLOCAL — Array containing the Gauss-Legendre points translated and normalized to the interval [XGRID(I), XGRID(I+1)]. (Input)  
The size of the array is equal to the degree of the Gauss-Legendre polynomials used for constructing the finite element matrices.
- QW — Array containing the Gauss-Legendre weights. (Input)  
The size of the array is equal to the degree of the Gauss-Legendre polynomials used for constructing the finite element matrices.
- U — Array of size size(XLOCAL) × 12 containing the basis function values that define  $\hat{\beta}(x)$  at the Gauss-Legendre points XLOCAL. (Input)  
Let

$$x \in [x_I, x_{I+1}], h_I := x_{I+1} - x_I, z(x) := (x - x_I) / h_I.$$

Using the local approximation in the I-th interval, defined by

$$f(x, t) = \sum_{k=-2}^3 y_{3I+k} \beta_{3I+k}(x),$$

and setting

$$u_{j,k} := U(j, k), \quad x_j := XLOCAL(j) \quad \text{and} \quad z(x_j) := z_j,$$

vector  $\hat{\beta}(x_j) = (\hat{\beta}_1(x_j), \dots, \hat{\beta}_6(x_j))$  is defined as

$$\begin{aligned} \hat{\beta}(x_j) &:= (\beta_{3I-2}(x_j), \dots, \beta_{3I+3}(x_j))^T \\ &:= (b_0(z_j), h_I b_1(z_j), h_I^2 b_2(z_j), b_0(1-z_j), -h_I b_1(1-z_j), h_I^2 b_2(1-z_j))^T \\ &:= (u_{j,1}, u_{j,2}, u_{j,3}, u_{j,7}, u_{j,8}, u_{j,9})^T. \end{aligned}$$

PHI — Array of size 6 containing a Gauss-Legendre approximation for the local

contribution  $\varphi_t^I := \int_{XGRID(I)}^{XGRID(I+1)} \phi(f, x, t) \hat{\beta}(x) dx$ , where  $t = T$  and

$$\hat{\beta}(x) := (\beta_{3I-2}(x), \dots, \beta_{3I+3}(x))^T. \quad (\text{Output})$$

Setting  $NDEG := \text{SIZE}(XLOCAL)$  and  $x_j := XLOCAL(j)$ , array PHI contains elements

$$PHI(i) = \text{WIDTH} * \sum_{j=1}^{NDEG} QW(j) \hat{\beta}_i(x_j) \phi(f, x_j, t)$$

for  $i = 1, \dots, 6$ .

DPHI — Array of size 6×6, a Gauss-Legendre approximation for the Jacobian of the local contribution  $\varphi_t^I$  at time point  $t = T$ ,

$$\frac{\partial \varphi_t^I}{\partial y} := \int_{XGRID(I)}^{XGRID(I+1)} \frac{\partial \phi(f, x, t)}{\partial f} \hat{\beta}(x) \hat{\beta}^T(x) dx. \quad (\text{Output})$$

The approximation to this symmetric matrix is stored row-wise, i.e.

$$DPHI(i, j) = \text{WIDTH} * \sum_{k=1}^{NDEG} QW(k) \hat{\beta}_i(x_k) \hat{\beta}_j(x_k) \left. \frac{\partial \phi}{\partial f} \right|_{x=XLOCAL(k), t=T}$$

or  $i, j = 1, \dots, 6$ .

### Optional Arguments

FCN\_DATA — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied

subroutine. For a detailed description of this argument see [FCN\\_DATA](#) below. (Input/Output)

`EKFORCE` must be declared `EXTERNAL` in the calling program.

If subroutine `EKFORCE` is not used as an optional argument then it is assumed that the forcing term  $\phi$  in the Feynman-Kac equation is identically zero.

***ATOL*** — Array of non-negative values containing absolute error tolerances used in the computation of each column of solution array `Y` via integration routine `DASPH`. (Input)  
The size of array `ATOL` can be 1 or  $3 \times m$ . In the first case, `ATOL(1:1)` is applied to all solution components, in the latter each component of `ATOL` is assigned to the corresponding solution component allowing for individual control of the error tolerances. At least one entry in arrays `ATOL` or `RTOL` must be greater than 0.  
Default: `ATOL(1:1) = 1.0e-3` for single and `1.0d-5` for double precision.

***RTOL*** — Array of non-negative values containing relative error tolerances used in the computation of each column of solution array `Y` via integration routine `DASPH`. (Input)  
The size of array `RTOL` can be 1 or  $3 \times m$ . In the first case, `RTOL(1:1)` is applied to all solution components, in the latter each component of `RTOL` is assigned to the corresponding solution component allowing for individual control of the error tolerances. At least one entry in arrays `ATOL` or `RTOL` must be greater than 0.  
Default: `RTOL(1:1) = 1.0e-3` for single and `1.0d-5` for double precision.

***NDEG*** — Degree of the Gauss-Legendre formulas used for constructing the finite element matrices. (Input)  
`NDEG`  $\geq 6$ .  
Default: `NDEG = 6`.

***RINITSTEP*** — Starting step size for the integration. (Input)  
`RINITSTEP` must be strictly negative since the integration is internally done from `T = 0` to `T = TGRID(n)` in a negative direction.  
Default: Program defined initial stepsize.

***MAXBDFORDER*** — Maximum order of the backward differentiation formulas (BDF) used in the integrator `DASPH`. (Input)  
 $1 \leq \text{MAXBDFORDER} \leq 5$ .  
Default: `MAXBDFORDER = 5`.

***RMAXSTEP*** — Maximum step size the integrator may take. (Input)  
`RMAXSTEP` must be strictly positive.  
Default: `RMAXSTEP = AMACH(2)`, the largest possible machine number.

***MAXIT*** — Maximum number of internal integration steps between two consecutive time points in `TGRID`. (Input)  
`MAXIT` must be strictly positive.  
Default: `MAXIT = 500000`.

**IMETHSTEPCTRL** — Indicates which step control algorithm is used in the integration. (Input)

If  $IMETHSTEPCTRL = 0$ , then the step control method of Söderlind is used. If  $IMETHSTEPCTRL = 1$ , then the method used by the original Petzold code SASSL is used.

| <b>IMETHSTEPCTRL</b> | <b>Method used</b>              |
|----------------------|---------------------------------|
| 0                    | Method of Söderlind..           |
| 1                    | Method from Petzold code SASSL. |

Default:  $IMETHSTEPCTRL = 0$ .

**TBARRIER** — Time barrier past which the integration routine `DASPH` will not go during integration. (Input)

$TBARRIER \geq TGRID(n)$ .

Default:  $TBARRIER = TGRID(n)$ .

**ISTATE** — Array of size 5 whose entries flag the state of computation for the matrices and vectors required in the integration. (Output)

For each entry, a zero indicates that no computation has been done or that there is a time dependence. A one indicates that the entry has been computed and there is no time dependence.

The `ISTATE` entries are as follows:

| <b>I</b> | <b>ISTATE(I)</b>                                 |
|----------|--------------------------------------------------|
| 1        | State of computation of Mass matrix, M.          |
| 2        | State of computation of Stiffness matrix, N.     |
| 3        | State of computation of Bending matrix, R.       |
| 4        | State of computation of Weighted mass matrix, K. |
| 5        | State of computation of initial data.            |

**NVAL** — Array of size 3 summarizing the number of evaluations required during the integration. (Output)

| <b>I</b> | <b>NVAL(I)</b>                                                                       |
|----------|--------------------------------------------------------------------------------------|
| 1        | Number of residual function evaluations of the DAE used in the model.                |
| 2        | Number of factorizations of the differential matrix associated with solving the DAE. |
| 3        | Number of linear system solve steps using the differential matrix.                   |

**ITDEPEND** — Logical array of size 7 indicating time dependence of the coefficients, boundary conditions and forcing term  $\phi$  in the Feynman-Kac equation. (Output)  
 If `ITDEPEND(I) = .FALSE.` then argument `I` is not time dependent, if `ITDEPEND(I) = .TRUE.` then argument `I` is time dependent.

| <b>I</b> | <b>ITDEPEND(I)</b>                            |
|----------|-----------------------------------------------|
| 1        | Time dependence of $\sigma'$ .                |
| 2        | Time dependence of $\sigma$ .                 |
| 3        | Time dependence of $\mu$ .                    |
| 4        | Time dependence of $\kappa$ .                 |
| 5        | Time dependence of left boundary conditions.  |
| 6        | Time dependence of right boundary conditions. |
| 7        | Time dependence of $\phi$ .                   |

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. (Input/Output)  
 The derived type, `s_fcn_data`, is defined as:

```

type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type

```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type.

Note that if user-supplied data are required in one of the user-defined functions or subroutines available for routine `FEYNMAN_KAC` then these data must be defined via `FCN_DATA`.

## FORTRAN 90 Interface

Generic: `CALL FEYNMAN_KAC (XGRID, TGRID, NLBC, NRBC, FKCOEF, FKINITCOND, FKBC, Y, YPRIME [, ...])`

Specific: The specific interface names are `S_FEYNMAN_KAC` and `D_FEYNMAN_KAC`.

## Description

The generalized Feynman-Kac differential equation has the form

$$f_t + \mu(x,t)f_x + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t)f = \phi(f, x, t)$$

where the initial data satisfies

$$f(x, T) = p(x).$$

The derivatives are  $f_t = \frac{\partial f}{\partial t}$ ,  $f_x = \frac{\partial f}{\partial x}$  etc.

FEYNMAN\_KAC uses a finite element Galerkin method over the rectangle<sup>3</sup>

$$[x_{\min}, x_{\max}] \times [\bar{T}, T]$$

in  $(x, t)$  to compute the approximate solution. The interval  $[x_{\min}, x_{\max}]$  is decomposed with a grid

$$(x_{\min} =) x_1 < x_2 < \dots < x_m (= x_{\max}).$$

On each subinterval the solution is represented by

$$\begin{aligned} f(x, t) = & f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f'_i b_1(z) \\ & - h_i f'_{i+1} b_1(1-z) + h_i^2 f''_i b_2(z) + h_i^2 f''_{i+1} b_2(1-z). \end{aligned}$$

The values  $f_i, f'_i, f''_i, f_{i+1}, f'_{i+1}, f''_{i+1}$  are time-dependent coefficients associated with each interval.

The basis functions  $b_0, b_1, b_2$  are given for  $x \in [x_i, x_{i+1}]$ ,  $h_i := x_{i+1} - x_i$ ,  $z(x) := (x - x_i) / h_i \in [0, 1]$ ,

by

$$\begin{aligned} b_0(z) &= -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3(6z^2 + 3z + 1) \\ b_1(z) &= -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z + 1) \\ b_2(z) &= \frac{1}{2}(-z^5 + 3z^4 - 3z^3 + z^2) = \frac{1}{2}(1-z)^3 z^2 \end{aligned}$$

The Galerkin principle is then applied. Using the provided initial and boundary conditions leads to an index 1 differential-algebraic equation (DAE) for the time-dependent coefficients

$$y_{3i-2} := f_i, \quad y_{3i-1} := f'_i, \quad y_{3i} := f''_i, \quad i = 1, \dots, m.$$

This system is integrated using the variable order, variable step algorithm `DASPH`. Solution values and their time derivatives are returned at a grid preceding time  $T$ , expressed in units of time remaining.

More mathematical details are found in Hanson, R. (2008) “[Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements](#)”.

---

<sup>3</sup>SuperLU is used to support the sparse matrix operations used by FEYNMAN\_KAC. SuperLU is well-tested. Distributed and threaded versions are available but these are not used in our software. SuperLU was developed by James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. The authors do not support the package in the context used in the IMSL Libraries.

## Examples

### Example 1 – A Diffusion Model For Call Options

In Beckers (1980) there is a model for a Stochastic Differential Equation of option pricing. The idea is a “constant elasticity of variance diffusion (or CEV) class”

$$dS = \mu S dt + \sigma S^{\alpha/2} dW, \quad 0 \leq \alpha < 2$$

The Black-Scholes model is the limiting case  $\alpha \rightarrow 2$ . A numerical solution of this diffusion model yields the price of a call option. Various values of the strike price  $K$ , time values,  $\sigma$  and power coefficient  $\alpha$  are used to evaluate the option price at values of the underlying price. The sets of parameters in the computation are:

1. power  $\alpha = \{2.0, 1.0, 0.0\}$
2. strike price  $K = \{15.0, 20.0, 25.0\}$
3. volatility  $\sigma = \{0.2, 0.3, 0.4\}$
4. times until expiration =  $\{1/12, 4/12, 7/12\}$
5. underlying prices =  $\{19.0, 20.0, 21.0\}$
6. interest rate  $r = 0.05$
7.  $x_{\min} = 0, x_{\max} = 60$
8.  $nx = 121, n = 3 \times nx = 363$

With this model the Feynman-Kac differential equation is defined by identifying:

$$x: S$$

$$\sigma(x, t): \sigma x^{\alpha/2}; \quad \frac{\partial \sigma}{\partial x} = \frac{a\sigma}{2} x^{\alpha/2-1}$$

$$\mu(x, t): rx$$

$$\kappa(x, t): r$$

$$\phi(f, x, t) \equiv 0$$

The payoff function is the “vanilla option”,  $p(x) = \max(x - K, 0)$ .

[Link to example source](#) (feynman\_kac\_ex1.f90)

```
! Compute Constant Elasticity of Variance Model for Vanilla Call
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use umach_int

 implicit none
```

```

! The set of strike prices
 real(kind(1e0)) :: ks(3)=(/15.0e0,20.0e0,25.0e0/)
! The set of sigma values
 real(kind(1e0)) :: sigma(3) = (/0.2e0, 0.3e0, 0.4e0/)
! The set of model diffusion powers
 real(kind(1e0)) :: alpha(3) = (/2.0e0,1.0e0,0.0e0/)
! Time values for the options
 integer, parameter :: nt = 3
 real(kind(1e0)) :: time(nt)=(/1.e0/12., 4.e0/12., 7.e0/12./)
! Values of the underlying where evaluation are made
 integer, parameter :: nv = 3, nlbc = 3, nrbc = 3
 real(kind(1e0)) :: xs(nv) = (/19.0e0,20.0e0,21.0e0/)
! Value of the interest rate and continuous dividend
 real(kind(1e0)) :: r = 0.05e0, dividend = 0.0e0
! Values of the min and max underlying values modeled
 real(kind(1e0)) :: x_min = 0.0e0, x_max = 60.0e0

! Define parameters for the integration step.
 integer, parameter :: nx = 121, nint = nx-1, n = 3*nx
 real(kind(1e0)) :: xgrid(nx), y(n,0:nt), yprime(n,0:nt), &
 dx, f(nv,nt)
 type(s_fcn_data) fcn_data
 integer :: nout
 real(kind(1e0)), external :: fkcoef, fkinitcond
 external fkbc

 integer :: i,i1,i2,i3,j
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
 allocate(fcn_data % rdata (6))
! Define an equally-spaced grid of points for the underlying price
 dx = (x_max-x_min)/real(nint)
 xgrid(1) = x_min
 xgrid(nx) = x_max
 do i = 2,nx-1
 xgrid(i) = xgrid(i-1) + dx
 end do

 call umach(2, nout)
 write(nout,'(T05,A)') "Constant Elasticity of Variance Model '//&
 "for Vanilla Call"
 write(nout,'(T10,"Interest Rate ", F7.3, T38,"Continuous '//&
 'Dividend ", F7.3)') r, dividend
 write(nout,'(T10,"Minimum and Maximum Prices of Underlying ", '//&
 '2F7.2)') x_min, x_max
 write(nout,'(T10,"Number of equally spaced spline knots ",I4,'//&
 '/T10,"Number of unknowns ",I4)')&
 nx-1,n
 write(nout,'(/T10,"Time in Years Prior to Expiration ",2X,'//&
 '3F7.4)') time
 write(nout,'(T10,"Option valued at Underlying Prices ", '//&
 '3F7.2)') xs

 do i1 = 1,3 ! Loop over power
 do i2=1,3 ! Loop over volatility

```

```

do i3=1,3 ! Loop over strike price
! Pass data through into evaluation routines.
 fcn_data % rdata =&
 (/ks(i3),x_max,sigma(i2),alpha(i1),r,dividend/)
 call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)
! Evaluate solution at vector of points XS(:), at each time value
! prior to expiration.
 do i=1,nt
 f(:,i) = hqsval (xs, xgrid, y(:,i))
 end do
 write(nout,'(T05,"Strike=",F5.2," Sigma=", F5.2,'//&
 '" Alpha=", F5.2,/(T25," Call Option Values ",'//&
 'X,3F7.4)') ks(I3),sigma(I2),&
 alpha(i1),(f(i,:),i=1,nv)
 end do !i3 - Strike price loop
end do !i2 - Sigma loop
end do !i1 - Alpha loop
end

! These functions and routines define the coefficients, payoff
! and boundary conditions.
function fkcoef (X, TX, iflag, fcn_data)
 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: X, TX
 integer, intent(inout) :: iflag
 type(s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: fkcoef
 real(kind(1e0)) :: sigma, interest_rate, alpha, dividend,&
 zero = 0.0e0, half = 0.5e0

 sigma = fcn_data % rdata(3)
 alpha = fcn_data % rdata(4)
 interest_rate = fcn_data % rdata(5)
 dividend = fcn_data % rdata(6)
 select case (iflag)
 case (1)
! The coefficient derivative d(sigma)/dx
 fkcoef = half*alpha*sigma*x**(alpha*half-1.0e0)
! The coefficient sigma(x)
 case (2)
 fkcoef = sigma*x**(alpha*half)
 case (3)
! The coefficient mu(x)
 fkcoef = (interest_rate - dividend) * x
 case (4)
! The coefficient kappa(x)
 fkcoef = interest_rate
 end select
! Note that there is no time dependence
 iflag = 0
 return
end function fkcoef

function fkinitcond(x, fcn_data)

```

```

 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: x
 type (s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: fkinitcond
 real(kind(1e0)) :: zero = 0.0e0
 real(kind(1e0)) :: strike_price

 strike_price = fcn_data % rdata(1)
! The payoff function
 fkinitcond = max(x - strike_price, zero)
 return
end function fkinitcond

subroutine fkbc (tx, iflag, bccoefs, fcn_data)
 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: tx
 integer, intent(inout) :: iflag
 real(kind(1e0)), dimension(:, ::), intent(out) :: bccoefs
 type (s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: x_max, df, interest_rate, strike_price

 strike_price = fcn_data % rdata(1)
 x_max = fcn_data % rdata(2)
 interest_rate = fcn_data % rdata(5)
 select case (iflag)
 case (1)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, 0.0e0/)
 bccoefs(2,1:4) = (/0.0e0, 1.0e0, 0.0e0, 0.0e0/)
 bccoefs(3,1:4) = (/0.0e0, 0.0e0, 1.0e0, 0.0e0/)
! Note no time dependence at left end
 iflag = 0
 case (2)
 df = exp(interest_rate * tx)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, &
 x_max - df*strike_price/)
 bccoefs(2,1:4) = (/0.0e0, 1.0e0, 0.0e0, 1.0e0/)
 bccoefs(3,1:4) = (/0.0e0, 0.0e0, 1.0e0, 0.0e0/)
 end select
end subroutine fkbc

```

## Output

```

Constant Elasticity of Variance Model for Vanilla Call
Interest Rate 0.050 Continuous Dividend 0.000
Minimum and Maximum Prices of Underlying 0.00 60.00
Number of equally spaced spline knots 120
Number of unknowns 363

Time in Years Prior to Expiration 0.0833 0.3333 0.5833
Option valued at Underlying Prices 19.00 20.00 21.00

Strike=15.00 Sigma= 0.20 Alpha= 2.00
Call Option Values 4.0624 4.2575 4.4730
Call Option Values 5.0624 5.2506 5.4490

```

|              |                         |        |        |        |
|--------------|-------------------------|--------|--------|--------|
|              | Call Option Values      | 6.0624 | 6.2486 | 6.4385 |
| Strike=20.00 | Sigma= 0.20 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.1310 | 0.5955 | 0.9699 |
|              | Call Option Values      | 0.5018 | 1.0887 | 1.5101 |
|              | Call Option Values      | 1.1977 | 1.7483 | 2.1752 |
| Strike=25.00 | Sigma= 0.20 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.0000 | 0.0112 | 0.0745 |
|              | Call Option Values      | 0.0000 | 0.0372 | 0.1621 |
|              | Call Option Values      | 0.0007 | 0.1027 | 0.3141 |
| Strike=15.00 | Sigma= 0.30 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 4.0637 | 4.3398 | 4.6622 |
|              | Call Option Values      | 5.0626 | 5.2944 | 5.5786 |
|              | Call Option Values      | 6.0624 | 6.2708 | 6.5240 |
| Strike=20.00 | Sigma= 0.30 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.3109 | 1.0276 | 1.5494 |
|              | Call Option Values      | 0.7326 | 1.5424 | 2.1017 |
|              | Call Option Values      | 1.3765 | 2.1690 | 2.7379 |
| Strike=25.00 | Sigma= 0.30 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.0006 | 0.1112 | 0.3543 |
|              | Call Option Values      | 0.0038 | 0.2169 | 0.5548 |
|              | Call Option Values      | 0.0184 | 0.3857 | 0.8222 |
| Strike=15.00 | Sigma= 0.40 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 4.0755 | 4.5138 | 4.9675 |
|              | Call Option Values      | 5.0662 | 5.4201 | 5.8326 |
|              | Call Option Values      | 6.0634 | 6.3579 | 6.7301 |
| Strike=20.00 | Sigma= 0.40 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.5115 | 1.4640 | 2.1273 |
|              | Call Option Values      | 0.9621 | 1.9951 | 2.6929 |
|              | Call Option Values      | 1.5814 | 2.6105 | 3.3216 |
| Strike=25.00 | Sigma= 0.40 Alpha= 2.00 |        |        |        |
|              | Call Option Values      | 0.0083 | 0.3286 | 0.7790 |
|              | Call Option Values      | 0.0285 | 0.5167 | 1.0657 |
|              | Call Option Values      | 0.0813 | 0.7687 | 1.4103 |
| Strike=15.00 | Sigma= 0.20 Alpha= 1.00 |        |        |        |
|              | Call Option Values      | 4.0624 | 4.2479 | 4.4311 |
|              | Call Option Values      | 5.0624 | 5.2479 | 5.4311 |
|              | Call Option Values      | 6.0624 | 6.2479 | 6.4311 |
| Strike=20.00 | Sigma= 0.20 Alpha= 1.00 |        |        |        |
|              | Call Option Values      | 0.0000 | 0.0218 | 0.1045 |
|              | Call Option Values      | 0.1498 | 0.4109 | 0.6485 |
|              | Call Option Values      | 1.0832 | 1.3314 | 1.5773 |
| Strike=25.00 | Sigma= 0.20 Alpha= 1.00 |        |        |        |
|              | Call Option Values      | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values      | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values      | 0.0000 | 0.0000 | 0.0000 |

|              |                    |             |        |        |        |
|--------------|--------------------|-------------|--------|--------|--------|
| Strike=15.00 | Sigma= 0.30        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 4.0624 | 4.2477 | 4.4309 |
|              | Call Option Values |             | 5.0624 | 5.2477 | 5.4309 |
|              | Call Option Values |             | 6.0624 | 6.2477 | 6.4309 |
| Strike=20.00 | Sigma= 0.30        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 0.0011 | 0.0781 | 0.2201 |
|              | Call Option Values |             | 0.1994 | 0.5000 | 0.7543 |
|              | Call Option Values |             | 1.0835 | 1.3443 | 1.6023 |
| Strike=25.00 | Sigma= 0.30        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0005 |
| Strike=15.00 | Sigma= 0.40        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 4.0624 | 4.2479 | 4.4312 |
|              | Call Option Values |             | 5.0624 | 5.2479 | 5.4312 |
|              | Call Option Values |             | 6.0624 | 6.2479 | 6.4312 |
| Strike=20.00 | Sigma= 0.40        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 0.0076 | 0.1563 | 0.3452 |
|              | Call Option Values |             | 0.2495 | 0.5907 | 0.8706 |
|              | Call Option Values |             | 1.0868 | 1.3779 | 1.6571 |
| Strike=25.00 | Sigma= 0.40        | Alpha= 1.00 |        |        |        |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0001 |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0008 |
|              | Call Option Values |             | 0.0000 | 0.0003 | 0.0063 |
| Strike=15.00 | Sigma= 0.20        | Alpha= 0.00 |        |        |        |
|              | Call Option Values |             | 4.0626 | 4.2479 | 4.4311 |
|              | Call Option Values |             | 5.0623 | 5.2480 | 5.4311 |
|              | Call Option Values |             | 6.0624 | 6.2480 | 6.4312 |
| Strike=20.00 | Sigma= 0.20        | Alpha= 0.00 |        |        |        |
|              | Call Option Values |             | 0.0001 | 0.0001 | 0.0002 |
|              | Call Option Values |             | 0.0816 | 0.3316 | 0.5748 |
|              | Call Option Values |             | 1.0818 | 1.3308 | 1.5748 |
| Strike=25.00 | Sigma= 0.20        | Alpha= 0.00 |        |        |        |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0000 |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0000 |
| Strike=15.00 | Sigma= 0.30        | Alpha= 0.00 |        |        |        |
|              | Call Option Values |             | 4.0625 | 4.2479 | 4.4312 |
|              | Call Option Values |             | 5.0623 | 5.2479 | 5.4312 |
|              | Call Option Values |             | 6.0624 | 6.2479 | 6.4312 |
| Strike=20.00 | Sigma= 0.30        | Alpha= 0.00 |        |        |        |
|              | Call Option Values |             | 0.0000 | 0.0000 | 0.0029 |
|              | Call Option Values |             | 0.0894 | 0.3326 | 0.5753 |
|              | Call Option Values |             | 1.0826 | 1.3306 | 1.5749 |
| Strike=25.00 | Sigma= 0.30        | Alpha= 0.00 |        |        |        |

|              |                    |             |        |        |
|--------------|--------------------|-------------|--------|--------|
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |
| Strike=15.00 | Sigma= 0.40        | Alpha= 0.00 |        |        |
|              | Call Option Values | 4.0624      | 4.2479 | 4.4312 |
|              | Call Option Values | 5.0623      | 5.2479 | 5.4312 |
|              | Call Option Values | 6.0624      | 6.2479 | 6.4312 |
| Strike=20.00 | Sigma= 0.40        | Alpha= 0.00 |        |        |
|              | Call Option Values | 0.0000      | 0.0002 | 0.0113 |
|              | Call Option Values | 0.0985      | 0.3383 | 0.5781 |
|              | Call Option Values | 1.0830      | 1.3306 | 1.5749 |
| Strike=25.00 | Sigma= 0.40        | Alpha= 0.00 |        |        |
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |
|              | Call Option Values | 0.0000      | 0.0000 | 0.0000 |

## Example 2 – American Option vs. European Option On a Vanilla Put

The value of the American Option on a Vanilla Put can be no smaller than its European counterpart. That is due to the American Option providing the opportunity to exercise at any time prior to expiration. This example compares this difference – or premium value of the American Option – at two time values using the Black-Scholes model. The example is based on Wilmott et al. (1996, p. 176), and uses the non-linear forcing or weighting term described in Hanson, R. (2008), “*Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements*”, for evaluating the price of the American Option. A call to the subroutine `fkinit_put` sets the initial conditions. One breakpoint is set exactly at the strike price.

The sets of parameters in the computation are:

1. Strike price  $K = \{10.0\}$
2. Volatility  $\sigma = \{0.4\}$
3. Times until expiration =  $\{1/4, 1/2\}$
4. Interest rate  $r = 0.1$
5.  $x_{\min} = 0.0$ ,  $x_{\max} = 30.0$
6.  $nx = 121$ ,  $n = 3 \times nx = 363$

The payoff function is the “vanilla option”,  $p(x) = \max(K - x, 0)$ .

[Link to example source](#) (feynman\_kac\_ex2.f90)

```
! Compute American Option Premium for Vanilla Put
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use umach_int
 implicit none
! The strike price
```

```

 real(kind(1e0)) :: ks = 10.0e0
! The sigma value
 real(kind(1e0)) :: sigma = 0.4e0

! Time values for the options
 integer, parameter :: nt = 2
 real(kind(1e0)) :: time(nt)=(/0.25e0, 0.5e0/)
! Values of the underlying where evaluations are made
 integer, parameter :: nv = 9
 integer, parameter :: nlbc = 2, nrbc = 3, ndeg = 6

 integer :: i
 real(kind(1e0)) :: xs(nv) = (/((i-1)*2.0e0,i=1,nv)/)
! Value of the interest rate and continuous dividend
 real(kind(1e0)) :: r = 0.1e0, dividend = 0.0e0
! Values of the min and max underlying values modeled
 real(kind(1e0)) :: x_min = 0.0e0, x_max = 30.0e0
 real(kind(1e0)) :: atol(1), rtol(1)

! Define parameters for the integration step.
 integer, parameter :: nx = 121, nint = nx-1, n = 3*nx
 real(kind(1e0)) :: xgrid(nx), ye(n,0:nt), yeprime(n,0:nt), &
 ya(n,0:nt), yaprime(n,0:nt), &
 dx, fe(nv,nt), fa(nv,nt)

 type(s_fcn_data) fcn_data
 integer :: nout
 real(kind(1e0)), external :: fkcoef_put, fkinitcond_put
 external fkbc_put, fkinit_put, fkforce_put

 call umach(2, nout)
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
 allocate(fcn_data % rdata (6), fcn_data % idata (1))
! Define an equally-spaced grid of points for the underlying price
 dx = (x_max-x_min)/real(nint)
 xgrid(1) = x_min
 xgrid(nx) = x_max
 do i=2,nx-1
 xgrid(i) = xgrid(i-1) + dx
 end do

! Place a breakpoint at the strike price.
 do i = 1,nx
 if (xgrid(i) > ks) then
 xgrid(i-1) = ks
 exit
 end if
 end do

! Request less accuracy than the default values provide.
 atol(1) = 0.5e-2
 rtol(1) = 0.5e-2
 fcn_data % rdata = (/ks,x_max,sigma,r,dividend,atol(1)/)
 fcn_data % idata = (/ndeg/)

! Compute European then American Put Option Values.
 call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef_put,&

```

```

 fkinitcond_put, fkbc_put, ye, yeprime,&
 FKINIT=fkinit_put, ATOL=atol,RTOL=rtol,&
 FCN_DATA = fcn_data)
call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef_put,&
 fkinitcond_put, fkbc_put, ya, yaprime,&
 FKINIT=fkinit_put, ATOL=atol, RTOL=rtol,&
 FKFORCE=fkforce_put, FCN_DATA = fcn_data)

! Evaluate solutions at vector of points XS(:), at each time value
! prior to expiration.
do i=1,nt
 fe(:,i) = hqsval (xs, xgrid, ye(:,I))
 fa(:,I) = hqsval (xs, xgrid, ya(:,I))
end do
write(nout,'(T05,A,/,T05,A)')&
 "American Option Premium for Vanilla Put, 3 and 6 Months "///&
 "Prior to", "Expiry"
write(nout,'(T08,"Number of equally spaced spline knots ",I4,'///&
 '/T08,"Number of unknowns ",I4)') nx,n
write(nout,'(T08,"Strike= ",F5.2,", Sigma=", F5.2,", Interest'///&
 ' Rate=",F5.2,/T08,"Underlying", T26,"European",'///&
 'T42,"American",/(T10,5F8.4)') ks,sigma,r,&
 (xs(i), fe(i,1:nt), fa(i,1:nt),i=1,nv)
end

! These routines define the coefficients, payoff, boundary
! conditions, forcing term and initial conditions for American and
! European Options.
function fkcoef_put(x, tx, iflag, fcn_data)
 use mp_types
 implicit none
 integer, intent(inout) :: iflag
 real(kind(1e0)), intent(in) :: x, tx
 type(s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: fkcoef_put

 real(kind(1e0)) :: sigma, strike_price, interest_rate, &
 dividend, zero=0.e0
 sigma = fcn_data % rdata(3)
 interest_rate = fcn_data % rdata(4)
 dividend = fcn_data % rdata(5)
 select case (iflag)
 case (1)
! The coefficient derivative d(sigma)/dx
 fkcoef_put = sigma
! The coefficient sigma(x)
 case (2)
 fkcoef_put = sigma*x
 case (3)
! The coefficient mu(x)
 fkcoef_put = (interest_rate - dividend)*x
 case (4)
! The coefficient kappa(x)
 fkcoef_put = interest_rate
 end select
! Note that there is no time dependence
 iflag = 0
 return
end function

```

```

end function fkcoef_put

function fkinitcond_put(x, fcn_data)
 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: x
 type (s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: fkinitcond_put
 real(kind(1e0)) :: zero = 0.0e0
 real(kind(1e0)) :: strike_price

 strike_price = fcn_data % rdata(1)
! The payoff function
 fkinitcond_put = max(strike_price - x, zero)
 return
end function fkinitcond_put

subroutine fkbc_put (tx, iflag, bccoeffs, fcn_data)
 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: tx
 integer, intent(inout) :: iflag
 real(kind(1e0)), dimension(:,:), intent(out) :: bccoeffs
 type (s_fcn_data), optional :: fcn_data

 select case (iflag)
 case (1)
 bccoeffs(1,1:4) = ((/0.0e0, 1.0e0, 0.0e0, -1.0e0/))
 bccoeffs(2,1:4) = ((/0.0e0, 0.0e0, 1.0e0, 0.0e0/))
 case (2)
 bccoeffs(1,1:4) = ((/1.0e0, 0.0e0, 0.0e0, 0.0e0/))
 bccoeffs(2,1:4) = ((/0.0e0, 1.0e0, 0.0e0, 0.0e0/))
 bccoeffs(3,1:4) = ((/0.0e0, 0.0e0, 1.0e0, 0.0e0/))
 end select
! Note no time dependence
 iflag = 0
end subroutine fkbc_put

subroutine fkforce_put (interval, t, hx, y, xlocal, qw, u, &
 phi, dphi, fcn_data)
 use mp_types
 implicit none
 integer, parameter :: local = 6
 integer :: i, j, l, ndeg
 integer, intent(in) :: interval
 real(kind(1e0)), intent(in) :: y(:), t, hx, qw(:), &
 xlocal(:), u(:, :)
 real(kind(1e0)), intent(out) :: phi(:), dphi(:, :)
 type (s_fcn_data), optional :: fcn_data

 real(kind(1e0)) :: yl(local), bf(local)
 real(kind(1e0)) :: value, strike_price, interest_rate, &
 zero=0.0e0, one=1.0e0, rt, mu

 yl = y(3*interval-2:3*interval+3)
 phi = zero

```

```

value = fcn_data % rdata(6)
strike_price = fcn_data % rdata(1)
interest_rate = fcn_data % rdata(4)
ndeg = fcn_data % idata(1)

mu = 2
! This is the local definition of the forcing term
do j=1,local
do l=1,ndeg
bf(1:3) = u(1,1:3)
bf(4:6) = u(1,7:9)
rt = dot_product(y1,bf)
rt = value/(rt + value - (strike_price - xlocal(1)))
phi(j) = phi(j) + qw(l) * bf(j) * rt**mu
end do
end do
phi = -phi*hx*interest_rate*strike_price

! This is the local derivative matrix for the forcing term

dphi = zero
do j =1,local
do i = 1,local
do l=1,ndeg
bf(1:3) = u(1,1:3)
bf(4:6) = u(1,7:9)
rt = dot_product(y1,bf)
rt = one/(rt + value - (strike_price - xlocal(1)))
dphi(i,j) = dphi(i,j) + qw(l) * bf(I) * bf(j) *&
rt**(mu+1)
end do
end do
end do
dphi = mu*dphi*hx*value**mu*interest_rate*strike_price
return
end subroutine fkforce_put

subroutine fkinit_put(xgrid,tgrid,t,yprime,y,atol,rtol,&
fcfn_data)

use mp_types
implicit none
real(kind(1e0)), intent(in) :: xgrid(:), tgrid(:), t,&
yprime(:)
real(kind(1e0)), intent(inout) :: y(:), atol(:), rtol(:)
type (s_fcn_data), optional :: fcn_data
integer :: i

if (t == 0.0e0) then
! Set initial data precisely. The strike price is a breakpoint.
! Average the derivative limit values from either side.
do i=1,size(xgrid)
if (xgrid(i) < fcn_data % rdata(1)) then
y(3*i-2) = fcn_data % rdata(1) - xgrid(i)
y(3*i-1) = -1.0e0
y(3*i) = 0.0e0
else if (xgrid(i) == fcn_data % rdata(1)) then

```

```

 y(3*i-2) = 0.0e0
 y(3*i-1) = -0.5e0
 y(3*i) = 0.0e0
 else
 y(3*i-2) = 0.0e0
 y(3*i-1) = 0.0e0
 y(3*i) = 0.0e0
 end if
end do
end if
end subroutine fkinit_put

```

## Output

American Option Premium for Vanilla Put, 3 and 6 Months Prior to Expiry

```

Number of equally spaced spline knots 121
Number of unknowns 363
Strike= 10.00, Sigma= 0.40, Interest Rate= 0.10
Underlying European American
 0.0000 9.7536 9.5137 10.0000 10.0000
 2.0000 7.7536 7.5138 8.0000 8.0000
 4.0000 5.7537 5.5156 6.0000 6.0000
 6.0000 3.7614 3.5680 4.0000 4.0000
 8.0000 1.9064 1.9162 2.0214 2.0909
10.0000 0.6516 0.8540 0.6767 0.9034
12.0000 0.1625 0.3365 0.1675 0.3515
14.0000 0.0369 0.1266 0.0374 0.1322
16.0000 0.0088 0.0481 0.0086 0.0504

```

## Example 3 – European Option With Two Payoff Strategies

This example evaluates the price of a European Option using two payoff strategies: *Cash-or-Nothing* and *Vertical Spread*. In the first case the payoff function is

$$p(x) = \begin{cases} 0, & x \leq K \\ B, & x > K \end{cases}$$

The value  $B$  is regarded as the bet on the asset price, see Wilmott et al. (1995, p. 39-40). The second case has the payoff function

$$p(x) = \max(x - K_1) - \max(x - K_2), \quad K_2 > K_1$$

Both problems use the same boundary conditions. Each case requires a separate integration of the Black-Scholes differential equation, but only the payoff function evaluation differs in each case. The sets of parameters in the computation are:

1. Strike and bet prices  $K_1 = \{10.0\}$ ,  $K_2 = \{15.0\}$ , and  $B = \{2.0\}$
2. Volatility  $\sigma = \{0.4\}$ .
3. Times until expiration =  $\{1/4, 1/2\}$ .
4. Interest rate  $r = 0.1$ .

5.  $x_{min} = 0, x_{max} = 30$ .
6.  $nx = 121, n = 3 \times nx = 363$ .

[Link to example source \(feynman\\_kac\\_ex3.f90\)](#)

```

! Compute European Option Premium for a Cash-or-Nothing
! and a Vertical Spread Call.
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use umach_int
 implicit none
! The strike price
 real(kind(1e0)) :: ks1 = 10.0e0
! The spread value
 real(kind(1e0)) :: ks2 = 15.0e0
! The Bet for the Cash-or-Nothing Call
 real(kind(1e0)) :: bet = 2.0e0
! The sigma value
 real(kind(1e0)) :: sigma = 0.4e0

! Time values for the options
 integer, parameter :: nt = 2
 real(kind(1e0)) :: time(nt)=(/0.25e0, 0.5e0/)
! Values of the underlying where evaluation are made
 integer, parameter :: nv = 12, nlbc = 3, nrbc = 3
 integer :: i
 real(kind(1e0)) :: xs(nv) = (/ (2+(I-1)*2.0e0, I=1, NV) /)
! Value of the interest rate and continuous dividend -
 real(kind(1e0)) :: r = 0.1e0, dividend = 0.0e0
! Values of the min and max underlying values modeled -
 real(kind(1e0)) :: x_min = 0.0e0, x_max = 30.0e0

! Define parameters for the integration step.
 integer, parameter :: nx = 61, nint = nx-1, n=3*nx
 real(kind(1e0)) :: xgrid(nx), yb(n,0:nt), ybprime(n,0:nt), &
 yv(n,0:nt), yvprime(n,0:nt), &
 dx, fb(nv,nt), fv(nv,nt)

 type(s_fcn_data) fcn_data
 integer :: nout
 real(kind(1e0)), external :: fkcoef_call, fkinitcond_call
 external fkbc_call

 call umach(2, nout)
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
 allocate(fcn_data % rdata (7), fcn_data % idata (1))
! Define an equally-spaced grid of points for the underlying price
 dx = (x_max-x_min)/real(nint)
 xgrid(1) = x_min
 xgrid(nx) = x_max
 do i = 2, nx-1
 xgrid(i) = xgrid(i-1) + dx
 end do

 fcn_data % rdata = (/ks1,bet,ks2,x_max,sigma,r,dividend/)

```

```

! Flag the difference in payoff functions -
! 1 for the Bet, 2 for the Vertical Spread
 fcn_data % idata(1) = 1
 call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef_call,&
 fkinitcond_call, fkbc_call, yb, ybprime,&
 FCN_DATA = fcn_data)

 fcn_data % idata(1) = 2
 call feynman_kac (Xgrid, time, nlbc, nrbc, fkcoef_call,&
 fkinitcond_call, fkbc_call, yv, yvprime,&
 FCN_DATA = fcn_data)

! Evaluate solutions at vector of points XS(:), at each time value
! prior to expiration.
 do i=1,nt
 fb(:,i) = hqsval (xs, xgrid, yb(:,I))
 fv(:,i) = hqsval (xs, xgrid, yv(:,I))
 end do
 write(nout,'(T05,A)') "European Option Value for A Bet",&
 " and a Vertical Spread, 3 and 6 Months "///&
 "Prior to Expiry"
 write(nout,'(T08,"Number of equally spaced spline knots "'//&
 ',I4,/T08,"Number of unknowns ",I4)') NX,N
 write(nout,'(T08,"Strike = ",F5.2,", Sigma =", F5.2,'//&
 '" , Interest Rate =",F5.2,'//&
 '/T08,"Bet = ",F5.2,", Spread Value = ", F5.2'//&
 '/T10,"Underlying", T32,"A Bet", T40,"Vertical Spread",'//&
 '/(T10,5F9.4)') ks1, sigma, r, bet, ks2, &
 (xs(i), fb(i,1:nt), fv(i,1:nt),i=1,nv)
end

! These routines define the coefficients, payoff, boundary
! conditions and forcing term for American and European Options.
 function fkcoef_call (x, tx, iflag, fcn_data) result(value)
 use mp_types
 implicit none

 integer, intent(inout) :: iflag
 real(kind(1e0)), intent(in) :: x, tx
 type(s_fcn_data), optional :: fcn_data
 real(kind(1e0)) :: value

 real(kind(1e0)) :: sigma, interest_rate, dividend
! Data passed through using allocated components
! of the derived type s_fcn_data
 sigma = fcn_data % rdata(5)
 interest_rate = fcn_data % rdata(6)
 dividend = fcn_data % rdata(7)
 select case (iflag)
 case (1)
! The coefficient derivative d(sigma)/dx
 value = sigma
! The coefficient sigma(x)
 case (2)
 value = sigma * x

```

```

 case (3)
! The coefficient mu(x)
 value = (interest_rate - dividend) * x
 case (4)
! The coefficient kappa(x)
 value = interest_rate
 end select
! Note that there is no time dependence
 iflag = 0
 return
end function fkcoef_call

function fkinitcond_call(x, fcn_data) result(value)
use mp_types
implicit none

real(kind(1e0)), intent(in) :: x
type(s_fcn_data), optional :: fcn_data
real(kind(1e0)) :: value

real(kind(1e0)) :: strike_price, spread, bet
real(kind(1e0)), parameter :: zero = 0.0e0

strike_price = fcn_data % rdata(1)
bet = fcn_data % rdata(2)
spread = fcn_data % rdata(3)
! The payoff function - Use flag passed to decide which
select case (fcn_data % idata(1))
 case(1)
! After reaching the strike price the payoff jumps
! from zero to the bet value.
 value = zero
 if (x > strike_price) value = bet
 case(2)
! Function is zero up to strike price.
! Then linear between strike price and spread.
! Then has constant value Spread-Strike Price after
! the value Spread.
 value = max(x-strike_price, zero) - max(x-spread, zero)
 end select
return
end function fkinitcond_call

subroutine fkbc_call (TX, iflag, bccoeffs, fcn_data)
use mp_types
implicit none

real(kind(1e0)), intent(in) :: tx
integer, intent(inout) :: iflag
real(kind(1e0)), dimension(:, :), intent(out) :: bccoeffs
type(s_fcn_data), optional :: fcn_data

real(kind(1e0)) :: strike_price, spread, bet, &
 interest_rate, df

strike_price = fcn_data % rdata(1)
bet = fcn_data % rdata(2)

```

```

spread = fcn_data % rdata(3)
interest_rate = fcn_data % rdata(6)
select case (iflag)
 case (1)
 bccoefs(1,1:4) = ((/1.0e0, 0.0e0, 0.0e0, 0.0e0/))
 bccoefs(2,1:4) = ((/0.0e0, 1.0e0, 0.0e0, 0.0e0/))
 bccoefs(3,1:4) = ((/0.0e0, 0.0e0, 1.0e0, 0.0e0/))
 case (2)
! This is the discount factor using the risk-free
! interest rate
 df = exp(interest_rate * tx)
! Use flag passed to decide on boundary condition -
 select case (fcn_data % idata(1))
 case(1)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, bet*df/)
 case(2)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, &
 (spread-strike_price)*df/)
 end select
 bccoefs(2,1:4) = (/0.0e0, 1.0e0, 0.0e0, 0.0e0/)
 bccoefs(3,1:4) = (/0.0e0, 0.0e0, 1.0e0, 0.0e0/)
 return
 end select
! Note no time dependence in case (1) for iflag
 iflag = 0
end subroutine fkbc_call

```

## Output

```

European Option Value for A Bet
and a Vertical Spread, 3 and 6 Months Prior to Expiry
Number of equally spaced spline knots 61
Number of unknowns 183
Strike= 10.00, Sigma= 0.40, Interest Rate= 0.10
Bet = 2.00, Spread Value = 15.00

```

| Underlying |        | A Bet  | Vertical | Spread |
|------------|--------|--------|----------|--------|
| 2.0000     | 0.0000 | 0.0000 | 0.0000   | 0.0000 |
| 4.0000     | 0.0000 | 0.0014 | 0.0000   | 0.0006 |
| 6.0000     | 0.0110 | 0.0723 | 0.0039   | 0.0447 |
| 8.0000     | 0.2691 | 0.4302 | 0.1478   | 0.3832 |
| 10.0000    | 0.9948 | 0.9781 | 0.8909   | 1.1926 |
| 12.0000    | 1.6094 | 1.4290 | 2.1911   | 2.2273 |
| 14.0000    | 1.8655 | 1.6922 | 3.4254   | 3.1553 |
| 16.0000    | 1.9338 | 1.8175 | 4.2263   | 3.8264 |
| 18.0000    | 1.9476 | 1.8700 | 4.6264   | 4.2492 |
| 20.0000    | 1.9501 | 1.8904 | 4.7911   | 4.4921 |
| 22.0000    | 1.9505 | 1.8979 | 4.8497   | 4.6231 |
| 24.0000    | 1.9506 | 1.9007 | 4.8685   | 4.6909 |

## Example 4 – Convertible Bonds

This example evaluates the price of a convertible bond. Here, convertibility means that the bond may, at any time of the holder's choosing, be converted to a multiple of the specified asset. Thus a convertible bond with price  $x$  returns an amount  $K$  at time  $T$  *unless* the owner has converted the

bond to  $\nu x$ ,  $\nu \geq 1$ , units of the asset at some time prior to  $T$ . This definition, the differential equation and boundary conditions are given in Chapter 18 of Wilmott et al. (1996). Using a constant interest rate and volatility factor, the parameters and boundary conditions are:

1. Bond face value  $K = \{1\}$ , conversion factor  $\nu = 1.125$
2. Volatility  $\sigma = \{0.25\}$
3. Times until expiration =  $\{1/2, 1\}$
4. Interest rate  $r = 0.1$ , dividend  $D = 0.02$
5.  $x_{\min} = 0$ ,  $x_{\max} = 4$
6.  $nx = 61$ ,  $n = 3 \times nx = 183$
7. Boundary conditions  $f(0, t) = K \exp(-r(T - t))$ ,  $f(x_{\max}, t) = \nu x_{\max}$
8. Terminal data  $f(x, T) = \max(K, \nu x)$
9. Constraint for bond holder  $f(x, t) \geq \nu x$

Note that the error tolerance is set to a pure absolute error of value  $10^{-3}$ . The free boundary constraint  $f(x, t) \geq \nu x$  is achieved by use of a non-linear forcing term in the subroutine `fkforce_cbond`. The terminal conditions are provided with the user subroutine `fkinit_cbond`.

[Link to example source \(feynman\\_kac\\_ex4.f90\)](#)

```
! Compute value of a Convertible Bond
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use umach_int

 implicit none
! The face value
 real(kind(1e0)) :: ks = 1.0e0
! The sigma or volatility value
 real(kind(1e0)) :: sigma = 0.25e0

! Time values for the options
 integer, parameter :: nt = 2
 real(kind(1e0)) :: time(nt)=(/0.5e0, 1.0e0/)
! Values of the underlying where evaluation are made
 integer, parameter :: nv = 13
 integer, parameter :: nlbc = 3, nrbc = 3, ndeg = 6
 integer :: i
 real(kind(1e0)) :: xs(nv) = (/((i-1)*0.25e0, i=1, nv)/)
! Value of the interest rate, continuous dividend and factor
 real(kind(1e0)) :: r = 0.1e0, dividend = 0.02e0, &
```

```

 factor =1.125e0
! Values of the min and max underlying values modeled
 real(kind(1e0)) :: x_min = 0.0e0, x_max = 4.0e0
! Define parameters for the integration step.
 integer, parameter :: nx = 61, nint = nx-1, n = 3*nx
 real(kind(1e0)) :: xgrid(nx), y(n,0:nt), yprime(n,0:nt), &
 dx, f(nv,0:nt)
! Relative and absolute error tolerances
 real(kind(1e0)) :: atol(1), rtol(1)
 type(s_fcn_data) fcn_data
 real(kind(1e0)), external :: fkcoef_cbond, fkinitcond_cbond
 external fkbc_cbond, fkforce_cbond, fkinit_cbond

 integer :: nout

 call umach(2,nout)
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
 allocate(fcn_data % rdata (7), fcn_data % idata (1))

! Define an equally-spaced grid of points for the underlying price
 dx = (x_max - x_min)/real(nint)
 xgrid(1) = x_min
 xgrid(nx) = x_max

 do i=2,nx-1
 xgrid(i) = xgrid(i-1) + dx
 end do

! Use a pure absolute error tolerance for the integration
! The default values require too much integration time.
 atol(1) = 1.0e-3
 rtol(1) = 0.0e0

! Pass the data for evaluation
 fcn_data % rdata = (/ks,x_max,sigma,r,dividend,factor,&
 atol(1)/)
 fcn_data % idata = (/ndeg/)

! Compute value of convertible bond
 call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef_cbond,&
 fkinitcond_cbond, fkbc_cbond, y, yprime,&
 ATOL=atol, RTOL=rtol, FKINIT = fkinit_cbond,&
 FKFORCE = fkforce_cbond, FCN_DATA = fcn_data)

! Evaluate and display solutions at vector of points XS(:), at each
! time value prior to expiration.
 do i=0,nt
 f(:,i) = hqsval (xs, xgrid, y(:,i))
 end do

 write(nout,'(T05,A)')&
 "Convertible Bond Value, 0+, 6 and 12 Months Prior to Expiry"

 write(nout,'(T08,"Number of equally spaced spline knots ",I4,'//&
 '/T08,"Number of unknowns ",I4)') NX,N

```

```

write(nout, '(T08,"Strike = ",F5.2,", Sigma =", F5.2, '//&
'T08,"Interest Rate =",F5.2,", Dividend =",F5.2, '//&
"', Factor = ",F5.3, '//T08,"Underlying", T26,"Bond Value", '//&
'/(T10,4F8.4))') ks,sigma,r,dividend,factor,&
(xs(i), f(i,0:nt),i=1,nv)
end

! These routines define the coefficients, payoff, boundary
! conditions and forcing term.
function fkcoef_cbond(x, tx, iflag, fcn_data) result(value)
use mp_types
implicit none
integer, intent(inout) :: iflag
real(kind(1e0)), intent(in) :: x, tx
type(s_fcn_data), optional :: fcn_data
real(kind(1e0)) :: value

real(kind(1e0)) :: sigma, interest_rate, &
dividend, zero = 0.e0

sigma = fcn_data % rdata(3)
interest_rate = fcn_data % rdata(4)
dividend = fcn_data % rdata(5)

select case (iflag)
case (1)
! The coefficient derivative d(sigma)/dx
value = sigma
! The coefficient sigma(x)
case (2)
value = sigma * x
case (3)
! The coefficient mu(x)
value = (interest_rate - dividend) * x
case (4)
! The coefficient kappa(x)
value = interest_rate
end select
! Note that there is no time dependence
iflag = 0
return
end function fkcoef_cbond

function fkinitcond_cbond(x, fcn_data) result(value)
use mp_types
implicit none
real(kind(1e0)), intent(in) :: x
type(s_fcn_data), optional :: fcn_data
real(kind(1e0)) :: value

real(kind(1e0)) :: strike_price, factor

strike_price = fcn_data % rdata(1)
factor = fcn_data % rdata(6)
value = max(factor * x, strike_price)
return

```

```

end function fkinitcond_cbond

subroutine fkbc_cbond (tx, iflag, bccoefs, fcn_data)
use mp_types
implicit none
real(kind(1e0)), intent(in) :: tx
integer, intent(inout) :: iflag
real(kind(1e0)), dimension(:, :), intent(out) :: bccoefs
type(s_fcn_data), optional :: fcn_data

real(kind(1e0)) :: interest_rate, strike_price, dp, &
 factor, x_max

select case (iflag)
case (1)
 strike_price = fcn_data % rdata(1)
 interest_rate = fcn_data % rdata(4)
 dp = strike_price * exp(tx*interest_rate)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, dp/)
 bccoefs(2,1:4) = (/0.0e0, 1.0e0, 0.0e0, 0.0e0/)
 bccoefs(3,1:4) = (/0.0e0, 0.0e0, 1.0e0, 0.0e0/)
 return
case (2)
 x_max = fcn_data % rdata(2)
 factor = fcn_data % rdata(6)
 bccoefs(1,1:4) = (/1.0e0, 0.0e0, 0.0e0, factor * x_max/)
 bccoefs(2,1:4) = (/0.0e0, 1.0e0, 0.0e0, factor/)
 bccoefs(3,1:4) = (/0.0e0, 0.0e0, 1.0e0, 0.0e0/)
end select
! Note no time dependence
iflag = 0
return
end subroutine fkbc_cbond

subroutine fkforce_cbond (interval, t, hx, y, xlocal, qw, u, &
 phi, dphi, fcn_data)
use mp_types
implicit none
integer :: i, j, l
integer, parameter :: local = 6
integer, intent(in) :: interval
real(kind(1.e0)), intent(in) :: y(:, t, hx, qw(:), xlocal(:), &
 u(:, :))
real(kind(1.e0)), intent(out) :: phi(:, dphi(:, :))

integer :: ndeg
real(kind(1.e0)) :: yl(local), bf(local)
real(kind(1.e0)) :: value, strike_price, interest_rate, &
 zero = 0.0e0, one = 1.0e0, rt, mu, factor
type(s_fcn_data), optional :: fcn_data

yl = y(3*interval-2:3*interval+3)
phi = zero
dphi = zero
value = fcn_data % rdata(7)
strike_price = fcn_data % rdata(1)

```

```

interest_rate = fcn_data % rdata(4)
factor = fcn_data % rdata(6)
ndeg = fcn_data % idata(1)
mu = 2
! This is the local definition of the forcing term
! It "forces" the constraint f >= factor*x.
do j=1,local
 do l = 1,ndeg
 bf(1:3) = u(l,1:3)
 bf(4:6) = u(l,7:9)
 rt = dot_product(y1,bf)
 rt = value/(rt + value - factor * xlocal(l))
 phi(j) = phi(j) + qw(l) * bf(j) * rt**mu
 end do
end do
phi = -phi * hx * factor * strike_price
! This is the local derivative matrix for the forcing term -
do j=1,local
 do i = 1,local
 do l=1,ndeg
 bf(1:3) = u(L,1:3)
 bf(4:6) = u(L,7:9)
 rt = dot_product(y1,bf)
 rt = one/(rt + value - factor * xlocal(l))
 dphi(i,j) = dphi(i,j) + qw(l) * bf(i) * bf(j) &
 * (value * rt)**mu * rt
 end do
 end do
end do
dphi = -mu * dphi * hx * factor * strike_price
return
end subroutine fkforce_cbond

subroutine fkinit_cbond(xgrid,tgrid,t,yprime,y,atol,rtol,&
 fcn_data)

use mp_types
implicit none
real(kind(1e0)), intent(inout) :: y(:), atol(:), rtol(:)
real(kind(1e0)), intent(in) :: xgrid(:), tgrid(:), yprime(:), &
 t
type(s_fcn_data), optional :: fcn_data

integer :: i
if (t == 0.0e0) then
! Set initial data precisely.
do i=1,size(Xgrid)
 if (xgrid(i)*fcn_data % rdata(6) <&
 fcn_data % rdata(1)) then
 y(3*i-2) = fcn_data % rdata(1)
 y(3*i-1) = 0.0e0
 y(3*i) = 0.0e0
 else
 y(3*i-2) = xgrid(i) * fcn_data % rdata(6)
 y(3*i-1) = fcn_data % rdata(6)
 y(3*i) = 0.0e0
 end if
end do

```

```

end if
end subroutine fkinit_cbond

```

## Output

```

Convertible Bond Value, 0+, 6 and 12 Months Prior to Expiry
Number of equally spaced spline knots 61
Number of unknowns 183
Strike= 1.00, Sigma= 0.25
Interest Rate= 0.10, Dividend= 0.02, Factor= 1.125

```

| Underlying | Bond Value |        |        |
|------------|------------|--------|--------|
| 0.0000     | 1.0000     | 0.9512 | 0.9048 |
| 0.2500     | 1.0000     | 0.9512 | 0.9049 |
| 0.5000     | 1.0000     | 0.9513 | 0.9065 |
| 0.7500     | 1.0000     | 0.9737 | 0.9605 |
| 1.0000     | 1.1250     | 1.1416 | 1.1464 |
| 1.2500     | 1.4062     | 1.4117 | 1.4121 |
| 1.5000     | 1.6875     | 1.6922 | 1.6922 |
| 1.7500     | 1.9688     | 1.9731 | 1.9731 |
| 2.0000     | 2.2500     | 2.2540 | 2.2540 |
| 2.2500     | 2.5312     | 2.5349 | 2.5349 |
| 2.5000     | 2.8125     | 2.8160 | 2.8160 |
| 2.7500     | 3.0938     | 3.0970 | 3.0970 |
| 3.0000     | 3.3750     | 3.3781 | 3.3781 |

## Example 5 – A Non-Standard American Option

This example illustrates a method for evaluating a certain “Bermudan Style” or non-standard American option. These options are American Style options restricted to certain dates where the option may be exercised. Since this agreement gives the holder more opportunity than a European option, it is worth more. But since the holder can only exercise at certain times it is worth no more than the American style option value that can be exercised at any time. Our solution method uses the same model and data as in Example 2, but allows exercise at weekly intervals. Thus we integrate, for half a year, over each weekly interval using a European style Black-Scholes model, but with initial data at each new week taken from the corresponding values of the American style option.

[Link to example source \(feynman\\_kac\\_ex5.f90\)](#)

```

! Compute Bermudan-Style Option Premium for Vanilla Put
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use umach_int

 implicit none
 integer :: nout
! The strike price
 real(kind(1e0)) :: ks = 10.0e0
! The sigma value
 real(kind(1e0)) :: sigma = 0.4e0
! Program working stores
 real(kind(1e0)) :: week

```

```

! Time values for the options
 integer, parameter :: nt = 26
 integer, parameter :: ndeg = 6
 real(kind(1e0)) :: time(nt), time_end = 0.5e0
! Values of the underlying where evaluation are made
 integer, parameter :: nv = 9, nlbc = 2, nrbc = 3
 integer :: i
 real(kind(1e0)) :: xs(nv) = (/((i-1)*2.0e0,i=1,nv)/)
! Value of the interest rate and continuous dividend
 real(kind(1e0)) :: r = 0.1e0, dividend = 0.0e0
! Values of the min and max underlying values modeled
 real(kind(1e0)) :: x_min = 0.0e0, x_max = 30.0e0

! Define parameters for the integration step.
 integer, parameter :: nx = 61, nint = nx-1, n = 3*nx
 real(kind(1e0)) :: xgrid(nx), yb(n,0:nt), ybprime(n,0:nt), &
 ya(n,0:nt), yaprime(n,0:nt), &
 ytemp(n,0:1), ytempprime(n,0:1), &
 dx, fb(nv,nt), fa(nv,nt)

 real(kind(1e0)) :: atol
 type(s_fcn_data) fcn_data_amer, fcn_data_berm
 real(kind(1e0)), external :: fkcoef_put, fkinitcond_put
 external fkbc_put, fkforce_put, fkinit_amer_put, fkinit_berm_put

 call umach(2, nout)
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
 allocate(fcn_data_amer % rdata (6), fcn_data_amer % idata (1))
! Define an equally-spaced grid of points for the underlying price
 dx = (x_max-x_min)/real(nint)
 xgrid(1) = x_min
 xgrid(nx) = x_max
 do i=2,nx-1
 xgrid(i) = xgrid(i-1) + dx
 end do

! Place a breakpoint at the strike price.
 do i=1,nx
 if (xgrid(i) > ks) then
 xgrid(i-1) = ks
 exit
 end if
 end do

! Compute time values where American option is computed
 week = time_end/real(nt,kind(week))
 time(1) = week
 do i=2,nt-1
 time(i) = time(i-1) + week
 end do
 time(nt) = time_end

 atol = 1.0e-3
 fcn_data_amer % rdata = (/ks,x_max,sigma,r,dividend,atol/)
 fcn_data_amer % idata = (/ndeg/)

! Compute American Put Option Values at the weekly grid.

```

```

 call feynman_kac (xgrid, time, nlbc, nrbc, fkcoef_put,&
 fkinitcond_put, fkbc_put, ya, yaprime,&
 FKINIT = fkinit_amer_put,&
 FKFORCE = fkforce_put,&
 FCN_DATA = fcn_data_amer)
! Integrate once again over the weekly grid, using the American
! Option values as initial data for a piece-wise European option
! integration.

! Allocate space to hold coefficient data and initial values.
 allocate(fcn_data_berm % rdata(5+n))
 fcn_data_berm % rdata(1:5) = fcn_data_amer % rdata(1:5)
! Copy initial data so the payoff value is the same for
! American and Bermudan option values.
 yb(1:n,0) = ya(1:n,0)
 ybprime(1:n,0) = ya(1:n,0)

 do i=0,nt-1
! Move American Option values into place as initial conditions,
! but now integrating with European style over each period of
! the weekly grid.
 fcn_data_berm % rdata(6:) = ya(1:n,i)
 if (i .eq. 0) then
 call feynman_kac (xgrid, (/time(1)/), nlbc, nrbc,&
 fkcoef_put, fkinitcond_put, fkbc_put,&
 ytemp(:,0:1), ytempprime(:,0:1),&
 FKINIT = fkinit_berm_put,&
 FCN_DATA = fcn_data_berm)
 else
 call feynman_kac (xgrid, (/time(i+1)-time(i)/),&
 nlbc, nrbc, fkcoef_put,&
 fkinitcond_put, fkbc_put,&
 ytemp(:,0:1), ytempprime(:,0:1),&
 FKINIT = fkinit_berm_put,&
 FCN_DATA = fcn_data_berm)
 end if
! Record values of the Bermudan option at the end of each integration.
 yb(1:n,i+1) = ytemp(1:n,1)
 ybprime(1:n,i+1) = ytempprime(1:n,1)
 end do
! Evaluate solutions at vector of points XS(:), at each time value
! prior to expiration.
 do i=1,nt
 fa(:,i) = hqsval (xs, xgrid, ya(:,i))
 fb(:,i) = hqsval (xs, xgrid, yb(:,i))
 end do

 write(nout,'(T05,A)')&
 "American Option Premium for Vanilla Put, 6 Months "///&
 "Prior to Expiry"
 write(nout,'(T05,A)')&
 "Exercise Opportunities At Weekly Intervals"
 write(nout,'(T08,"Number of equally spaced spline knots ",'///&
 'I4,/T08,"Number of unknowns ",I4)') nx, n
 write(nout,'(T08,"Strike = ",F5.2,", Sigma =", F5.2,'///&
 '"', Interest Rate =",F5.2,///T08,"Underlying", '///&

```

```

 'T20,"Bermudan Style", T42,"American",'//&
 '/'(T10,F8.4, T26, F8.4, T42, F8.4)')&
 KS,SIGMA,R,&
 (xs(i), fb(i,nt:nt), fa(i,nt:nt),i=1,nv)
 end

! These subprograms set the coefficients, payoff, boundary
! conditions and forcing term for American and European Options.
 function fkcoef_put(x, tx, iflag, fcn_data_amer) &
 result(value)

 use mp_types
 implicit none
 integer, intent(inout) :: iflag
 real(kind(1e0)), intent(in) :: x, tx
 type(s_fcn_data), optional :: fcn_data_amer
 real(kind(1e0)) :: value

 real(kind(1e0)) :: sigma, interest_rate, dividend, zero=0.0e0

 sigma = fcn_data_amer % rdata(3)
 interest_rate = fcn_data_amer % rdata(4)
 dividend = fcn_data_amer % rdata(5)
 select case (iflag)
 case (1)
! The coefficient derivative d(sigma)/dx
 value = sigma
 case (2)
! The coefficient sigma(x)
 value = sigma * x
 case (3)
! The coefficient mu(x)
 value = (interest_rate - dividend) * x
 case (4)
! The coefficient kappa(x)
 value = interest_rate
 end select
! Note that there is no time dependence
 iflag = 0
 return
 end function fkcoef_put

 function fkinitcond_put(x, fcn_data_amer) result(value)
 use mp_types
 implicit none

 real(kind(1e0)), intent(in) :: x
 type(s_fcn_data), optional :: fcn_data_amer

 real(kind(1e0)) :: value
 real(kind(1e0)) :: strike_price, zero = 0.0e0

 strike_price = fcn_data_amer % rdata(1)
! The payoff function
 value = max(strike_price - x, zero)
 return
 end function fkinitcond_put

```

```

subroutine fkbc_put (tx, iflag, bccoeffs, fcn_data)
use mp_types
implicit none
real(kind(1e0)), intent(in) :: tx
integer, intent(inout) :: iflag
real(kind(1e0)), dimension(:,:), intent(out) :: bccoeffs
type (s_fcn_data), optional :: fcn_data

select case (iflag)
case (1)
 bccoeffs(1,1:4) = ((/0.0e0, 1.0e0, 0.0e0, -1.0e0/))
 bccoeffs(2,1:4) = ((/0.0e0, 0.0e0, 1.0e0, 0.0e0/))
case (2)
 bccoeffs(1,1:4) = ((/1.0e0, 0.0e0, 0.0e0, 0.0e0/))
 bccoeffs(2,1:4) = ((/0.0e0, 1.0e0, 0.0e0, 0.0e0/))
 bccoeffs(3,1:4) = ((/0.0e0, 0.0e0, 1.0e0, 0.0e0/))
end select
! Note no time dependence
iflag = 0
end subroutine fkbc_put

subroutine fkforce_put (interval, t, hx, y, xlocal, qw, u,&
 phi, dphi, fcn_data_amer)

use mp_types
implicit none
integer, parameter :: local = 6
integer :: i, j, l, ndeg
integer, intent(in) :: interval
real(kind(1.e0)), intent(in) :: y(:,), t, hx, qw(:,),&
 xlocal(:,), u(:,)
real(kind(1.e0)), intent(out) :: phi(:,), dphi(:,)
type (s_fcn_data), optional :: fcn_data_amer

real(kind(1.e0)) :: yl(local), bf(local)
real(kind(1.e0)) :: value, strike_price, interest_rate,&
 zero = 0.e0, one = 1.e0, rt, mu

yl = y(3*interval-2:3*interval+3)
phi = zero
value = fcn_data_amer % rdata(6)
strike_price = fcn_data_amer % rdata(1)
interest_rate = fcn_data_amer % rdata(4)
ndeg = fcn_data_amer % idata(1)

mu = 2
! This is the local definition of the forcing term
do j=1,local
 do l=1,ndeg
 bf(1:3) = U(L,1:3)
 bf(4:6) = U(L,7:9)
 rt = dot_product(YL,BF)
 rt = value/(rt + value-(strike_price-xlocal(1)))
 phi(j) = phi(j) + qw(l) * bf(j) * rt**mu
 end do
end do
phi = -phi * hx * interest_rate * strike_price

```

```

! This is the local derivative matrix for the forcing term
dphi = zero
do j=1,local
 do i = 1,local
 do l=1,ndeg
 bf(1:3) = u(L,1:3)
 bf(4:6) = u(L,7:9)
 rt = dot_product(y1,bf)
 rt = one/(rt + value - (strike_price - xlocal(l)))
 dphi(i,j) = dphi(i,j) + qw(l) * bf(i) * bf(j) *&
 rt**(mu+1)
 end do
 end do
end do
dphi = mu * dphi * hx * value**mu * interest_rate *&
 strike_price
end subroutine fkforce_put

subroutine fkinit_amer_put(xgrid,tgrid,t,yprime,y,atol,rtol,&
 fcn_data_amer)

use mp_types
implicit none

real(kind(1e0)), intent(in) :: xgrid(:), tgrid(:), t,&
 yprime(:)
real(kind(1e0)), intent(inout) :: y(:), atol(:), rtol(:)
type(s_fcn_data), optional :: fcn_data_amer
integer :: i

if (t == 0.0e0) then
! Set initial data precisely. The strike price is a breakpoint.
! Average the derivative limit values from either side.
 do i=1,size(xgrid)
 if (xgrid(i) < fcn_data_amer % rdata(1)) then
 y(3*i-2) = fcn_data_amer % rdata(1) - xgrid(i)
 y(3*i-1) = -1.0e0
 y(3*i) = 0.0e0
 else if (xgrid(i) == fcn_data_amer % rdata(1)) then
 y(3*i-2) = 0.0e0
 y(3*i-1) = -0.5e0
 y(3*i) = 0.0e0
 else
 y(3*i-2) = 0.0e0
 y(3*i-1) = 0.0e0
 y(3*i) = 0.0e0
 end if
 end do
end if
end subroutine fkinit_amer_put

subroutine fkinit_berm_put(xgrid,tgrid,t,yprime,y,atol,rtol,&
 fcn_data_berm)

use mp_types
implicit none

real(kind(1e0)), intent(in) :: xgrid(:), tgrid(:), t,&
 yprime(:)

```

```

real(kind(1e0)), intent(inout) :: y(:), atol(:), rtol(:)
type(s_fcn_data), optional :: fcn_data_berm
integer :: i

if (t == 0.0e0) then
! Set initial data for each week at the previously computed
! American Option values. These coefficients are passed
! in the derived type fcn_data_berm.
do i=1,size(xgrid)
y(3*i-2) = fcn_data_berm % rdata(3+3*i)
y(3*i-1) = fcn_data_berm % rdata(4+3*i)
y(3*i) = fcn_data_berm % rdata(5+3*i)
end do
end if
end subroutine fkinit_berm_put

```

## Output

```

American Option Premium for Vanilla Put, 6 Months Prior to Expiry
Exercise Opportunities At Weekly Intervals
Number of equally spaced spline knots 61
Number of unknowns 183
Strike= 10.00, Sigma= 0.40, Interest Rate= 0.10

```

| Underlying | Bermudan Style | American |
|------------|----------------|----------|
| 0.0000     | 9.9808         | 10.0000  |
| 2.0000     | 7.9808         | 8.0000   |
| 4.0000     | 5.9808         | 6.0000   |
| 6.0000     | 3.9808         | 4.0000   |
| 8.0000     | 2.0924         | 2.0926   |
| 10.0000    | 0.9139         | 0.9138   |
| 12.0000    | 0.3570         | 0.3569   |
| 14.0000    | 0.1309         | 0.1309   |
| 16.0000    | 0.0468         | 0.0469   |

## Example 6 – Oxygen Diffusion Problem

Our previous examples are from the field of financial engineering. A final example is a physical model. The Oxygen Diffusion Problem is summarized in Crank [4], p. 10-20, 261-262. We present the numerical treatment of the transformed one-dimensional system

$$f_t + f_{xx} = 1, 0 \leq x \leq s(t), f(0, x) = \frac{1}{2}(1-x)^2, 0 \leq x \leq 1,$$

$$f(t, 1) = f_x(t, 0) = 0, t < 0.$$

A slight difference from the Crank development is that we have reflected the time variable  $t \rightarrow -t$  to match our form of the Feynman-Kac equation. We have a free boundary problem because the interface  $s(t)$  is implicit. This interface is implicitly defined by solving the variational relation  $(f_t + f_{xx} - 1)f = 0, f \geq 0$ . The first factor is zero for  $0 \leq x < s(t)$  and the second factor is zero for  $s(t) \leq x \leq 1$ . We list the Feynman-Kac equation coefficients, forcing term and boundary conditions, followed by comments.

$$\sigma(x,t) = \sqrt{2}; \quad \frac{\partial \sigma}{\partial x} = 0$$

$$\mu(x,t) = 0$$

$$\kappa(x,t) = 0$$

$$p(x) = \frac{1}{2}(1-x)^2$$

$$\phi(f,x,t) \equiv 1 - \left( \frac{\varepsilon}{f + \varepsilon} \right)^\mu, \quad \varepsilon = ATOL, \mu = 2$$

$$f_x(0,t) = \exp(t/\varepsilon^2)$$

$$f(1,t) = f_x(1,t) = 0, t < 0$$

The  $\phi$  forcing term has the property of being almost the value 1 when the solution is larger than the factor  $\varepsilon$ . As the solution  $f \downarrow 0$ , the forcing term  $\phi$  is almost the value zero. These properties combine to approximately achieve the variational relation that defines the free boundary. Note that the arc of the free boundary is not explicitly available with this numerical method. We have used  $\varepsilon = ATOL$ , the requested absolute error tolerance for the numerical integration.

The boundary condition  $f_x(t,0) = 0, t < 0$ , is discontinuous as  $t \uparrow 0$ , since the initial data yields  $f_x(0,0) = 1$ . For the numerical integration we have chosen a boundary value function that starts with the value 1 at  $t = 0$  and rapidly blends to the value zero as the integration proceeds in the negative direction. It is necessary to give the integrator the opportunity to severely limit the step size near  $t = 0$  to achieve continuity during the blending.

In the example code, values of  $f(0,t)$  are checked against published values for certain values of  $t$ . Also checked are values of  $f(0,s(t)) = 0$  at published values of the free boundary, for the same values of  $t$ .

[Link to example source](#) (feynman\_kac\_ex6.f90)

```
! Integrate Oxygen Diffusion Model found in the book
! Crank, John. Free and Moving Boundary Problems,
! Oxford Univ. Press, (1984), p. 19-20 and p. 261-262.
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use norm_int
 use umach_int
 implicit none

 integer :: nout
 real(kind(1e0)), allocatable :: xgrid(:), tgrid(:), y(:,:), &
 yprime(:,:), f(:,:), s(:)
 real(kind(1e0)) :: dx, x_min, x_max, zero=0.0e0, one=1.0e0
```

```

real(kind(1e0)) :: atol(1), rtol(1)
type(s_fcn_data) :: fk_ox2
integer :: i, nint, n, nunit, ntimes = 8
integer, parameter :: ndeg = 6, nlbc = 1, nrbc = 2
real(kind(1e0)), external :: fkcoef_ox2, fkinitcond_ox2
external fkbc_ox2, fkforce_ox2

call umach(2,nout)

! Define number of equally spaced intervals for elements
nint = 100
! Allocate the space needed for the integration process
n = 3*(nint+1)
allocate(xgrid(nint+1), y(n,0:ntimes), yprime(n,0:ntimes), &
 tgrid(ntimes), f(1,ntimes), s(ntimes))

! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
allocate(fk_ox2 % rdata (1), fk_ox2 % idata (1))

atol(1) = 0.5e-2
rtol(1) = 0.5e-2
fk_ox2 % rdata(1) = atol(1)
fk_ox2 % idata(1) = ndeg

! Define interval endpoints
x_min = zero
x_max = one
! Define interval widths
dx = (x_max-x_min)/real(nint)
xgrid(1) = x_min
xgrid(nint+1) = x_max
! Define grid points of interval
do i=2,nint
 xgrid(i) = xgrid(i-1) + dx
end do
! Define time integration output points
! These correspond to published values in Crank's book, p. 261-262
tgrid = (/0.04e0,0.06e0,0.10e0,0.12e0,0.14e0,&
 0.16e0,0.18e0,0.185e0/)
call feynman_kac (xgrid, tgrid, nlbc, nrbc, fkcoef_ox2,&
 fkinitcond_ox2, fkbc_ox2, y, yprime,&
 ATOL = atol, RTOL = rtol,&
 FKFORCE = fkforce_ox2, FCN_DATA = fk_ox2)
! Summarize output at the left end
do i=1,ntimes
 f(:,i)= hqsval (/zero/), xgrid, y(:,i))
end do
! Check differences of evaluation and published left end values
f(1,:) = f(1,:) - (/2.743e-01, 2.236e-01, 1.432e-01,&
 1.091e-01, 7.786e-02, 4.883e-02, 2.179e-02, 1.534e-02/)
write(nout,*) "Oxygen Depletion Model, from Crank's "///&
 "Book, p. 261-262,"
write(nout,*) "'Free and Moving Boundary Value Problems'"
if (norm(f(1,:)) < ntimes * atol(1)) then
 write(nout,*) "FEYNMAN_KAC Example 6 - Fixed Sealed "///&

```

```

 "Surface Values are correct"
 else
 write(nout,*) "FEYNMAN_KAC Example 6 - does not agree with"//&
 " published left end values"
 end if
! Define known position of free boundary at the time points
 s = (/0.9992e0,0.9918e0,0.9350e0,0.8792e0,&
 0.7989e0,0.6834e0,0.5011e0,0.4334e0/)

! Evaluate and verify solution is small near free boundary -
 do i=1,ntimes
 f(:,i) = hqsval ((/s(i)/), xgrid, y(:,i))
 end do

 if (norm(f(1,:)) < ntimes * atol(1)) then
 write(nout,*) "FEYNMAN_KAC Example 6 - Free Boundary " //&
 "Position Values are correct"
 else
 write(nout,*) "FEYNMAN_KAC Example 6 - does not agree " //&
 "with published free boundary values"
 end if
end

function fkcoef_ox2 (x, tx, iflag, fk_ox2) result(value)
 use mp_types
 implicit none
! Coefficient valuation routine for the Oxygen Diffusion
! Model found in Crank's book, p. 20
! Input/Ouput variables
 integer, intent(inout) :: iflag
 real(kind(1e0)), intent(in) :: x, tx
 type(s_fcn_data), optional :: fk_ox2
 real(kind(1e0)) :: value

! Local variables
 real(kind(1e0)) :: zero = 0.0e0, two = 2.0e0

 select case (iflag)
 case (1) ! Factor DSigma/Dx(x,t)
 value = zero
 case (2) ! Factor Sigma(x,t)
 value = sqrt(two)
 case (3) ! Factor Mu (x,t)
 value = zero
 case (4) ! Factor Kappa (x,t)
 value = zero
 end select
! Signal no dependence on tx=t=time for any coefficient.
 iflag = 0
 return
end function fkcoef_ox2

function fkinitcond_ox2(x, fk_ox2) result(value)
 use mp_types
 implicit none
 real(kind(1e0)), intent(in) :: x
 type (s_fcn_data), optional :: fk_ox2

```

```

real(kind(1e0)) :: value

real(kind(1e0)) :: half = 0.5e0, one = 1.0e0

value = half * (one - x)**2
return
end function fkinitcond_ox2

subroutine fkbc_ox2 (tx, iflag, bccoeffs, fk_ox2)
use mp_types
implicit none
! Evaluation routine for Oxygen Diffusion Model
! boundary conditions.
! Input/Output variables
real(kind(1e0)), intent(in) :: tx
integer, intent(inout) :: iflag
real(kind(1e0)), dimension(:, :), intent(out) :: bccoeffs
type (s_fcn_data), optional :: fk_ox2

! Local variables
real(kind(1e0)) :: zero = 0.0e0, one = 1.0e0, atol

atol = fk_ox2 % rdata(1)
select case (iflag)
case (1) ! Left Boundary Condition, at X_min=0
! There is a rapid blending of the boundary condition to achieve
! a zero derivative value at the left end.
! The initial data has the derivative with value one.
! This boundary condition essentially abruptly changes that
! derivative value to zero.
! Returning iflag=1 signals time dependence. This is important
! for this problem.
bccoeffs(1,1:4) = (/0.0e0, one, 0.0e0, exp(tx/atol**2)/)
return
case (2) ! Right Boundary Condition, at X_max=1
bccoeffs(1,1:4) = (/one, 0.0e0, 0.0e0, 0.0e0/)
bccoeffs(2,1:4) = (/0.0e0, one, 0.0e0, 0.0e0/)
end select
iflag = 0 ! Signal no dependence on tx=time.
end subroutine fkbc_ox2

subroutine fkforce_ox2 (interval, t, hx, y, xlocal, qw, u, &
phi, dphi, fk_ox2)
! Evaluation routine for Oxygen Diffusion model forcing function.
use mp_types
implicit none
integer, parameter :: local = 6
integer :: i, j, l, mu, ndeg
integer, intent(in) :: interval

real(kind(1e0)), intent(in) :: y(:), t, hx, qw(:), &
xlocal(:), u(:, :)
real(kind(1e0)), intent(out) :: phi(:), dphi(:, :)
type (s_fcn_data), optional :: fk_ox2

real(kind(1e0)) :: yl(local), bf(local)

```

```

real(kind(1e0)) :: value, zero = 0.0e0, one = 1.0e0, rt

y1 = y(3*interval-2:3*interval+3)
phi = zero

value = fk_ox2 % rdata(1)
ndeg = fk_ox2 % idata(1)

mu = 2
do j=1,local
 do l=1,ndeg
 bf(1:3) = u(1,1:3)
 bf(4:6) = u(1,7:9)
 rt = dot_product(y1,bf)
 rt = one - (value/(rt + value))*mu
 phi(j) = phi(j) + qw(l) * bf(j) * RT
 end do
end do
phi = phi * hx
! This is the local derivative matrix for the forcing term -
dphi = zero
do j=1,local
 do i = 1,local
 do l=1,ndeg
 bf(1:3) = u(1,1:3)
 bf(4:6) = u(1,7:9)
 rt = dot_product(y1,bf)
 rt = one/(rt + value)
 dphi(i,j) = dphi(i,j) + qw(l) * bf(i) * bf(j) *&
 rt**(mu+1)
 end do
 end do
end do
dphi = mu * dphi * hx * value**mu
return
end subroutine fkforce_ox2

```

## Output

Oxygen Depletion Model, from Crank's Book, p. 261-262,  
 'Free and Moving Boundary Value Problems'  
 FEYNMAN\_KAC Example 6 - Fixed Sealed Surface Values are correct  
 FEYNMAN\_KAC Example 6 - Free Boundary Position Values are correct

## Example 7 – Calculating the Greeks

In this example, routine FEYNMAN\_KAC is used to solve for the Greeks, i.e. various derivatives of Feynman-Kac (FK) solutions applicable to the pricing of options and related financial derivatives. In order to illustrate and verify these calculations, the Greeks are calculated by two methods. The first method involves the FK solution to the diffusion model for call options given in Example 1 for the Black-Scholes (BS) case, i.e.  $\alpha = 2$ . The second method calculates the Greeks using the closed-form BS evaluations which can be found at [http://en.wikipedia.org/wiki/The\\_Greeks](http://en.wikipedia.org/wiki/The_Greeks).

This example calculates FK and BS solutions  $V(S,t)$  to the BS problem and the following Greeks:

- $Delta = \frac{\partial V}{\partial S}$

is the first derivative of the *Value*,  $V(S, t)$ , of a portfolio of derivative security derived from underlying instrument with respect to the underlying instrument's price  $S$ ;

- $Gamma = \frac{\partial^2 V}{\partial S^2}$  ;

- $Theta = -\frac{\partial V}{\partial t}$  is the negative first derivative of  $V$  with respect to time  $t$ ;

- $Charm = \frac{\partial^2 V}{\partial S \partial t}$  ;

- $Color = \frac{\partial^3 V}{\partial S^2 \partial t}$  ;

- $Rho = -\frac{\partial V}{\partial r}$  is the first derivative of  $V$  with respect to the risk-free rate  $r$ ;

- $Vega = \frac{\partial V}{\partial \sigma}$  measures sensivity to volatility parameter  $\sigma$  of the underlying  $S$ ;

- $Volga = \frac{\partial^2 V}{\partial \sigma^2}$  ;

- $Vanna = \frac{\partial^2 V}{\partial S \partial \sigma}$  ;

- $Speed = \frac{\partial^3 V}{\partial S^3}$  .

*Intrinsic Greeks* include derivatives involving only  $S$  and  $t$ , the intrinsic FK arguments. In the above list, *Value*, *Delta*, *Gamma*, *Theta*, *Charm*, *Color* and *Speed* are all intrinsic Greeks. As is discussed in Hanson, R. (2008) "[Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements](#)", the expansion of the FK solution function  $V(S, t)$  in terms of quintic polynomial functions defined on  $S$ -grid subintervals and subject to continuity constraints in derivatives 0, 1 and 2 across the boundaries of these subintervals allows *Value*, *Delta*, *Gamma*, *Theta*, *Charm* and *Color* to be calculated directly by routines FEYNMAN\_KAC and HQSVAL.

*Non-intrinsic Greeks* are derivatives of  $V$  involving FK parameters other than the intrinsic arguments  $S$  and  $t$ , such as  $r$  and  $\sigma$ . Non-intrinsic Greeks in the above list include *Rho*, *Vega*, *Volga* and *Vanna*. In order to calculate non-intrinsic Greek (parameter) derivatives or intrinsic Greek  $S$ -derivatives beyond the second (such as *Speed*) or  $t$ -derivatives beyond the first, the entire FK solution must be calculated 3 times (for a parabolic fit) or five times (for a quartic fit), at the

point where the derivative is to be evaluated and at nearby points located symmetrically on either side.

Using a Taylor series expansion of  $f(\sigma + \varepsilon)$  truncated to  $m + 1$  terms (to allow an  $m$ -degree polynomial fit of  $m + 1$  data points),

$$f(\sigma + \varepsilon) = \sum_{n=0}^m \frac{f^{(n)}(\sigma)}{n!} \varepsilon^n,$$

we are able to derive the following parabolic (3 point) estimation of first and second derivatives  $f^{(1)}(\sigma)$  and  $f^{(2)}(\sigma)$  in terms of the three values  $f(\sigma - \varepsilon)$ ,  $f(\sigma)$  and  $f(\sigma + \varepsilon)$ , where  $\varepsilon = \varepsilon_{frac} \sigma$  and  $0 < \varepsilon_{frac} \ll 1$ :

$$f^{(1)}(\sigma) \equiv \frac{\partial f(\sigma)}{\partial \sigma} \approx f^{[1]}(\sigma, \varepsilon) \equiv \frac{f(\sigma + \varepsilon) - f(\sigma - \varepsilon)}{2\varepsilon},$$

$$f^{(2)}(\sigma) \equiv \frac{\partial^2 f(\sigma)}{\partial \sigma^2} \approx f^{[2]}(\sigma, \varepsilon) \equiv \frac{f(\sigma + \varepsilon) + f(\sigma - \varepsilon) - 2f(\sigma)}{\varepsilon^2}.$$

Similarly, the quartic (5 point) estimation of  $f^{(1)}(\sigma)$  and  $f^{(2)}(\sigma)$  in terms of  $f(\sigma - 2\varepsilon)$ ,  $f(\sigma - \varepsilon)$ ,  $f(\sigma)$ ,  $f(\sigma + \varepsilon)$  and  $f(\sigma + 2\varepsilon)$  is:

$$f^{(1)}(\sigma) \approx \frac{4}{3} f^{[1]}(\sigma, \varepsilon) - \frac{1}{3} f^{[1]}(\sigma, 2\varepsilon)$$

$$f^{(2)}(\sigma) \approx \frac{4}{3} f^{[2]}(\sigma, \varepsilon) - \frac{1}{3} f^{[2]}(\sigma, 2\varepsilon).$$

For our example, the quartic estimate does not appear to be significantly better than the parabolic estimate, so we have produced only parabolic estimates by setting variable `iquart` to 0. The user may try the example with the quartic estimate simply by setting `iquart` to 1.

As is pointed out in [Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements](#), the quintic polynomial expansion function used by `FEYNMAN_KAC` only allows for continuous derivatives through the second derivative. While up to fifth derivatives can be calculated from the quintic expansion (indeed function `HQSVAL` will allow the third derivative to be calculated by setting optional argument `IDERIV` to 3, as is done in this example), the accuracy is compromised by the inherent lack of continuity across grid points (i.e. subinterval boundaries).

The accurate second derivatives in  $S$  returned by function `HQSVAL` can be leveraged into a third derivative estimate by calculating three FK second derivative solutions, the first solution for grid and evaluation point set  $\{S, f^{(2)}(S)\}$  and the second and third solutions for solution grid and evaluation point sets  $\{S + \varepsilon, f^{(2)}(S + \varepsilon)\}$  and  $\{S - \varepsilon, f^{(2)}(S - \varepsilon)\}$ , where the solution grid and evaluation point sets are shifted up and down by  $\varepsilon$ . In this example,  $\varepsilon$  is set to  $\varepsilon_{frac} \bar{S}$ , where  $\bar{S}$  is the average value of  $S$  over the range of grid values and  $0 < \varepsilon_{frac} \ll 1$ . The third derivative solution can then be obtained using the parabolic estimate

$$f^{(3)}(S) \equiv \frac{\partial f^{(2)}(\sigma)}{\partial S} \approx \frac{f^{(2)}(S + \varepsilon) + f^{(2)}(S - \varepsilon)}{2\varepsilon}$$

This procedure is implemented in the current example to calculate the Greek *Speed*. (For comparison purposes, *Speed* is also calculated directly by setting the optional argument `IDERIV` to 3 in the call to function `HQSVAL`. The output from this direct calculation is called “*Speed2*”.)

To reach better accuracy results, all computations are done in double precision.

The average and maximum relative errors (defined as the absolute value of the difference between the BS and FK values divided by the BS value) for each of the Greeks is given at the end of the output. (These relative error statistics are given for nine combinations of Strike Price and volatility, but only one of the nine combinations is actually printed in the output.) Both intrinsic and non-intrinsic Greeks have good accuracy (average relative error is in the range 0.01 – 0.00001) except for *Volga*, which has an average relative error of about 0.05. This is probably a result of the fact that *Volga* involves differences of differences, which will more quickly erode accuracy than calculations using only one difference to approximate a derivative. Possible ways to improve upon the 2 to 4 significant digits of accuracy achieved in this example include increasing FK integration accuracy by reducing the initial stepsize (via optional argument `RINITSTEPSIZE`), by choosing more closely spaced  $S$  and  $t$  grid points (by adjusting `FEYNMAN_KAC`’s input arrays `XGRID` and `TGRID`) and by adjusting  $\varepsilon_{frac}$  so that the central differences used to calculate the derivatives are not too small to compromise accuracy.

[Link to example source](#) (feynman\_kac\_ex7.f90)

```
! Greeks computation
 use feynman_kac_int
 use hqsval_int
 use mp_types
 use anordf_int
 use const_int
 use umach_int

 implicit none

 real(kind(1d0)), external :: fkcoef, fkinitcond
 external fkbc

! The set of strike prices
 real(kind(1d0)) :: ks(3) = (/15.0d0,20.0d0,25.0d0/)
! The set of sigma values
 real(kind(1d0)) :: sigma(3) = (/0.2d0, 0.3d0, 0.4d0/)
! The set of model diffusion powers: alpha = 2.0 ==> Black Scholes
 real(kind(1d0)) :: alpha(3) = (/2.0d0, 1.0d0, 0.0d0/)
! Time values for the options
 integer, parameter :: nt = 3
 real(kind(1d0)) :: time(nt)=(/1.d0/12., 4.d0/12., 7.d0/12./)
! Values of the min and max underlying values modeled
 real(kind(1d0)) :: x_min = 0.0d0, x_max = 60.0d0
! Value of the interest rate and continuous dividend
 real(kind(1d0)) :: r = 0.05d0, dividend = 0.0d0
! Values of the underlying where evaluations are made
```

```

integer, parameter :: nv = 3
real(kind(1d0)) :: eval_points(nv) = &
 (/19.0d0, 20.0d0, 21.0d0/)
! Define parameters for the integration step.
integer, parameter :: nx = 121, nint = nx-1, n = 3*nx
real(kind(1d0)) :: xgrid(nx), y(n,0:nt), yprime(n,0:nt)
type(d_fcn_data) fcn_data
! Number of left/right boundary conditions
integer, parameter :: nlbc = 3, nrbc = 3
! Further parameters for the integration step
real(kind(1d0)) :: dx, dx2, pi, sqrt2pi
! used to calc derivatives
real(kind(1d0)) :: epsfrac = .001d0
character(len=6) :: greek_name(12) = (/&
 " Value", " Delta", " Gamma", " Theta",&
 " Charm", " Color", " Vega", " Volga",&
 " Vanna", " Rho", " Speed", "Speed2"/)
! Time values for the options
real(kind(1d0)) :: rex(12), reavg(12)
integer :: irect(12)
integer :: i, i2, i3, j, ig, iquart, nout

real(kind(1d0)) ::&
 spline_values(nv,nt,12), spline_values1(nv,nt),&
 spline_valuesp(nv,nt), spline_valuesm(nv,nt),&
 spline_valuespp(nv,nt), spline_valuesmm(nv,nt),&
 xgridp(nx), xgridm(nx),xgridpp(nx), xgridmm(nx),&
 eval_pointsp(nv), eval_pointsp(nv),&
 eval_pointsm(nv), eval_pointsmm(nv),&
 BS_values(nv,nt,12), sVo_array(nt)

call umach(2, nout)
! Allocate space inside the derived type for holding
! data values. These are for the evaluation routines.
allocate(fcn_data % rdata (6))

pi = const('pi')
sqrt2pi = sqrt(2.0 * pi)
dx2 = epsfrac * 0.5d0 * (x_min + x_max)

! Compute Constant Elasticity of Variance Model for Vanilla Call
! Define equally-spaced grid of points for the underlying price

dx = (x_max - x_min)/real(nint)
xgrid(1) = x_min
xgrid(nx) = x_max
do i = 2,nx-1
 xgrid(i) = xgrid(i-1) + dx
end do

write(nout,'(T05,A)') "Constant Elasticity of Variance"//&
 " Model for Vanilla Call Option"
write(nout,'(T10,"Interest Rate: ", F7.3, T38,'//&
 '"Continuous Dividend: ", F7.3)') r, dividend
write(nout,'(T10,"Minimum and Maximum Prices of '//&
 ' Underlying: ", 2F7.2)') x_min, x_max
write(nout,'(T10,"Number of equally spaced spline knots:",'//&

```

```

 'I4') nx - 1
write(nout, '(T10,"Number of unknowns: ",I4)') n
write(nout,*)
write(nout, '(/T10,"Time in Years Prior to Expiration: ",'//&
 '2X,3F7.4)') time
write(nout, '(T10,"Option valued at Underlying Prices: ",'//&
 '3F7.2)') eval_points
write(nout,*)

!
! iquart = 0 : derivatives estimated with 3-point fitted parabola
! iquart = 1 : derivatives estimated with 5-point fitted quartic
!
! polynomial
!

iquart = 0
if (iquart == 0) then
 write(nout, '(T10,"3 point (parabolic) estimate of '//&
 'parameter derivatives"')
else
 write(nout, '(T10,"5 point (quartic) estimate of parameter'//&
 ' derivatives"')
end if
write(nout, '(T10,"epsfrac = ", F11.8)') epsfrac

irect = 0
reavg = 0.0d0
rex = 0.0d0

! alpha: Black-Scholes
do i2 = 1, 3
! Loop over volatility
do i3 = 1, 3
! Loop over strike price
call calc_Greeks(i2, i3, iquart)
end do
end do

write(nout,*)
do ig = 1, 12
 reavg(ig) = reavg(ig)/irect(ig)
 write (nout, '(" Greek: ", A6, "; avg rel err: ",'//&
 'F15.12, "; max rel err: ", F15.12)')&
 greek_name(ig), reavg(ig), rex(ig)
end do

contains

subroutine calc_Greeks(volatility, strike_price, iquart)
implicit none
integer, intent(in) :: volatility, strike_price, iquart
! Local variables
integer :: i1 = 1, j, iSderiv, gNi, l, k
integer :: nt = 3
real(kind(1d0)) :: x_maxp, x_maxm, x_maxpp, x_maxmm
real(kind(1d0)) :: eps, tau, sigsqtau, sqrt_sigsqtau, sigsq

```

```

real(kind(ld0)) :: d1, d2, n01pdf_d1, nu, relerr, relerrmax
real(kind(ld0)) :: sVo, BSVo, S

if ((volatility == 1) .and. (strike_price == 1)) then
 write(nout,*)
 write(nout,'(/T10,"Strike = ",F5.2,", Sigma =", F5.2,'//&
 '"', Alpha =",F5.2,":")') ks(strike_price),&
 sigma(volatility), alpha(i1)
 write(nout,*)
 write(nout,'(T10,"years to expiration: ", 3F7.4)')&
 (time(j), j=1,3)
 write(nout,*)
end if

fcn_data % rdata = (/ks(strike_price), x_max,&
 sigma(volatility), alpha(i1), r, dividend/)

call feynman_kac(xgrid, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)

! Compute Value, Delta, Gamma, Theta, Charm and Color
do l = 0,2
 do i=1,NT
 spline_values(:,i,l+1) = hqsval(eval_points, xgrid,&
 y(:,i), IDERIV=1)
 spline_values(:,i,l+4) = hqsval(eval_points, xgrid,&
 yprime(:,i), IDERIV=1)
 end do
end do
! Signs for Charm and Color must be inverted because FEYNMAN_KAC
! computes -d/dt instead of d/dt
spline_values(:, :, 5:6) = -spline_values(:, :, 5:6)

! Speed2
do i=1,nt
 spline_values(:,i,12) = hqsval(eval_points, xgrid, Y(:,i),&
 IDERIV=3)
end do

! Compute Vega, Volga, Vanna, Rho, Speed
! l = 7 8 9 10 11
do l = 7,11
 xgridp = xgrid
 xgridm = xgrid
 eval_pointsp = eval_points
 eval_pointsm = eval_points
 x_maxp = x_max
 x_maxm = x_max
 fcn_data % rdata(3) = sigma(volatility)
 fcn_data % rdata(5) = r
 iSderiv = 0
 if (l == 9) iSderiv = 1 ! Vanna
 if (l == 11) iSderiv = 2 ! Speed
 if (l == 10) then
 fcn_data % rdata(5) = r * (1.0 + epsfrac) ! Rho
 else if (l < 10) then

```

```

 fcn_data % rdata(3) = sigma(volatility) * (1.0 + epsfrac)
end if
if (l == 11) then
 xgridp = xgrid + dx2
 xgridm = xgrid - dx2
 eval_pointsp = eval_points + dx2
 eval_pointsm = eval_points - dx2
 x_maxp = x_max + dx2
 x_maxm = x_max - dx2
end if
fcn_data % rdata(2) = x_maxp
call feynman_kac(xgridp, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)
do i=1,nt
 spline_valuessp(:,i) = hqsval(eval_pointsp, xgridp,&
 y(:,i), IDERIV=iSderiv)
end do

if (l == 10) then
 fcn_data % rdata(5) = r * (1.0 - epsfrac) ! Rho
else if (l < 10) then
 fcn_data % rdata(3) = sigma(volatility) *&
 (1.0 - epsfrac)
end if
fcn_data % rdata(2) = x_maxm
! calculate spline values for sigmaM = sigmaI2-1*(1. - epsfrac) or
! rM = r*(1. - epsfrac):
call feynman_kac(xgridm, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)
do i=1,NT
 spline_valuessm(:,i) = hqsval(eval_pointsm, xgridm,&
 y(:,i), IDERIV=iSderiv)
end do

if (iquart == 1) then
 xgridpp = xgrid
 xgridmm = xgrid
 eval_pointsp = eval_points
 eval_pointsm = eval_points
 x_maxpp = x_max
 x_maxmm = x_max

 if (l == 11) then ! Speed
 xgridpp = xgrid + 2.0 * dx2
 xgridmm = xgrid - 2.0 * dx2
 eval_pointsp = eval_points + 2.0 * dx2
 eval_pointsm = eval_points - 2.0 * dx2
 x_maxpp = x_max + 2.0 * dx2
 x_maxmm = x_max - 2.0 * dx2
 end if

 fcn_data % rdata(2) = x_maxpp
 if (l == 10) then
! calculate spline values for rPP = r*(1. + 2.*epsfrac):

```

```

 fcn_data % rdata(5) = r * (1.0 + 2.0 * epsfrac)
 else if (l < 10) then
! calculate spline values for sigmaPP = sigma(i2)*(1. + 2.*epsfrac):
 fcn_data % rdata(3) = sigma(volatility) *&
 (1.0 + 2.0 * epsfrac)
 end if
 call feynman_kac (xgridpp, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)

 do i=1,nt
 spline_valuespp(:,i) = hqsval(eval_pointspp, xgridpp,&
 Y(:,i), IDERIV=iSderiv)
 end do

 fcn_data % rdata(2) = x_maxmm
! calculate spline values for sigmaMM = sigma(i2-1)*(1. - 2.*epsfrac)
! or rMM = r*(1. - 2.*epsfrac)
 if (l == 10) then
 fcn_data % rdata(5) = r * (1.0 - 2.0 * epsfrac)
 else if (l < 10) then
 fcn_data % rdata(3) = sigma(volatility) *&
 (1.0 - 2.0 * epsfrac)
 end if
 call feynman_kac (xgridmm, time, nlbc, nrbc, fkcoef,&
 fkinitcond, fkbc, y, yprime,&
 FCN_DATA = fcn_data)

 do i=1,nt
 spline_valuesmm(:,i) = hqsval(eval_pointsmm, xgridmm,&
 y(:,i), IDERIV=iSderiv)
 end do
end if ! if (iquart == 1)

if (l /= 8) then
 eps = sigma(volatility) * epsfrac
 if (l == 10) eps = r * epsfrac ! Rho
 if (l == 11) eps = dx2 ! Speed

 spline_values(:, :, 1) = &
 (spline_valuesp - spline_valuesm) / (2.0 * eps)
 if (iquart /= 0) then
 spline_values1 = &
 (spline_valuespp - spline_valuesmm) / (4.0 * eps)
 spline_values(:, :, 1) = &
 (4.0 * spline_values(:, :, 1) - spline_values1) / 3.0
 end if
end if

if (l == 8) then ! Volga
 eps = sigma(volatility) * epsfrac
 spline_values(:, :, 1) = &
 (spline_valuesp + spline_valuesm - 2.0 * &
 spline_values(:, :, 1)) / (eps * eps)

 if (iquart /= 0) then
 spline_values1 = &

```

```

 (spline_valuespp + spline_valuesmm - 2.0 * &
 spline_values(:, :, 1)) / (4.0 * eps * eps)
spline_values(:, :, 1) = &
 (4.0 * spline_values(:, :, 1) - spline_values1) / 3.0
 end if
 end if
end do
! Evaluate BS solution at vector eval_points,
! at each time value prior to expiration.

 do i = 1, nt
!
! Black-Scholes (BS) European call option
! value = ValBSEC(S,t) = exp(-q*tau)*S*N01CDF(d1) -
! exp(-r*tau)*K*N01CDF(d2),
! where:
! tau = time to maturity = T - t;
! q = annual dividend yield;
! r = risk free rate;
! K = strike price;
! S = stock price;
! N01CDF(x) = N(0,1)_CDF(x);
! d1 = (log(S/K) +
! (r - q + 0.5*sigma**2) * tau) /
! (sigma * sqrt(tau));
! d2 = d1 - sigma * sqrt(tau)
!
! BS option values for tau = time(i):
 tau = time(i)
 sigsqtau = (sigma(volatility)**2) * tau
 sqrt_sigsqtau = sqrt(sigsqtau)
 sigsq = sigma(volatility) * sigma(volatility)
 do j = 1, nv
! Values of the underlying price where evaluations are made:
 S = eval_points(j)
 d1 = (log(S / ks(strike_price)) + (r - dividend) &
 * tau + 0.5 * sigsqtau) / sqrt_sigsqtau
 n01pdf_d1 = exp((-0.5) * d1 * d1) / sqrt2pi
 nu = exp((-dividend) * tau) * S * n01pdf_d1 * sqrt(tau)

 d2 = d1 - sqrt_sigsqtau
 BS_values(j,i,1) = exp((-dividend) * tau) * S * &
 anordf(d1) - exp((-r) * tau) * &
 ks(strike_price) * anordf(d2)
! greek = Rho
 BS_values(j,i,10) = exp((-r) * tau) * ks(strike_price) * &
 tau * anordf(d2)
! greek = Vega
 BS_values(j,i,7) = nu
! greek = Volga
 BS_values(j,i,8) = nu * d1 * d2 / sigma(volatility)
! greek = delta
 BS_values(j,i,2) = exp((-dividend) * tau) * anordf(d1)
! greek = Vanna
 BS_values(j,i,9) = (nu / S) * (1.0 - d1 / sqrt_sigsqtau)
! greek = dgamma

```

```

 BS_values(j,i,3) = exp((-dividend) * tau) * &
 n01pdf_d1 / (S * sqrt_sigsqtau)
! greek = speed
 BS_values(j,i,11) = (-exp((-dividend) * tau)) * &
 n01pdf_d1 * (1.0 + d1 / sqrt_sigsqtau) &
 / (S * S * sqrt_sigsqtau)
! greek = speed
 BS_values(j,i,12) = (-exp((-dividend) * tau)) * &
 n01pdf_d1 * (1.0 + d1 / sqrt_sigsqtau) / &
 (S * S * sqrt_sigsqtau)
 d2 = d1 - sqrt_sigsqtau
! greek = theta
 BS_values(j,i,4) = exp((-dividend) * tau) * S * &
 (dividend * anordf(d1) - 0.5 * sigsq * &
 n01pdf_d1 / sqrt_sigsqtau) - r * &
 exp((-r) * tau) * ks(strike_price) * &
 anordf(d2)
! greek = charm
 BS_values(j,i,5) = exp((-dividend) * tau) * ((-dividend) &
 * anordf(d1) + n01pdf_d1 * &
 ((r - dividend) * tau - 0.5 * d2 * &
 sqrt_sigsqtau) / (tau * sqrt_sigsqtau))
! greek = color
 BS_values(j,i,6) = &
 (-exp((-dividend) * tau)) * n01pdf_d1 * &
 (2.0 * dividend * tau + 1.0 + d1 * &
 (2.0 * (r - dividend) * tau - d2 * &
 sqrt_sigsqtau) / sqrt_sigsqtau) / &
 (2.0 * S * tau * sqrt_sigsqtau)

 end do
 end do

do l=1,12
 relerrmax = 0.0
 do i = 1,nv
 do j = 1,nt
 sVo = spline_values(i,j,1)
 BSVo = BS_values(i,j,1)
 relerr = abs((sVo - BSVo) / BSVo)
 if (relerr > relerrmax) relerrmax = relerr
 reavg(l) = reavg(l) + relerr
 irect(l) = irect(l) + 1
 end do
 end do
 if (relerrmax > rex(l)) rex(l) = relerrmax

 if ((volatility == 1) .and. (strike_price == 1)) then
 do i=1,nv
 sVo_array(1:nt) = spline_values(i,1:nt,1)
 write(nout, ("underlying price: ", F4.1, "; FK ", '//&
 'A6," : ', 3(F13.10,TR1))') eval_points(i), &
 greek_name(1), &
 (sVo_array(k), k=1,nt)
 write(nout, '(T25, "BS ", A6," : ', 3(F13.10,TR1))') &
 greek_name(1), (BS_values(i,k,1), k=1,nt)
 end do
 end if
end if

```

```

end do
end subroutine calc_Greeks
end

! These functions and routines define the coefficients, payoff and boundary
conditions.
function fkcoef (x, tx, iflag, fcn_data)
 use mp_types
 implicit none
 real(kind(1d0)), intent(in) :: x, tx
 integer, intent(inout) :: iflag
 type(d_fcn_data), optional :: fcn_data
 real(kind(1d0)) :: fkcoef

 real(kind(1d0)) :: sigma, interest_rate, alpha, dividend, &
 half = 0.5d0
 sigma = fcn_data % rdata(3)
 alpha = fcn_data % rdata(4)
 interest_rate = fcn_data % rdata(5)
 dividend = fcn_data % rdata(6)
 select case (iflag)
 case (1)
! The coefficient derivative d(sigma)/dx
 fkcoef = half*alpha*sigma*x**(alpha*half-1.0d0)
! The coefficient sigma(x)
 case (2)
 fkcoef = sigma*x**(alpha*half)
 case (3)
! The coefficient mu(x)
 fkcoef = (interest_rate - dividend) * x
 case (4)
! The coefficient kappa(x)
 fkcoef = interest_rate
 end select
! Note that there is no time dependence
 iflag = 0
 return
 end function fkcoef

 function fkinitcond(x, fcn_data)
 use mp_types
 implicit none
 real(kind(1d0)), intent(in) :: x
 type (d_fcn_data), optional :: fcn_data
 real(kind(1d0)) :: fkinitcond
 real(kind(1d0)) :: zero = 0.0d0
 real(kind(1d0)) :: strike_price

 strike_price = fcn_data % rdata(1)
! The payoff function
 fkinitcond = max(x - strike_price, zero)
 return
 end function fkinitcond

 subroutine fkbc (tx, iflag, bccoefs, fcn_data)
 use mp_types

```

```

implicit none
real(kind(1d0)), intent(in) :: tx
integer, intent(inout) :: iflag
real(kind(1d0)), dimension(:, :), intent(out) :: bccoefs
type (d_fcn_data), optional :: fcn_data
real(kind(1d0)) :: x_max, df, interest_rate, strike_price

strike_price = fcn_data % rdata(1)
x_max = fcn_data % rdata(2)
interest_rate = fcn_data % rdata(5)
select case (iflag)
 case (1)
 bccoefs(1,1:4) = (/1.0d0, 0.0d0, 0.0d0, 0.0d0/)
 bccoefs(2,1:4) = (/0.0d0, 1.0d0, 0.0d0, 0.0d0/)
 bccoefs(3,1:4) = (/0.0d0, 0.0d0, 1.0d0, 0.0d0/)
! Note no time dependence at left end
 iflag = 0
 case (2)
 df = exp(interest_rate * tx)
 bccoefs(1,1:4) = (/1.0d0, 0.0d0, 0.0d0, x_max -&
 df*strike_price/)
 bccoefs(2,1:4) = (/0.0d0, 1.0d0, 0.0d0, 1.0d0/)
 bccoefs(3,1:4) = (/0.0d0, 0.0d0, 1.0d0, 0.0d0/)
end select
end subroutine fkbc

```

## Output

```

Constant Elasticity of Variance Model for Vanilla Call Option
Interest Rate: 0.050 Continuous Dividend: 0.000
Minimum and Maximum Prices of Underlying: 0.00 60.00
Number of equally spaced spline knots: 120
Number of unknowns: 363

```

```

Time in Years Prior to Expiration: 0.0833 0.3333 0.5833
Option valued at Underlying Prices: 19.00 20.00 21.00

```

```

3 point (parabolic) estimate of parameter derivatives
epsfrac = 0.00100000

```

```

Strike =15.00 Sigma = 0.20 Alpha = 2.00:

```

```

years to expiration: 0.0833 0.3333 0.5833

```

```

underlying price: 19.0; FK Value: 4.0623732450 4.2575924184 4.4733805278
 BS Value: 4.0623732509 4.2575929678 4.4733814062
underlying price: 20.0; FK Value: 5.0623700127 5.2505145764 5.4492418798
 BS Value: 5.0623700120 5.2505143129 5.4492428547
underlying price: 21.0; FK Value: 6.0623699727 6.2485587059 6.4385718831
 BS Value: 6.0623699726 6.2485585270 6.4385720688
underlying price: 19.0; FK Delta: 0.9999864098 0.9877532309 0.9652249945
 BS Delta: 0.9999863811 0.9877520034 0.9652261127
underlying price: 20.0; FK Delta: 0.9999998142 0.9964646548 0.9842482622
 BS Delta: 0.9999998151 0.9964644003 0.9842476147
underlying price: 21.0; FK Delta: 0.9999999983 0.9990831687 0.9932459040

```

|                         |    |         |               |               |               |
|-------------------------|----|---------|---------------|---------------|---------------|
|                         | BS | Delta:  | 0.9999999985  | 0.9990834124  | 0.9932451927  |
| underlying price: 19.0; | FK | Gamma:  | 0.0000543456  | 0.0144908955  | 0.0264849216  |
|                         | BS | Gamma:  | 0.0000547782  | 0.0144911447  | 0.0264824761  |
| underlying price: 20.0; | FK | Gamma:  | 0.0000008315  | 0.0045912854  | 0.0129288434  |
|                         | BS | Gamma:  | 0.0000008437  | 0.0045925328  | 0.0129280372  |
| underlying price: 21.0; | FK | Gamma:  | 0.0000000080  | 0.0012817012  | 0.0058860348  |
|                         | BS | Gamma:  | 0.0000000077  | 0.0012818272  | 0.0058865489  |
| underlying price: 19.0; | FK | Theta:  | -0.7472631891 | -0.8301000450 | -0.8845209253 |
|                         | BS | Theta:  | -0.7472638978 | -0.8301108199 | -0.8844992143 |
| underlying price: 20.0; | FK | Theta:  | -0.7468881086 | -0.7706770630 | -0.8152217385 |
|                         | BS | Theta:  | -0.7468880640 | -0.7706789470 | -0.8152097697 |
| underlying price: 21.0; | FK | Theta:  | -0.7468815742 | -0.7479185416 | -0.7728950748 |
|                         | BS | Theta:  | -0.7468815673 | -0.7479153725 | -0.7728982104 |
| underlying price: 19.0; | FK | Charm:  | -0.0014382828 | -0.0879903285 | -0.0843323992 |
|                         | BS | Charm:  | -0.0014397520 | -0.0879913927 | -0.0843403333 |
| underlying price: 20.0; | FK | Charm:  | -0.0000284881 | -0.0364107814 | -0.0547260337 |
|                         | BS | Charm:  | -0.0000285354 | -0.0364209077 | -0.0547074804 |
| underlying price: 21.0; | FK | Charm:  | -0.0000003396 | -0.0126436426 | -0.0313343015 |
|                         | BS | Charm:  | -0.0000003190 | -0.0126437838 | -0.0313252716 |
| underlying price: 19.0; | FK | Color:  | 0.0051622176  | 0.0685064195  | 0.0299871130  |
|                         | BS | Color:  | 0.0051777484  | 0.0684737183  | 0.0300398444  |
| underlying price: 20.0; | FK | Color:  | 0.0001188761  | 0.0355826975  | 0.0274292189  |
|                         | BS | Color:  | 0.0001205713  | 0.0355891884  | 0.0274307898  |
| underlying price: 21.0; | FK | Color:  | 0.0000015432  | 0.0143174420  | 0.0190897159  |
|                         | BS | Color:  | 0.0000015141  | 0.0143247729  | 0.0190752019  |
| underlying price: 19.0; | FK | Vega:   | 0.0003289870  | 0.3487168323  | 1.1153520921  |
|                         | BS | Vega:   | 0.0003295819  | 0.3487535501  | 1.1153536190  |
| underlying price: 20.0; | FK | Vega:   | 0.0000056652  | 0.1224632724  | 0.6032458218  |
|                         | BS | Vega:   | 0.0000056246  | 0.1224675413  | 0.6033084039  |
| underlying price: 21.0; | FK | Vega:   | 0.0000000623  | 0.0376974472  | 0.3028275297  |
|                         | BS | Vega:   | 0.0000000563  | 0.0376857196  | 0.3028629419  |
| underlying price: 19.0; | FK | Volga:  | 0.0286254576  | 8.3705173459  | 16.7944554708 |
|                         | BS | Volga:  | 0.0286064650  | 8.3691191978  | 16.8219823169 |
| underlying price: 20.0; | FK | Volga:  | 0.0007137402  | 4.2505025277  | 12.9315441466 |
|                         | BS | Volga:  | 0.0007186004  | 4.2519372748  | 12.9612638820 |
| underlying price: 21.0; | FK | Volga:  | 0.0000100364  | 1.7613083436  | 8.6626161799  |
|                         | BS | Volga:  | 0.0000097963  | 1.7617504949  | 8.6676581034  |
| underlying price: 19.0; | FK | Vanna:  | -0.0012418872 | -0.3391850563 | -0.6388552010 |
|                         | BS | Vanna:  | -0.0012431594 | -0.3391932673 | -0.6387423326 |
| underlying price: 20.0; | FK | Vanna:  | -0.0000244490 | -0.1366771953 | -0.3945466661 |
|                         | BS | Vanna:  | -0.0000244825 | -0.1367114682 | -0.3945405194 |
| underlying price: 21.0; | FK | Vanna:  | -0.0000002904 | -0.0466333335 | -0.2187406645 |
|                         | BS | Vanna:  | -0.0000002726 | -0.0466323413 | -0.2187858632 |
| underlying price: 19.0; | FK | Rho:    | 1.2447807022  | 4.8365676561  | 8.0884594648  |
|                         | BS | Rho:    | 1.2447806658  | 4.8365650322  | 8.0884502627  |
| underlying price: 20.0; | FK | Rho:    | 1.2448021850  | 4.8929216544  | 8.3041708173  |
|                         | BS | Rho:    | 1.2448021908  | 4.8929245641  | 8.3041638392  |
| underlying price: 21.0; | FK | Rho:    | 1.2448024992  | 4.9107294560  | 8.4114197621  |
|                         | BS | Rho:    | 1.2448024996  | 4.9107310444  | 8.4114199038  |
| underlying price: 19.0; | FK | Speed:  | -0.0002124684 | -0.0156265453 | -0.0179534748 |
|                         | BS | Speed:  | -0.0002123854 | -0.0156192867 | -0.0179536520 |
| underlying price: 20.0; | FK | Speed:  | -0.0000037247 | -0.0055877024 | -0.0097502607 |
|                         | BS | Speed:  | -0.0000037568 | -0.0055859333 | -0.0097472434 |
| underlying price: 21.0; | FK | Speed:  | -0.0000000385 | -0.0017085830 | -0.0048143174 |
|                         | BS | Speed:  | -0.0000000378 | -0.0017082128 | -0.0048130214 |
| underlying price: 19.0; | FK | Speed2: | -0.0002310655 | -0.0156276977 | -0.0179516855 |

```

underlying price: 20.0; BS Speed2: -0.0002123854 -0.0156192867 -0.0179536520
underlying price: 21.0; FK Speed2: -0.0000043215 -0.0055923924 -0.0097502997
underlying price: 21.0; BS Speed2: -0.0000037568 -0.0055859333 -0.0097472434
underlying price: 21.0; FK Speed2: -0.0000000475 -0.0017117661 -0.0048153107
underlying price: 21.0; BS Speed2: -0.0000000378 -0.0017082128 -0.0048130214

Greek: Value; avg rel err: 0.000146171196; max rel err: 0.009030737566
Greek: Delta; avg rel err: 0.000035817272; max rel err: 0.001158483076
Greek: Gamma; avg rel err: 0.001088392379; max rel err: 0.044845800289
Greek: Theta; avg rel err: 0.000054196359; max rel err: 0.001412847300
Greek: Charm; avg rel err: 0.001213347059; max rel err: 0.064576457415
Greek: Color; avg rel err: 0.003323954467; max rel err: 0.136355681544
Greek: Vega; avg rel err: 0.001514753397; max rel err: 0.106255126885
Greek: Volga; avg rel err: 0.058531380389; max rel err: 1.639564208085
Greek: Vanna; avg rel err: 0.001061525805; max rel err: 0.065629483069
Greek: Rho; avg rel err: 0.000146868262; max rel err: 0.009438788128
Greek: Speed; avg rel err: 0.002065441607; max rel err: 0.073086615101
Greek: Speed2; avg rel err: 0.008429883935; max rel err: 0.255746328973

```

---

## HQSVAL

This rank-1 array function evaluates a Hermite quintic spline or one of its derivatives for an array of input points. In particular, it computes solution values for the Feynman-Kac PDE handled by routine [FEYNMAN\\_KAC](#).

### Function Return Value

**HQSVAL** — Rank-1 array containing the values or derivatives of the Hermite quintic spline at the points given in array *XVAL*. (Output)  
*size* = *size(XVAL)*.

### Required Arguments

**XVAL** — Rank-1 array containing the points at which the Hermite quintic spline is to be evaluated. (Input)

Let  $NXVAL = \text{size}(XVAL)$ . The points in *XVAL* must lie within the range of array *BREAK*, i.e.  $BREAK(1) \leq XVAL(I) \leq BREAK(NXVAL)$ ,  $I=1, \dots, NXVAL$ .

**BREAK** — Rank-1 array containing the breakpoints for the Hermite quintic spline representation. (Input)

When applied to the output from routine [FEYNMAN\\_KAC](#), array *BREAK* is identical to array *XGRID*.

Let  $NBREAK = \text{size}(BREAK)$ . *NBREAK* must be at least 2 and the values in *BREAK* must be in strictly increasing order.

**COEFFS** — Rank-1 array of size  $3 * NBREAK$  containing the coefficients of the Hermite quintic spline representation. (Input)

When applied to the output arrays *Y* or *YPRIME* from routine [FEYNMAN\\_KAC](#), array *COEFFS* is identical to one of the columns of arrays *Y* or *YPRIME*, respectively.

## Optional Argument

**IDERIV** — Order of the derivative to be evaluated. (Input)

It is required that `IDERIV = 0, 1, 2` or `3`. Use `0` for the function itself, `1` for the first derivative, etc.

Default: `IDERIV = 0`.

## FORTRAN 90 Interface

Generic: `HQSVAL (XVAL, BREAK, COEFFS [, ...])`

Specific: The specific interface names are `S_HQSVAL` and `D_HQSVAL`.

## Description

The Hermite quintic spline interpolation is done over the composite interval  $[x_{\min}, x_{\max}]$ , where `BREAK(I) = xi` are given by  $(x_{\min} =) x_1 < x_2 < \dots < x_m (= x_{\max})$ .

The Hermite quintic spline function is constructed using three primary functions, defined by

$$b_0(z) = -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3(6z^2 + 3z + 1),$$

$$b_1(z) = -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z + 1),$$

$$b_2(z) = \frac{1}{2}(-z^5 + 3z^4 - 3z^3 + z^2) = \frac{1}{2}(1-z)^3 z^2.$$

For each

$$x \in [x_i, x_{i+1}], h_i = x_{i+1} - x_i, z_i = (x - x_i) / h_i, i = 1, \dots, m-1,$$

the spline is locally defined by

$$H(x) = y_{3i-2}b_0(z) + y_{3i+1}b_0(1-z) + h_i y_{3i-1}b_1(z) \\ - h_i y_{3i+2}b_1(1-z) + h_i^2 y_{3i}b_2(z) + h_i^2 y_{3i+3}b_2(1-z),$$

where

$$y_{3i-2} = f(x_i),$$

$$y_{3i-1} = (\partial f / \partial x)(x_i) = f_x(x_i),$$

$$y_{3i} = (\partial^2 f / \partial x^2)(x_i) = f_{xx}(x_i), i = 1, \dots, m.$$

are the values of a given twice continuously differentiable function  $f$  and its first two derivatives at the breakpoints.

The approximating function  $H(x)$  is twice continuously differentiable on  $[x_{\min}, x_{\max}]$ , whereas

the third derivative is in general only continuous within the interior of the intervals  $[x_i, x_{i+1}]$ .

From the local representation of  $H(x)$  it follows that

$$H(x_i) = f(x_i) = y_{3i-2},$$

$$H'(x_i) = f'(x_i) = y_{3i-1},$$

$$H''(x_i) = y_{3i}, i = 1, \dots, m.$$

The spline coefficients  $y_i, i = 1, \dots, 3m$ , are stored as successive triplets in array `COEFFS`. For a given  $w \in [x_{\min}, x_{\max}]$ , function `HQSVAL` uses the information in `COEFFS` together with the values of  $b_0, b_1, b_2$  and its derivatives at  $w$  to compute  $H^{(d)}(w), d = 0, \dots, 3$  using the local representation on the particular subinterval containing  $w$ .

When using arrays `XGRID` and `Y(:, I)` from routine `FEYNMAN_KAC` as input arrays `BREAK` and `COEFFS`, function `HQSVAL` allows for computation of approximate solutions  $f, f_x, f_{xx}, f_{xxx}$  to the Feynman-Kac PDE for `IDERIV=0,1,2,3`, respectively. The solution values are evaluated at the array of points  $(XVAL(:, I), TGRID(I))$  for  $I=1, \dots, \text{size}(TGRID)$  and  $(XVAL(:, 0), 0)$  for  $I=0$ . Similarly, using arrays `XGRID` and `YPRIME(:, I)` allows for computation of approximate solutions  $f_t, f_{tx}, f_{txx}, f_{txxx}$  to the Feynman-Kac PDE.

### Example: Exact Interpolation with Hermite Quintic Splines

Consider function  $f(x) = x^5$ , a polynomial of degree 5, on the interval  $[-1,1]$  with breakpoints  $\pm 1$ . Then, the end derivative values are

$$y_1 = f(-1) = -1, \quad y_2 = f'(-1) = 5, \quad y_3 = f''(-1) = -20$$

and

$$y_4 = f(1) = 1, \quad y_5 = f'(1) = 5, \quad y_6 = f''(1) = 20.$$

Since the Hermite quintic interpolates all polynomials up to degree 5 exactly, the spline interpolation on  $[-1,1]$  must agree with the exact function values up to rounding errors.

```

use hqsval_int
use umach_int

implicit none

integer :: i, nout
real(kind(1e0)) :: break(2), xval(7), coeffs(6), interpolant(7)

! Define arrays
break = (/ -1.0, 1.0 /)
xval = (/ -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75 /)
coeffs = (/ -1.0, 5.0, -20.0, 1.0, 5.0, 20.0 /)

! Compute interpolant
```

```

interpolant = hqsval(xval, break, coeffs)

call umach(2, nout)
! Compare interpolant with exact function.
write(nout, '(A6,A10,A15,A10)') 'x', 'F(x)', 'Interpolant', 'Error'
write(nout, '(f8.3,f9.3,f11.3,f15.7)') (xval(i), F(xval(i)), &
 interpolant(i), abs(F(xval(i))-interpolant(i)), &
 i=1,7)

contains
function F(x)
 implicit none
 real(kind(1e0)) :: F, x

 F = x**5
 return
end function F
end

```

## Output

| x      | F(x)   | Interpolant | Error     |
|--------|--------|-------------|-----------|
| -0.750 | -0.237 | -0.237      | 0.0000000 |
| -0.500 | -0.031 | -0.031      | 0.0000000 |
| -0.250 | -0.001 | -0.001      | 0.0000000 |
| 0.000  | 0.000  | 0.000       | 0.0000000 |
| 0.250  | 0.001  | 0.001       | 0.0000000 |
| 0.500  | 0.031  | 0.031       | 0.0000000 |
| 0.750  | 0.237  | 0.237       | 0.0000000 |

---

## FPS2H



Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the `HODIE` finite-difference scheme on a uniform mesh.

### Required Arguments

**PRHS** — User-supplied `FUNCTION` to evaluate the right side of the partial differential equation. The form is `PRHS(X, Y)`, where

`X` — `X`-coordinate value. (Input)

`Y` — `Y`-coordinate value. (Input)

`PRHS` — Value of the right side at  $(X, Y)$ . (Output)

PRHS must be declared EXTERNAL in the calling program.

**BRHS** — User-supplied FUNCTION to evaluate the right side of the boundary conditions. The form is BRHS(ISIDE, X, Y), where

ISIDE — Side number. (Input)

See IBCTY below for the definition of the side numbers.

X — X-coordinate value. (Input)

Y — Y-coordinate value. (Input)

BRHS — Value of the right side of the boundary condition at (X, Y). (Output)

BRHS must be declared EXTERNAL in the calling program.

**COEFU** — Value of the coefficient of U in the differential equation. (Input)

**NX** — Number of grid lines in the X-direction. (Input)  
NX must be at least 4. See Comment 2 for further restrictions on NX.

**NY** — Number of grid lines in the Y-direction. (Input)  
NY must be at least 4. See Comment 2 for further restrictions on NY.

**AX** — The value of X along the left side of the domain. (Input)

**BX** — The value of X along the right side of the domain. (Input)

**AY** — The value of Y along the bottom of the domain. (Input)

**BY** — The value of Y along the top of the domain. (Input)

**IBCTY** — Array of size 4 indicating the type of boundary condition on each side of the domain or that the solution is periodic. (Input)  
The sides are numbered 1 to 4 as follows:

| Side       | Location |
|------------|----------|
| 1 - Right  | (X = BX) |
| 2 - Bottom | (Y = AY) |
| 3 - Left   | (X = AX) |
| 4 - Top    | (Y = BY) |

There are three boundary condition types.

| <b>IBCTY</b> | <b>Boundary Condition</b>                                                                             |
|--------------|-------------------------------------------------------------------------------------------------------|
| 1            | Value of $U$ is given. (Dirichlet)                                                                    |
| 2            | Value of $dU/dX$ is given (sides 1 and/or 3). (Neumann) Value of $dU/dY$ is given (sides 2 and/or 4). |
| 3            | Periodic.                                                                                             |

$U$  — Array of size  $NX$  by  $NY$  containing the solution at the grid points. (Output)

### Optional Arguments

***IORDER*** — Order of accuracy of the finite-difference approximation. (Input)  
It can be either 2 or 4. Usually,  $IORDER = 4$  is used.  
Default:  $IORDER = 4$ .

***LDU*** — Leading dimension of  $U$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDU = \text{size}(U,1)$ .

### FORTRAN 90 Interface

Generic: `CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY, IBCTY, U [, ...])`

Specific: The specific interface names are `S_FPS2H` and `D_FPS2H`.

### FORTRAN 77 Interface

Single: `CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY, IBCTY, IORDER, U, LDU)`

Double: The double precision name is `DFPS2H`.

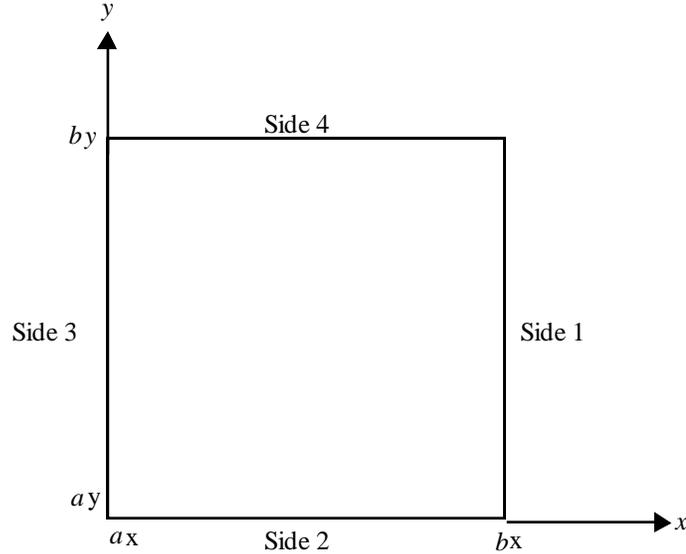
### Description

Let  $c = \text{COEFU}$ ,  $a_x = \text{AX}$ ,  $b_x = \text{BX}$ ,  $a_y = \text{AY}$ ,  $b_y = \text{BY}$ ,  $n_x = \text{NX}$  and  $n_y = \text{NY}$ .

`FPS2H` is based on the code `HFFT2D` by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = p$$

on the rectangular domain  $(a_x, b_x) \times (a_y, b_y)$  with a user-specified combination of Dirichlet (solution prescribed), Neumann (first-derivative prescribed), or periodic boundary conditions. The sides are numbered clockwise, starting with the right side.



When  $c = 0$  and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum  $\infty$ -norm is returned.

The solution is computed using either a second- or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies upon the fact that  $n_x - 1$  is highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If  $n_x - 1$  is highly composite then the execution time of `FPS2H` is proportional to  $n_x n_y \log_2 n_x$ . If evaluations of  $p(x, y)$  are inexpensive, then the difference in running time between `IORDER = 2` and `IORDER = 4` is small.

### Comments

1. Workspace may be explicitly provided, if desired, by use of `F2S2H/DF2S2H`. The reference is:

```
CALL F2S2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY,
 IBCTY, IORDER, U, LDU, UWORK, WORK)
```

The additional arguments are as follows:

**UWORK** — Work array of size  $NX + 2$  by  $NY + 2$ . If the actual dimensions of `U` are large enough, then `U` and `UWORK` can be the same array.

**WORK** — Work array of length  $(NX + 1)(NY + 1)(IORDER - 2)/2 + 6(NX + NY) + NX/2 + 16$ .

2. The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas  $HX = (BX - AX)/(NX - 1)$  and  $HY = (BY - AY)/(NY - 1)$ . The grid spacings in the X and Y directions must be the same, i.e., NX and NY must be such that HX equals HY. Also, as noted above, NX and NY must both be at least 4. To increase the speed of the fast Fourier transform, NX - 1 should be the product of small primes. Good choices are 17, 33, and 65.
3. If -COEFU is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

### Example

In this example, the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 3u = -2 \sin(x + 2y) + 16e^{2x+3y}$$

with the boundary conditions  $\partial u / \partial y = 2 \cos(x + 2y) + 3 \exp(2x + 3y)$  on the bottom side and  $u = \sin(x + 2y) + \exp(2x + 3y)$  on the other three sides. The domain is the rectangle  $[0, 1/4] \times [0, 1/2]$ . The output of FPS2H is a  $17 \times 33$  table of U values. The quadratic interpolation routine QD2VL is used to print a table of values.

```

USE FPS2H_INT
USE QD2VL_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NCVAL, NX, NXTABL, NY, NYTABL
PARAMETER (NCVAL=11, NX=17, NXTABL=5, NY=33, NYTABL=5)
!
INTEGER I, IBCTY(4), IORDER, J, NOUT
REAL AX, AY, BRHS, BX, BY, COEFU, ERROR, FLOAT, PRHS, &
TRUE, U(NX,NY), UTABL, X, XDATA(NX), Y, YDATA(NY)
INTRINSIC FLOAT
EXTERNAL BRHS, PRHS
!
! Set rectangle size
AX = 0.0
BX = 0.25
AY = 0.0
BY = 0.50
!
! Set boundary condition types
IBCTY(1) = 1
IBCTY(2) = 2
IBCTY(3) = 1
IBCTY(4) = 1
!
! Coefficient of U
COEFU = 3.0
!
! Order of the method
IORDER = 4
!
! Solve the PDE
CALL FPS2H (PRHS, BRHS, COEFU, NX, NY, AX, BX, AY, BY, IBCTY, U)

```

```

! Setup for quadratic interpolation
 DO 10 I=1, NX
 XDATA(I) = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NX-1)
10 CONTINUE
 DO 20 J=1, NY
 YDATA(J) = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NY-1)
20 CONTINUE
! Print the solution
 CALL UMACH (2, NOUT)
 WRITE (NOUT, '(8X,A,11X,A,11X,A,8X,A)') 'X', 'Y', 'U', 'Error'
 DO 40 J=1, NYTABL
 DO 30 I=1, NXTABL
 X = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NXTABL-1)
 Y = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NYTABL-1)
 UTABL = QD2VL(X,Y,XDATA,YDATA,U)
 TRUE = SIN(X+2.*Y) + EXP(2.*X+3.*Y)
 ERROR = TRUE - UTABL
 WRITE (NOUT, '(4F12.4)') X, Y, UTABL, ERROR
30 CONTINUE
40 CONTINUE
 END
!
 REAL FUNCTION PRHS (X, Y)
 REAL X, Y
!
 REAL EXP, SIN
 INTRINSIC EXP, SIN
!
 PRHS = -2.*SIN(X+2.*Y) + 16.*EXP(2.*X+3.*Y)
 RETURN
 END
!
 REAL FUNCTION BRHS (ISIDE, X, Y)
 INTEGER ISIDE
 REAL X, Y
!
 REAL COS, EXP, SIN
 INTRINSIC COS, EXP, SIN
!
 Define the boundary conditions
 IF (ISIDE .EQ. 2) THEN
 BRHS = 2.*COS(X+2.*Y) + 3.*EXP(2.*X+3.*Y)
 ELSE
 BRHS = SIN(X+2.*Y) + EXP(2.*X+3.*Y)
 END IF
 RETURN
 END

```

## Output

| X      | Y      | U      | Error  |
|--------|--------|--------|--------|
| 0.0000 | 0.0000 | 1.0000 | 0.0000 |
| 0.0625 | 0.0000 | 1.1956 | 0.0000 |
| 0.1250 | 0.0000 | 1.4087 | 0.0000 |
| 0.1875 | 0.0000 | 1.6414 | 0.0000 |
| 0.2500 | 0.0000 | 1.8961 | 0.0000 |
| 0.0000 | 0.1250 | 1.7024 | 0.0000 |
| 0.0625 | 0.1250 | 1.9562 | 0.0000 |

|        |        |        |        |
|--------|--------|--------|--------|
| 0.1250 | 0.1250 | 2.2345 | 0.0000 |
| 0.1875 | 0.1250 | 2.5407 | 0.0000 |
| 0.2500 | 0.1250 | 2.8783 | 0.0000 |
| 0.0000 | 0.2500 | 2.5964 | 0.0000 |
| 0.0625 | 0.2500 | 2.9322 | 0.0000 |
| 0.1250 | 0.2500 | 3.3034 | 0.0000 |
| 0.1875 | 0.2500 | 3.7148 | 0.0000 |
| 0.2500 | 0.2500 | 4.1720 | 0.0000 |
| 0.0000 | 0.3750 | 3.7619 | 0.0000 |
| 0.0625 | 0.3750 | 4.2163 | 0.0000 |
| 0.1250 | 0.3750 | 4.7226 | 0.0000 |
| 0.1875 | 0.3750 | 5.2878 | 0.0000 |
| 0.2500 | 0.3750 | 5.9199 | 0.0000 |
| 0.0000 | 0.5000 | 5.3232 | 0.0000 |
| 0.0625 | 0.5000 | 5.9520 | 0.0000 |
| 0.1250 | 0.5000 | 6.6569 | 0.0000 |
| 0.1875 | 0.5000 | 7.4483 | 0.0000 |
| 0.2500 | 0.5000 | 8.3380 | 0.0000 |

---

## FPS3H



Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the `HODIE` finite-difference scheme on a uniform mesh.

### Required Arguments

**PRHS** — User-supplied `FUNCTION` to evaluate the right side of the partial differential equation. The form is `PRHS (X, Y, Z)`, where

`X` — The  $x$ -coordinate value. (Input)

`Y` — The  $y$ -coordinate value. (Input)

`Z` — The  $z$ -coordinate value. (Input)

`PRHS` — Value of the right side at  $(X, Y, Z)$ . (Output)

`PRHS` must be declared `EXTERNAL` in the calling program.

**BRHS** — User-supplied `FUNCTION` to evaluate the right side of the boundary conditions. The form is `BRHS (ISIDE, X, Y, Z)`, where

`ISIDE` — Side number. (Input)

See *IBCTY* for the definition of the side numbers.

*X* – The *x*-coordinate value. (Input)

*Y* – The *y*-coordinate value. (Input)

*Z* – The *z*-coordinate value. (Input)

*BRHS* – Value of the right side of the boundary condition at (*x*, *y*, *z*). (Output)

*BRHS* must be declared *EXTERNAL* in the calling program.

*COEFU* — Value of the coefficient of *U* in the differential equation. (Input)

*NX* — Number of grid lines in the *x*-direction. (Input)  
*NX* must be at least 4. See Comment 2 for further restrictions on *NX*.

*NY* — Number of grid lines in the *y*-direction. (Input)  
*NY* must be at least 4. See Comment 2 for further restrictions on *NY*.

*NZ* — Number of grid lines in the *z*-direction. (Input)  
*NZ* must be at least 4. See Comment 2 for further restrictions on *NZ*.

*AX* — Value of *x* along the left side of the domain. (Input)

*BX* — Value of *x* along the right side of the domain. (Input)

*AY* — Value of *y* along the bottom of the domain. (Input)

*BY* — Value of *y* along the top of the domain. (Input)

*AZ* — Value of *z* along the front of the domain. (Input)

*BZ* — Value of *z* along the back of the domain. (Input)

*IBCTY* — Array of size 6 indicating the type of boundary condition on each face of the domain or that the solution is periodic. (Input)  
The sides are numbers 1 to 6 as follows:

| Side       | Location     |
|------------|--------------|
| 1 - Right  | ( $X = BX$ ) |
| 2 - Bottom | ( $Y = AY$ ) |
| 3 - Left   | ( $X = AX$ ) |
| 4 - Top    | ( $Y = BY$ ) |

5 - Front            ( $Z = BZ$ )

6 - Back            ( $Z = AZ$ )

There are three boundary condition types.

**IBCTY    Boundary Condition**

1            Value of  $U$  is given. (Dirichlet)

2            Value of  $dU/dX$  is given (sides 1 and/or 3). (Neumann) Value of  $dU/dY$  is given (sides 2 and/or 4). Value of  $dU/dZ$  is given (sides 5 and/or 6).

3            Periodic.

$U$  — Array of size  $NX$  by  $NY$  by  $NZ$  containing the solution at the grid points. (Output)

### Optional Arguments

**IORDER** — Order of accuracy of the finite-difference approximation. (Input)

It can be either 2 or 4. Usually,  $IORDER = 4$  is used.

Default:  $IORDER = 4$ .

**LDU** — Leading dimension of  $U$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDU = \text{size}(U,1)$ .

**MDU** — Middle dimension of  $U$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $MDU = \text{size}(U,2)$ .

### FORTRAN 90 Interface

Generic:    `CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, BZ, IBCTY, U [, ...])`

Specific:    The specific interface names are `S_FPS3H` and `D_FPS3H`.

### FORTRAN 77 Interface

Single:     `CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, BZ, IBCTY, IORDER, U, LDU, MDU)`

Double:     The double precision name is `DFPS3H`.

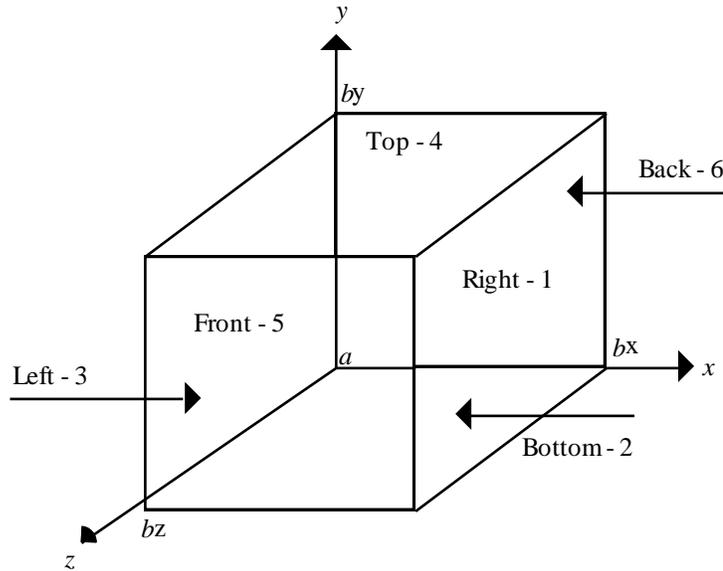
## Description

Let  $c = \text{COEFU}$ ,  $a_x = \text{AX}$ ,  $b_x = \text{BX}$ ,  $n_x = \text{NX}$ ,  $a_y = \text{AY}$ ,  $b_y = \text{BY}$ ,  $n_y = \text{NY}$ ,  $a_z = \text{AZ}$ ,  $b_z = \text{BZ}$ , and  $n_z = \text{NZ}$ .

`FPS3H` is based on the code `HFFT3D` by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + cu = p$$

on the domain  $(a_x, b_x) \times (a_y, b_y) \times (a_z, b_z)$  (a box) with a user-specified combination of Dirichlet (solution prescribed), Neumann (first derivative prescribed), or periodic boundary conditions. The six sides are numbered as shown in the following diagram.



When  $c = 0$  and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum  $\infty$ -norm is returned.

The solution is computed using either a second- or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies upon the fact that  $n_x - 1$  and  $n_z - 1$  are highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If  $n_x - 1$  and  $n_z - 1$  are highly composite, then the execution time of `FPS3H` is proportional to

$$n_x n_y n_z \left( \log_2^2 n_x + \log_2^2 n_z \right)$$

If evaluations of  $p(x, y, z)$  are inexpensive, then the difference in running time between `IORDER = 2` and `IORDER = 4` is small.

## Comments

1. Workspace may be explicitly provided, if desired, by use of F2S3H/DF2S3H. The reference is:

```
CALL F2S3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY,
 AZ, BZ, IBCTY, IORDER, U, LDU, MDU, UWORK, WORK)
```

The additional arguments are as follows:

**UWORK** — Work array of size  $NX + 2$  by  $NY + 2$  by  $NZ + 2$ . If the actual dimensions of  $U$  are large enough, then  $U$  and  $UWORK$  can be the same array.

**WORK** — Work array of length  $(NX + 1)(NY + 1)(NZ + 1)(IORDER - 2)/2 + 2(NX * NY + NX * NZ + NY * NZ) + 2(NX + NY + 1) + \text{MAX}(2 * NX * NY, 2 * NX + NY + 4 * NZ + (NX + NZ)/2 + 29)$

2. The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas

$$HX = (BX - AX)/(NX - 1),$$

$$HY = (BY - AY)/(NY - 1), \text{ and}$$

$$HZ = (BZ - AZ)/(NZ - 1).$$

The grid spacings in the  $x$ ,  $y$  and  $z$  directions must be the same, i.e.,  $NX$ ,  $NY$  and  $NZ$  must be such that  $HX = HY = HZ$ . Also, as noted above,  $NX$ ,  $NY$  and  $NZ$  must all be at least 4. To increase the speed of the Fast Fourier transform,  $NX - 1$  and  $NZ - 1$  should be the product of small primes. Good choices for  $NX$  and  $NZ$  are 17, 33 and 65.

3. If  $-COEFU$  is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

## Example

This example solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + 10u = -4 \cos(3x + y - 2z) + 12e^{x-z} + 10$$

with the boundary conditions  $\partial u / \partial z = -2 \sin(3x + y - 2z) - \exp(x - z)$  on the front side and  $u = \cos(3x + y - 2z) + \exp(x - z) + 1$  on the other five sides. The domain is the box  $[0, 1/4] \times [0, 1/2] \times [0, 1/2]$ . The output of F2S3H is a  $9 \times 17 \times 17$  table of  $U$  values. The quadratic interpolation routine QD3VL is used to print a table of values.

```
USE F2S3H_INT
USE UMACH_INT
USE QD3VL_INT
```

```
IMPLICIT NONE
```

```

! SPECIFICATIONS FOR PARAMETERS
INTEGER LDU, MDU, NX, NXTABL, NY, NYTABL, NZ, NZTABL
PARAMETER (NX=5, NXTABL=4, NY=9, NYTABL=3, NZ=9, &
 NZTABL=3, LDU=NX, MDU=NY)
!
INTEGER I, IBCTY(6), IORDER, J, K, NOUT
REAL AX, AY, AZ, BRHS, BX, BY, BZ, COEFU, FLOAT, PRHS, &
 U(LDU,MDU,NZ), UTABL, X, ERROR, TRUE, &
 XDATA(NX), Y, YDATA(NY), Z, ZDATA(NZ)
INTRINSIC COS, EXP, FLOAT
EXTERNAL BRHS, PRHS
!
! Define domain
AX = 0.0
BX = 0.125
AY = 0.0
BY = 0.25
AZ = 0.0
BZ = 0.25
!
! Set boundary condition types
IBCTY(1) = 1
IBCTY(2) = 1
IBCTY(3) = 1
IBCTY(4) = 1
IBCTY(5) = 2
IBCTY(6) = 1
!
! Coefficient of U
COEFU = 10.0
!
! Order of the method
IORDER = 4
!
! Solve the PDE
CALL FPS3H (PRHS, BRHS, COEFU, NX, NY, NZ, AX, BX, AY, BY, AZ, &
 BZ, IBCTY, U)
!
! Set up for quadratic interpolation
DO 10 I=1, NX
 XDATA(I) = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NX-1)
10 CONTINUE
DO 20 J=1, NY
 YDATA(J) = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NY-1)
20 CONTINUE
DO 30 K=1, NZ
 ZDATA(K) = AZ + (BZ-AZ)*FLOAT(K-1)/FLOAT(NZ-1)
30 CONTINUE
!
! Print the solution
CALL UMACH (2, NOUT)
WRITE (NOUT, '(8X,5(A,11X))') 'X', 'Y', 'Z', 'U', 'Error'
DO 60 K=1, NZTABL
 DO 50 J=1, NYTABL
 DO 40 I=1, NXTABL
 X = AX + (BX-AX)*FLOAT(I-1)/FLOAT(NXTABL-1)
 Y = AY + (BY-AY)*FLOAT(J-1)/FLOAT(NYTABL-1)
 Z = AZ + (BZ-AZ)*FLOAT(K-1)/FLOAT(NZTABL-1)
 UTABL = QD3VL(X,Y,Z,XDATA,YDATA,ZDATA,U, CHECK=.false.)
 TRUE = COS(3.0*X+Y-2.0*Z) + EXP(X-Z) + 1.0
 ERROR = UTABL - TRUE
 WRITE (NOUT, '(5F12.4)') X, Y, Z, UTABL, ERROR
 40 CONTINUE
 50 CONTINUE
 60 CONTINUE

```

```

60 CONTINUE
 END
!
 REAL FUNCTION PRHS (X, Y, Z)
 REAL X, Y, Z
!
 REAL COS, EXP
 INTRINSIC COS, EXP
!
 Right side of the PDE
 PRHS = -4.0*COS(3.0*X+Y-2.0*Z) + 12*EXP(X-Z) + 10.0
 RETURN
 END
!
 REAL FUNCTION BRHS (ISIDE, X, Y, Z)
 INTEGER ISIDE
 REAL X, Y, Z
!
 REAL COS, EXP, SIN
 INTRINSIC COS, EXP, SIN
!
 Boundary conditions
 IF (ISIDE .EQ. 5) THEN
 BRHS = -2.0*SIN(3.0*X+Y-2.0*Z) - EXP(X-Z)
 ELSE
 BRHS = COS(3.0*X+Y-2.0*Z) + EXP(X-Z) + 1.0
 END IF
 RETURN
 END

```

## Output

| X      | Y      | Z      | U      | Error   |
|--------|--------|--------|--------|---------|
| 0.0000 | 0.0000 | 0.0000 | 3.0000 | 0.0000  |
| 0.0417 | 0.0000 | 0.0000 | 3.0348 | 0.0000  |
| 0.0833 | 0.0000 | 0.0000 | 3.0558 | 0.0001  |
| 0.1250 | 0.0000 | 0.0000 | 3.0637 | 0.0001  |
| 0.0000 | 0.1250 | 0.0000 | 2.9922 | 0.0000  |
| 0.0417 | 0.1250 | 0.0000 | 3.0115 | 0.0000  |
| 0.0833 | 0.1250 | 0.0000 | 3.0175 | 0.0000  |
| 0.1250 | 0.1250 | 0.0000 | 3.0107 | 0.0000  |
| 0.0000 | 0.2500 | 0.0000 | 2.9690 | 0.0001  |
| 0.0417 | 0.2500 | 0.0000 | 2.9731 | 0.0000  |
| 0.0833 | 0.2500 | 0.0000 | 2.9645 | 0.0000  |
| 0.1250 | 0.2500 | 0.0000 | 2.9440 | -0.0001 |
| 0.0000 | 0.0000 | 0.1250 | 2.8514 | 0.0000  |
| 0.0417 | 0.0000 | 0.1250 | 2.9123 | 0.0000  |
| 0.0833 | 0.0000 | 0.1250 | 2.9592 | 0.0000  |
| 0.1250 | 0.0000 | 0.1250 | 2.9922 | 0.0000  |
| 0.0000 | 0.1250 | 0.1250 | 2.8747 | 0.0000  |
| 0.0417 | 0.1250 | 0.1250 | 2.9211 | 0.0010  |
| 0.0833 | 0.1250 | 0.1250 | 2.9524 | 0.0010  |
| 0.1250 | 0.1250 | 0.1250 | 2.9689 | 0.0000  |
| 0.0000 | 0.2500 | 0.1250 | 2.8825 | 0.0000  |
| 0.0417 | 0.2500 | 0.1250 | 2.9123 | 0.0000  |
| 0.0833 | 0.2500 | 0.1250 | 2.9281 | 0.0000  |
| 0.1250 | 0.2500 | 0.1250 | 2.9305 | 0.0000  |

|        |        |        |        |         |
|--------|--------|--------|--------|---------|
| 0.0000 | 0.0000 | 0.2500 | 2.6314 | -0.0249 |
| 0.0417 | 0.0000 | 0.2500 | 2.7420 | -0.0004 |
| 0.0833 | 0.0000 | 0.2500 | 2.8112 | -0.0042 |
| 0.1250 | 0.0000 | 0.2500 | 2.8609 | -0.0138 |
| 0.0000 | 0.1250 | 0.2500 | 2.7093 | 0.0000  |
| 0.0417 | 0.1250 | 0.2500 | 2.8153 | 0.0344  |
| 0.0833 | 0.1250 | 0.2500 | 2.8628 | 0.0237  |
| 0.1250 | 0.1250 | 0.2500 | 2.8825 | 0.0000  |
| 0.0000 | 0.2500 | 0.2500 | 2.7351 | -0.0127 |
| 0.0417 | 0.2500 | 0.2500 | 2.8030 | -0.0011 |
| 0.0833 | 0.2500 | 0.2500 | 2.8424 | -0.0040 |
| 0.1250 | 0.2500 | 0.2500 | 2.8735 | -0.0012 |

---

## SLEIG

Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u = \lambda r(x)u \text{ for } x \text{ in } (a,b)$$

with boundary conditions (at regular points)

$$a_1u - a_2(pu') = \lambda(a_1'u - a_2'(pu')) \text{ at } a$$

$$b_1u + b_2(pu') = 0 \text{ at } b$$

### Required Arguments

**CONS** — Array of size eight containing

$$a_1, a_1', a_2, a_2', b_1, b_2, a \text{ and } b$$

in locations CONS(1) through CONS(8), respectively. (Input)

**COEFFN** — User-supplied subroutine to evaluate the coefficient functions. The usage is

CALL COEFFN (X, PX, QX, RX)

X — Independent variable. (Input)

PX — The value of  $p(x)$  at X. (Output)

QX — The value of  $q(x)$  at X. (Output)

RX — The value of  $r(x)$  at X. (Output)

COEFFN must be declared EXTERNAL in the calling program.

**ENDFIN** — Logical array of size two. ENDFIN(1) = .true. if the endpoint  $a$  is finite.

ENDFIN(2) = .true. if endpoint  $b$  is finite. (Input)

**INDEX** — Vector of size NUMEIG containing the indices of the desired eigenvalues. (Input)

*EVAL* — Array of length *NUMEIG* containing the computed approximations to the eigenvalues whose indices are specified in *INDEX*. (Output)

### Optional Arguments

*NUMEIG* — The number of eigenvalues desired. (Input)  
Default: *NUMEIG* = size (*INDEX*,1).

*TEVLAB* — Absolute error tolerance for eigenvalues. (Input)  
Default: *TEVLAB* = 10.\* machine precision.

*TEVLRL* — Relative error tolerance for eigenvalues. (Input)  
Default: *TEVLRL* = SQRT(machine precision).

### FORTRAN 90 Interface

Generic:     CALL SLEIG (CONS, COEFFN, ENDFIN, INDEX, EVAL [, ...])

Specific:    The specific interface names are S\_SLEIG and D\_SLEIG.

### FORTRAN 77 Interface

Single:     CALL SLEIG (CONS, COEFFN, ENDFIN, NUMEIG, INDEX, TEVLAB, TEVLRL, EVAL)

Double:     The double precision name is DSLEIG.

### Description

This subroutine is designed for the calculation of eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u = \lambda r(x)u \text{ for } x \text{ in } (a,b) \quad (1)$$

with boundary conditions (at regular points)

$$\begin{aligned} a_1u - a_2(pu') &= \lambda(a'_1u - a'_2(pu')) \text{ at } a \\ b_1u + b_2(pu') &= 0 \text{ at } b \end{aligned}$$

We assume that

$$a'_1a_2 - a_1a'_2 > 0$$

when  $a'_1 \neq 0$  and  $a'_2 \neq 0$ . The problem is considered regular if and only if

- $a$  and  $b$  are finite,
- $p(x)$  and  $r(x)$  are positive in  $(a, b)$ ,

- $1/p(x)$ ,  $q(x)$  and  $r(x)$  are locally integrable near the endpoints.

Otherwise the problem is called singular. The theory assumes that  $p$ ,  $p'$ ,  $q$ , and  $r$  are at least continuous on  $(a, b)$ , though a finite number of jump discontinuities can be handled by suitably defining an input mesh.

For regular problems, there are an infinite number of eigenvalues

$$\lambda_0 < \lambda_1 < \dots < \lambda_k, k \rightarrow \infty$$

Each eigenvalue has an associated eigenfunction which is unique up to a constant. For singular problems, there is a wide range in the behavior of the eigenvalues.

As presented in Pruess and Fulton (1993) the approach is to replace (1) by a new problem

$$-(\hat{p}\hat{u}')' + \hat{q}\hat{u} = \hat{\lambda}\hat{r}\hat{u} \quad (2)$$

with analogous boundary conditions

$$\begin{aligned} a_1\hat{u}(a) - a_2(\hat{p}\hat{u}')'(a) &= \hat{\lambda} \left[ a_1\hat{u}(a) - a_2'(\hat{p}\hat{u}')'(a) \right] \\ b_1\hat{u}(b) + b_2(\hat{p}\hat{u}')'(b) &= 0 \end{aligned}$$

where

$$\hat{p}, \hat{q} \text{ and } \hat{r}$$

are step function approximations to  $p$ ,  $q$ , and  $r$ , respectively. Given the mesh

$a = x_1 < x_2 < \dots < x_{N+1} = b$ , the usual choice for the step functions uses midpoint interpolation, i. e.,

$$\hat{p}(x) = p_n \equiv p\left(\frac{x_n + x_{n+1}}{2}\right)$$

for  $x$  in  $(x_n, x_{n+1})$  and similarly for the other coefficient functions. This choice works well for regular problems. Some singular problems require a more sophisticated technique to capture the asymptotic behavior. For the midpoint interpolants, the differential equation (2) has the known closed form solution in

$(x_n, x_{n+1})$

$$\hat{u}(x) = \hat{u}(x_n)\phi_n'(x - x_n) + (\hat{p}\hat{u}')'(x_n)\phi_n(x - x_n)/p_n$$

with

$$\phi_n(t) = \begin{cases} \sin \omega_n t / \omega_n, \tau_n > 0 \\ \sinh \omega_n t / \omega_n, \tau_n < 0 \\ t, \tau = 0 \end{cases}$$

where

$$\tau_n = (\hat{\lambda}r_n - q_n)/p_n$$

and

$$\omega_n = \sqrt{|\tau_n|}$$

Starting with,

$$\hat{u}(a) \text{ and } (\hat{p}\hat{u}')(a)$$

consistent with the boundary condition,

$$\begin{aligned}\hat{u}(a) &= a_2 - a'_2 \hat{\lambda} \\ (\hat{p}\hat{u}')(a) &= a_1 - a'_1 \hat{\lambda}\end{aligned}$$

an algorithm is to compute for  $n = 1, 2, \dots, N$ ,

$$\begin{aligned}\hat{u}(x_{n+1}) &= \hat{u}(x_n)\phi'_n(h_n) + (\hat{p}\hat{u}')(x_n)\phi_n(h_n)/p_n \\ (\hat{p}\hat{u}')(x_{n+1}) &= -\tau_n p_n \hat{u}(x_n)\phi'_n(h_n) + (\hat{p}\hat{u}')(x_n)\phi_n(h_n)\end{aligned}$$

which is a shooting method. For a fixed mesh we can iterate on the approximate eigenvalue until the boundary condition at  $b$  is satisfied. This will yield an  $O(h^2)$  approximation

$$\hat{\lambda}_k$$

to some  $\lambda_k$ .

The problem (2) has a step spectral function given by

$$\hat{\rho}(t) = \sum \frac{1}{\int \hat{r}(x)\hat{u}_k^2(x)dx + \alpha}$$

where the sum is taken over  $k$  such that

$$\hat{\lambda}_k \leq t$$

and

$$\alpha = a'_1 a_2 - a_1 a'_2$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of S2EIG/DS2EIG. The reference is:

```
CALL S2EIG (CONS, COEFFN, ENDFIN, NUMEIG, INDEX, TEVLAB,
 TEVLRL, EVAL, JOB, IPRINT, TOLS, NUMX, XEF, NRHO, T,
 TYPE, EF, PDEF, RHO, IFLAG, WORK, IWORK)
```

The additional arguments are as follows:

**JOB** — Logical array of length five. (Input)

JOB(1) = *.true.* if a set of eigenvalues are to be computed but not their eigenfunctions.

JOB(2) = *.true.* if a set of eigenvalue and eigenfunction pairs are to be computed.

JOB(3) = *.true.* if the spectral function is to be computed over some subinterval of the essential spectrum.

JOB(4) = *.true.* if the normal automatic classification is overridden. If JOB(4) = *.true.* then TYPE(\*,\*) must be entered correctly. Most users will not want to override the classification process, but it might be appropriate for users experimenting with problems for which the coefficient functions do not have power-like behavior near the singular endpoints. The classification is considered sufficiently important for spectral density function calculations that JOB(4) is ignored with JOB(3) = *.true.*.

JOB(5) = *.true.* if mesh distribution is chosen by SLEIG. If JOB(5) = *.true.* and NUMX is zero, the number of mesh points are also chosen by SLEIG. If NUMX > 0 then NUMX mesh points will be used. If JOB(5) = *.false.*, the number NUMX and distribution XEF(\*) must be input by the user.

**IPRINT** — Control levels of internal printing. (Input)

No printing is performed if IPRINT = 0. If either JOB(1) or JOB(2) is true:

| <b>IPRINT</b> | <b>Printed Output</b>                                                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| 1             | Initial mesh (the first 51 or fewer points), eigenvalue estimate at each level.                                                                     |
| 4             | The above and at each level matching point for eigenfunction shooting, X(*), EF(*) and PDEF(*) values.                                              |
| 5             | The above and at each level the brackets for the eigenvalue search, intermediate shooting information for the eigenfunction and eigenfunction norm. |

If JOB(3) = *.true.*

| <b>IPRINT</b> | <b>Printed Output</b>                                                                                          |
|---------------|----------------------------------------------------------------------------------------------------------------|
| 1             | The actual (a, b) used at each iteration and the total number of eigenvalues computed.                         |
| 2             | The above and switchover points to the asymptotic formulas, and some intermediate ( <i>t</i> ) approximations. |

| <b>IPRINT</b> | <b>Printed Output</b>                                                                                                                                |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1             | The actual (a, b) used at each iteration and the total number of eigenvalues computed.                                                               |
| 3             | The above and initial meshes for each iteration, the index of the largest eigenvalue which may be computed, and various eigenvalue and $R_N$ values. |
| 4             | The above and $\hat{\rho}$ values at each level.                                                                                                     |
| 5             | The above and $R_N$ add eigenvalues below the switchover point                                                                                       |

If `JOB(4) = .false.`

| <b>IPRINT</b> | <b>Printed Output</b>                                                  |
|---------------|------------------------------------------------------------------------|
| 2             | Output a description of the spectrum.                                  |
| 3             | The above and the constants for the Friedrichs' boundary condition(s). |
| 5             | The above and intermediate details of the classification calculation.  |

**TOLS** — Array of length 4 containing tolerances. (Input)

`TOLS(1)` — absolute error tolerance for eigenfunctions

`TOLS(2)` — relative error tolerance for eigenfunctions

`TOLS(3)` — absolute error tolerance for eigenfunction derivatives

`TOLS(4)` — relative error tolerance for eigenfunction derivatives

The absolute tolerances must be positive.

The relative tolerances must be at least `100 *amach(4)`

**NUMX** — Integer whose value is the number of output points where each eigenfunction is to be evaluated (the number of entries in `XEF(*)`) when `JOB(2) = .true.`. If `JOB(5) = .false.` and `NUMX` is greater than zero, then `NUMX` is the number of points in the initial mesh used. If `JOB(5) = .false.`, the points in `XEF(*)` should be chosen with a reasonable distribution. Since the endpoints  $a$  and  $b$  must be part of any mesh, `NUMX` cannot be one in this case. If `JOB(5) = .false.` and `JOB(3) = .true.`, then `NUMX` must be positive. On output, `NUMX` is set to the number of points for eigenfunctions when input `NUMX = 0`, and `JOB(2)` or `JOB(5) = .true.`. (Input/Output)

**XEF** — Array of points on input where eigenfunction estimates are desired, if `JOB(2) = .true.`. Otherwise, if `JOB(5) = .false.` and `NUMX` is greater than zero, the user's initial mesh is entered. The entries must be ordered so that  $a = XEF(1) < XEF(2) < \dots < XEF(NUMX) = b$ . If either endpoint is infinite, the corresponding `XEF(1)` or `XEF(NUMX)` is ignored. However, it is required that `XEF(2)` be negative when `ENDFIN(1) = .false.`, and that `XEF(NUMX-1)` be positive when `ENDFIN(2) = .false.` On output, `XEF(*)` is changed only if `JOB(2)` and `JOB(5)` are true. If `JOB(2) = .false.`, this vector is not referenced. If `JOB(2) = .true.` and `NUMX` is greater than zero on input, `XEF(*)` should be dimensioned at least `NUMX + 16`. If `JOB(2)` is true and `NUMX` is zero on input, `XEF(*)` should be dimensioned at least 31.

**NRHO** — The number of output values desired for the array `RHO(*)`. `NRHO` is not used if `JOB(3) = .false.` (Input)

**T** — Real vector of size `NRHO` containing values where the spectral function `RHO(*)` is desired. The entries must be sorted in increasing order. The existence and location of a continuous spectrum can be determined by calling `SLEIG` with the first four entries of `JOB` set to false and `IPRINT` set to 1. `T(*)` is not used if `JOB(3) = .false.` (Input)

**TYPE** — 4 by 2 logical matrix. Column 1 contains information about endpoint  $a$  and column 2 refers to endpoint  $b$ .  
`TYPE(1,*) = .true.` if and only if the endpoint is regular  
`TYPE(2,*) = .true.` if and only if the endpoint is limit circle  
`TYPE(3,*) = .true.` if and only if the endpoint is nonoscillatory for all eigenvalues  
`TYPE(4,*) = .true.` if and only if the endpoint is oscillatory for all eigenvalues  
Note: all of these values must be correctly input if `JOB(4) = .true.`.  
Otherwise, `TYPE(*,*)` is output. (Input/Output)

**EF** — Array of eigenfunction values. `EF((k-1)*NUMX + i)` is the estimate of  $u(XEF(i))$  corresponding to the eigenvalue in `EV(k)`. If `JOB(2) = .false.` then this vector is not referenced. If `JOB(2) = .true.` and `NUMX` is greater than zero on entry, then `EF(*)` should be dimensioned at least `NUMX * NUMEIG`. If `JOB(2) = .true.` and `NUMX` is zero on input, then `EF(*)` should be dimensioned `31 * NUMEIG`. (Output)

**PDEF** — Array of eigenfunction derivative values. `PDEF((k-1)*NUMX + i)` is the estimate of  $(pu')$  (`XEF(i)`) corresponding to the eigenvalue in `EV(k)`. If `JOB(2) = .false.` this vector is not referenced. If `JOB(2) = .true.`, it must be dimensioned the same as `EF(*)`. (Output)

**RHO** — Array of size `NRHO` containing values for the spectral density function  $\rho(t)$ , `RHO(I) = \rho(T(I))`. This vector is not referenced if `JOB(3)` is false. (Output)

**IFLAG** — Array of size `max(1, numeig)` containing information about the output. `IFLAG(K)` refers to the  $K$ -th eigenvalue, when `JOB(1)` or `JOB(2) = .true.`

Otherwise, only IFLAG(1) is used. Negative values are associated with fatal errors, and the calculations are ceased. Positive values indicate a warning.

(Output)

IFLAG(K)

| IFLAG (K) | Description                                                                                                                                                                                                                                                                                                            |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -1        | too many levels needed for the eigenvalue calculation; problem seems too difficult at this tolerance. Are the coefficient functions nonsmooth?                                                                                                                                                                         |
| -2        | too many levels needed for the eigenfunction calculation; problem seems too difficult at this tolerance. Are the eigenfunctions ill-conditioned?                                                                                                                                                                       |
| -3        | too many levels needed for the spectral density calculation; problem seems too difficult at this tolerance.                                                                                                                                                                                                            |
| -4        | the user has requested the spectral density function for a problem which has no continuous spectrum.                                                                                                                                                                                                                   |
| -5        | the user has requested the spectral density function for a problem with both endpoints generating essential spectrum, i.e. both endpoints either OSC or O-NO.                                                                                                                                                          |
| -6        | the user has requested the spectral density function for a problem in spectral category 2 for which a proper normalization of the solution at the NONOSC endpoint is not known; for example, problems with an irregular singular point or infinite endpoint at one end and continuous spectrum generated at the other. |
| -7        | problems were encountered in obtaining a bracket.                                                                                                                                                                                                                                                                      |
| -8        | too small a step was used in the integration. The TOLS(*) values may be too small for this problem.                                                                                                                                                                                                                    |
| -9        | too small a step was used in the spectral density function calculation for which the continuous spectrum is generated by a finite endpoint.                                                                                                                                                                            |
| -10       | an argument to the circular trig functions is too large. Try running the problem again with a finer initial mesh or, for singular problems, use interval truncation.                                                                                                                                                   |
| -15       | $p(x)$ and $r(x)$ are not positive in the interval $(a, b)$ .                                                                                                                                                                                                                                                          |
| -20       | eigenvalues and/or eigenfunctions were requested for a problem with an OSC singular endpoint. Interval truncation must be used on such problems.                                                                                                                                                                       |
| 1         | Failure in the bracketing procedure probably due to a cluster of eigenvalues which the code cannot separate. Calculations have continued but any eigenfunction results are suspect. Try running the problem again with tighter input tolerances to separate the cluster.                                               |

- 2      there is uncertainty in the classification for this problem. Because of the limitations of floating point arithmetic, and the nature of the finite sampling, the routine cannot be certain about the classification information at the requested tolerance.
- 3      there may be some eigenvalues embedded in the essential spectrum. Use of `IPRINT` greater than zero will provide additional output giving the location of the approximating eigenvalues for the step function problem. These could be extrapolated to estimate the actual eigenvalue embedded in the essential spectrum.
- 4      a change of variables was made to avoid potentially slow convergence. However, the global error estimates may not be as reliable. Some experimentation using different tolerances is recommended.
- 6      there were problems with eigenfunction convergence in a spectral density calculation. The output  $\rho(t)$  may not be accurate.

**WORK** — Array of size `MAX(1000, NUMEIG + 22)` used for workspace.

**IWORK** — Integer array of size `NUMEIG + 3` used for workspace.

### Example 1

This example computes the first ten eigenvalues of the problem from Titchmarsh (1962) given by

$$p(x) = r(x) = 1$$

$$q(x) = x$$

$$[a, b] = [0, \infty]$$

$$u(a) = u(b) = 0$$

The eigenvalues are known to be the zeros of

$$f(\lambda) = J_{1/3}\left(\frac{2}{3}\lambda^{3/2}\right) + J_{-1/3}\left(\frac{2}{3}\lambda^{3/2}\right)$$

For each eigenvalue  $\lambda_k$ , the program prints  $k$ ,  $\lambda_k$  and  $f(\lambda_k)$ .

```

USE SLEIG_INT
USE CBJS_INT

IMPLICIT NONE

! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, INDEX(10), NUMEIG, NOUT
REAL CONS(8), EVAL(10), LAMBDA, TEVLAB, &
TEVLRL, XNU
COMPLEX CBS1(1), CBS2(1), Z
LOGICAL ENDFIN(2)

! SPECIFICATIONS FOR INTRINSICS
```

```

 INTRINSIC CMPLX, SQRT
 REAL SQRT
 COMPLEX CMPLX
!
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
 EXTERNAL COEFF
!
 CALL UMACH (2, NOUT)
!
! Define boundary conditions
 CONS(1) = 1.0
 CONS(2) = 0.0
 CONS(3) = 0.0
 CONS(4) = 0.0
 CONS(5) = 1.0
 CONS(6) = 0.0
 CONS(7) = 0.0
 CONS(8) = 0.0
!
 ENDFIN(1) = .TRUE.
 ENDFIN(2) = .FALSE.
!
! Compute the first 10 eigenvalues
 NUMEIG = 10
 DO 10 I=1, NUMEIG
 INDEX(I) = I - 1
10 CONTINUE
!
! Set absolute and relative tolerance
!
 CALL SLEIG (CONS, COEFF, ENDFIN, INDEX, EVAL)
!
 XNU = -1.0/3.0
 WRITE(NOUT,99998)
 DO 20 I=1, NUMEIG
 LAMBDA = EVAL(I)
 Z = CMPLX(2.0/3.0*LAMBDA*SQRT(LAMBDA),0.0)
 CALL CBJS (XNU, Z, 1, CBS1)
 CALL CBJS (-XNU, Z, 1, CBS2)
 WRITE (NOUT,99999) I-1, LAMBDA, REAL(CBS1(1) + CBS2(1))
20 CONTINUE
!
99998 FORMAT(/, 2X, 'index', 5X, 'lambda', 5X, 'f(lambda)',/)
99999 FORMAT(I5, F13.4, E15.4)
 END
!
 SUBROUTINE COEFF (X, PX, QX, RX)
!
! SPECIFICATIONS FOR ARGUMENTS
 REAL X, PX, QX, RX
!
 PX = 1.0
 QX = X
 RX = 1.0
 RETURN
 END

```

## Output

| index | lambda  | f(lambda)   |
|-------|---------|-------------|
| 0     | 2.3381  | -0.8285E-05 |
| 1     | 4.0879  | -0.1651E-04 |
| 2     | 5.5205  | 0.6843E-04  |
| 3     | 6.7867  | -0.4523E-05 |
| 4     | 7.9440  | 0.8952E-04  |
| 5     | 9.0227  | 0.1123E-04  |
| 6     | 10.0401 | 0.1031E-03  |
| 7     | 11.0084 | -0.7913E-04 |
| 8     | 11.9361 | -0.5095E-04 |
| 9     | 12.8293 | 0.4645E-03  |

## Additional Examples

### Example 2

In this problem from Scott, Shampine and Wing (1969),

$$p(x) = r(x) = 1$$

$$q(x) = x^2 + x^4$$

$$[a, b] = [-\infty, \infty]$$

$$u(a) = u(b) = 0$$

the first eigenvalue and associated eigenfunction, evaluated at selected points, are computed. As a rough check of the correctness of the results, the magnitude of the residual

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u - \lambda r(x)u$$

is printed. We compute a spline interpolant to  $u'$  and use the function CSDER to estimate the quantity  $-(p(x)u')'$ .

```

USE S2EIG_INT
USE CSDER_INT
USE UMACH_INT
USE CSAKM_INT

IMPLICIT NONE

! SPECIFICATIONS FOR LOCAL VARIABLES

INTEGER I, IFLAG(1), INDEX(1), IWORK(100), NINTV, NOUT, NRHO, &
NUMEIG, NUMX
REAL BRKUP(61), CONS(8), CSCFUP(4,61), EF(61), EVAL(1), &
LAMBDA, PDEF(61), PX, QX, RESIDUAL, RHO(1), RX, T(1), &
TEVLAB, TEVLR, TOLS(4), WORK(3000), X, XEF(61)
LOGICAL ENDFIN(2), JOB(5), TYPE(4,2)

! SPECIFICATIONS FOR INTRINSICS

INTRINSIC ABS, REAL
REAL ABS, REAL

! SPECIFICATIONS FOR SUBROUTINES

EXTERNAL COEFF

! Define boundary conditions
CONS(1) = 1.0

```

```

CONS(2) = 0.0
CONS(3) = 0.0
CONS(4) = 0.0
CONS(5) = 1.0
CONS(6) = 0.0
CONS(7) = 0.0
CONS(8) = 0.0
!
! Compute eigenvalue and eigenfunctions
JOB(1) = .FALSE.
JOB(2) = .TRUE.
JOB(3) = .FALSE.
JOB(4) = .FALSE.
JOB(5) = .FALSE.
!
ENDFIN(1) = .FALSE.
ENDFIN(2) = .FALSE.
!
! Compute eigenvalue with index 0
NUMEIG = 1
INDEX(1) = 0
!
TEVLAB = 1.0E-3
TEVLRL = 1.0E-3
TOLS(1) = TEVLAB
TOLS(2) = TEVLRL
TOLS(3) = TEVLAB
TOLS(4) = TEVLRL
NRHO = 0
!
! Set up mesh, points at which u and
! u' will be computed
NUMX = 61
DO 10 I=1, NUMX
 XEF(I) = 0.05*REAL(I-31)
10 CONTINUE
!
CALL S2EIG (CONS, COEFF, ENDFIN, NUMEIG, INDEX, TEVLAB, TEVLRL, &
 EVAL, JOB, 0, TOLS, NUMX, XEF, NRHO, T, TYPE, EF, &
 PDEF, RHO, IFLAG, WORK, IWORK)
!
LAMBDA = EVAL(1)
20 CONTINUE
!
! Compute spline interpolant to u'
!
CALL CSAKM (XEF, PDEF, BRKUP, CSCFUP)
NINTV = NUMX - 1
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99997) ' lambda = ', LAMBDA
WRITE (NOUT,99999)
!
! At a subset of points from the
! input mesh, compute residual =
! abs(-(u')' + q(x)u - lambda*u).
! We know p(x) = 1 and r(x) = 1.
DO 30 I=1, 41, 2
 X = XEF(I+10)
 CALL COEFF (X, PX, QX, RX)
!

```

```

! Use the spline fit to u' to
! estimate u'' with CSDER
!
 RESIDUAL = ABS(-CSDER(1,X,BRKUP,CSCFUP)+QX*EF(I+10)- &
 LAMBDA*EF(I+10))
 WRITE (NOUT,99998) X, EF(I+10), PDEF(I+10), RESIDUAL
30 CONTINUE
!
99997 FORMAT (/, A14, F10.5, /)
99998 FORMAT (5X, F4.1, 3F15.5)
99999 FORMAT (7X, 'x', 11X, 'u(x)', 10X, 'u''(x)', 9X, 'residual', /)
END
!
 SUBROUTINE COEFF (X, PX, QX, RX)
! SPECIFICATIONS FOR ARGUMENTS
 REAL X, PX, QX, RX
!
 PX = 1.0
 QX = X*X + X*X*X*X
 RX = 1.0
 RETURN
 END

```

## Output

```

lambda = 1.39247
 x u(x) u'(x) residual
-1.0 0.38632 0.65019 0.00189
-0.9 0.45218 0.66372 0.00081
-0.8 0.51837 0.65653 0.00023
-0.7 0.58278 0.62827 0.00113
-0.6 0.64334 0.57977 0.00183
-0.5 0.69812 0.51283 0.00230
-0.4 0.74537 0.42990 0.00273
-0.3 0.78366 0.33393 0.00265
-0.2 0.81183 0.22811 0.00273
-0.1 0.82906 0.11570 0.00278
 0.0 0.83473 0.00000 0.00136
 0.1 0.82893 -0.11568 0.00273
 0.2 0.81170 -0.22807 0.00273
 0.3 0.78353 -0.33388 0.00267
 0.4 0.74525 -0.42983 0.00265
 0.5 0.69800 -0.51274 0.00230
 0.6 0.64324 -0.57967 0.00182
 0.7 0.58269 -0.62816 0.00113
 0.8 0.51828 -0.65641 0.00023
 0.9 0.45211 -0.66361 0.00081
 1.0 0.38626 -0.65008 0.00189

```

---

## SLCNT

Calculates the indices of eigenvalues of a Sturm-Liouville problem of the form for

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u = \lambda r(x)u \text{ for } x \text{ in } [a, b]$$

with boundary conditions (at regular points)

$$a_1 u - a_2 (pu') = \lambda (a'_1 u - a'_2 (pu')) \text{ at } a$$
$$b_1 u + b_2 (pu') = 0 \text{ at } b$$

in a specified subinterval of the real line,  $[\alpha, \beta]$ .

### Required Arguments

**ALPHA** — Value of the left end point of the search interval. (Input)

**BETAR** — Value of the right end point of the search interval. (Input)

**CONS** — Array of size eight containing

$$a_1, a'_1, a_2, a'_2, b_1, b_2, a \text{ and } b$$

in locations `CONS (1) ... CONS (8)`, respectively. (Input)

**COEFFN** — User-supplied subroutine to evaluate the coefficient functions. The usage is

`CALL COEFFN (X, PX, QX, RX)`

`X` — Independent variable. (Input)

`PX` — The value of  $p(x)$  at  $x$ . (Output)

`QX` — The value of  $q(x)$  at  $x$ . (Output)

`RX` — The value of  $r(x)$  at  $x$ . (Output)

`COEFFN` must be declared `EXTERNAL` in the calling program.

**ENDFIN** — Logical array of size two. `ENDFIN = .true.` if and only if the endpoint  $a$  is finite. `ENDFIN(2) = .true.` if and only if endpoint  $b$  is finite. (Input)

**IFIRST** — The index of the first eigenvalue greater than  $\alpha$ . (Output)

**NTOTAL** — Total number of eigenvalues in the interval  $[\alpha, \beta]$ . (Output)

### FORTRAN 90 Interface

Generic: `CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)`

Specific: The specific interface names are `S_SLCNT` and `D_SLCNT`.

### FORTRAN 77 Interface

Single: `CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)`

Double: The double precision name is DSLCNT.

## Description

This subroutine computes the indices of eigenvalues, if any, in a subinterval of the real line for Sturm-Liouville problems in the form

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) + q(x)u = \lambda r(x)u \text{ for } x \text{ in } [a, b]$$

with boundary conditions (at regular points)

$$\begin{aligned} a_1 u - a_2 (pu') &= \lambda (a'_1 u - a'_2 (pu')) \text{ at } a \\ b_1 u + b_2 (pu') &= 0 \text{ at } b \end{aligned}$$

It is intended to be used in conjunction with SLEIG. SLCNT is based on the routine INTERV from the package SLEDGE.

## Example

Consider the harmonic oscillator (Titchmarsh) defined by

$$\begin{aligned} p(x) &= 1 \\ q(x) &= x^2 \\ r(x) &= 1 \\ [a, b] &= [-\infty, \infty] \\ u(a) &= 0 \\ u(b) &= 0 \end{aligned}$$

The eigenvalues of this problem are known to be

$$\lambda_k = 2k + 1, k = 0, 1, \dots$$

Therefore in the interval [10, 16] we expect SLCNT to note three eigenvalues, with the first of these having index five.

```
USE SLCNT_INT
USE UMACH_INT

IMPLICIT NONE
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IFIRST, NOUT, NTOTAL
REAL ALPHA, BETAR, CONS(8)
LOGICAL ENDFIN(2)
!
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL COEFFN
!
CALL UMACH (2, NOUT)
! set u(a) = 0, u(b) = 0
CONS(1) = 1.0E0
```

```

CONS(2) = 0.0E0
CONS(3) = 0.0E0
CONS(4) = 0.0E0
CONS(5) = 1.0E0
CONS(6) = 0.0E0
CONS(7) = 0.0E0
CONS(8) = 0.0E0
!
ENDFIN(1) = .FALSE.
ENDFIN(2) = .FALSE.
!
ALPHA = 10.0
BETAR = 16.0
!
CALL SLCNT (ALPHA, BETAR, CONS, COEFFN, ENDFIN, IFIRST, NTOTAL)
!
WRITE (NOUT,99998) ALPHA, BETAR, IFIRST
WRITE (NOUT,99999) NTOTAL
!
99998 FORMAT (/, 'Index of first eigenvalue in [', F5.2, ',', F5.2, &
 ']' IS ', I2)
99999 FORMAT ('Total number of eigenvalues in this interval: ', I2)
!
END
!
SUBROUTINE COEFFN (X, PX, QX, RX)
!
REAL X, PX, QX, RX
!
 SPECIFICATIONS FOR ARGUMENTS
!
PX = 1.0E0
QX = X*X
RX = 1.0E0
RETURN
END

```

## Output

```

Index of first eigenvalue in [10.00,16.00] is 5
Total number of eigenvalues in this interval: 3

```

# Chapter 6: Transforms

---

## Routines

|                                                                                   |                           |      |
|-----------------------------------------------------------------------------------|---------------------------|------|
| <b>6.1. Real Trigonometric FFT</b>                                                |                           |      |
| Computes the Discrete Fourier Transform of a rank-1 complex array, x .....        | <a href="#">FAST_DFT</a>  | 1220 |
| Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, x ..... | <a href="#">FAST_2DFT</a> | 1227 |
| Computes the Discrete Fourier Transform 2DFT) of a rank-3 complex array, x .....  | <a href="#">FAST_3DFT</a> | 1233 |
| Forward transform .....                                                           | <a href="#">FFTRF</a>     | 1236 |
| Backward or inverse transform .....                                               | <a href="#">FFTRB</a>     | 1240 |
| Initialization routine for FFTR* .....                                            | <a href="#">FFTRI</a>     | 1243 |
| <b>6.2. Complex Exponential FFT</b>                                               |                           |      |
| Forward transform.....                                                            | <a href="#">FFTCF</a>     | 1245 |
| Backward or inverse transform .....                                               | <a href="#">FFTCB</a>     | 1248 |
| Initialization routine for FFTC* .....                                            | <a href="#">FFTCI</a>     | 1251 |
| <b>6.3. Real Sine and Cosine FFTs</b>                                             |                           |      |
| Forward and inverse sine transform .....                                          | <a href="#">FSINT</a>     | 1253 |
| Initialization routine for FSINT .....                                            | <a href="#">FSINI</a>     | 1255 |
| Forward and inverse cosine transform .....                                        | <a href="#">FCOST</a>     | 1257 |
| Initialization routine for FCOST.....                                             | <a href="#">FCOSI</a>     | 1259 |
| <b>6.4. Real Quarter Sine and Quarter Cosine FFTs</b>                             |                           |      |
| Forward quarter sine transform .....                                              | <a href="#">QSINF</a>     | 1261 |
| Backward or inverse transform .....                                               | <a href="#">QSINB</a>     | 1263 |
| Initialization routine for QSIN*.....                                             | <a href="#">QSINI</a>     | 1266 |
| Forward quarter cosine transform.....                                             | <a href="#">QCOSF</a>     | 1268 |
| Backward or inverse transform .....                                               | <a href="#">QCOSB</a>     | 1270 |
| Initialization routine for QCOS* .....                                            | <a href="#">QCOSI</a>     | 1272 |
| <b>6.5. Two- and Three-Dimensional Complex FFTs</b>                               |                           |      |
| Forward transform .....                                                           | <a href="#">FFT2D</a>     | 1274 |
| Backward or inverse transform .....                                               | <a href="#">FFT2B</a>     | 1277 |
| Forward transform .....                                                           | <a href="#">FFT3F</a>     | 1281 |
| Backward or inverse transform .....                                               | <a href="#">FFT3B</a>     | 1285 |
| <b>6.6. Convolutions and Correlations</b>                                         |                           |      |
| Real convolution .....                                                            | <a href="#">RCONV</a>     | 1289 |

|                                                      |       |      |
|------------------------------------------------------|-------|------|
| Complex convolution .....                            | CCONV | 1294 |
| Real correlation .....                               | RCORL | 1299 |
| Complex correlation .....                            | CCORL | 1304 |
| <b>6.7. Laplace Transform</b>                        |       |      |
| Inverse Laplace transform.....                       | INLAP | 1309 |
| Inverse Laplace transform for smooth functions ..... | SINLP | 1311 |

## Usage Notes

### Fast Fourier Transforms

A Fast Fourier Transform (FFT) is simply a discrete Fourier transform that can be computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately  $N^2$  operations where  $N$  is the number of points in the transform, while the FFT (which computes the same values) takes approximately  $N \log N$  operations. The algorithms in this chapter are modeled on the Cooley-Tukey (1965) algorithm; hence, the computational savings occur, not for all integers  $N$ , but for  $N$  which are highly composite. That is,  $N$  (or in certain cases  $N + 1$  or  $N - 1$ ) should be a product of small primes.

All of the FFT routines compute a *discrete* Fourier transform. The routines accept a vector  $x$  of length  $N$  and return a vector

$$\hat{x}$$

defined by

$$\hat{x}_m := \sum_{n=1}^N x_n \omega_{nm}$$

The various transforms are determined by the selection of  $\omega$ . In the following table, we indicate the selection of  $\omega$  for the various transforms. This table should not be mistaken for a definition since the precise transform definitions (at times) depend on whether  $N$  or  $m$  is even or odd.

| Routine | $\omega_{nm}$                             |
|---------|-------------------------------------------|
| FFTRF   | $\cos$ or $\sin \frac{(m-1)(n-1)2\pi}{N}$ |
| FFTRB   | $\cos$ or $\sin \frac{(m-1)(n-1)2\pi}{N}$ |
| FFTCF   | $\exp^{-2\pi i(n-1)(m-1)/N}$              |
| FFTCB   | $\exp^{2\pi i(n-1)(m-1)/N}$               |
| FSINT   | $\sin \frac{nm\pi}{N+1}$                  |
| FCOST   | $\cos \frac{(n-1)(m-1)\pi}{N-1}$          |
| QSINF   | $2 \sin \frac{(2m-1)n\pi}{2N}$            |
| QSINB   | $4 \sin \frac{(2n-1)m\pi}{2N}$            |
| QCOSE   | $2 \cos \frac{(2m-1)(n-1)\pi}{2N}$        |
| QCOSEB  | $4 \cos \frac{(2n-1)(m-1)\pi}{2N}$        |

For many of the routines listed above, there is a corresponding “I” (for initialization) routine. Use these routines *only* when repeatedly transforming sequences of the same length. In this situation, the “I” routine will compute the initial setup once, and then the user will call the corresponding “2” routine. This can result in substantial computational savings. For more information on the usage of these routines, the user should consult the documentation under the appropriate routine name.

In addition to the one-dimensional transformations described above, we also provide complex two and three-dimensional FFTs and their inverses based on calls to either `FFTCF` or `FFTCB`. If you need a higher dimensional transform, then you should consult the example program for `FFTCI`, which suggests a basic strategy one could employ.

## Continuous versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham, 1974) as

$$\hat{f}(\omega) = (F f)(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \omega t} dt$$

We begin by making the following approximation:

$$\begin{aligned}\hat{f}(\omega) &\approx \int_{-T/2}^{T/2} f(t) e^{-2\pi i \omega t} dt \\ &= \int_0^T f(t - T/2) e^{-2\pi i \omega (t - T/2)} dt \\ &= e^{\pi i \omega T} \int_0^T f(t - T/2) e^{-2\pi i \omega t} dt\end{aligned}$$

If we approximate the last integral using the rectangle rule with spacing  $h = T/N$ , we have

$$\hat{f}(\omega) \approx e^{\pi i \omega T} h \sum_{k=0}^{N-1} e^{-2\pi i \omega kh} f(kh - T/2)$$

Finally, setting  $\omega = j/T$  for  $j = 0, \dots, N-1$  yields

$$\hat{f}(j/T) \approx e^{\pi i j} h \sum_{k=0}^{N-1} e^{-2\pi i jk/N} f(kh - T/2) = (-1)^j h \sum_{k=0}^{N-1} e^{-2\pi i jk/N} f_k^h$$

where the vector  $f^h = (f(-T/2), \dots, f((N-1)h - T/2))$ . Thus, after scaling the components by  $(-1)^j h$ , the discrete Fourier transform as computed in `FFTCF` (with input  $f^h$ ) is related to an approximation of the continuous Fourier transform by the above formula. This is seen more clearly by making a change of variables in the last sum. Set

$$n = k + 1, \quad m = j + 1, \quad \text{and} \quad f_k^h = x_n$$

then, for  $m = 1, \dots, N$  we have

$$\hat{f}((m-1)/T) \approx -(-1)^m h \hat{x}_m = -(-1)^m h \sum_{n=1}^N e^{-2\pi i (m-1)(n-1)/N} x_n$$

If the function  $f$  is expressed as a FORTRAN function routine, then the continuous Fourier transform

$$\hat{f}$$

can be approximated using the IMSL routine `QDAWF` (see [Chapter 4, Integration and Differentiation](#)).

## Inverse Laplace Transform

The last two routines described in this chapter, `INLAP` and `SINLP`, compute the inverse Laplace transforms.

## FAST\_DFT

Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array,  $x$ .

## Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

## Optional Arguments

`forward_in = x` (Input)

Stores the input complex array of rank-1 to be transformed.

`forward_out = y` (Output)

Stores the output complex array of rank-1 resulting from the transform.

`inverse_in = y` (Input)

Stores the input complex array of rank-1 to be inverted.

`inverse_out = x` (Output)

Stores the output complex array of rank-1 resulting from the inverse transform.

`ndata = n` (Input)

Uses the sub-array of size `n` for the numbers.

Default value: `n = size(x)`.

`ido = ido` (Input/Output)

Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. Example 3 illustrates this usage. The general algorithm for this usage is to enter `fast_dft` with `ido = 0`. A return occurs thereafter with `ido < 0`. The optional rank-1 complex array `w(:)` with `size(w) >= -ido` must be re-allocated. Then, re-enter `fast_dft`. The next return from `fast_dft` has the output value `ido = 1`. The variables required for the transform and its inverse are saved in `w(:)`. Thereafter, when the routine is entered with `ido = 1` and for the same value of `n`, the contents of `w(:)` will be used for the working variables. The expensive initialization step is avoided. The optional arguments “`ido=`” and “`work_array=`” must be used together.

`work_array = w(:)` (Output/Input)

Complex array of rank-1 used to store working variables and values between calls to `fast_dft`. The value for `size(w)` must be at least as large as the value `-ido` for the value of `ido < 0`.

`iopt = iopt(:)` (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `fast_dft`. The options are as follows:

| Packaged Options for <code>FAST_DFT</code> |                                       |              |
|--------------------------------------------|---------------------------------------|--------------|
| Option Prefix = ?                          | Option Name                           | Option Value |
| <code>c_, z_</code>                        | <code>fast_dft_scan_for_NaN</code>    | 1            |
| <code>c_, z_</code>                        | <code>fast_dft_near_power_of_2</code> | 2            |
| <code>c_, z_</code>                        | <code>fast_dft_scale_forward</code>   | 3            |
| <code>c_, z_</code>                        | <code>Fast_dft_scale_inverse</code>   | 4            |

```
iopt(IO) = ?_options(?_fast_dft_scan_for_NaN, ?_dummy)
```

Examines each input array entry to find the first value such that `isNaN(x(i)) == .true.`

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs.

```
iopt(IO) = ?_options(?_fast_dft_near_power_of_2, ?_dummy)
```

Nearest power of  $2 \geq n$  is returned as an output in `iopt(IO + 1)%idummy`.

```
iopt(IO) = ?_options(?_fast_dft_scale_forward, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the forward transformed array.

Default value is 1.

```
iopt(IO) = ?_options(?_fast_dft_scale_inverse, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the inverse transformed array.

Default value is 1.

## FORTRAN 90 Interface

Generic: None

Specific: The specific interface names are `S_FAST_DFT`, `D_FAST_DFT`, `C_FAST_DFT`, and `Z_FAST_DFT`.

## Description

The `fast_dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776). The maximum computing efficiency occurs when the size of the array can be factored in the form

$$n = 2^{i_1} 3^{i_2} 4^{i_3} 5^{i_4}$$

using non-negative integer values  $\{i_1, i_2, i_3, i_4\}$ . There is no further restriction on  $n \geq 1$ .

## Fatal and Terminal Messages

See the *messages.gls* file for error messages for `FAST_DFT`. These error messages are numbered 651–661; 701–711.

### Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_dft_int
use rand_gen_int

implicit none

! This is Example 1 for FAST_DFT.

integer, parameter :: n=1024
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) err, y(2*n)
complex(kind(1e0)), dimension(n) :: a, b, c

! Generate a random complex sequence.
call rand_gen(y)
a = cmplx(y(1:n), y(n+1:2*n), kind(one))
c = a

! Transform and then invert the sequence back.
call c_fast_dft(forward_in=a, &
 forward_out=b)
call c_fast_dft(inverse_in=b, &
 inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
err = maxval(abs(c-a))/maxval(abs(c))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for FAST_DFT is correct.'
end if

end
```

## Output

```
Example 1 for FAST_DFT is correct.
```

## Additional Examples

### Example 2: Cyclical Data with a Linear Trend

This set of data is sampled from a function  $x(t) = at + b + y(t)$ , where  $y(t)$  is a harmonic series. The independent variable is normalized as  $-1 \leq t \leq 1$ . Thus, the data is said *to have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver `LIN_SOL_LSQ`, [Chapter 1](#). Then, the residuals are transformed and the resulting frequencies are analyzed.

```

use fast_dft_int
use lin_sol_lsq_int
use rand_gen_int
use sort_real_int

implicit none

! This is Example 2 for FAST_DFT.

integer i
integer, parameter :: n=64, k=4
integer ip(n)
real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
real(kind(1e0)) delta_t, pi
real(kind(1e0)) y(k), z(2), indx(k), t(n), temp(n)
complex(kind(1e0)) a_trend(n,2), a, b_trend(n,1), b, c(k), f(n), &
 r(n), x(n), x_trend(2,1)

! Generate random data for linear trend and harmonic series.
call rand_gen(z)
a = z(1); b = z(2)
call rand_gen(y)
! This emphasizes harmonics 2 through k+1.
c = y + one

! Determine sampling interval.
delta_t = two/n
t = (/(-one+i*delta_t, i=0,n-1)/)

! Compute pi.
pi = atan(one)*4E0
indx = (/ (i*pi, i=1,k) /)

! Make up data set as a linear trend plus harmonics.
x = a + b*t + &
 matmul(exp(cmplx(zero,spread(t,2,k)*spread(indx,1,n),kind(one))),c)

! Define least-squares matrix data for a linear trend.
a_trend(1:,1) = one
a_trend(1:,2) = t
b_trend(1:,1) = x

! Solve for a linear trend.
call lin_sol_lsq(a_trend, b_trend, x_trend)

! Compute harmonic residuals.
r = x - reshape(matmul(a_trend,x_trend), (/n/))

! Transform harmonic residuals.
call c_fast_dft(forward_in=r, forward_out=f)
ip = (/ (i, i=1,n) /)

! The dominant frequencies should be 2 through k+1.
! Sort the magnitude of the transform first.
call s_sort_real(-(abs(f)), temp, iperm=ip)

```

```

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with 2 through k+1.
 call s_sort_real(real(ip(1:k)), temp)
 ip(1:k)=(/ (i,i=2,k+1)/)

! Check the results.
 if (count(int(temp(1:k)) /= ip(1:k)) == 0) then
 write (*,*) 'Example 2 for FAST_DFT is correct.'
 end if

end

```

## Output

Example 2 for FAST\_DFT is correct.

### Example 3: Several Transforms with Initialization

In this example, the optional arguments `ido` and `work_array` are used to save working variables in the calling program unit. This results in maximum efficiency of the transform and its inverse since the working variables do not have to be precomputed following each entry to routine `fast_dft`.

```

 use fast_dft_int
 use rand_gen_int

 implicit none

! This is Example 3 for FAST_DFT.

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.
 integer, parameter :: n=64
 integer ido_value
 real(kind(1e0)) :: one=1e0
 real(kind(1e0)) err, y(2*n)
 complex(kind(1e0)), dimension(n) :: a, b, save_a
 complex(kind(1e0)), allocatable :: work(:)

! Generate a random complex array.
 call rand_gen(y)
 a = cmplx(y(1:n), y(n+1:2*n), kind(one))
 save_a = a

! Transform and then invert the sequence using the pre-computed
! working values.
 ido_value = 0
 do
 if(allocated(work)) deallocate(work)

! Allocate the space required for work(:).
 if (ido_value <= 0) allocate(work(-ido_value))

 call c_fast_dft(forward_in=a, forward_out=b, &

```

```

 ido=ido_value, work_array=work)

 if (ido_value == 1) exit
end do

! Re-enter routine with working values available in work(:).
 call c_fast_dft(inverse_in=b, inverse_out=a, &
 ido=ido_value, work_array=work)

! Deallocate the space used for work(:).
 if (allocated(work)) deallocate(work)

! Check the results.
 err = maxval(abs(save_a-a))/maxval(abs(save_a))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 3 for FAST_DFT is correct.'
 end if

end

```

## Output

Example 3 for FAST\_DFT is correct.

## Example 4: Convolutions using Fourier Transforms

In this example we compute sums

$$c_k = \sum_{j=0}^{n-1} a_j b_{k-j}, k = 0, \dots, n-1$$

The definition implies a matrix-vector product. A direct approach requires about  $n^2$  operations consisting of an add and multiply. An efficient method consisting of computing the products of the transforms of the

$$\{a_j\} \text{ and } \{b_j\}$$

then inverting this product, is preferable to the matrix-vector approach for large problems. The example is also illustrated in `operator_ex37`, [Chapter 10](#) using the generic function interface FFT and IFFT.

```

 use fast_dft_int
 use rand_gen_int

 implicit none

! This is Example 4 for FAST_DFT.

 integer j
 integer, parameter :: n=40
 real(kind(1e0)) :: one=1e0
 real(kind(1e0)) err

```

```

 real(kind(1e0)), dimension(n) :: x, y, yy(n,n)
 complex(kind(1e0)), dimension(n) :: a, b, c, d, e, f

! Generate two random complex sequence 'a' and 'b'.

 call rand_gen(x)
 call rand_gen(y)
 a=x; b=y

! Compute the convolution 'c' of 'a' and 'b'.
! Use matrix times vector for test results.
 yy(1:,1)=y
 do j=2,n
 yy(2:,j)=yy(1:n-1,j-1)
 yy(1,j)=yy(n,j-1)
 end do

 c=matmul(yy,x)

! Transform the 'a' and 'b' sequences into 'd' and 'e'.

 call c_fast_dft(forward_in=a, &
 forward_out=d)
 call c_fast_dft(forward_in=b, &
 forward_out=e)

! Invert the product d*e.

 call c_fast_dft(inverse_in=d*e, &
 inverse_out=f)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).

 err = maxval(abs(c-f))/maxval(abs(c))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for FAST_DFT is correct.'
 end if

 end

```

## Output

Example 4 for FAST\_DFT is correct.

---

## FAST\_2DFT

Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array,  $x$ .

### Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

## Optional Arguments

`forward_in = x` (Input)

Stores the input complex array of rank-2 to be transformed.

`forward_out = y` (Output)

Stores the output complex array of rank-2 resulting from the transform.

`inverse_in = y` (Input)

Stores the input complex array of rank-2 to be inverted.

`inverse_out = x` (Output)

Stores the output complex array of rank-2 resulting from the inverse transform.

`mdata = m` (Input)

Uses the sub-array in first dimension of size `m` for the numbers.

Default value: `m = size(x, 1)`.

`ndata = n` (Input)

Uses the sub-array in the second dimension of size `n` for the numbers.

Default value: `n = size(x, 2)`.

`ido = ido` (Input/Output)

Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_2dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. Example 3 illustrates this usage. The general algorithm for this usage is to enter `fast_2dft` with `ido = 0`. A return occurs thereafter with `ido < 0`. The optional rank-1 complex array `w(:)` with `size(w) >= -ido` must be re-allocated. Then, re-enter `fast_2dft`. The next return from `fast_2dft` has the output value `ido = 1`. The variables required for the transform and its inverse are saved in `w(:)`. Thereafter, when the routine is entered with `ido = 1` and for the same values of `m` and `n`, the contents of `w(:)` will be used for the working variables. The expensive initialization step is avoided. The optional arguments “`ido=`” and “`work_array=`” must be used together.

`work_array = w(:)` (Output/Input)

Complex array of rank-1 used to store working variables and values between calls to `fast_2dft`. The value for `size(w)` must be at least as large as the value `-ido` for the value of `ido < 0`.

`iopt = iopt(:)` (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `fast_2dft`. The options are as follows:

|                                                    |
|----------------------------------------------------|
| <b>Packaged Options for <code>FAST_2DFT</code></b> |
|----------------------------------------------------|

| Packaged Options for <code>FAST_2DFT</code> |                                        |              |
|---------------------------------------------|----------------------------------------|--------------|
| Option Prefix = ?                           | Option Name                            | Option Value |
| <code>c_, z_</code>                         | <code>fast_2dft_scan_for_NaN</code>    | 1            |
| <code>c_, z_</code>                         | <code>fast_2dft_near_power_of_2</code> | 2            |
| <code>c_, z_</code>                         | <code>fast_2dft_scale_forward</code>   | 3            |
| <code>c_, z_</code>                         | <code>fast_2dft_scale_inverse</code>   | 4            |

```
iopt(IO) = ?_options(?_fast_2dft_scan_for_NaN, ?_dummy)
```

Examines each input array entry to find the first value such that `isNaN(x(i,j)) == .true.`

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs.

```
iopt(IO) = ?_options(?_fast_2dft_near_power_of_2, ?_dummy)
```

Nearest powers of  $2 \geq m$  and  $\geq n$  are returned as an outputs in `iopt(IO + 1)%idummy` and `iopt(IO + 2)%idummy`.

```
iopt(IO) = ?_options(?_fast_2dft_scale_forward, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the forward transformed array.

Default value is 1.

```
iopt(IO) = ?_options(?_fast_2dft_scale_inverse, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the inverse transformed array.

Default value is 1.

## FORTRAN 90 Interface

Generic: None

Specific: The specific interface names are `S_FAST_2DFT`, `D_FAST_2DFT`, `C_FAST_2DFT`, and `Z_FAST_2DFT`.

## Description

The `fast_2dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

## Fatal and Terminal Messages

See the `messages.gls` file for error messages for `FAST_2DFT`. These error messages are numbered 670–680; 720–730.

## Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_2dft_int
use rand_int

implicit none

! This is Example 1 for FAST_2DFT.

integer, parameter :: n=24
integer, parameter :: m=40
real(kind(1e0)) :: err, one=1e0
complex(kind(1e0)), dimension(n,m) :: a, b, c

! Generate a random complex sequence.
a=rand(a); c=a

! Transform and then invert the transform.
call c_fast_2dft(forward_in=a, &
 forward_out=b)
call c_fast_2dft(inverse_in=b, &
 inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
err = maxval(abs(c-a))/maxval(abs(c))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for FAST_2DFT is correct.'
end if

end
```

## Output

```
Example 1 for FAST_2DFT is correct.
```

## Additional Examples

### Example 2: Cyclical 2D Data with a Linear Trend

This set of data is sampled from a function  $x(s, t) = a + bs + ct + y(s, t)$ , where  $y(s, t)$  is an harmonic series. The independent variables are normalized as  $-1 \leq s \leq 1$  and  $-1 \leq t \leq 1$ . Thus, the data is said to *have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver . Then, the residuals are transformed and the resulting frequencies are analyzed.

```
use fast_2dft_int
use lin_sol_lsqr_int
use sort_real_int
use rand_int
implicit none
```

```

! This is Example 2 for FAST_2DFT.

integer i
integer, parameter :: n=8, k=15
integer ip(n*n), order(k)
real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
real(kind(1e0)) delta_t
real(kind(1e0)) rn(3), s(n), t(n), temp(n*n), new_order(k)
complex(kind(1e0)) a, b, c, a_trend(n*n,3), b_trend(n*n,1), &
 f(n,n), r(n,n), x(n,n), x_trend(3,1)
complex(kind(1e0)), dimension(n,n) :: g=zero, h=zero

! Generate random data for planar trend.
rn = rand(rn)
a = rn(1)
b = rn(2)
c = rn(3)

! Generate the frequency components of the harmonic series.
! Non-zero random amplitudes given on two edges of the square domain.
g(1:,1)=rand(g(1:,1))
g(1,1:)=rand(g(1,1:))

! Invert 'g' into the harmonic series 'h' in time domain.
call c_fast_2dft(inverse_in=g, inverse_out=h)

! Compute sampling interval.
delta_t = two/n
s = (/(-one + (i-1)*delta_t, i=1,n)/)
t = (/(-one + (i-1)*delta_t, i=1,n)/)

! Make up data set as a linear trend plus harmonics.
x = a + b*spread(s,dim=2,ncopies=n) + &
 c*spread(t,dim=1,ncopies=n) + h

! Define least-squares matrix data for a planar trend.
a_trend(1:,1) = one
a_trend(1:,2) = reshape(reshape(s,dim=2,ncopies=n), (/n*n/))
a_trend(1:,3) = reshape(reshape(t,dim=1,ncopies=n), (/n*n/))
b_trend(1:,1) = reshape(x, (/n*n/))

! Solve for a linear trend.
call lin_sol_lsq(a_trend, b_trend, x_trend)

! Compute harmonic residuals.
r = x - reshape(matmul(a_trend,x_trend), (/n,n/))

! Transform harmonic residuals.
call c_fast_2dft(forward_in=r, forward_out=f)

ip = (/ (i,i=1,n**2) /)

! Sort the magnitude of the transform.
call s_sort_real(-(abs(reshape(f, (/n*n/)))), &
 temp, iperm=ip)

```

```

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with the original frequency order.
 call s_sort_real(real(ip(1:k)), new_order)

 order(1:n) = (/ (i,i=1,n) /)
 order(n+1:k) = (/ ((i-n)*n+1,i=n+1,k) /)

! Check the results.
 if (count(order /= int(new_order)) == 0) then
 write (*,*) 'Example 2 for FAST_2DFT is correct.'
 end if

end

```

## Output

Example 2 for FAST\_2DFT is correct.

### Example 3: Several 2D Transforms with Initialization

In this example, the optional arguments `ido` and `work_array` are used to save working variables in the calling program unit. This results in maximum efficiency of the transform and its inverse since the working variables do not have to be precomputed following each entry to routine `fast_2dft`.

```

 use fast_2dft_int

 implicit none

! This is Example 3 for FAST_2DFT.

 integer i, j
 integer, parameter :: n=256
 real(kind(1e0)), parameter :: one=1e0, zero=0e0
 real(kind(1e0)) r(n,n), err
 complex(kind(1e0)) a(n,n), b(n,n), c(n,n)

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.

 integer ido_value
 complex(kind(1e0)), allocatable :: work(:)

! Fill in value one for points inside the circle with r=64.
 a = zero
 r = reshape(/(((i-n/2)**2 + (j-n/2)**2, i=1,n), &
 j=1,n)/), (/n,n/))
 where (r <= (n/4)**2) a = one
 c = a

! Transform and then invert the sequence using the pre-computed
! working values.

```

```

ido_value = 0
do
 if(allocated(work)) deallocate(work)
! Allocate the space required for work(:).
 if (ido_value <= 0) allocate(work(-ido_value))
! Transform the image and then invert it back.
 call c_fast_2dft(forward_in=a, &
 forward_out=b, IDO=ido_value, work_array=work)
 if (ido_value == 1) exit
end do
call c_fast_2dft(inverse_in=b, &
 inverse_out=a, IDO=ido_value, work_array=work)
! Deallocate the space used for work(:).
 if (allocated(work)) deallocate(work)
! Check that inverse(transform(image)) = image.
 err = maxval(abs(c-a))/maxval(abs(c))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 3 for FAST_2DFT is correct.'
 end if
end
end

```

## Output

Example 3 for FAST\_2DFT is correct.

---

# FAST\_3DFT

Computes the Discrete Fourier Transform (2DFT) of a rank-3 complex array.

## Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

## Optional Arguments

`forward_in = x` (Input)

Stores the input complex array of rank-3 to be transformed.

`forward_out = y` (Output)

Stores the output complex array of rank-3 resulting from the transform.

`inverse_in = y` (Input)

Stores the input complex array of rank-3 to be inverted.

`inverse_out = x` (Output)

Stores the output complex array of rank-3 resulting from the inverse transform.

`mdata = m` (Input)

Uses the sub-array in first dimension of size `m` for the numbers.

Default value: `m = size(x, 1)`.

`ndata = n` (Input)

Uses the sub-array in the second dimension of size `n` for the numbers.

Default value: `n = size(x, 2)`.

`kdata = k` (Input)

Uses the sub-array in the third dimension of size `k` for the numbers.

Default value: `k = size(x, 3)`.

`ido = ido` (Input/Output)

Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_3dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. The general algorithm for this usage is to enter `fast_3dft` with `ido = 0`. A return occurs thereafter with `ido < 0`. The optional rank-1 complex array `w(:)` with `size(w) >= -ido` must be re-allocated. Then, re-enter `fast_3dft`. The next return from `fast_3dft` has the output value `ido = 1`. The variables required for the transform and its inverse are saved in `w(:)`. Thereafter, when the routine is entered with `ido = 1` and for the same values of `m` and `n`, the contents of `w(:)` will be used for the working variables. The expensive initialization step is avoided. The optional arguments “`ido=`” and “`work_array=`” must be used together.

`work_array = w(:)` (Output/Input)

Complex array of rank-1 used to store working variables and values between calls to `fast_3dft`. The value for `size(w)` must be at least as large as the value `-ido` for the value of `ido < 0`.

`iopt = iopt(:)` (Input/Output)

Derived type array with the same precision as the input array; used for passing optional data to `fast_3dft`. The options are as follows:

| Packaged Options for <code>FAST_3DFT</code> |                                        |              |
|---------------------------------------------|----------------------------------------|--------------|
| Option Prefix = ?                           | Option Name                            | Option Value |
| <code>C_, z_</code>                         | <code>fast_3dft_scan_for_NaN</code>    | 1            |
| <code>C_, z_</code>                         | <code>fast_3dft_near_power_of_2</code> | 2            |
| <code>C_, z_</code>                         | <code>fast_3dft_scale_forward</code>   | 3            |
| <code>C_, z_</code>                         | <code>fast_3dft_scale_inverse</code>   | 4            |

`iopt(IO) = ?_options(?_fast_3dft_scan_for_NaN, ?_dummy)`

Examines each input array entry to find the first value such that

```
isNaN(x(i,j,k)) ==.true.
```

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs.

```
iopt(IO) = ?_options(?_fast_3dft_near_power_of_2, ?_dummy)
Nearest powers of $2 \geq m$, $\geq n$, and $\geq k$ are returned as an outputs in
iopt(IO+1)%idummy, iopt(IO+2)%idummy and iopt(IO+3)%idummy
```

```
iopt(IO) = ?_options(?_fast_3dft_scale_forward, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the forward transformed array.

Default value is 1.

```
iopt(IO) = ?_options(?_fast_3dft_scale_inverse, real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```

Complex number defined by the factor

`cmplx(real_part_of_scale, imaginary_part_of_scale)` is multiplied by the inverse transformed array.

Default value is 1.

## FORTRAN 90 Interface

Generic: None

Specific: The specific interface names are `S_FAST_3DFT`, `D_FAST_3DFT`, `C_FAST_3DFT`, and `Z_FAST_3DFT`.

## Description

The `fast_3dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

## Fatal and Terminal Messages

See the `messages.gls` file for error messages for `FAST_3DFT`. These error messages are numbered 685–695; 740–750.

## Example: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_3dft_int

implicit none

! This is Example 1 for FAST_3DFT.

integer i, j, k
```

```

integer, parameter :: n=64
real(kind(1e0)), parameter :: one=1e0, zero=0e0
real(kind(1e0)) r(n,n,n), err
complex(kind(1e0)) a(n,n,n), b(n,n,n), c(n,n,n)

! Fill in value one for points inside the sphere
! with radius=16.
a = zero
do i=1,n
 do j=1,n
 do k=1,n
 r(i,j,k) = (i-n/2)**2+(j-n/2)**2+(k-n/2)**2
 end do
 end do
end do
where (r <= (n/4)**2) a = one
c = a

! Transform the image and then invert it back.
call c_fast_3dft(forward_in=a, &
 forward_out=b)
call c_fast_3dft(inverse_in=b, &
 inverse_out=a)

! Check that inverse(transform(image)) = image.
err = maxval(abs(c-a))/maxval(abs(c))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for FAST_3DFT is correct.'
end if

end

```

## Output

Example 1 for FAST\_3DFT is correct.

---

# FFTRF



Computes the Fourier coefficients of a real periodic sequence.

## Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*SEQ* — Array of length *N* containing the periodic sequence. (Input)

*COEF* — Array of length *N* containing the Fourier coefficients. (Output)

## FORTRAN 90 Interface

Generic:    CALL FFTRF (N, SEQ, COEF)

Specific:    The specific interface names are S\_FFTRF and D\_FFTRF.

## FORTRAN 77 Interface

Single:     CALL FFTRF (N, SEQ, COEF)

Double:     The double precision name is DFFTRF.

## Description

The routine FFTRF computes the discrete Fourier transform of a real vector of size  $N$ . It uses the Intel<sup>®</sup> Math Kernel Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm that is most efficient when  $N$  is a product of small prime factors. If  $N$  satisfies this condition, then the computational effort is proportional to  $N \log N$ .

Specifically, given an  $N$ -vector  $s = \text{SEQ}$ , FFTRF returns in  $c = \text{COEF}$ , if  $N$  is even:

$$c_{2m-2} = \sum_{n=1}^N s_n \cos \left[ \frac{(m-1)(n-1)2\pi}{N} \right] \quad m = 2, \dots, N/2 + 1$$
$$c_{2m-1} = -\sum_{n=1}^N s_n \sin \left[ \frac{(m-1)(n-1)2\pi}{N} \right] \quad m = 2, \dots, N/2$$
$$c_1 = \sum_{n=1}^N s_n$$

If  $N$  is odd,  $c_m$  is defined as above for  $m$  from 2 to  $(N+1)/2$ .

We now describe a fairly common usage of this routine. Let  $f$  be a real valued function of time. Suppose we sample  $f$  at  $N$  equally spaced time intervals of length  $\Delta$  seconds starting at time  $t_0$ . That is, we have

$$\text{SEQ}_i = f(t_0 + (i-1)\Delta) \quad i = 1, 2, \dots, N$$

The routine FFTRF treats this sequence as if it were periodic of period  $N$ . In particular, it assumes that  $f(t_0) = f(t_0 + N\Delta)$ . Hence, the period of the function is assumed to be  $T = N\Delta$ .

Now, FFTRF accepts as input SEQ and returns as output coefficients  $c = \text{COEF}$  that satisfy the following relation when  $N$  is odd ( $N$  even is similar):

$$\text{SEQ}_i = \frac{1}{N} \left[ c_1 + 2 \sum_{n=2}^{(N+1)/2} c_{2n-2} \cos \left[ \frac{2\pi(n-1)(i-1)}{N} \right] - 2 \sum_{n=2}^{(N+1)/2} c_{2n-1} \sin \left[ \frac{2\pi(n-1)(i-1)}{N} \right] \right]$$

This formula is very revealing. It can be interpreted in the following manner. The coefficients produced by `FFTRF` produce an interpolating trigonometric polynomial to the data. That is, if we define

$$g(t) := \frac{1}{N} \left[ c_1 + 2 \sum_{n=2}^{(N+1)/2} c_{2n-2} \cos \left[ \frac{2\pi(n-1)(t-t_0)}{N\Delta} \right] - 2 \sum_{n=2}^{(N+1)/2} c_{2n-1} \sin \left[ \frac{2\pi(n-1)(t-t_0)}{N\Delta} \right] \right]$$

$$= \frac{1}{N} \left[ c_1 + 2 \sum_{n=2}^{(N+1)/2} c_{2n-2} \cos \left[ \frac{2\pi(n-1)(t-t_0)}{T} \right] - 2 \sum_{n=2}^{(N+1)/2} c_{2n-1} \sin \left[ \frac{2\pi(n-1)(t-t_0)}{T} \right] \right]$$

then, we have

$$f(t_0 + (i-1)\Delta) = g(t_0 + (i-1)\Delta)$$

Now, suppose we want to discover the dominant frequencies. One forms the vector  $P$  of length  $N/2$  as follows:

$$P_1 := |c_1|$$

$$P_k := \sqrt{c_{2k-2}^2 + c_{2k-1}^2} \quad k = 2, 3, \dots, (N+1)/2$$

These numbers correspond to the energy in the spectrum of the signal. In particular,  $P_k$  corresponds to the energy level at frequency

$$\frac{k-1}{T} = \frac{k-1}{N\Delta} \quad k = 1, 2, \dots, \frac{N+1}{2}$$

Furthermore, note that there are only  $(N+1)/2 \approx T/(2\Delta)$  resolvable frequencies when  $N$  observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal.

Similar relations hold for the case when  $N$  is even.

Finally, note that the Fourier transform has an (unnormalized) inverse that is implemented in `FFTRB`. The routine `FFTRF` is based on the real `FFT` in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2TRF/DF2TRF`. The reference is:

```
CALL F2TRF (N, SEQ, COEF, WFFTR)
```

The additional argument is

**WFFTR** — Array of length  $2N + 15$  initialized by `FFTRI`. (Input)  
The initialization depends on  $N$ .

If the Intel® Math Kernel Library or IBM Engineering and Scientific Subroutine Library is used, `WFFTR` is not referenced.

2. The routine `FFTRF` is most efficient when `N` is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.
4. If `FFTRF/FFTRB` is used repeatedly with the same value of `N`, then call `FFTRI` followed by repeated calls to `F2TRF/F2TRB`. This is more efficient than repeated calls to `FFTRF/FFTRB`.

If the Intel® Math Kernel Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTRI` are not used. In this case, there is no need to call `FFTRI`.

## Example

In this example, a pure cosine wave is used as a data vector, and its Fourier series is recovered. The Fourier series is a vector with all components zero except at the appropriate frequency where it has an  $N$ .

```

USE FFTRF_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, NOUT
REAL COEF(N), COS, FLOAT, TWOPI, SEQ(N)
INTRINSIC COS, FLOAT
TWOPI = CONST('PI')
!
TWOPI = 2.0*TWOPI
!
CALL UMACH (2, NOUT) Get output unit number
!
! This loop fills out the data vector
! with a pure exponential signal
DO 10 I=1, N
 SEQ(I) = COS(FLOAT(I-1)*TWOPI/FLOAT(N))
10 CONTINUE
!
CALL FFTRF (N, SEQ, COEF) Compute the Fourier transform of SEQ
!
! Print results
WRITE (NOUT,99998)
99998 FORMAT (9X, 'INDEX', 5X, 'SEQ', 6X, 'COEF')
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
END

```

## Output

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.62  | 3.50 |
| 3     | -0.22 | 0.00 |
| 4     | -0.90 | 0.00 |
| 5     | -0.90 | 0.00 |
| 6     | -0.22 | 0.00 |
| 7     | 0.62  | 0.00 |

---

## FFTRB



Computes the real periodic sequence from its Fourier coefficients.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*COEF* — Array of length *N* containing the Fourier coefficients. (Input)

*SEQ* — Array of length *N* containing the periodic sequence. (Output)

### FORTRAN 90 Interface

Generic:    CALL FFTRB (N, COEF, SEQ [, ...])

Specific:   The specific interface names are S\_FFTRB and D\_FFTRB.

### FORTRAN 77 Interface

Single:     CALL FFTRB (N, COEF, SEQ)

Double:     The double precision name is DFFTRB.

### Description

The routine FFTRB is the unnormalized inverse of the routine FFTRF. This routine computes the discrete inverse Fourier transform of a real vector of size *N*. It uses the Intel<sup>®</sup> Math Kernel Library or IBM Engineering and Scientific Subroutine Library for the computation, if available.

Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *c* = COEF, FFTRB returns in *s* = SEQ, if *N* is even:

$$s_m = c_1 + (-1)^{(m-1)} c_N + 2 \sum_{n=2}^{N/2} c_{2n-2} \cos \frac{[(n-1)(m-1)2\pi]}{N} - 2 \sum_{n=2}^{N/2} c_{2n-1} \sin \frac{[(n-1)(m-1)2\pi]}{N}$$

If  $N$  is odd:

$$s_m = c_1 + 2 \sum_{n=2}^{(N+1)/2} c_{2n-2} \cos \frac{[(n-1)(m-1)2\pi]}{N} - 2 \sum_{n=2}^{(N+1)/2} c_{2n-1} \sin \frac{[(n-1)(m-1)2\pi]}{N}$$

The routine `FFTRB` is based on the inverse real FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2TRB/DF2TRB`. The reference is:

```
CALL F2TRB (N, COEF, SEQ, WFFTR)
```

The additional argument is

**WFFTR** — Array of length  $2N + 15$  initialized by `FFTRI`. (Input)  
The initialization depends on  $N$ .

If the Intel<sup>®</sup> Math Kernel Library or IBM Engineering and Scientific Subroutine Library is used, `WFFTR` is not referenced.

2. The routine `FFTRB` is most efficient when  $N$  is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.
4. If `FFTRF/FFTRB` is used repeatedly with the same value of  $N$ , then call `FFTRI` followed by repeated calls to `F2TRF/F2TRB`. This is more efficient than repeated calls to `FFTRF/FFTRB`.

If the Intel<sup>®</sup> Math Kernel Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTRI` are not used. In this case, there is no need to call `FFTRI`.

## Example

We compute the forward real FFT followed by the inverse operation. In this example, we first compute the Fourier transform

$$\hat{x} = \text{COEF}$$

of the vector  $x$ , where  $x_j = (-1)^j$  for  $j = 1$  to  $N$ . This vector

$$\hat{x}$$

is now input into `FFTRB` with the resulting output  $s = Nx$ , that is,  $s_j = (-1)^j N$  for  $j = 1$  to  $N$ .

```

 USE FFTRB_INT
 USE CONST_INT
 USE FFTRF_INT
 USE UMACH_INT

 IMPLICIT NONE
 INTEGER N
 PARAMETER (N=7)
!
 INTEGER I, NOUT
 REAL COEF(N), FLOAT, SEQ(N), TWOPI, X(N)
 INTRINSIC FLOAT
 TWOPI = CONST('PI')
!
 TWOPI = TWOPI
!
 CALL UMACH (2, NOUT) Get output unit number
!
 CALL UMACH (2, NOUT) Fill the data vector
!
 DO 10 I=1, N
 X(I) = FLOAT((-1)**I)
10 CONTINUE
!
 CALL FFTRF (N, X, COEF) Compute the forward transform of X
!
 WRITE (NOUT,99994) Print results
 WRITE (NOUT,99995)
99994 FORMAT (9X, 'Result after forward transform')
99995 FORMAT (9X, 'INDEX', 5X, 'X', 8X, 'COEF')
 WRITE (NOUT,99996) (I, X(I), COEF(I), I=1,N)
99996 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
!
 CALL FFTRB (N, COEF, SEQ) Compute the backward transform of
! COEF
!
 CALL FFTRB (N, COEF, SEQ) Print results
!
 WRITE (NOUT,99997)
 WRITE (NOUT,99998)
99997 FORMAT (/, 9X, 'Result after backward transform')
99998 FORMAT (9X, 'INDEX', 4X, 'COEF', 6X, 'SEQ')
 WRITE (NOUT,99999) (I, COEF(I), SEQ(I), I=1,N)
99999 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
 END

```

## Output

```

Result after forward transform
INDEX X COEF
 1 -1.00 -1.00

```

|   |       |       |
|---|-------|-------|
| 2 | 1.00  | -1.00 |
| 3 | -1.00 | -0.48 |
| 4 | 1.00  | -1.00 |
| 5 | -1.00 | -1.25 |
| 6 | 1.00  | -1.00 |
| 7 | -1.00 | -4.38 |

Result after backward transform

| INDEX | COEF  | SEQ   |
|-------|-------|-------|
| 1     | -1.00 | -7.00 |
| 2     | -1.00 | 7.00  |
| 3     | -0.48 | -7.00 |
| 4     | -1.00 | 7.00  |
| 5     | -1.25 | -7.00 |
| 6     | -1.00 | 7.00  |
| 7     | -4.38 | -7.00 |

---

## FFTRI

Computes parameters needed by `FFTRF` and `FFTRB`.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*WFFTR* — Array of length  $2N + 15$  containing parameters needed by `FFTRF` and `FFTRB`.  
(Output)

### FORTRAN 90 Interface

Generic:    `CALL FFTRI (N, WFFTR)`

Specific:   The specific interface names are `S_FFTRI` and `D_FFTRI`.

### FORTRAN 77 Interface

Single:     `CALL FFTRI (N, WFFTR)`

Double:     The double precision name is `DFFFTRI`.

### Description

The routine `FFTRI` initializes the routines `FFTRF` and `FFTRB`. An efficient way to make multiple calls for the same *N* to routine `FFTRF` or `FFTRB`, is to use routine `FFTRI` for initialization. (In this case, replace `FFTRF` or `FFTRB` with `F2TRF` or `F2TRB`, respectively.) The routine `FFTRI` is based on the routine `REFTI` in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the Intel<sup>®</sup> Math Kernel Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTRI` are not used. In this case, there is no need to call `FFTRI`.

## Comments

Different WFFTR arrays are needed for different values of N.

## Example

In this example, we compute three distinct real FFTs by calling FFTRI once and then calling F2TRF three times.

```
USE FFTRI_INT
USE CONST_INT
USE F2TRF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, K, NOUT
REAL COEF(N), COS, FLOAT, TWOPI, WFFTR(29), SEQ(N)
INTRINSIC COS, FLOAT
!
TWOPI = CONST('PI')
TWOPI = 2* TWOPI
!
CALL UMACH (2, NOUT) Get output unit number
!
CALL FFTRI (N, WFFTR) Set the work vector
!
DO 20 K=1, 3
!
! This loop fills out the data vector
! with a pure exponential signal
 DO 10 I=1, N
 SEQ(I) = COS(FLOAT(K*(I-1))*TWOPI/FLOAT(N))
10 CONTINUE
!
! Compute the Fourier transform of SEQ
 CALL F2TRF (N, SEQ, COEF, WFFTR)
!
! Print results
 WRITE (NOUT,99998)
99998 FORMAT (/, 9X, 'INDEX', 5X, 'SEQ', 6X, 'COEF')
 WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999 FORMAT (1X, I11, 5X, F5.2, 5X, F5.2)
!
20 CONTINUE
END
```

## Output

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.62  | 3.50 |
| 3     | -0.22 | 0.00 |
| 4     | -0.90 | 0.00 |
| 5     | -0.90 | 0.00 |

|   |       |      |
|---|-------|------|
| 6 | -0.22 | 0.00 |
| 7 | 0.62  | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | -0.22 | 0.00 |
| 3     | -0.90 | 0.00 |
| 4     | 0.62  | 3.50 |
| 5     | 0.62  | 0.00 |
| 6     | -0.90 | 0.00 |
| 7     | -0.22 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | -0.90 | 0.00 |
| 3     | 0.62  | 0.00 |
| 4     | -0.22 | 0.00 |
| 5     | -0.22 | 0.00 |
| 6     | 0.62  | 3.50 |
| 7     | -0.90 | 0.00 |

---

## FFTCF



Computes the Fourier coefficients of a complex periodic sequence.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*SEQ* — Complex array of length *N* containing the periodic sequence. (Input)

*COEF* — Complex array of length *N* containing the Fourier coefficients. (Output)

### FORTRAN 90 Interface

Generic:    CALL FFTCF (N, SEQ, COEF)

Specific:   The specific interface names are *S\_FFTCF* and *D\_FFTCF*.

### FORTRAN 77 Interface

Single:     CALL FFTCF (N, SEQ, COEF)

Double:     The double precision name is *DFFTCF*.

## Description

The routine `FFTCF` computes the discrete complex Fourier transform of a complex vector of size  $N$ . It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$  is a product of small prime factors. If  $N$  satisfies this condition, then the computational effort is proportional to  $N \log N$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N$ -vector  $x$ , `FFTCF` returns in  $c = \text{COEF}$

$$c_m = \sum_{n=1}^N x_n e^{-2\pi i(n-1)(m-1)/N}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$\sqrt{NS}$$

Finally, note that we can invert the Fourier transform as follows:

$$x_n = \frac{1}{N} \sum_{m=1}^N c_m e^{2\pi i(m-1)(n-1)/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. An unnormalized inverse is implemented in `FFTCB`. `FFTCF` is based on the complex FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2TCF/DF2TCF`. The reference is:

```
CALL F2TCF (N, SEQ, COEF, WFFTC, CPY)
```

The additional arguments are as follows:

**WFFTC** — Real array of length  $4 * N + 15$  initialized by `FFTCI`. The initialization depends on  $N$ . (Input)

**CPY** — Real array of length  $2 * N$ . (Workspace)

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, `WFFTC` and `CPY` are not referenced.

2. The routine `FFTCF` is most efficient when  $N$  is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.

4. If FFTCF/FFTCB is used repeatedly with the same value of  $N$ , then call FFTCI followed by repeated calls to F2TCF/F2TCB. This is more efficient than repeated calls to FFTCF/FFTCB.

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by FFTCI are not used. In this case, there is no need to call FFTCI.

## Example

In this example, we input a pure exponential data vector and recover its Fourier series, which is a vector with all components zero except at the appropriate frequency where it has an  $N$ . Notice that the norm of the input vector is

$$\sqrt{N}$$

but the norm of the output vector is  $N$ .

```

USE FFTCF_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)
!
INTEGER I, NOUT
REAL TWOPI
COMPLEX C, CEXP, COEF(N), H, SEQ(N)
INTRINSIC CEXP
!
C = (0.,1.)
TWOPI = CONST('PI')
TWOPI = 2.0 * TWOPI
!
H = (TWOPI*C/N)*3. Here we compute (2*pi*i/N)*3.
!
! This loop fills out the data vector
! with a pure exponential signal of
! frequency 3.
DO 10 I=1, N
 SEQ(I) = CEXP((I-1)*H)
10 CONTINUE
!
CALL FFTCF (N, SEQ, COEF) Compute the Fourier transform of SEQ
!
! Get output unit number and print
! results
CALL UMACH (2, NOUT)
WRITE (NOUT,99998)
99998 FORMAT (9X, 'INDEX', 8X, 'SEQ', 15X, 'COEF')
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99999 FORMAT (1X, I11, 5X, '(', F5.2, ', ', F5.2, ')', &
 5X, '(', F5.2, ', ', F5.2, ')')
END

```

## Output

| INDEX | SEQ            | COEF          |
|-------|----------------|---------------|
| 1     | ( 1.00, 0.00)  | ( 0.00, 0.00) |
| 2     | (-0.90, 0.43)  | ( 0.00, 0.00) |
| 3     | ( 0.62, -0.78) | ( 0.00, 0.00) |
| 4     | (-0.22, 0.97)  | ( 7.00, 0.00) |
| 5     | (-0.22, -0.97) | ( 0.00, 0.00) |
| 6     | ( 0.62, 0.78)  | ( 0.00, 0.00) |
| 7     | (-0.90, -0.43) | ( 0.00, 0.00) |

---

## FFTCB



Computes the complex periodic sequence from its Fourier coefficients.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*COEF* — Complex array of length *N* containing the Fourier coefficients. (Input)

*SEQ* — Complex array of length *N* containing the periodic sequence. (Output)

### FORTRAN 90 Interface

Generic:     CALL FFTCB (N, COEF, SEQ)

Specific:    The specific interface names are S\_FFCTCB and D\_FFCTCB.

### FORTRAN 77 Interface

Single:     CALL FFTCB (N, COEF, SEQ)

Double:     The double precision name is DFFTCB.

### Description

The routine FFTCB computes the inverse discrete complex Fourier transform of a complex vector of size *N*. It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to  $N \log N$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N$ -vector  $c = \text{COEF}$ , `FFTCB` returns in  $s = \text{SEQ}$

$$s_m = \sum_{n=1}^N c_n e^{2\pi i(n-1)(m-1)/N}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$\sqrt{N}S$$

Finally, note that we can invert the inverse Fourier transform as follows:

$$c_n = \frac{1}{N} \sum_{m=1}^N s_m e^{-2\pi i(n-1)(m-1)/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. `FFTCB` is based on the complex inverse FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2TCB/DF2TCB`. The reference is:

```
CALL F2TCB (N, COEF, SEQ, WFFTC, CPY)
```

The additional arguments are as follows:

**WFFTC** — Real array of length  $4 * N + 15$  initialized by `FFTCI`. The initialization depends on `N`. (Input)

**CPY** — Real array of length  $2 * N$ . (Workspace)

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, `WFFTC` and `CPY` are not referenced.

2. The routine `FFTCB` is most efficient when `N` is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.
4. If `FFTCF/FFTCB` is used repeatedly with the same value of `N`; then call `FFTCI` followed by repeated calls to `F2TCF/F2TCB`. This is more efficient than repeated calls to `FFTCF/FFTCB`.

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTCI` are not used. In this case, there is no need to call `FFTCI`.

## Example

In this example, we first compute the Fourier transform of the vector  $x$ , where  $x_j = j$  for  $j = 1$  to  $N$ . Note that the norm of  $x$  is  $(N[N+1][2N+1]/6)^{1/2}$ , and hence, the norm of the transformed vector

$$\hat{x} = c$$

is  $N([N+1][2N+1]/6)^{1/2}$ . The vector

$$\hat{x}$$

is used as input into FFTCB with the resulting output  $s = Nx$ , that is,  $s_j = jN$ , for  $j = 1$  to  $N$ .

```
USE FFTCB_INT
USE FFTCF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, NOUT
COMPLEX CMLPX, SEQ(N), COEF(N), X(N)
INTRINSIC CMLPX

! This loop fills out the data vector
! with X(I)=I, I=1,N
DO 10 I=1, N
 X(I) = CMLPX(I,0)
10 CONTINUE

! Compute the forward transform of X
CALL FFTCF (N, X, COEF)

! Compute the backward transform of
! COEF
CALL FFTCB (N, COEF, SEQ)

! Get output unit number
CALL UMACH (2, NOUT)

! Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I, X(I), COEF(I), SEQ(I), I=1,N)
99998 FORMAT (5X, 'INDEX', 9X, 'INPUT', 9X, 'FORWARD TRANSFORM', 3X, &
 'BACKWARD TRANSFORM')
99999 FORMAT (1X, I7, 7X, '(',F5.2,',',F5.2,')', &
 7X, '(',F5.2,',',F5.2,')', &
 7X, '(',F5.2,',',F5.2,')')

END
```

## Output

| INDEX | INPUT         | FORWARD TRANSFORM | BACKWARD TRANSFORM |
|-------|---------------|-------------------|--------------------|
| 1     | ( 1.00, 0.00) | (28.00, 0.00)     | ( 7.00, 0.00)      |
| 2     | ( 2.00, 0.00) | (-3.50, 7.27)     | (14.00, 0.00)      |
| 3     | ( 3.00, 0.00) | (-3.50, 2.79)     | (21.00, 0.00)      |
| 4     | ( 4.00, 0.00) | (-3.50, 0.80)     | (28.00, 0.00)      |
| 5     | ( 5.00, 0.00) | (-3.50,-0.80)     | (35.00, 0.00)      |

|   |               |               |               |
|---|---------------|---------------|---------------|
| 6 | ( 6.00, 0.00) | (-3.50,-2.79) | (42.00, 0.00) |
| 7 | ( 7.00, 0.00) | (-3.50,-7.27) | (49.00, 0.00) |

---

## FFTCI

Computes parameters needed by FFTCF and FFTCB.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*WFFTC* — Array of length  $4N + 15$  containing parameters needed by FFTCF and FFTCB.  
(Output)

### FORTRAN 90 Interface

Generic:    CALL FFTCI (N,WFFTC)

Specific:    The specific interface names are S\_FFTCI and D\_FFTCI.

### FORTRAN 77 Interface

Single:     CALL FFTCI (N,WFFTC)

Double:     The double precision name is DFFTCI.

### Description

The routine FFTCI initializes the routines FFTCF and FFTCB. An efficient way to make multiple calls for the same *N* to IMSL routine FFTCF or FFTCB is to use routine FFTCI for initialization. (In this case, replace FFTCF or FFTCB with F2TCF or F2TCB, respectively.) The routine FFTCI is based on the routine CFFTI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by FFTCI are not used. In this case, there is no need to call FFTCI.

### Comments

Different WFFTC arrays are needed for different values of *N*.

### Example

In this example, we compute a two-dimensional complex FFT by making one call to FFTCI followed by  $2N$  calls to F2TCF.

```
USE FFTCI_INT
USE CONST_INT
USE F2TCF_INT
```

```

USE UMACH_INT

IMPLICIT NONE

! SPECIFICATIONS FOR PARAMETERS
INTEGER N
PARAMETER (N=4)

!
INTEGER I, IR, IS, J, NOUT
REAL FLOAT, TWOPI, WFFTC(35), CPY(2*N)
COMPLEX CEXP, CMPLX, COEF(N,N), H, SEQ(N,N), TEMP
INTRINSIC CEXP, CMPLX, FLOAT

!
TWOPI = CONST('PI')
TWOPI = 2*TWOPI
IR = 3
IS = 1

! Here we compute e**(2*pi*i/N)
TEMP = CMPLX(0.0, TWOPI/FLOAT(N))
H = CEXP(TEMP)

! Fill SEQ with data
DO 20 I=1, N
 DO 10 J=1, N
 SEQ(I,J) = H**((I-1)*(IR-1)+(J-1)*(IS-1))
10 CONTINUE
20 CONTINUE

! Print out SEQ
! Get output unit number
CALL UMACH (2, NOUT)
WRITE (NOUT,99997)
DO 30 I=1, N
 WRITE (NOUT,99998) (SEQ(I,J),J=1,N)
30 CONTINUE

! Set initialization vector
CALL FFTCI (N, WFFTC)

! Transform the columns of SEQ
DO 40 I=1, N
 CALL F2TCF (N, SEQ(1:,I), COEF(1:,I), WFFTC, CPY)
40 CONTINUE

! Take transpose of the result
DO 60 I=1, N
 DO 50 J=I + 1, N
 TEMP = COEF(I,J)
 COEF(I,J) = COEF(J,I)
 COEF(J,I) = TEMP
50 CONTINUE
60 CONTINUE

! Transform the columns of this result
DO 70 I=1, N
 CALL F2TCF (N, COEF(1:,I), SEQ(1:,I), WFFTC, CPY)
70 CONTINUE

! Take transpose of the result
DO 90 I=1, N
 DO 80 J=I + 1, N
 TEMP = SEQ(I,J)
 SEQ(I,J) = SEQ(J,I)
 SEQ(J,I) = TEMP

```

```

 80 CONTINUE
 90 CONTINUE
!
! Print results
 WRITE (NOUT,99999)
 DO 100 I=1, N
 WRITE (NOUT,99998) (SEQ(I,J),J=1,N)
 100 CONTINUE
!
99997 FORMAT (1X, 'The input matrix is below')
99998 FORMAT (1X, 4(' (',F5.2,',',F5.2,')'))
99999 FORMAT (/, 1X, 'Result of two-dimensional transform')
 END

```

## Output

```

The input matrix is below
(1.00, 0.00) (1.00, 0.00) (1.00, 0.00) (1.00, 0.00)
(-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00)
(1.00, 0.00) (1.00, 0.00) (1.00, 0.00) (1.00, 0.00)
(-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00) (-1.00, 0.00)

Result of two-dimensional transform
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(16.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)

```

---

# FSINT

Computes the discrete Fourier sine transformation of an odd sequence.

## Required Arguments

*N* — Length of the sequence to be transformed. It must be greater than 1. (Input)

*SEQ* — Array of length *N* containing the sequence to be transformed. (Input)

*COEF* — Array of length *N* + 1 containing the transformed sequence. (Output)

## FORTRAN 90 Interface

Generic:    CALL FSINT (N, SEQ, COEF)

Specific:   The specific interface names are S\_FSINT and D\_FSINT.

## FORTRAN 77 Interface

Single:     CALL FSINT (N, SEQ, COEF)

Double:     The double precision name is DFSINT.

## Description

The routine `FSINT` computes the discrete Fourier sine transform of a real vector of size  $N$ . The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N + 1$  is a product of small prime factors. If  $N$  satisfies this condition, then the computational effort is proportional to  $N \log N$ .

Specifically, given an  $N$ -vector  $s = \text{SEQ}$ , `FSINT` returns in  $c = \text{COEF}$

$$c_m = 2 \sum_{n=1}^N s_n \sin\left(\frac{mn\pi}{N+1}\right)$$

Finally, note that the Fourier sine transform is its own (unnormalized) inverse. The routine `FSINT` is based on the sine FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2INT/DF2INT`. The reference is:

```
CALL F2INT (N, SEQ, COEF, WFSIN)
```

The additional argument is:

**WFSIN** — Array of length `INT(2.5 * N + 15)` initialized by `FSINI`. The initialization depends on `N`. (Input)

2. The routine `FSINT` is most efficient when  $N + 1$  is the product of small primes.
3. The routine `FSINT` is its own (unnormalized) inverse. Applying `FSINT` twice will reproduce the original sequence multiplied by  $2 * (N + 1)$ .
4. The arrays `COEF` and `SEQ` may be the same, if `SEQ` is also dimensioned at least  $N + 1$ .
5. `COEF(N + 1)` is needed as workspace.
6. If `FSINT` is used repeatedly with the same value of `N`, then call `FSINI` followed by repeated calls to `F2INT`. This is more efficient than repeated calls to `FSINT`.

## Example

In this example, we input a pure sine wave as a data vector and recover its Fourier sine series, which is a vector with all components zero except at the appropriate frequency it has an  $N$ .

```
USE FSINT_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
```

```

 INTEGER N
 PARAMETER (N=7)
!
 INTEGER I, NOUT
 REAL COEF(N+1), FLOAT, PI, SIN, SEQ(N)
 INTRINSIC FLOAT, SIN
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Fill the data vector SEQ
! with a pure sine wave
 PI = CONST('PI')
 DO 10 I=1, N
 SEQ(I) = SIN(FLOAT(I)*PI/FLOAT(N+1))
10 CONTINUE
!
! Compute the transform of SEQ
 CALL FSINT (N, SEQ, COEF)
!
! Print results
 WRITE (NOUT,99998)
 WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
 END

```

## Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 0.38 | 8.00 |
| 2     | 0.71 | 0.00 |
| 3     | 0.92 | 0.00 |
| 4     | 1.00 | 0.00 |
| 5     | 0.92 | 0.00 |
| 6     | 0.71 | 0.00 |
| 7     | 0.38 | 0.00 |

---

## FSINI

Computes parameters needed by FSINT.

### Required Arguments

*N* — Length of the sequence to be transformed. *N* must be greater than 1. (Input)

*WFSIN* — Array of length  $\text{INT}(2.5 * N + 15)$  containing parameters needed by FSINT.  
(Output)

### FORTRAN 90 Interface

Generic:    CALL FSINI (N,WFSIN)

Specific:   The specific interface names are *S\_FSINI* and *D\_FSINI*.

## FORTRAN 77 Interface

Single:      CALL FSINI (N,WFSIN)

Double:      The double precision name is DFSINI.

## Description

The routine FSINI initializes the routine FSINT. An efficient way to make multiple calls for the same  $N$  to IMSL routine FSINT, is to use routine FSINI for initialization. (In this case, replace FSINT with F2INT.) The routine FSINI is based on the routine SINTI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

Different WFSIN arrays are needed for different values of  $N$ .

## Example

In this example, we compute three distinct sine FFTs by calling FSINI once and then calling F2INT three times.

```
 USE FSINI_INT
 USE UMACH_INT
 USE CONST_INT
 USE F2INT_INT

 IMPLICIT NONE
 INTEGER N
 PARAMETER (N=7)

!
 INTEGER I, K, NOUT
 REAL COEF(N+1), FLOAT, PI, SIN, WFSIN(32), SEQ(N)
 INTRINSIC FLOAT, SIN
!
 CALL UMACH (2, NOUT) Get output unit number
!
 CALL FSINI (N, WFSIN) Initialize the work vector WFSIN
!
! Different frequencies of the same
! wave will be transformed
 DO 20 K=1, 3
!
! Fill the data vector SEQ
! with a pure sine wave
 PI = CONST('PI')
 DO 10 I=1, N
 SEQ(I) = SIN(FLOAT(K*I)*PI/FLOAT(N+1))
10 CONTINUE
!
! Compute the transform of SEQ
 CALL F2INT (N, SEQ, COEF, WFSIN)
!
! Print results
 WRITE (NOUT,99998)
 WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
20 CONTINUE
```

```

99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
END

```

## Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 0.38 | 8.00 |
| 2     | 0.71 | 0.00 |
| 3     | 0.92 | 0.00 |
| 4     | 1.00 | 0.00 |
| 5     | 0.92 | 0.00 |
| 6     | 0.71 | 0.00 |
| 7     | 0.38 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 0.71  | 0.00 |
| 2     | 1.00  | 8.00 |
| 3     | 0.71  | 0.00 |
| 4     | 0.00  | 0.00 |
| 5     | -0.71 | 0.00 |
| 6     | -1.00 | 0.00 |
| 7     | -0.71 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 0.92  | 0.00 |
| 2     | 0.71  | 0.00 |
| 3     | -0.38 | 8.00 |
| 4     | -1.00 | 0.00 |
| 5     | -0.38 | 0.00 |
| 6     | 0.71  | 0.00 |
| 7     | 0.92  | 0.00 |

---

## FCOST

Computes the discrete Fourier cosine transformation of an even sequence.

### Required Arguments

*N* — Length of the sequence to be transformed. It must be greater than 1. (Input)

*SEQ* — Array of length *N* containing the sequence to be transformed. (Input)

*COEF* — Array of length *N* containing the transformed sequence. (Output)

### FORTRAN 90 Interface

Generic:    CALL FCOST (N, SEQ, COEF)

Specific:   The specific interface names are S\_FCOST and D\_FCOST.

## FORTRAN 77 Interface

Single:      CALL FCOST (N, SEQ, COEF)

Double:     The double precision name is DFCOST.

## Description

The routine FCOST computes the discrete Fourier cosine transform of a real vector of size  $N$ . It uses the IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N - 1$  is a product of small prime factors. If  $N$  satisfies this condition, then the computational effort is proportional to  $N \log N$ .

Specifically, given an  $N$ -vector  $s = \text{SEQ}$ , FCOST returns in  $c = \text{COEF}$

$$c_m = 2 \sum_{n=2}^{N-1} s_n \cos \left[ \frac{(m-1)(n-1)\pi}{N-1} \right] + s_1 + s_N (-1)^{(m-1)}$$

Finally, note that the Fourier cosine transform is its own (unnormalized) inverse. Two applications of FCOST to a vector  $s$  produces  $(2N - 2)s$ . The routine FCOST is based on the cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of F2OST/DF2OST. The reference is:

```
CALL F2OST (N, SEQ, COEF, WFCOS)
```

The additional argument is

**WFCOS** — Array of length  $3 * N + 15$  initialized by FCOSI. The initialization depends on  $N$ . (Input)

If the IBM Engineering and Scientific Subroutine Library is used, WFCOS is not referenced.

2. The routine FCOST is most efficient when  $N - 1$  is the product of small primes.
3. The routine FCOST is its own (unnormalized) inverse. Applying FCOST twice will reproduce the original sequence multiplied by  $2 * (N - 1)$ .
4. The arrays COEF and SEQ may be the same.
5. If FCOST is used repeatedly with the same value of  $N$ , then call FCOSI followed by repeated calls to F2OST. This is more efficient than repeated calls to FCOST.

If the IBM Engineering and Scientific Subroutine Library is used, parameters computed by FCOSI are not used. In this case, there is no need to call FCOSI.

## Example

In this example, we input a pure cosine wave as a data vector and recover its Fourier cosine series, which is a vector with all components zero except at the appropriate frequency it has an  $N - 1$ .

```

USE FCOST_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)
!
INTEGER I, NOUT
REAL COEF(N), COS, FLOAT, PI, SEQ(N)
INTRINSIC COS, FLOAT
!
CALL UMACH (2, NOUT)
!
! Fill the data vector SEQ
! with a pure cosine wave
PI = CONST('PI')
DO 10 I=1, N
 SEQ(I) = COS(FLOAT(I-1)*PI/FLOAT(N-1))
10 CONTINUE
!
! Compute the transform of SEQ
CALL FCOST (N, SEQ, COEF)
!
! Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
END

```

## Output

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.87  | 6.00 |
| 3     | 0.50  | 0.00 |
| 4     | 0.00  | 0.00 |
| 5     | -0.50 | 0.00 |
| 6     | -0.87 | 0.00 |
| 7     | -1.00 | 0.00 |

---

## FCOSI

Computes parameters needed by FCOST.

### Required Arguments

$N$  — Length of the sequence to be transformed.  $N$  must be greater than 1. (Input)

*WFCOS* — Array of length  $3N + 15$  containing parameters needed by *FCOST*. (Output)

## **FORTRAN 90 Interface**

Generic:    CALL *FCOSI* (N, *WFCOS*)

Specific:   The specific interface names are *S\_FCOSI* and *D\_FCOSI*.

## **FORTRAN 77 Interface**

Single:     CALL *FCOSI* (N, *WFCOS*)

Double:     The double precision name is *DFCOSI*.

## **Description**

The routine *FCOSI* initializes the routine *FCOST*. An efficient way to make multiple calls for the same *N* to IMSL routine *FCOST* is to use routine *FCOSI* for initialization. (In this case, replace *FCOST* with *F2OST*.) The routine *FCOSI* is based on the routine *COSTI* in *FFTPACK*. The package *FFTPACK* was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the IBM Engineering and Scientific Subroutine Library is used, parameters computed by *FCOSI* are not used. In this case, there is no need to call *FCOSI*.

## **Comments**

Different *WFCOS* arrays are needed for different values of *N*.

## **Example**

In this example, we compute three distinct cosine FFTs by calling *FCOSI* once and then calling *F2OST* three times.

```
USE FCOSI_INT
USE CONST_INT
USE F2OST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, K, NOUT
REAL COEF(N), COS, FLOAT, PI, WFCOS(36), SEQ(N)
INTRINSIC COS, FLOAT
!
CALL UMACH (2, NOUT) Get output unit number
!
CALL FCOSI (N, WFCOS) Initialize the work vector WFCOS
!
! Different frequencies of the same
! wave will be transformed
```

```

 PI = CONST('PI')
 DO 20 K=1, 3
!
! Fill the data vector SEQ
! with a pure cosine wave
 DO 10 I=1, N
 SEQ(I) = COS(FLOAT(K*(I-1))*PI/FLOAT(N-1))
 10 CONTINUE
!
! Compute the transform of SEQ
 CALL F2OST (N, SEQ, COEF, WFCOS)
!
! Print results
 WRITE (NOUT,99998)
 WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
 20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
 END

```

## Output

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.87  | 6.00 |
| 3     | 0.50  | 0.00 |
| 4     | 0.00  | 0.00 |
| 5     | -0.50 | 0.00 |
| 6     | -0.87 | 0.00 |
| 7     | -1.00 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.50  | 0.00 |
| 3     | -0.50 | 6.00 |
| 4     | -1.00 | 0.00 |
| 5     | -0.50 | 0.00 |
| 6     | 0.50  | 0.00 |
| 7     | 1.00  | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.00  | 0.00 |
| 3     | -1.00 | 0.00 |
| 4     | 0.00  | 6.00 |
| 5     | 1.00  | 0.00 |
| 6     | 0.00  | 0.00 |
| 7     | -1.00 | 0.00 |

---

## QSINF

Computes the coefficients of the sine Fourier transform with only odd wave numbers.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*SEQ* — Array of length *N* containing the sequence. (Input)

*COEF* — Array of length  $N$  containing the Fourier coefficients. (Output)

### **FORTRAN 90 Interface**

Generic:     CALL QSINF (N, SEQ, COEF)

Specific:    The specific interface names are S\_QSINF and D\_QSINF.

### **FORTRAN 77 Interface**

Single:      CALL QSINF (N, SEQ, COEF)

Double:      The double precision name is DQSINF.

### **Description**

The routine QSINF computes the discrete Fourier quarter sine transform of a real vector of size  $N$ . The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$  is a product of small prime factors. If  $N$  satisfies this condition, then the computational effort is proportional to  $N \log N$ .

Specifically, given an  $N$ -vector  $s = \text{SEQ}$ , QSINF returns in  $c = \text{COEF}$

$$c_m = 2 \sum_{n=1}^{N-1} s_n \sin \left[ \frac{(2m-1)n\pi}{2N} \right] + s_N (-1)^{m-1}$$

Finally, note that the Fourier quarter sine transform has an (unnormalized) inverse, which is implemented in the IMSL routine QSINB. The routine QSINF is based on the quarter sine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

### **Comments**

1.    Workspace may be explicitly provided, if desired, by use of Q2INF/DQ2INF. The reference is:

```
CALL Q2INF (N, SEQ, COEF, WQSIN)
```

The additional argument is:

*WQSIN* — Array of length  $3 * N + 15$  initialized by QSINI. The initialization depends on  $N$ . (Input)

2.    The routine QSINF is most efficient when  $N$  is the product of small primes.
3.    The arrays COEF and SEQ may be the same.

- If `QSINF/QSINB` is used repeatedly with the same value of `N`, then call `QSINI` followed by repeated calls to `Q2INF/Q2INB`. This is more efficient than repeated calls to `QSINF/QSINB`.

### Example

In this example, we input a pure quarter sine wave as a data vector and recover its Fourier quarter sine series.

```

USE QSINF_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, NOUT
REAL COEF(N), FLOAT, PI, SIN, SEQ(N)
INTRINSIC FLOAT, SIN
!
CALL UMACH (2, NOUT) Get output unit number
!
! Fill the data vector SEQ
! with a pure sine wave
PI = CONST('PI')
DO 10 I=1, N
 SEQ(I) = SIN(FLOAT(I)*(PI/2.0)/FLOAT(N))
10 CONTINUE
!
CALL QSINF (N, SEQ, COEF) Compute the transform of SEQ
!
! Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
END

```

### Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 0.22 | 7.00 |
| 2     | 0.43 | 0.00 |
| 3     | 0.62 | 0.00 |
| 4     | 0.78 | 0.00 |
| 5     | 0.90 | 0.00 |
| 6     | 0.97 | 0.00 |
| 7     | 1.00 | 0.00 |

---

## QSINB

Computes a sequence from its sine Fourier coefficients with only odd wave numbers.

## Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*COEF* — Array of length *N* containing the Fourier coefficients. (Input)

*SEQ* — Array of length *N* containing the sequence. (Output)

## FORTRAN 90 Interface

Generic:    CALL QSINB (N, COEF, SEQ)

Specific:    The specific interface names are S\_QSINB and D\_QSINB.

## FORTRAN 77 Interface

Single:     CALL QSINB (N, COEF, SEQ)

Double:     The double precision name is DQSINB.

## Description

The routine QSINB computes the discrete (unnormalized) inverse Fourier quarter sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *c* = COEF, QSINB returns in *s* = SEQ

$$s_m = 4 \sum_{n=1}^N c_n \sin \left( \frac{(2n-1)m\pi}{2N} \right)$$

Furthermore, a vector *x* of length *N* that is first transformed by QSINF and then by QSINB will be returned by QSINB as 4*Nx*. The routine QSINB is based on the inverse quarter sine FFT in FFTPACK which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of Q2INB/DQ2INB. The reference is:

CALL Q2INB (N, SEQ, COEF, WQSIN)

The additional argument is:

**WQSIN** — ray of length 3 \* *N* + 15 initialized by QSINI. The initialization depends on *N*.(Input)

2. The routine QSINB is most efficient when N is the product of small primes.
3. The arrays COEF and SEQ may be the same.
4. If QSINF/QSINB is used repeatedly with the same value of N, then call QSINI followed by repeated calls to Q2INF/Q2INB. This is more efficient than repeated calls to QSINF/QSINB.

## Example

In this example, we first compute the quarter wave sine Fourier transform  $c$  of the vector  $x$  where  $x_n = n$  for  $n = 1$  to  $N$ . We then compute the inverse quarter wave Fourier transform of  $c$  which is  $4Nx = s$ .

```

USE QSINB_INT
USE QSINF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, NOUT
REAL SEQ(N), COEF(N), X(N)
INTRINSIC FLOAT

! Get output unit number
CALL UMACH (2, NOUT)

! Fill the data vector X
! with X(I) = I, I=1,N
DO 10 I=1, N
 X(I) = FLOAT(I)
10 CONTINUE

! Compute the forward transform of X
CALL QSINF (N, X, COEF)

! Compute the backward transform of W
CALL QSINB (N, COEF, SEQ)

!C Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (X(I), COEF(I), SEQ(I), I=1,N)
99998 FORMAT (5X, 'INPUT', 5X, 'FORWARD TRANSFORM', 3X, 'BACKWARD ', &
 'TRANSFORM')
99999 FORMAT (3X, F6.2, 10X, F6.2, 15X, F6.2)
END

```

## Output

| INPUT | FORWARD TRANSFORM | BACKWARD TRANSFORM |
|-------|-------------------|--------------------|
| 1.00  | 39.88             | 28.00              |
| 2.00  | -4.58             | 56.00              |
| 3.00  | 1.77              | 84.00              |
| 4.00  | -1.00             | 112.00             |
| 5.00  | 0.70              | 140.00             |
| 6.00  | -0.56             | 168.00             |
| 7.00  | 0.51              | 196.00             |

---

# QSINI

Computes parameters needed by `QSINF` and `QSINB`.

## Required Arguments

$N$  — Length of the sequence to be transformed. (Input)

$WQSIN$  — Array of length  $3N + 15$  containing parameters needed by `QSINF` and `QSINB`. (Output)

## FORTRAN 90 Interface

Generic:     `CALL QSINI (N, WQSIN)`

Specific:    The specific interface names are `S_QSINI` and `D_QSINI`.

## FORTRAN 77 Interface

Single:      `CALL QSINI (N, WQSIN)`

Double:      The double precision name is `DQSINI`.

## Description

The routine `QSINI` initializes the routines `QSINF` and `QSINB`. An efficient way to make multiple calls for the same  $N$  to IMSL routine `QSINF` or `QSINB` is to use routine `QSINI` for initialization. (In this case, replace `QSINF` or `QSINB` with `Q2INF` or `Q2INB`, respectively.) The routine `QSINI` is based on the routine `SINQI` in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

Different `WQSIN` arrays are needed for different values of  $N$ .

## Example

In this example, we compute three distinct quarter sine transforms by calling `QSINI` once and then calling `Q2INF` three times.

```
USE QSINI_INT
USE CONST_INT
USE Q2INF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)
!
```

```

 INTEGER I, K, NOUT
 REAL COEF(N), FLOAT, PI, SIN, WQSIN(36), SEQ(N)
 INTRINSIC FLOAT, SIN
!
! Get output unit number
 CALL UMACH (2, NOUT)
!
! Initialize the work vector WQSIN
 CALL QSINI (N, WQSIN)
!
! Different frequencies of the same
! wave will be transformed
 PI = CONST('PI')
 DO 20 K=1, 3
!
! Fill the data vector SEQ
! with a pure sine wave
 DO 10 I=1, N
 SEQ(I) = SIN(FLOAT((2*K-1)*I)*(PI/2.0)/FLOAT(N))
10 CONTINUE
!
! Compute the transform of SEQ
 CALL Q2INF (N, SEQ, COEF, WQSIN)
!
! Print results
 WRITE (NOUT,99998)
 WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
 END

```

## Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 0.22 | 7.00 |
| 2     | 0.43 | 0.00 |
| 3     | 0.62 | 0.00 |
| 4     | 0.78 | 0.00 |
| 5     | 0.90 | 0.00 |
| 6     | 0.97 | 0.00 |
| 7     | 1.00 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 0.62  | 0.00 |
| 2     | 0.97  | 7.00 |
| 3     | 0.90  | 0.00 |
| 4     | 0.43  | 0.00 |
| 5     | -0.22 | 0.00 |
| 6     | -0.78 | 0.00 |
| 7     | -1.00 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 0.90  | 0.00 |
| 2     | 0.78  | 0.00 |
| 3     | -0.22 | 7.00 |
| 4     | -0.97 | 0.00 |
| 5     | -0.62 | 0.00 |
| 6     | 0.43  | 0.00 |
| 7     | 1.00  | 0.00 |

---

# QCOSF

Computes the coefficients of the cosine Fourier transform with only odd wave numbers.

## Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*SEQ* — Array of length *N* containing the sequence. (Input)

*COEF* — Array of length *N* containing the Fourier coefficients. (Output)

## FORTRAN 90 Interface

Generic:     CALL QCOSF (N, SEQ, COEF [, ...])

Specific:    The specific interface names are S\_QCOSF and D\_QCOSF.

## FORTRAN 77 Interface

Single:      CALL QCOSF (N, SEQ, COEF)

Double:     The double precision name is DQCOSF.

## Description

The routine QCOSF computes the discrete Fourier quarter cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*.

Specifically, given an *N*-vector *s* = SEQ, QCOSF returns in *c* = COEF

$$c_m = s_1 + 2 \sum_{n=2}^N s_n \cos\left(\frac{(2m-1)(n-1)\pi}{2N}\right)$$

Finally, note that the Fourier quarter cosine transform has an (unnormalized) inverse which is implemented in QCOSB. The routine QCOSF is based on the quarter cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of Q2OSF/DQ2OSF. The reference is:

```
CALL Q2OSF (N, SEQ, COEF, WQCOS)
```

The additional argument is:

**WQCOS** — Array of length  $3 * N + 15$  initialized by `QCOSI`. The initialization depends on `N`. (Input)

2. The routine `QCOSF` is most efficient when `N` is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.
4. If `QCOSF/QCOSB` is used repeatedly with the same value of `N`, then call `QCOSI` followed by repeated calls to `Q2OSF/Q2OSB`. This is more efficient than repeated calls to `QCOSF/QCOSB`.

## Example

In this example, we input a pure quarter cosine wave as a data vector and recover its Fourier quarter cosine series.

```
USE QCOSF_INT
USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, NOUT
REAL COEF(N), COS, FLOAT, PI, SEQ(N)
INTRINSIC COS, FLOAT
!
CALL UMACH (2, NOUT) Get output unit number
!
CALL UMACH (2, NOUT) Fill the data vector SEQ
!
! with a pure cosine wave
PI = CONST('PI')
DO 10 I=1, N
 SEQ(I) = COS (FLOAT(I-1) * (PI/2.0) / FLOAT(N))
10 CONTINUE

!
! Compute the transform of SEQ
CALL QCOSF (N, SEQ, COEF)
!
! Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
99998 FORMAT (9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
END
```

## Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 1.00 | 7.00 |
| 2     | 0.97 | 0.00 |
| 3     | 0.90 | 0.00 |
| 4     | 0.78 | 0.00 |

|   |      |      |
|---|------|------|
| 5 | 0.62 | 0.00 |
| 6 | 0.43 | 0.00 |
| 7 | 0.22 | 0.00 |

---

## QCOSB

Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*COEF* — Array of length *N* containing the Fourier coefficients. (Input)

*SEQ* — Array of length *N* containing the sequence. (Output)

### FORTRAN 90 Interface

Generic:    CALL QCOSB (N, COEF, SEQ)

Specific:   The specific interface names are S\_QCOSB and D\_QCOSB.

### FORTRAN 77 Interface

Single:     CALL QCOSB (N, COEF, SEQ)

Double:     The double precision name is DQCOSB.

### Description

The routine QCOSB computes the discrete (unnormalized) inverse Fourier quarter cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log *N*. Specifically, given an *N*-vector *c* = COEF, QCOSB returns in *s* = SEQ

$$s_m = 4 \sum_{n=1}^N c_n \cos \left( \frac{(2n-1)(m-1)\pi}{2N} \right)$$

Furthermore, a vector *x* of length *N* that is first transformed by QCOSF and then by QCOSB will be returned by QCOSB as 4*Nx*. The routine QCOSB is based on the inverse quarter cosine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

### Comments

1.    Workspace may be explicitly provided, if desired, by use of Q2OSB/DQ2OSB. The reference is:

```
CALL Q2OSB (N, COEF, SEQ, WQCOS)
```

The additional argument is:

**WQCOS** — Array of length  $3 * N + 15$  initialized by `QCOSI`. The initialization depends on `N`. (Input)

2. The routine `QCOSB` is most efficient when `N` is the product of small primes.
3. The arrays `COEF` and `SEQ` may be the same.
4. If `QCOSF/QCOSB` is used repeatedly with the same value of `N`, then call `QCOSI` followed by repeated calls to `Q2OSF/Q2OSB`. This is more efficient than repeated calls to `QCOSF/QCOSB`.

### Example

In this example, we first compute the quarter wave cosine Fourier transform  $c$  of the vector  $x$ , where  $x_n = n$  for  $n = 1$  to  $N$ . We then compute the inverse quarter wave Fourier transform of  $c$  which is  $4Nx = s$ .

```

USE QCOSB_INT
USE QCOSF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)
!
INTEGER I, NOUT
REAL FLOAT, SEQ(N), COEF(N), X(N)
INTRINSIC FLOAT
!
CALL UMACH (2, NOUT) Get output unit number
!
! Fill the data vector X
! with X(I) = I, I=1,N
DO 10 I=1, N
 X(I) = FLOAT(I)
10 CONTINUE
!
CALL QCOSF (N, X, COEF) Compute the forward transform of X
!
! Compute the backward transform of
! COEF
CALL QCOSB (N, COEF, SEQ)
!
! Print results
WRITE (NOUT,99998)
DO 20 I=1, N
 WRITE (NOUT,99999) X(I), COEF(I), SEQ(I)
20 CONTINUE
99998 FORMAT (5X, 'INPUT', 5X, 'FORWARD TRANSFORM', 3X, 'BACKWARD ', &
 'TRANSFORM')
99999 FORMAT (3X, F6.2, 10X, F6.2, 15X, F6.2)
END

```

## Output

| INPUT | FORWARD TRANSFORM | BACKWARD TRANSFORM |
|-------|-------------------|--------------------|
| 1.00  | 31.12             | 28.00              |
| 2.00  | -27.45            | 56.00              |
| 3.00  | 10.97             | 84.00              |
| 4.00  | -9.00             | 112.00             |
| 5.00  | 4.33              | 140.00             |
| 6.00  | -3.36             | 168.00             |
| 7.00  | 0.40              | 196.00             |

---

## QCOSI

Computes parameters needed by `QCOSF` and `QCOSB`.

### Required Arguments

*N* — Length of the sequence to be transformed. (Input)

*WQCOS* — Array of length  $3N + 15$  containing parameters needed by `QCOSF` and `QCOSB`. (Output)

### FORTRAN 90 Interface

Generic:     `CALL QCOSI (N, WQCOS)`

Specific:    The specific interface names are `S_QCOSI` and `D_QCOSI`.

### FORTRAN 77 Interface

Single:      `CALL QCOSI (N, WQCOS)`

Double:      The double precision name is `DQCOSI`.

### Description

The routine `QCOSI` initializes the routines `QCOSF` and `QCOSB`. An efficient way to make multiple calls for the same *N* to IMSL routine `QCOSF` or `QCOSB` is to use routine `QCOSI` for initialization. (In this case, replace `QCOSF` or `QCOSB` with `Q2OSF` or `Q2OSB`, respectively.) The routine `QCOSI` is based on the routine `COSQI` in `FFTPACK`, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

### Comments

Different *WQCOS* arrays are needed for different values of *N*.

## Example

In this example, we compute three distinct quarter cosine transforms by calling `QCOSI` once and then calling `Q2OSF` three times.

```
USE QCOSI_INT
USE CONST_INT
USE Q2OSF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=7)

!
INTEGER I, K, NOUT
REAL COEF(N), COS, FLOAT, PI, WQCOS(36), SEQ(N)
INTRINSIC COS, FLOAT

!
CALL UMACH (2, NOUT) Get output unit number
!
CALL Q2OSF (N, WQCOS) Initialize the work vector WQCOS
!
! Different frequencies of the same
! wave will be transformed
PI = CONST('PI')
DO 20 K=1, 3
!
! Fill the data vector SEQ
! with a pure cosine wave
DO 10 I=1, N
 SEQ(I) = COS(FLOAT((2*K-1)*(I-1))*(PI/2.0)/FLOAT(N))
10 CONTINUE
!
! Compute the transform of SEQ
CALL Q2OSF (N, SEQ, COEF, WQCOS)
!
! Print results
WRITE (NOUT,99998)
WRITE (NOUT,99999) (I, SEQ(I), COEF(I), I=1,N)
20 CONTINUE
99998 FORMAT (/, 9X, 'INDEX', 6X, 'SEQ', 7X, 'COEF')
99999 FORMAT (1X, I11, 5X, F6.2, 5X, F6.2)
END
```

## Output

| INDEX | SEQ  | COEF |
|-------|------|------|
| 1     | 1.00 | 7.00 |
| 2     | 0.97 | 0.00 |
| 3     | 0.90 | 0.00 |
| 4     | 0.78 | 0.00 |
| 5     | 0.62 | 0.00 |
| 6     | 0.43 | 0.00 |
| 7     | 0.22 | 0.00 |

| INDEX | SEQ   | COEF |
|-------|-------|------|
| 1     | 1.00  | 0.00 |
| 2     | 0.78  | 7.00 |
| 3     | 0.22  | 0.00 |
| 4     | -0.43 | 0.00 |
| 5     | -0.90 | 0.00 |

|       |       |      |
|-------|-------|------|
| 6     | -0.97 | 0.00 |
| 7     | -0.62 | 0.00 |
| INDEX | SEQ   | COEF |
| 1     | 1.00  | 0.00 |
| 2     | 0.43  | 0.00 |
| 3     | -0.62 | 7.00 |
| 4     | -0.97 | 0.00 |
| 5     | -0.22 | 0.00 |
| 6     | 0.78  | 0.00 |
| 7     | 0.90  | 0.00 |

---

## FFT2D



Computes Fourier coefficients of a complex periodic two-dimensional array.

### Required Arguments

*A* — *NRA* by *NCA* complex matrix containing the periodic data to be transformed. (Input)

*COEF* — *NRA* by *NCA* complex matrix containing the Fourier coefficients of *A*. (Output)

### Optional Arguments

*NRA* — The number of rows of *A*. (Input)  
Default: *NRA* = size (*A*,1).

*NCA* — The number of columns of *A*. (Input)  
Default: *NCA* = size (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = size (*A*,1).

*LDCOEF* — Leading dimension of *COEF* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDCOEF* = size (*COEF*,1).

### FORTRAN 90 Interface

Generic:    CALL FFT2D (*A*, *COEF* [, ...])

Specific:   The specific interface names are *S\_FFT2D* and *D\_FFT2D*.

## FORTRAN 77 Interface

Single:      CALL FFT2D (NRA, NCA, A, LDA, COEF, LDcoef)

Double:     The double precision name is DFFT2D.

## Description

The routine FFT2D computes the discrete complex Fourier transform of a complex two dimensional array of size  $(NRA = N) \times (NCA = M)$ . It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$  and  $M$  are each products of small prime factors. If  $N$  and  $M$  satisfy this condition, then the computational effort is proportional to  $NM \log NM$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N \times M$  array  $a$ , FFT2D returns in  $c = COEF$

$$c_{jk} = \sum_{n=1}^N \sum_{m=1}^M a_{nm} e^{-2\pi i(j-1)(n-1)/N} e^{-2\pi i(k-1)(m-1)/M}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$\sqrt{NMS}$$

Finally, note that an unnormalized inverse is implemented in FFT2B. The routine FFT2D is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of F2T2D/DF2T2D. The reference is:

```
CALL F2T2D (NRA, NCA, A, LDA, COEF, LDcoef, WFF1, WFF2, CWK, CPY)
```

The additional arguments are as follows:

**WFF1** — Real array of length  $4 * NRA + 15$  initialized by FFTCI. The initialization depends on NRA. (Input)

**WFF2** — Real array of length  $4 * NCA + 15$  initialized by FFTCI. The initialization depends on NCA. (Input)

**CWK** — Complex array of length 1. (Workspace)

**CPY** — Real array of length  $2 * \text{MAX}(NRA, NCA)$ . (Workspace)

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, WFF1, WFF2, CWK, and CPY are not referenced.

2. The routine `FFT2D` is most efficient when `NRA` and `NCA` are the product of small primes.
3. The arrays `COEF` and `A` may be the same.
4. If `FFT2D/FFT2B` is used repeatedly, with the same values for `NRA` and `NCA`, then use `FFTCI` to fill `WFF1(N = NRA)` and `WFF2(N = NCA)`. Follow this with repeated calls to `F2T2D/F2T2B`. This is more efficient than repeated calls to `FFT2D/FFT2B`.

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTCI` are not used. In this case, there is no need to call `FFTCI`.

### Example

In this example, we compute the Fourier transform of the pure frequency input for a  $5 \times 4$  array

$$a_{nm} = e^{2\pi i(n-1)2/N} e^{2\pi i(m-1)3/M}$$

for  $1 \leq n \leq 5$  and  $1 \leq m \leq 4$  using the IMSL routine `FFT2D`. The result

$$\hat{a} = c$$

has all zeros except in the (3, 4) position.

```

USE FFT2D_INT
USE CONST_INT
USE WRCRN_INT

IMPLICIT NONE
INTEGER I, IR, IS, J, NCA, NRA
REAL FLOAT, TWOPI
COMPLEX A(5,4), C, CEXP, CMPLX, COEF(5,4), H
CHARACTER TITLE1*26, TITLE2*26
INTRINSIC CEXP, CMPLX, FLOAT
!
TITLE1 = 'The input matrix is below '
TITLE2 = 'The output matrix is below'
NRA = 5
NCA = 4
IR = 3
IS = 4
!
 Fill A with initial data
TWOPI = CONST('PI')
TWOPI = 2.0*TWOPI
C = CMPLX(0.0,1.0)
H = CEXP(TWOPI*C)
DO 10 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = CEXP(TWOPI*C*((FLOAT((I-1)*(IR-1))/FLOAT(NRA) + &
 FLOAT((J-1)*(IS-1))/FLOAT(NCA))))
10 CONTINUE
!
CALL WRCRN (TITLE1, A)

```

```

!
CALL FFT2D (A, COEF)
!
CALL WRCRN (TITLE2, COEF)
!
END

```

## Output

The input matrix is below

|   | 1               | 2               | 3               | 4               |
|---|-----------------|-----------------|-----------------|-----------------|
| 1 | ( 1.000, 0.000) | ( 0.000,-1.000) | (-1.000, 0.000) | ( 0.000, 1.000) |
| 2 | (-0.809, 0.588) | ( 0.588, 0.809) | ( 0.809,-0.588) | (-0.588,-0.809) |
| 3 | ( 0.309,-0.951) | (-0.951,-0.309) | (-0.309, 0.951) | ( 0.951, 0.309) |
| 4 | ( 0.309, 0.951) | ( 0.951,-0.309) | (-0.309,-0.951) | (-0.951, 0.309) |
| 5 | (-0.809,-0.588) | (-0.588, 0.809) | ( 0.809, 0.588) | ( 0.588,-0.809) |

The Output matrix is below

|   | 1             | 2             | 3             | 4              |
|---|---------------|---------------|---------------|----------------|
| 1 | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00)  |
| 2 | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00)  |
| 3 | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 20.00, 0.00) |
| 4 | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00)  |
| 5 | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00) | ( 0.00, 0.00)  |

---

## FFT2B



Computes the inverse Fourier transform of a complex periodic two-dimensional array.

### Required Arguments

**COEF** — NRCOEF by NCCOEF complex array containing the Fourier coefficients to be transformed. (Input)

**A** — NRCOEF by NCCOEF complex array containing the Inverse Fourier coefficients of COEF. (Output)

### Optional Arguments

**NRCOEF** — The number of rows of COEF. (Input)  
 Default: NRCOEF = size (COEF, 1).

**NCCOEF** — The number of columns of COEF. (Input)  
 Default: NCCOEF = size (COEF, 2).

**LDCOEF** — Leading dimension of `COEF` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDCOEF = size (COEF, 1)`.

**LDA** — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDA = size (A, 1)`.

### **FORTRAN 90 Interface**

Generic: `CALL FFT2B (COEF, A [, ...])`

Specific: The specific interface names are `S_FFT2B` and `D_FFT2B`.

### **FORTRAN 77 Interface**

Single: `CALL FFT2B (NRCOEF, NCCOEF, COEF, LDCOEF, A, LDA)`

Double: The double precision name is `DFFT2B`.

### **Description**

The routine `FFT2B` computes the inverse discrete complex Fourier transform of a complex two-dimensional array of size  $(NRCOEF = N) \times (NCCOEF = M)$ . It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$  and  $M$  are both products of small prime factors. If  $N$  and  $M$  satisfy this condition, then the computational effort is proportional to  $NM \log NM$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N \times M$  array  $c = COEF$ , `FFT2B` returns in  $a$

$$a_{jk} = \sum_{n=1}^N \sum_{m=1}^M c_{nm} e^{2\pi i(j-1)(n-1)/N} e^{2\pi i(k-1)(m-1)/M}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$S\sqrt{NM}$$

Finally, note that an unnormalized inverse is implemented in `FFT2D`. The routine `FFT2B` is based on the complex FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of `F2T2B/DF2T2B`. The reference is:

```
CALL F2T2B (NRCOEF, NCCOEF, A, LDA, COEF, LDCOEF, WFF1, WFF2, CWK,
 CPY)
```

The additional arguments are as follows:

**WFF1** — Real array of length  $4 * \text{NRCOEF} + 15$  initialized by `FFTCI`. The initialization depends on `NRCOEF`. (Input)

**WFF2** — Real array of length  $4 * \text{NCCOEF} + 15$  initialized by `FFTCI`. The initialization depends on `NCCOEF`. (Input)

**CWK** — Complex array of length 1. (Workspace)

**CPY** — Real array of length  $2 * \text{MAX}(\text{NRCOEF}, \text{NCCOEF})$ . (Workspace)

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, `WFF1`, `WFF2`, `CWK`, and `CPY` are not referenced.

2. The routine `FFT2B` is most efficient when `NRCOEF` and `NCCOEF` are the product of small primes.
3. The arrays `COEF` and `A` may be the same.
4. If `FFT2D/FFT2B` is used repeatedly, with the same values for `NRCOEF` and `NCCOEF`, then use `FFTCI` to fill `WFF1(N = NRCOEF)` and `WFF2(N = NCCOEF)`. Follow this with repeated calls to `F2T2D/F2T2B`. This is more efficient than repeated calls to `FFT2D/FFT2B`.

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by `FFTCI` are not used. In this case, there is no need to call `FFTCI`.

## Example

In this example, we first compute the Fourier transform of the  $5 \times 4$  array

$$x_{nm} = n + 5(m - 1)$$

for  $1 \leq n \leq 5$  and  $1 \leq m \leq 4$  using the IMSL routine `FFT2D`. The result

$$\hat{x} = c$$

is then inverted by a call to `FFT2B`. Note that the result is an array  $a$  satisfying  $a = (5)(4)x = 20x$ . In general, `FFT2B` is an unnormalized inverse with expansion factor  $NM$ .

```
USE FFT2B_INT
USE FFT2D_INT
USE WRCRN_INT

IMPLICIT NONE
INTEGER M, N, NCA, NRA
COMPLEX CMPLX, X(5,4), A(5,4), COEF(5,4)
```

```

CHARACTER TITLE1*26, TITLE2*26, TITLE3*26
INTRINSIC CMPLX
!
TITLE1 = 'The input matrix is below '
TITLE2 = 'After FFT2D '
TITLE3 = 'After FFT2B '
NRA = 5
NCA = 4
!
 Fill X with initial data
DO 20 N=1, NRA
 DO 10 M=1, NCA
 X(N,M) = CMPLX(FLOAT(N+5*M-5),0.0)
10 CONTINUE
20 CONTINUE
!
CALL WRCRN (TITLE1, X)
!
CALL FFT2D (X, COEF)
!
CALL WRCRN (TITLE2, COEF)
!
CALL FFT2B (COEF, A)
!
CALL WRCRN (TITLE3, A)
!
END

```

## Output

```

 The input matrix is below
 1 2 3 4
1 (1.00, 0.00) (6.00, 0.00) (11.00, 0.00) (16.00, 0.00)
2 (2.00, 0.00) (7.00, 0.00) (12.00, 0.00) (17.00, 0.00)
3 (3.00, 0.00) (8.00, 0.00) (13.00, 0.00) (18.00, 0.00)
4 (4.00, 0.00) (9.00, 0.00) (14.00, 0.00) (19.00, 0.00)
5 (5.00, 0.00) (10.00, 0.00) (15.00, 0.00) (20.00, 0.00)

```

```

 After FFT2D
 1 2 3 4
1 (210.0, 0.0) (-50.0, 50.0) (-50.0, 0.0) (-50.0, -50.0)
2 (-10.0, 13.8) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0)
3 (-10.0, 3.2) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0)
4 (-10.0, -3.2) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0)
5 (-10.0, -13.8) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0)

```

```

 After FFT2B
 1 2 3 4
1 (20.0, 0.0) (120.0, 0.0) (220.0, 0.0) (320.0, 0.0)
2 (40.0, 0.0) (140.0, 0.0) (240.0, 0.0) (340.0, 0.0)
3 (60.0, 0.0) (160.0, 0.0) (260.0, 0.0) (360.0, 0.0)
4 (80.0, 0.0) (180.0, 0.0) (280.0, 0.0) (380.0, 0.0)
5 (100.0, 0.0) (200.0, 0.0) (300.0, 0.0) (400.0, 0.0)

```

---

# FFT3F



Computes Fourier coefficients of a complex periodic three-dimensional array.

## Required Arguments

*A* — Three-dimensional complex matrix containing the data to be transformed. (Input)

*B* — Three-dimensional complex matrix containing the Fourier coefficients of *A*. (Output)  
The matrices *A* and *B* may be the same.

## Optional Arguments

*N1* — Limit on the first subscript of matrices *A* and *B*. (Input)  
Default:  $N1 = \text{size}(A, 1)$

*N2* — Limit on the second subscript of matrices *A* and *B*. (Input)  
Default:  $N2 = \text{size}(A, 2)$

*N3* — Limit on the third subscript of matrices *A* and *B*. (Input)  
Default:  $N3 = \text{size}(A, 3)$

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

*MDA* — Middle dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDA = \text{size}(A, 2)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{size}(B, 1)$ .

*MDB* — Middle dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDB = \text{size}(B, 2)$ .

## FORTRAN 90 Interface

Generic:    CALL FFT3F (A, B [, ...])

Specific:   The specific interface names are `S_FFT3F` and `D_FFT3F`.

## FORTRAN 77 Interface

Single:       CALL FFT3F (N1, N2, N3, A, LDA, MDA, B, LDB, MDB)

Double:       The double precision name is DFFT3F.

## Description

The routine `FFT3F` computes the forward discrete complex Fourier transform of a complex three-dimensional array of size  $(N1 = N) \times (N2 = M) \times (N3 = L)$ . It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$ ,  $M$ , and  $L$  are each products of small prime factors. If  $N$ ,  $M$ , and  $L$  satisfy this condition, then the computational effort is proportional to  $NML \log NML$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N \times M \times L$  array  $a$ , `FFT3F` returns in  $c = \text{COEF}$

$$c_{jkl} = \sum_{n=1}^N \sum_{m=1}^M \sum_{l=1}^L a_{nml} e^{-2\pi i(j-1)(n-1)/N} e^{-2\pi i(k-1)(m-1)/M} e^{-2\pi i(l-1)(l-1)/L}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$\sqrt{NMLS}$$

Finally, note that an unnormalized inverse is implemented in `FFT3B`. The routine `FFT3F` is based on the complex FFT in `FFTPACK`. The package `FFTPACK` was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `F2T3F/DF2T3F`. The reference is:

```
CALL F2T3F (N1, N2, N3, A, LDA, MDA, B, LDB, MDB, WFF1, WFF2, WFF3, CPY)
```

The additional arguments are as follows:

**WFF1** — Real array of length  $4 * N1 + 15$  initialized by `FFTCI`. The initialization depends on  $N1$ . (Input)

**WFF2** — Real array of length  $4 * N2 + 15$  initialized by `FFTCI`. The initialization depends on  $N2$ . (Input)

**WFF3** — Real array of length  $4 * N3 + 15$  initialized by `FFTCI`. The initialization depends on  $N3$ . (Input)

**CPY** — Real array of size  $2 * \text{MAX}(N1, N2, N3)$ . (Workspace)

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, WFFT1, WFF2, WFF3, and CPY are not referenced.

2. The routine FFT3F is most efficient when N1, N2, and N3 are the product of small primes.
3. If FFT3F/FFT3B is used repeatedly with the same values for N1, N2 and N3, then use FFTCI to fill WFF1(N = N1), WFF2(N = N2), and WFF3(N = N3). Follow this with repeated calls to F2T3F/F2T3B. This is more efficient than repeated calls to FFT3F/FFT3B.

If the Intel® Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by FFTCI are not used. In this case, there is no need to call FFTCI.

### Example

In this example, we compute the Fourier transform of the pure frequency input for a  $2 \times 3 \times 4$  array

$$a_{nml} = e^{2\pi i(n-1)l/2} e^{2\pi i(m-1)2/3} e^{2\pi i(l-1)2/4}$$

for  $1 \leq n \leq 2$ ,  $1 \leq m \leq 3$ , and  $1 \leq l \leq 4$  using the IMSL routine FFT3F. The result

$$\hat{a} = c$$

has all zeros except in the (2, 3, 3) position.

```

USE FFT3F_INT
USE UMACH_INT
USE CONST_INT

IMPLICIT NONE
INTEGER LDA, LDB, MDA, MDB, NDA, NDB
PARAMETER (LDA=2, LDB=2, MDA=3, MDB=3, NDA=4, NDB=4)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, J, K, L, M, N, N1, N2, N3, NOUT
REAL PI
COMPLEX A(LDA,MDA,NDA), B(LDB,MDB,NDB), C, H
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC CEXP, CMPLX
COMPLEX CEXP, CMPLX
! SPECIFICATIONS FOR SUBROUTINES
! SPECIFICATIONS FOR FUNCTIONS
! Get output unit number
CALL UMACH (2, NOUT)
PI = CONST('PI')
C = CMPLX(0.0,2.0*PI)
! Set array A
DO 30 N=1, 2
 DO 20 M=1, 3
 DO 10 L=1, 4
 H = C*(N-1)*1/2 + C*(M-1)*2/3 + C*(L-1)*2/4
 A(N,M,L) = CEXP(H)

```

```

10 CONTINUE
20 CONTINUE
30 CONTINUE
!
 CALL FFT3F (A, B)
!
 WRITE (NOUT,99996)
 DO 50 I=1, 2
 WRITE (NOUT,99998) I
 DO 40 J=1, 3
 WRITE (NOUT,99999) (A(I,J,K),K=1,4)
40 CONTINUE
50 CONTINUE
!
 WRITE (NOUT,99997)
 DO 70 I=1, 2
 WRITE (NOUT,99998) I
 DO 60 J=1, 3
 WRITE (NOUT,99999) (B(I,J,K),K=1,4)
60 CONTINUE
70 CONTINUE
!
99996 FORMAT (13X, 'The input for FFT3F is')
99997 FORMAT (/, 13X, 'The results from FFT3F are')
99998 FORMAT (/, ' Face no. ', I1)
99999 FORMAT (1X, 4('(',F6.2,',',F6.2,')',3X))
 END

```

## Output

The input for FFT3F is

```

Face no. 1
(1.00, 0.00) (-1.00, 0.00) (1.00, 0.00) (-1.00, 0.00)
(-0.50, -0.87) (0.50, 0.87) (-0.50, -0.87) (0.50, 0.87)
(-0.50, 0.87) (0.50, -0.87) (-0.50, 0.87) (0.50, -0.87)

```

```

Face no. 2
(-1.00, 0.00) (1.00, 0.00) (-1.00, 0.00) (1.00, 0.00)
(0.50, 0.87) (-0.50, -0.87) (0.50, 0.87) (-0.50, -0.87)
(0.50, -0.87) (-0.50, 0.87) (0.50, -0.87) (-0.50, 0.87)

```

The results from FFT3F are

```

Face no. 1
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)

```

```

Face no. 2
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (24.00, 0.00) (0.00, 0.00)

```

---

# FFT3B



Computes the inverse Fourier transform of a complex periodic three-dimensional array.

## Required Arguments

*A* — Three-dimensional complex matrix containing the data to be transformed. (Input)

*B* — Three-dimensional complex matrix containing the inverse Fourier coefficients of *A*.  
(Output)  
The matrices *A* and *B* may be the same.

## Optional Arguments

*N1* — Limit on the first subscript of matrices *A* and *B*. (Input)  
Default:  $N1 = \text{size}(A,1)$ .

*N2* — Limit on the second subscript of matrices *A* and *B*. (Input)  
Default:  $N2 = \text{size}(A,2)$ .

*N3* — Limit on the third subscript of matrices *A* and *B*. (Input)  
Default:  $N3 = \text{size}(A,3)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{size}(A, 1)$ .

*MDA* — Middle dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDA = \text{size}(A, 2)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{size}(B, 1)$ .

*MDB* — Middle dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $MDB = \text{size}(B, 2)$ .

## FORTRAN 90 Interface

Generic: `CALL FFT3B (A, B [, ...])`

Specific: The specific interface names are `S_FFT3B` and `D_FFT3B`.

## FORTRAN 77 Interface

Single:      CALL FFT3B (N1, N2, N3, A, LDA, MDA, B, LDB, MDB)

Double:     The double precision name is DFFT3B.

## Description

The routine FFT3B computes the inverse discrete complex Fourier transform of a complex three-dimensional array of size  $(N1 = N) \times (N2 = M) \times (N3 = L)$ . It uses the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when  $N$ ,  $M$ , and  $L$  are each products of small prime factors. If  $N$ ,  $M$ , and  $L$  satisfy this condition, then the computational effort is proportional to  $NML \log NML$ . This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an  $N \times M \times L$  array  $a$ , FFT3B returns in  $b$

$$b_{jkl} \sum_{n=1}^N \sum_{m=1}^M \sum_{l=1}^L a_{nml} e^{2\pi i(j-1)(n-1)/N} e^{2\pi i(k-1)(m-1)/M} e^{2\pi i(l-1)(l-1)/L}$$

Furthermore, a vector of Euclidean norm  $S$  is mapped into a vector of norm

$$\sqrt{NMLS}$$

Finally, note that an unnormalized inverse is implemented in FFT3F. The routine FFT3B is based on the complex FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of F2T3B/DF2T3B. The reference is:

```
CALL F2T3B (N1, N2, N3, A, LDA, MDA, B, LDB, MDB, WFF1, WFF2, WFF3, CPY)
```

The additional arguments are as follows:

**WFF1** — Real array of length  $4 * N1 + 15$  initialized by FFTCI. The initialization depends on N1. (Input)

**WFF2** — Real array of length  $4 * N2 + 15$  initialized by FFTCI. The initialization depends on N2. (Input)

**WFF3** — Real array of length  $4 * N3 + 15$  initialized by FFTCI. The initialization depends on N3. (Input)

**CPY** — Real array of size  $2 * \text{MAX}(N1, N2, N3)$ . (Workspace)

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, WFFT1, WFF2, WFF3, and CPY are not referenced.

2. The routine FFT3B is most efficient when N1, N2, and N3 are the product of small primes.
3. If FFT3F/FFT3B is used repeatedly with the same values for N1, N2 and N3, then use FFTCI to fill WFF1(N = N1), WFF2(N = N2), and WFF3(N = N3). Follow this with repeated calls to F2T3F/F2T3B. This is more efficient than repeated calls to FFT3F/FFT3B.

If the Intel<sup>®</sup> Math Kernel Library, Sun Performance Library or IBM Engineering and Scientific Subroutine Library is used, parameters computed by FFTCI are not used. In this case, there is no need to call FFTCI.

### Example

In this example, we compute the Fourier transform of the  $2 \times 3 \times 4$  array

$$x_{nml} = n + 2(m-1) + 2(3)(l-1)$$

for  $1 \leq n \leq 2$ ,  $1 \leq m \leq 3$ , and  $1 \leq l \leq 4$  using the IMSL routine FFT3F. The result

$$a = \hat{x}$$

is then inverted using FFT3B. Note that the result is an array  $b$  satisfying  $b = 2(3)(4)x = 24x$ . In general, FFT3B is an unnormalized inverse with expansion factor  $NML$ .

```

USE FFT3B_INT
USE FFT3F_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER LDA, LDB, MDA, MDB, NDA, NDB
PARAMETER (LDA=2, LDB=2, MDA=3, MDB=3, NDA=4, NDB=4)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I, J, K, L, M, N, N1, N2, N3, NOUT
COMPLEX A(LDA,MDA,NDA), B(LDB,MDB,NDB), X(LDB,MDB,NDB)
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC CEXP, CMPLX
COMPLEX CEXP, CMPLX
! SPECIFICATIONS FOR SUBROUTINES
! Get output unit number
CALL UMACH (2, NOUT)
N1 = 2
N2 = 3
N3 = 4
! Set array X
DO 30 N=1, 2
 DO 20 M=1, 3
 DO 10 L=1, 4
 X(N,M,L) = N + 2*(M-1) + 2*3*(L-1)
10 CONTINUE
20 CONTINUE
30 CONTINUE

```

```

!
CALL FFT3F (X, A)
CALL FFT3B (A, B)
!
WRITE (NOUT,99996)
DO 50 I=1, 2
 WRITE (NOUT,99998) I
 DO 40 J=1, 3
 WRITE (NOUT,99999) (X(I,J,K),K=1,4)
40 CONTINUE
50 CONTINUE
!
WRITE (NOUT,99997)
DO 70 I=1, 2
 WRITE (NOUT,99998) I
 DO 60 J=1, 3
 WRITE (NOUT,99999) (A(I,J,K),K=1,4)
60 CONTINUE
70 CONTINUE
!
WRITE (NOUT, 99995)
DO 90 I=1, 2
 WRITE (NOUT,99998) I
 DO 80 J=1, 3
 WRITE (NOUT,99999) (B(I,J,K),K=1,4)
80 CONTINUE
90 CONTINUE
99995 FORMAT (13X, 'The unnormalized inverse is')
99996 FORMAT (13X, 'The input for FFT3F is')
99997 FORMAT (/, 13X, 'The results from FFT3F are')
99998 FORMAT (/, ' Face no. ', I1)
99999 FORMAT (1X, 4('(',F6.2,',',',',F6.2,')',3X))
END

```

## Output

The input for FFT3F is

```

Face no. 1
(1.00, 0.00) (7.00, 0.00) (13.00, 0.00) (19.00, 0.00)
(3.00, 0.00) (9.00, 0.00) (15.00, 0.00) (21.00, 0.00)
(5.00, 0.00) (11.00, 0.00) (17.00, 0.00) (23.00, 0.00)

```

```

Face no. 2
(2.00, 0.00) (8.00, 0.00) (14.00, 0.00) (20.00, 0.00)
(4.00, 0.00) (10.00, 0.00) (16.00, 0.00) (22.00, 0.00)
(6.00, 0.00) (12.00, 0.00) (18.00, 0.00) (24.00, 0.00)

```

The results from FFT3F are

```

Face no. 1
(300.00, 0.00) (-72.00, 72.00) (-72.00, 0.00) (-72.00,-72.00)
(-24.00, 13.86) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(-24.00,-13.86) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)

```

```

Face no. 2
(-12.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)
(0.00, 0.00) (0.00, 0.00) (0.00, 0.00) (0.00, 0.00)

```

The unnormalized inverse is

```

Face no. 1
(24.00, 0.00) (168.00, 0.00) (312.00, 0.00) (456.00, 0.00)
(72.00, 0.00) (216.00, 0.00) (360.00, 0.00) (504.00, 0.00)
(120.00, 0.00) (264.00, 0.00) (408.00, 0.00) (552.00, 0.00)

```

```

Face no. 2
(48.00, 0.00) (192.00, 0.00) (336.00, 0.00) (480.00, 0.00)
(96.00, 0.00) (240.00, 0.00) (384.00, 0.00) (528.00, 0.00)
(144.00, 0.00) (288.00, 0.00) (432.00, 0.00) (576.00, 0.00)

```

## RCONV



Computes the convolution of two real vectors.

### Required Arguments

*X* — Real vector of length *NX*. (Input)

*Y* — Real vector of length *NY*. (Input)

*Z* — Real vector of length *NZ* containing the convolution of *X* and *Y*. (Output)

*ZHAT* — Real vector of length *NZ* containing the discrete Fourier transform of *Z*. (Output)

### Optional Arguments

*IDO* — Flag indicating the usage of *RCONV*. (Input)

Default: *IDO* = 0.

#### *IDO* Usage

0 If this is the only call to *RCONV*.

If *RCONV* is called multiple times in sequence with the same *NX*, *NY*, and *IPAD*, *IDO* should be set to

1 on the first call

2 on the intermediate calls

3 on the final call.

***NX*** — Length of the vector *X*. (Input)

Default:  $NX = \text{size}(X, 1)$ .

***NY*** — Length of the vector *Y*. (Input)

Default:  $NY = \text{size}(Y, 1)$ .

***IPAD*** — *IPAD* should be set to zero for periodic data or to one for nonperiodic data. (Input)

Default:  $IPAD = 0$ .

***NZ*** — Length of the vector *Z*. (Input/Output)

Upon input: When *IPAD* is zero, *NZ* must be at least  $\text{MAX}(NX, NY)$ . When *IPAD* is one, *NZ* must be greater than or equal to the smallest integer greater than or equal to

$(NX + NY - 1)$  of the form  $(2^\alpha) * (3^\beta) * (5^\gamma)$  where alpha, beta, and gamma are nonnegative integers. Upon output, the value for *NZ* that was used by *RCONV*.

Default:  $NZ = \text{size}(Z, 1)$ .

## **FORTRAN 90 Interface**

Generic: `CALL RCONV (X, Y, Z, ZHAT [, ...])`

Specific: The specific interface names are `S_RCONV` and `D_RCONV`.

## **FORTRAN 77 Interface**

Single: `CALL RCONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT)`

Double: The double precision name is `DRCONV`.

## **Description**

The routine *RCONV* computes the discrete convolution of two sequences *x* and *y*. More precisely, let  $n_x$  be the length of *x* and  $n_y$  denote the length of *y*. If a circular convolution is desired, then *IPAD* must be set to zero. We set

$$n_z := \max\{n_x, n_y\}$$

and we pad out the shorter vector with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on *x* is interpreted as a positive number between 1 and  $n_z$ , modulo  $n_z$ .

The technique used to compute the  $z_i$ 's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of *z* is given by

$$\hat{z}(n) = \hat{x}(n) \hat{y}(n)$$

where

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of  $x$  and  $y$ , multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that  $n_z$  is a product of small primes if `IPAD` is set to zero. If  $n_z$  is a product of small primes, then the computational effort will be proportional to  $n_z \log(n_z)$ . If `IPAD` is one, then a good value is chosen for  $n_z$  so that the Fourier transforms are efficient and  $n_z \geq n_x + n_y - 1$ . This will mean that both vectors will be padded with zeroes.

We point out that no complex transforms of  $x$  or  $y$  are taken since both sequences are real, we can take real transforms and simulate the complex transform above. This can produce a savings of a factor of six in time as well as save space over using the complex transform.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `R2ONV/DR2ONV`. The reference is:

```
CALL R2ONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT, XWK,
 YWK, WK)
```

The additional arguments are as follows:

**XWK** — Real work array of length `NZ`.

**YWK** — Real work array of length `NZ`.

**WK** — Real work array of length  $2 * \text{NZ} + 15$ .

2. Informational error

Type Code

|   |   |                                                                                                                                         |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The length of the vector <code>Z</code> must be large enough to hold the results. An acceptable length is returned in <code>NZ</code> . |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------|

## Example

In this example, we compute both a periodic and a non-periodic convolution. The idea here is that one can compute a moving average of the type found in digital filtering using this routine. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. The periodic case tries to recover a noisy sin function by averaging five nearby values. The nonperiodic case tries to recover the values of an exponential function contaminated by noise. The large error for the last value printed has to do with the fact that the convolution is averaging the zeroes in the “pad” rather than function values. Notice that the signal size is 100, but we only report the errors at ten points.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER NFLTR, NY, A
PARAMETER (NFLTR=5, NY=100)
!
INTEGER I, IPAD, K, MOD, NOUT, NZ
REAL ABS, EXP, F1, F2, FLOAT, FLTR(NFLTR), &
 FLTRER, ORIGER, SIN, TOTAL1, TOTAL2, TWOPI, X, &
 Y(NY), Z(2*(NFLTR+NY-1)), ZHAT(2*(NFLTR+NY-1))
INTRINSIC ABS, EXP, FLOAT, MOD, SIN
!
 DEFINE FUNCTIONS
F1(X) = SIN(X)
F2(X) = EXP(X)
!
CALL RNSET (1234579)
CALL UMACH (2, NOUT)
TWOPI = CONST('PI')
TWOPI = 2.0*TWOPI
!
 SET UP THE FILTER
DO 10 I=1, 5
 FLTR(I) = 0.2
10 CONTINUE
!
 SET UP Y-VECTOR FOR THE PERIODIC
CASE.
DO 20 I=1, NY
 X = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
 Y(I) = RNUNF()
 Y(I) = F1(X) + 0.5*Y(I) - 0.25
20 CONTINUE
!
 CALL THE CONVOLUTION ROUTINE FOR THE
PERIODIC CASE.
NZ = 2*(NFLTR+NY-1)
CALL RCONV (FLTR, Y, Z, ZHAT, IPAD=0, NZ=NZ)
!
 PRINT RESULTS
WRITE (NOUT,99993)
WRITE (NOUT,99995)
TOTAL1 = 0.0
TOTAL2 = 0.0
DO 30 I=1, NY
!
 COMPUTE THE OFFSET FOR THE Z-VECTOR
IF (I .GE. NY-1) THEN
 K = I - NY + 2
ELSE
 K = I + 2
END IF
!
X = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
ORIGER = ABS(Y(I)-F1(X))
FLTRER = ABS(Z(K)-F1(X))
IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F1(X), ORIGER, &
 FLTRER
TOTAL1 = TOTAL1 + ORIGER
TOTAL2 = TOTAL2 + FLTRER
30 CONTINUE

```

```

WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
!
! SET UP Y-VECTOR FOR THE NONPERIODIC
! CASE.
DO 40 I=1, NY
 A = FLOAT(I-1)/FLOAT(NY-1)
 Y(I) = RNUNF()
 Y(I) = F2(A) + 0.5*Y(I) - 0.25
40 CONTINUE
!
! CALL THE CONVOLUTION ROUTINE FOR THE
! NONPERIODIC CASE.
NZ = 2*(NFLTR+NY-1)
CALL RCONV (FLTR, Y, Z, ZHAT, IPAD=1)
!
! PRINT RESULTS
WRITE (NOUT,99994)
WRITE (NOUT,99996)
TOTAL1 = 0.0
TOTAL2 = 0.0
DO 50 I=1, NY
 X = FLOAT(I-1)/FLOAT(NY-1)
 ORIGER = ABS(Y(I)-F2(X))
 FLTRER = ABS(Z(I+2)-F2(X))
 IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F2(X), ORIGER, &
 FLTRER
 TOTAL1 = TOTAL1 + ORIGER
 TOTAL2 = TOTAL2 + FLTRER
50 CONTINUE
WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
99993 FORMAT (' Periodic Case')
99994 FORMAT (/, ' Nonperiodic Case')
99995 FORMAT (8X, 'x', 9X, 'sin(x)', 6X, 'Original Error', 5X, &
 'Filtered Error')
99996 FORMAT (8X, 'x', 9X, 'exp(x)', 6X, 'Original Error', 5X, &
 'Filtered Error')
99997 FORMAT (1X, F10.4, F13.4, 2F18.4)
99998 FORMAT (' Average absolute error before filter:', F10.5)
99999 FORMAT (' Average absolute error after filter:', F11.5)
END

```

## Output

| Periodic Case                         |         |                |                |
|---------------------------------------|---------|----------------|----------------|
| x                                     | sin(x)  | Original Error | Filtered Error |
| 0.0000                                | 0.0000  | 0.0811         | 0.0587         |
| 0.6981                                | 0.6428  | 0.0226         | 0.0781         |
| 1.3963                                | 0.9848  | 0.1526         | 0.0529         |
| 2.0944                                | 0.8660  | 0.0959         | 0.0125         |
| 2.7925                                | 0.3420  | 0.1747         | 0.0292         |
| 3.4907                                | -0.3420 | 0.1035         | 0.0238         |
| 4.1888                                | -0.8660 | 0.0402         | 0.0595         |
| 4.8869                                | -0.9848 | 0.0673         | 0.0798         |
| 5.5851                                | -0.6428 | 0.1044         | 0.0074         |
| 6.2832                                | 0.0000  | 0.0154         | 0.0018         |
| Average absolute error before filter: |         | 0.12481        |                |
| Average absolute error after filter:  |         | 0.04778        |                |

Nonperiodic Case

| x                                     | exp(x) | Original Error | Filtered Error |
|---------------------------------------|--------|----------------|----------------|
| 0.0000                                | 1.0000 | 0.1476         | 0.3915         |
| 0.1111                                | 1.1175 | 0.0537         | 0.0326         |
| 0.2222                                | 1.2488 | 0.1278         | 0.0932         |
| 0.3333                                | 1.3956 | 0.1136         | 0.0987         |
| 0.4444                                | 1.5596 | 0.1617         | 0.0964         |
| 0.5556                                | 1.7429 | 0.0071         | 0.0662         |
| 0.6667                                | 1.9477 | 0.1248         | 0.0713         |
| 0.7778                                | 2.1766 | 0.1556         | 0.0158         |
| 0.8889                                | 2.4324 | 0.1529         | 0.0696         |
| 1.0000                                | 2.7183 | 0.2124         | 1.0562         |
| Average absolute error before filter: |        |                | 0.12538        |
| Average absolute error after filter:  |        |                | 0.07764        |

---

## CCONV



Computes the convolution of two complex vectors.

### Required Arguments

*X* — Complex vector of length *NX*. (Input)

*Y* — Complex vector of length *NY*. (Input)

*Z* — Complex vector of length *NZ* containing the convolution of *X* and *Y*. (Output)

*ZHAT* — Complex vector of length *NZ* containing the discrete complex Fourier transform of *Z*. (Output)

### Optional Arguments

*IDO* — Flag indicating the usage of *CCONV*. (Input)  
Default: *IDO* = 0.

#### *IDO* Usage

0 If this is the only call to *CCONV*.

If *CCONV* is called multiple times in sequence with the same *NX*, *NY*, and *IPAD*, *IDO* should be set to:

1 on the first call

2 on the intermediate calls

3 on the final call.

***NX*** — Length of the vector *X*. (Input)

Default:  $NX = \text{size}(X, 1)$ .

***NY*** — Length of the vector *Y*. (Input)

Default:  $NY = \text{size}(Y, 1)$ .

***IPAD*** — *IPAD* should be set to zero for periodic data or to one for nonperiodic data. (Input)

Default:  $IPAD = 0$ .

***NZ*** — Length of the vector *Z*. (Input/Output)

Upon input: When *IPAD* is zero, *NZ* must be at least  $\text{MAX}(NX, NY)$ . When *IPAD* is one, *NZ* must be greater than or equal to the smallest integer greater than or equal to  $(NX + NY - 1)$  of the form  $(2^\alpha) * (3^\beta) * (5^\gamma)$  where alpha, beta, and gamma are nonnegative integers. Upon output, the value for *NZ* that was used by *CCONV*.

Default:  $NZ = \text{size}(Z, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL CCONV (X, Y, Z, ZHAT [, ...])`

Specific: The specific interface names are `S_CCONV` and `D_CCONV`.

## FORTRAN 77 Interface

Single: `CALL CCONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT)`

Double: The double precision name is `DCCONV`.

## Description

The subroutine *CCONV* computes the discrete convolution of two complex sequences *x* and *y*. More precisely, let  $n_x$  be the length of *x* and  $n_y$  denote the length of *y*. If a circular convolution is desired, then *IPAD* must be set to zero. We set

$$n_z := \max\{n_x, n_y\}$$

and we pad out the shorter vector with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on *x* is interpreted as a positive number between 1 and  $n_z$ , modulo  $n_z$ .

The technique used to compute the  $z_i$ 's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of *z* is given by

$$\hat{z}(n) = \hat{x}(n) \hat{y}(n)$$

where

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of  $x$  and  $y$ , multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that  $n_z$  is a product of small primes if `IPAD` is set to zero. If  $n_z$  is a product of small primes, then the computational effort will be proportional to  $n_z \log(n_z)$ . If `IPAD` is one, then a good value is chosen for  $n_z$  so that the Fourier transforms are efficient and  $n_z \geq n_x + n_y - 1$ . This will mean that both vectors will be padded with zeroes.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2ONV/DC2ONV`. The reference is:

```
CALL C2ONV (IDO, NX, X, NY, Y, IPAD, NZ, Z, ZHAT, XWK, YWK,
 WK)
```

The additional arguments are as follows:

**XWK** — Complex work array of length `NZ`.

**YWK** — Complex work array of length `NZ`.

**WK** — Real work array of length `6 * NZ + 15`.

2. Informational error

Type Code

|   |   |                                                                                                                                         |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The length of the vector <code>Z</code> must be large enough to hold the results. An acceptable length is returned in <code>NZ</code> . |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------|

## Example

In this example, we compute both a periodic and a non-periodic convolution. The idea here is that one can compute a moving average of the type found in digital filtering using this routine. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. The periodic case tries to recover a noisy function  $f_1(x) = \cos(x) + i \sin(x)$  by averaging five nearby values. The nonperiodic case tries to recover the values of the function  $f_2(x) = e^x f_1(x)$  contaminated by noise. The large error for the first and last value printed has to do with the fact that the convolution is averaging the zeroes in the “pad” rather than function values. Notice that the signal size is 100, but we only report the errors at ten points.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
```

```

INTEGER NFLTR, NY
PARAMETER (NFLTR=5, NY=100)
!
INTEGER I, IPAD, K, MOD, NOUT, NZ
REAL CABS, COS, EXP, FLOAT, FLTRER, ORIGER, &
 SIN, TOTAL1, TOTAL2, TWOPI, X, T1, T2
COMPLEX CMPLX, F1, F2, FLTR(NFLTR), Y(NY), Z(2*(NFLTR+NY-1)), &
 ZHAT(2*(NFLTR+NY-1))
INTRINSIC CABS, CMPLX, COS, EXP, FLOAT, MOD, SIN
!
 DEFINE FUNCTIONS
F1(X) = CMPLX(COS(X), SIN(X))
F2(X) = EXP(X)*CMPLX(COS(X), SIN(X))
!
CALL RNSET (1234579)
CALL UMACH (2, NOUT)
TWOPI = CONST('PI')
TWOPI = 2.0*TWOPI
!
 SET UP THE FILTER
CALL CSET(NFLTR, (0.2,0.0), FLTR,1)
!
 SET UP Y-VECTOR FOR THE PERIODIC
! CASE.
DO 20 I=1, NY
 X = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
 T1 = RNUNF()
 T2 = RNUNF()
 Y(I) = F1(X) + CMPLX(0.5*T1-0.25,0.5*T2-0.25)
20 CONTINUE
!
 CALL THE CONVOLUTION ROUTINE FOR THE
! PERIODIC CASE.
NZ = 2*(NFLTR+NY-1)
CALL CCONV (FLTR, Y, Z, ZHAT)
!
 PRINT RESULTS
WRITE (NOUT,99993)
WRITE (NOUT,99995)
TOTAL1 = 0.0
TOTAL2 = 0.0
DO 30 I=1, NY
!
 COMPUTE THE OFFSET FOR THE Z-VECTOR
 IF (I .GE. NY-1) THEN
 K = I - NY + 2
 ELSE
 K = I + 2
 END IF
!
 X = TWOPI*FLOAT(I-1)/FLOAT(NY-1)
 ORIGER = CABS(Y(I)-F1(X))
 FLTRER = CABS(Z(K)-F1(X))
 IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F1(X), ORIGER, &
 FLTRER
 TOTAL1 = TOTAL1 + ORIGER
 TOTAL2 = TOTAL2 + FLTRER
30 CONTINUE
WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
!
 SET UP Y-VECTOR FOR THE NONPERIODIC
! CASE.
DO 40 I=1, NY

```

```

 X = FLOAT(I-1)/FLOAT(NY-1)
 T1 = RNUNF()
 T2 = RNUNF()
 Y(I) = F2(X) + CMPLX(0.5*T1-0.25,0.5*T2-0.25)
40 CONTINUE
!
! CALL THE CONVOLUTION ROUTINE FOR THE
! NONPERIODIC CASE.
 NZ = 2*(NFLTR+NY-1)
 CALL CCONV (FLTR, Y, Z, ZHAT, IPAD=1)
!
! PRINT RESULTS
 WRITE (NOUT,99994)
 WRITE (NOUT,99996)
 TOTAL1 = 0.0
 TOTAL2 = 0.0
 DO 50 I=1, NY
 X = FLOAT(I-1)/FLOAT(NY-1)
 ORIGER = CABS(Y(I)-F2(X))
 FLTRER = CABS(Z(I+2)-F2(X))
 IF (MOD(I,11) .EQ. 1) WRITE (NOUT,99997) X, F2(X), ORIGER, &
 FLTRER
 TOTAL1 = TOTAL1 + ORIGER
 TOTAL2 = TOTAL2 + FLTRER
50 CONTINUE
 WRITE (NOUT,99998) TOTAL1/FLOAT(NY)
 WRITE (NOUT,99999) TOTAL2/FLOAT(NY)
99993 FORMAT (' Periodic Case')
99994 FORMAT (/, ' Nonperiodic Case')
99995 FORMAT (8X, 'x', 15X, 'f1(x)', 8X, 'Original Error', 5X, &
 'Filtered Error')
99996 FORMAT (8X, 'x', 15X, 'f2(x)', 8X, 'Original Error', 5X, &
 'Filtered Error')
99997 FORMAT (1X, F10.4, 5X, '(', F7.4, ', ', F8.4, ')', 5X, F8.4, &
 10X, F8.4)
99998 FORMAT (' Average absolute error before filter:', F11.5)
99999 FORMAT (' Average absolute error after filter:', F12.5)
 END

```

## Output

Periodic Case

| x                                     | f1(x)               | Original Error | Filtered Error |
|---------------------------------------|---------------------|----------------|----------------|
| 0.0000                                | ( 1.0000, 0.0000 )  | 0.1666         | 0.0773         |
| 0.6981                                | ( 0.7660, 0.6428 )  | 0.1685         | 0.1399         |
| 1.3963                                | ( 0.1736, 0.9848 )  | 0.1756         | 0.0368         |
| 2.0944                                | (-0.5000, 0.8660 )  | 0.2171         | 0.0142         |
| 2.7925                                | (-0.9397, 0.3420 )  | 0.1147         | 0.0200         |
| 3.4907                                | (-0.9397, -0.3420 ) | 0.0998         | 0.0331         |
| 4.1888                                | (-0.5000, -0.8660 ) | 0.1137         | 0.0586         |
| 4.8869                                | ( 0.1736, -0.9848 ) | 0.2217         | 0.0843         |
| 5.5851                                | ( 0.7660, -0.6428 ) | 0.1831         | 0.0744         |
| 6.2832                                | ( 1.0000, 0.0000 )  | 0.3234         | 0.0893         |
| Average absolute error before filter: |                     | 0.19315        |                |
| Average absolute error after filter:  |                     | 0.08296        |                |

Nonperiodic Case

| x                                     | f2(x)              | Original Error | Filtered Error |
|---------------------------------------|--------------------|----------------|----------------|
| 0.0000                                | ( 1.0000, 0.0000 ) | 0.0783         | 0.4336         |
| 0.1111                                | ( 1.1106, 0.1239 ) | 0.2434         | 0.0477         |
| 0.2222                                | ( 1.2181, 0.2752 ) | 0.1819         | 0.0584         |
| 0.3333                                | ( 1.3188, 0.4566 ) | 0.0703         | 0.1267         |
| 0.4444                                | ( 1.4081, 0.6706 ) | 0.1458         | 0.0868         |
| 0.5556                                | ( 1.4808, 0.9192 ) | 0.1946         | 0.0930         |
| 0.6667                                | ( 1.5307, 1.2044 ) | 0.1458         | 0.0734         |
| 0.7778                                | ( 1.5508, 1.5273 ) | 0.1815         | 0.0690         |
| 0.8889                                | ( 1.5331, 1.8885 ) | 0.0805         | 0.0193         |
| 1.0000                                | ( 1.4687, 2.2874 ) | 0.2396         | 1.1708         |
| Average absolute error before filter: |                    | 0.18549        |                |
| Average absolute error after filter:  |                    | 0.09636        |                |

## RCORL



Computes the correlation of two real vectors.

### Required Arguments

*X* — Real vector of length *N*. (Input)

*Y* — Real vector of length *N*. (Input)

*Z* — Real vector of length *NZ* containing the correlation of *X* and *Y*. (Output)

*ZHAT* — Real vector of length *NZ* containing the discrete Fourier transform of *Z*. (Output)

### Optional Arguments

*IDO* — Flag indicating the usage of RCORL. (Input)  
Default: *IDO* = 0.

#### *IDO* Usage

0 If this is the only call to RCORL.

If RCORL is called multiple times in sequence with the same *NX*, *NY*, and *IPAD*, *IDO* should be set to:

1 on the first call

2 on the intermediate calls

3 on the final call.

***N*** — Length of the *X* and *Y* vectors. (Input)

Default:  $N = \text{size}(X, 1)$ .

***IPAD*** — *IPAD* should be set as follows. (Input)

Default:  $IPAD = 0$ .

#### **IPAD Value**

*IPAD* 0 for periodic data with *X* and *Y* different.

*IPAD* 1 for nonperiodic data with *X* and *Y* different.

*IPAD* 2 for periodic data with *X* and *Y* identical.

*IPAD* 3 for nonperiodic data with *X* and *Y* identical.

***NZ*** — Length of the vector *Z*. (Input/Output)

Upon input: When *IPAD* is zero or two, *NZ* must be at least  $(2 * N - 1)$ . When *IPAD* is one or three, *NZ* must be greater than or equal to the smallest integer greater than or equal to  $(2 * N - 1)$  of the form  $(2^\alpha) * (3^\beta) * (5^\gamma)$  where alpha, beta, and gamma are nonnegative integers. Upon output, the value for *NZ* that was used by *RCORL*.

Default:  $NZ = \text{size}(Z, 1)$ .

## **FORTRAN 90 Interface**

Generic: `CALL RCORL (X, Y, Z, ZHAT [, ...])`

Specific: The specific interface names are `S_RCORL` and `D_RCORL`.

## **FORTRAN 77 Interface**

Single: `CALL RCORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT)`

Double: The double precision name is `DRCORL`.

## **Description**

The subroutine *RCORL* computes the discrete correlation of two sequences *x* and *y*. More precisely, let *n* be the length of *x* and *y*. If a circular correlation is desired, then *IPAD* must be set to zero (for *x* and *y* distinct) and two (for *x* = *y*). We set (on output)

$$n_z = n \quad \text{if } IPAD = 0, 2$$
$$n_z = 2^\alpha 3^\beta 5^\gamma \geq 2n - 1 \quad \text{if } IPAD = 1, 3$$

where  $\alpha, \beta, \gamma$  are nonnegative integers yielding the smallest number of the type  $2^\alpha 3^\beta 5^\gamma$  satisfying the inequality. Once  $n_z$  is determined, we pad out the vectors with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on  $x$  is interpreted as a positive number between one and  $n_z$ , modulo  $n_z$ . Note that this means that

$$z_{n_z-k}$$

contains the correlation of  $x(\cdot - k - 1)$  with  $y$  as  $k = 0, 1, \dots, n_z/2$ . Thus, if  $x(k - 1) = y(k)$  for all  $k$ , then we would expect

$$z_{n_z}$$

to be the largest component of  $z$ .

The technique used to compute the  $z_i$ 's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of  $z$  is given by

$$\hat{z}_j = \hat{x}_j \bar{\hat{y}}_j$$

where

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of  $x$  and the conjugate of the discrete Fourier transform of  $y$ , multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that  $n_z$  is a product of small primes if `IPAD` is set to zero or two. If  $n_z$  is a product of small primes, then the computational effort will be proportional to  $n_z \log(n_z)$ . If `IPAD` is one or three, then a good value is chosen for  $n_z$  so that the Fourier transforms are efficient and  $n_z \geq 2n - 1$ . This will mean that both vectors will be padded with zeroes.

We point out that no complex transforms of  $x$  or  $y$  are taken since both sequences are real, and we can take real transforms and simulate the complex transform above. This can produce a savings of a factor of six in time as well as save space over using the complex transform.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `R2ORL/DR2ORL`. The reference is:

```
CALL R2ORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT, XWK, YWK, WK)
```

The additional arguments are as follows:

**XWK** — Real work array of length `NZ`.

**YWK** — Real work array of length `NZ`.

**WK** — Real work array of length  $2 * NZ + 15$ .

2. Informational error

Type Code

4 1 The length of the vector  $z$  must be large enough to hold the results.  
An acceptable length is returned in  $NZ$ .

**Example**

In this example, we compute both a periodic and a non-periodic correlation between two distinct signals  $x$  and  $y$ . In the first case we have 100 equally spaced points on the interval  $[0, 2\pi]$  and  $f_1(x) = \sin(x)$ . We define  $x$  and  $y$  as follows

$$x_i = f_1\left(2\pi \frac{i-1}{n-1}\right) \quad i = 1, \dots, n$$

$$y_i = f_1\left(2\pi \frac{i-1}{n-1} + \frac{\pi}{2}\right) \quad i = 1, \dots, n$$

Note that the maximum value of  $z$  (the correlation of  $x$  with  $y$ ) occurs at  $i = 26$ , which corresponds to the offset.

The nonperiodic case uses the function  $f_2(x) = \sin(x^2)$ . The two input signals are on the interval  $[0, 4\pi]$ .

$$x_i = f_2\left(4\pi \frac{i-1}{n-1}\right) \quad i = 1, \dots, n$$

$$y_i = f_2\left(4\pi \frac{i-1}{n-1} + \pi\right) \quad i = 1, \dots, n$$

The offset of  $x$  to  $y$  is again (roughly) 26 and this is where  $z$  has its maximum value.

```

USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER N
PARAMETER (N=100)

!
INTEGER I, IPAD, K, NOUT, NZ
REAL A, F1, F2, FLOAT, PI, SIN, X(N), XNORM, &
 Y(N), YNORM, Z(4*N), ZHAT(4*N)
INTRINSIC FLOAT, SIN

! Define functions
F1(A) = SIN(A)
F2(A) = SIN(A*A)

!
CALL UMACH (2, NOUT)
PI = CONST('pi')

! Set up the vectors for the
! periodic case.
DO 10 I=1, N

```

```

 X(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1))
 Y(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI/2.0)
10 CONTINUE
! Call the correlation routine for the
! periodic case.
 NZ = 2*N
 CALL RCORL (X, Y, Z, ZHAT)
! Find the element of Z with the
! largest normalized value.
 XNORM = SNRM2(N,X,1)
 YNORM = SNRM2(N,Y,1)
 DO 20 I=1, N
 Z(I) = Z(I)/(XNORM*YNORM)
20 CONTINUE
 K = ISMAX(N,Z,1)
! Print results for the periodic
! case.
 WRITE (NOUT,99995)
 WRITE (NOUT,99994)
 WRITE (NOUT,99997)
 WRITE (NOUT,99998) K
 WRITE (NOUT,99999) K, Z(K)
! Set up the vectors for the
! nonperiodic case.
 DO 30 I=1, N
 X(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1))
 Y(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI)
30 CONTINUE
! Call the correlation routine for the
! nonperiodic case.
 NZ = 4*N
 CALL RCORL (X, Y, Z, ZHAT, IPAD=1)
! Find the element of Z with the
! largest normalized value.
 XNORM = SNRM2(N,X,1)
 YNORM = SNRM2(N,Y,1)
 DO 40 I=1, N
 Z(I) = Z(I)/(XNORM*YNORM)
40 CONTINUE
 K = ISMAX(N,Z,1)
! Print results for the nonperiodic
! case.
 WRITE (NOUT,99996)
 WRITE (NOUT,99994)
 WRITE (NOUT,99997)
 WRITE (NOUT,99998) K
 WRITE (NOUT,99999) K, Z(K)
99994 FORMAT (1X, 28('-'))
99995 FORMAT (' Case #1: Periodic data')
99996 FORMAT (/, ' Case #2: Nonperiodic data')
99997 FORMAT (' The element of Z with the largest normalized ')
99998 FORMAT (' value is Z(', I2, ').')
99999 FORMAT (' The normalized value of Z(', I2, ') is', F6.3)
 END

```

## Output

Example #1: Periodic case

-----  
The element of  $Z$  with the largest normalized value is  $Z(26)$ .  
The normalized value of  $Z(26)$  is 1.000

Example #2: Nonperiodic case

-----  
The element of  $Z$  with the largest normalized value is  $Z(26)$ .  
The normalized value of  $Z(26)$  is 0.661

---

## CCORL



Computes the correlation of two complex vectors.

### Required Arguments

$X$  — Complex vector of length  $N$ . (Input)

$Y$  — Complex vector of length  $N$ . (Input)

$Z$  — Complex vector of length  $NZ$  containing the correlation of  $X$  and  $Y$ . (Output)

$ZHAT$  — Complex vector of length  $NZ$  containing the inverse discrete complex Fourier transform of  $Z$ . (Output)

### Optional Arguments

$IDO$  — Flag indicating the usage of `CCORL`. (Input)  
Default:  $IDO = 0$ .

#### $IDO$ Usage

0     If this is the only call to `CCORL`.

If `CCORL` is called multiple times in sequence with the same  $NX$ ,  $NY$ , and  $IPAD$ ,  $IDO$  should be set to:

1     on the first call

2     on the intermediate calls

3     on the final call.

*N* — Length of the *X* and *Y* vectors. (Input)

Default:  $N = \text{size}(X, 1)$ .

*IPAD* — *IPAD* should be set as follows. (Input)

*IPAD* = 0 for periodic data with *X* and *Y* different. *IPAD* = 1 for nonperiodic data with *X* and *Y* different. *IPAD* = 2 for periodic data with *X* and *Y* identical. *IPAD* = 3 for nonperiodic data with *X* and *Y* identical.

Default: *IPAD* = 0.

*NZ* — Length of the vector *Z*. (Input/Output)

Upon input: When *IPAD* is zero or two, *NZ* must be at least  $(2 * N - 1)$ . When *IPAD* is one or three, *NZ* must be greater than or equal to the smallest integer greater than or equal to  $(2 * N - 1)$  of the form  $(2^\alpha) * (3^\beta) * (5^\gamma)$  where alpha, beta, and gamma are nonnegative integers. Upon output, the value for *NZ* that was used by *CCORL*.

Default:  $NZ = \text{size}(Z, 1)$ .

## FORTRAN 90 Interface

Generic: `CALL CCORL (X, Y, Z, ZHAT [, ...])`

Specific: The specific interface names are `S_CCORL` and `D_CCORL`.

## FORTRAN 77 Interface

Single: `CALL CCORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT)`

Double: The double precision name is `DCCORL`.

## Description

The subroutine *CCORL* computes the discrete correlation of two complex sequences *x* and *y*. More precisely, let *n* be the length of *x* and *y*. If a circular correlation is desired, then *IPAD* must be set to zero (for *x* and *y* distinct) and two (for *x* = *y*). We set (on output)

$$n_z = n \quad \text{if } IPAD = 0, 2$$
$$n_z = 2^\alpha 3^\beta 5^\gamma \geq 2n - 1 \quad \text{if } IPAD = 1, 3$$

where  $\alpha, \beta, \gamma$  are nonnegative integers yielding the smallest number of the type  $2^\alpha 3^\beta 5^\gamma$  satisfying the inequality. Once  $n_z$  is determined, we pad out the vectors with zeroes. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} \bar{y}_j$$

where the index on *x* is interpreted as a positive number between one and  $n_z$ , modulo  $n_z$ . Note that this means that

$$z_{n_z-k}$$

contains the correlation of  $x(\cdot - k - 1)$  with  $y$  as  $k = 0, 1, \dots, n_z/2$ . Thus, if  $x(k - 1) = y(k)$  for all  $k$ , then we would expect

$$\Re z_{n_z}$$

to be the largest component of  $\Re z$ .

The technique used to compute the  $z_i$ 's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of  $z$  is given by

$$\hat{z}_j = \hat{x}_j \bar{\hat{y}}_j$$

where

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of  $x$  and the conjugate of the discrete Fourier transform of  $y$ , multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that  $n_z$  is a product of small primes if `IPAD` is set to zero or two. If  $n_z$  is a product of small primes, then the computational effort will be proportional to  $n_z \log(n_z)$ . If `IPAD` is one or three, then a good value is chosen for  $n_z$  so that the Fourier transforms are efficient and  $n_z \geq 2n - 1$ . This will mean that both vectors will be padded with zeroes.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `C2ORL/DC2ORL`. The reference is:

```
CALL C2ORL (IDO, N, X, Y, IPAD, NZ, Z, ZHAT, XWK, YWK, WK)
```

The additional arguments are as follows:

**XWK** — Complex work array of length `NZ`.

**YWK** — Complex work array of length `NZ`.

**WK** — Real work array of length `6 * NZ + 15`.

2. Informational error

Type Code

4 1 The length of the vector `Z` must be large enough to hold the results. An acceptable length is returned in `NZ`.

## Example

In this example, we compute both a periodic and a non-periodic correlation between two distinct signals  $x$  and  $y$ . In the first case, we have 100 equally spaced points on the interval  $[0, 2\pi]$  and  $f_1(x) = \cos(x) + i \sin(x)$ . We define  $x$  and  $y$  as follows

$$x_i = f_1\left(2\pi \frac{i-1}{n-1}\right) \quad i = 1, \dots, n$$
$$y_i = f_1\left(2\pi \frac{i-1}{n-1} + \frac{\pi}{2}\right) \quad i = 1, \dots, n$$

Note that the maximum value of  $z$  (the correlation of  $x$  with  $y$ ) occurs at  $i = 26$ , which corresponds to the offset.

The nonperiodic case uses the function  $f_2(x) = \cos(x^2) + i \sin(x^2)$ . The two input signals are on the interval  $[0, 4\pi]$ .

$$x_i = f_2\left(4\pi \frac{i-1}{n-1}\right) \quad i = 1, \dots, n$$
$$y_i = f_2\left(4\pi \frac{i-1}{n-1} + \pi\right) \quad i = 1, \dots, n$$

The offset of  $x$  to  $y$  is again (roughly) 26 and this is where  $z$  has its maximum value.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER N
PARAMETER (N=100)

!
INTEGER I, IPAD, K, NOUT, NZ
REAL A, COS, F1, F2, FLOAT, PI, SIN, &
 XNORM, YNORM, ZREAL(4*N)
COMPLEX CMPLX, X(N), Y(N), Z(4*N), ZHAT(4*N)
INTRINSIC CMPLX, COS, FLOAT, SIN

! Define functions
F1(A) = CMPLX(COS(A), SIN(A))
F2(A) = CMPLX(COS(A*A), SIN(A*A))

!

CALL RNSET (1234579)
CALL UMACH (2, NOUT)
PI = CONST('pi')

! Set up the vectors for the
! periodic case.
DO 10 I=1, N
 X(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1))
 Y(I) = F1(2.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI/2.0)
10 CONTINUE

! Call the correlation routine for the
! periodic case.
NZ = 2*N
CALL CCORL (X, Y, Z, ZHAT, IPAD=0, NZ=NZ)

! Find the element of Z with the
! largest normalized real part.
```

```

XNORM = SCNRM2(N,X,1)
YNORM = SCNRM2(N,Y,1)
DO 20 I=1, N
 ZREAL1(I) = REAL(Z(I))/(XNORM*YNORM)
20 CONTINUE
K = ISMAX(N,ZREAL1,1)
!
! Print results for the periodic
! case.
WRITE (NOUT,99995)
WRITE (NOUT,99994)
WRITE (NOUT,99997)
WRITE (NOUT,99998) K
WRITE (NOUT,99999) K, ZREAL1(K)
!
! Set up the vectors for the
! nonperiodic case.
DO 30 I=1, N
 X(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1))
 Y(I) = F2(4.0*PI*FLOAT(I-1)/FLOAT(N-1)+PI)
30 CONTINUE
!
! Call the correlation routine for the
! nonperiodic case.
NZ = 4*N
CALL CCORL (X, Y, Z, ZHAT, IPAD=1, NZ=NZ)
!
! Find the element of z with the
! largest normalized real part.
XNORM = SCNRM2(N,X,1)
YNORM = SCNRM2(N,Y,1)
DO 40 I=1, N
 ZREAL1(I) = REAL(Z(I))/(XNORM*YNORM)
40 CONTINUE
K = ISMAX(N,ZREAL1,1)
!
! Print results for the nonperiodic
! case.
WRITE (NOUT,99996)
WRITE (NOUT,99994)
WRITE (NOUT,99997)
WRITE (NOUT,99998) K
WRITE (NOUT,99999) K, ZREAL1(K)
99994 FORMAT (1X, 28('-'))
99995 FORMAT (' Case #1: periodic data')
99996 FORMAT (/, ' Case #2: nonperiodic data')
99997 FORMAT (' The element of Z with the largest normalized ')
99998 FORMAT (' real part is Z(', I2, ').')
99999 FORMAT (' The normalized value of real(Z(', I2, ') is', F6.3)
END

```

## Output

Example #1: periodic case

```

The element of Z with the largest normalized real part is Z(26).
The normalized value of real(Z(26)) is 1.000

```

Example #2: nonperiodic case

```

```

The element of  $Z$  with the largest normalized real part is  $Z(26)$ .  
The normalized value of  $\text{real}(Z(26))$  is 0.638

---

## INLAP

Computes the inverse Laplace transform of a complex function.

### Required Arguments

**F** — User-supplied FUNCTION to which the inverse Laplace transform will be computed. The form is  $F(Z)$ , where

$Z$  — Complex argument. (Input)

$F$  — The complex function value. (Output)

$F$  must be declared EXTERNAL in the calling program.  $F$  should also be declared COMPLEX.

**T** — Array of length  $N$  containing the points at which the inverse Laplace transform is desired. (Input)

$T(I)$  must be greater than zero for all  $I$ .

**FINV** — Array of length  $N$  whose  $I$ -th component contains the approximate value of the Laplace transform at the point  $T(I)$ . (Output)

### Optional Arguments

**N** — Number of points at which the inverse Laplace transform is desired. (Input)

Default:  $N = \text{size}(T, 1)$ .

**ALPHA** — An estimate for the maximum of the real parts of the singularities of  $F$ . If unknown, set  $ALPHA = 0$ . (Input)

Default:  $ALPHA = 0.0$ .

**KMAX** — The number of function evaluations allowed for each  $T(I)$ . (Input)

Default:  $KMAX = 500$ .

**RELERR** — The relative accuracy desired. (Input)

Default:  $RELERR = 1.1920929e-5$  for single precision and  $2.22d-10$  for double precision.

### FORTRAN 90 Interface

Generic: `CALL INLAP (F, T, FINV [, ...])`

Specific: The specific interface names are `S_INLAP` and `D_INLAP`.

### FORTRAN 77 Interface

Single: `CALL INLAP (F, N, T, ALPHA, RELERR, KMAX, FINV)`

Double: The double precision name is `DINLAP`.

## Description

The routine `INLAP` computes the inverse Laplace transform of a complex-valued function. Recall that if  $f$  is a function that vanishes on the negative real axis, then we can define the Laplace transform of  $f$  by

$$L[f](s) := \int_0^{\infty} e^{-sx} f(x) dx$$

It is assumed that for some value of  $s$  the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on applying the epsilon algorithm to the complex Fourier series obtained as a discrete approximation to the inversion integral. The initial algorithm was proposed by K.S. Crump (1976) but was significantly improved by de Hoog et al. (1982). Given a complex-valued transform  $F(s) = L[f](s)$ , the trapezoidal rule gives the approximation to the inverse transform

$$g(t) = \left( e^{\alpha t} / T \right) \Re \left\{ \frac{1}{2} F(\alpha) + \sum_{k=1}^{\infty} F\left(\alpha + \frac{ik\pi}{T}\right) \exp\left(\frac{ik\pi t}{T}\right) \right\}$$

This is the real part of the sum of a complex power series in  $z = \exp(i\pi t/T)$ , and the algorithm accelerates the convergence of the partial sums of this power series by using the epsilon algorithm to compute the corresponding diagonal Pade approximants. The algorithm attempts to choose the order of the Pade approximant to obtain the specified relative accuracy while not exceeding the maximum number of function evaluations allowed. The parameter  $\alpha$  is an estimate for the maximum of the real parts of the singularities of  $F$ , and an incorrect choice of  $\alpha$  may give false convergence. Even in cases where the correct value of  $\alpha$  is unknown, the algorithm will attempt to estimate an acceptable value. Assuming satisfactory convergence, the discretization error  $E := g - f$  satisfies

$$E = \sum_{n=1}^{\infty} e^{-2n\alpha T} f(2nT + t)$$

It follows that if  $|f(t)| \leq Me^{\beta t}$ , then we can estimate the expression above to obtain (for  $0 \leq t \leq 2T$ )

$$E \leq Me^{\alpha t} / \left( e^{2T(\alpha-\beta)} - 1 \right)$$

## Comments

Informational errors

Type Code

- |   |   |                                                                                                                                         |
|---|---|-----------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The algorithm was not able to achieve the accuracy requested within <code>KMAX</code> function evaluations for some <code>T(I)</code> . |
| 4 | 2 | Overflow is occurring for a particular value of <code>T</code> .                                                                        |

## Example

We invert the Laplace transform of the simple function  $(s - 1)^{-2}$  and print the computed answer, the true solution and the difference at five different points. The correct inverse transform is  $xe^x$ .

```
USE INLAP_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER I, KMAX, N, NOUT
REAL ALPHA, DIF(5), EXP, FINV(5), FLOAT, RELERR, T(5), &
 TRUE(5)
COMPLEX F
INTRINSIC EXP, FLOAT
EXTERNAL F

! Get output unit number
CALL UMACH (2, NOUT)

!
DO 10 I=1, 5
 T(I) = FLOAT(I) - 0.5
10 CONTINUE
N = 5
ALPHA = 1.0E0
RELERR = 5.0E-4
CALL INLAP (F, T, FINV, ALPHA=ALPHA, RELERR=RELERR)
! Evaluate the true solution and the
! difference
DO 20 I=1, 5
 TRUE(I) = T(I)*EXP(T(I))
 DIF(I) = TRUE(I) - FINV(I)
20 CONTINUE

!
WRITE (NOUT,99999) (T(I),FINV(I),TRUE(I),DIF(I),I=1,5)
99999 FORMAT (7X, 'T', 8X, 'FINV', 9X, 'TRUE', 9X, 'DIFF', /, &
 5(1X,E9.1,3X,1PE10.3,3X,1PE10.3,3X,1PE10.3,/))
END

!
COMPLEX FUNCTION F (S)
COMPLEX S
F = 1./(S-1.)**2
RETURN
END
```

## Output

| T       | FINV      | TRUE      | DIFF       |
|---------|-----------|-----------|------------|
| 0.5E+00 | 8.244E-01 | 8.244E-01 | -4.768E-06 |
| 1.5E+00 | 6.723E+00 | 6.723E+00 | -3.481E-05 |
| 2.5E+00 | 3.046E+01 | 3.046E+01 | -1.678E-04 |
| 3.5E+00 | 1.159E+02 | 1.159E+02 | -6.027E-04 |
| 4.5E+00 | 4.051E+02 | 4.051E+02 | -2.106E-03 |

---

## SINLP

Computes the inverse Laplace transform of a complex function.

## Required Arguments

**F** — User-supplied FUNCTION to which the inverse Laplace transform will be computed. The form is  $F(Z)$ , where

**Z** — Complex argument. (Input)

**F** — The complex function value. (Output)

**F** must be declared EXTERNAL in the calling program. **F** must also be declared COMPLEX.

**T** — Vector of length **N** containing points at which the inverse Laplace transform is desired. (Input)

$T(I)$  must be greater than zero for all **I**.

**FINV** — Vector of length **N** whose **I**-th component contains the approximate value of the inverse Laplace transform at the point  $T(I)$ . (Output)

## Optional Arguments

**N** — The number of points at which the inverse Laplace transform is desired. (Input)

Default:  $N = \text{size}(T, 1)$ .

**SIGMA0** — An estimate for the maximum of the real parts of the singularities of **F**. (Input)

If unknown, set  $SIGMA0 = 0.0$ .

Default:  $SIGMA0 = 0.e0$ .

**EPSTOL** — The required absolute uniform pseudo accuracy for the coefficients and inverse Laplace transform values. (Input)

Default:  $EPSTOL = 1.1920929e-5$  for single precision and  $2.22d-10$  for double precision.

**ERRVEC** — Vector of length eight containing diagnostic information. (Output)

All components depend on the intermediately generated Laguerre coefficients. See [Comments](#).

## FORTRAN 90 Interface

Generic: `CALL SINLP (F, T, FINV [, ...])`

Specific: The specific interface names are `S_SINLP` and `D_SINLP`.

## FORTRAN 77 Interface

Single: `CALL SINLP (F, N, T, SIGMA0, EPSTOL, ERRVEC, FINV)`

Double: The double precision name is `DSINLP`.

## Description

The routine `SINLAP` computes the inverse Laplace transform of a complex-valued function. Recall that if  $f$  is a function that vanishes on the negative real axis, then we can define the Laplace transform of  $f$  by

$$L[f](s) := \int_0^{\infty} e^{-sx} f(x) dx$$

It is assumed that for some value of  $s$  the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on a modification of Weeks' method (see W.T. Weeks (1966)) due to B.S. Garbow et. al. (1988). This method is suitable when  $f$  has continuous derivatives of all orders on  $[0, \infty)$ . In this situation, this routine should be used in place of the IMSL routine `INLAP`. It is especially efficient when multiple function values are desired. In particular, given a complex-valued function  $F(s) = L[f](s)$ , we can expand  $f$  in a Laguerre series whose coefficients are determined by  $F$ . This is fully described in B.S. Garbow et. al. (1988) and Lyness and Giunta (1986).

The algorithm attempts to return approximations  $g(t)$  to  $f(t)$  satisfying

$$\left| \frac{g(t) - f(t)}{e^{\sigma t}} \right| < \varepsilon$$

where  $\varepsilon := \text{EPSTOL}$  and  $\sigma := \text{SIGMA} > \text{SIGMA0}$ . The expression on the left is called the pseudo error. An estimate of the pseudo error is available in `ERRVEC(1)`.

The first step in the method is to transform  $F$  to  $\phi$  where

$$\phi(z) = \frac{b}{1-z} F\left(\frac{b}{1-z} - \frac{b}{2} + \sigma\right)$$

Then, if  $f$  is smooth, it is known that  $\phi$  is analytic in the unit disc of the complex plane and hence has a Taylor series expansion

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s$$

which converges for all  $z$  whose absolute value is less than the radius of convergence  $R_c$ . This number is estimated in `ERRVEC(6)`. In `ERRVEC(5)`, we estimate the smallest number  $K$  which satisfies

$$|a_s| < \frac{K}{R^s}$$

for all  $R < R_c$ .

The coefficients of the Taylor series for  $\phi$  can be used to expand  $f$  in a Laguerre series

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt)$$

## Comments

1. Workspace may be explicitly provided, if desired, by use of `S2NLP/DS2NLP`. The reference is:

```
CALL S2NLP(F, N, T, SIGMA0, EPSTOL, ERRVEC, FINV, SIGMA,
 BVALUE, MTOP, WK, IFLOVC)
```

The additional arguments are as follows:

**SIGMA** — The first parameter of the Laguerre expansion. If `SIGMA` is not greater than `SIGMA0`, it is reset to `SIGMA0 + 0.7`. (Input)

**BVALUE** — The second parameter of the Laguerre expansion. If `BVALUE` is less than  $2.0 * (SIGMA - SIGMA0)$ , it is reset to  $2.5 * (SIGMA - SIGMA0)$ . (Input)

**MTOP** — An upper limit on the number of coefficients to be computed in the Laguerre expansion. `MTOP` must be a multiple of four. Note that the maximum number of Laplace transform evaluations is  $MTOP/2 + 2$ . (Default: 1024.) (Input)

**WK** — Real work vector of length  $9 * MTOP/4$ .

**IFLOVC** — Integer vector of length `N`, the `I`-th component of which contains the overflow/underflow indicator for the computed value of `FINV(I)`. (Output)  
See Comment 3.

2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                                                                            |
|---|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | 1 | Normal termination, but with estimated error bounds slightly larger than <code>EPSTOL</code> . Note, however, that the actual errors on the final results may be smaller than <code>EPSTOL</code> as bounds independent of <code>T</code> are pessimistic. |
| 3 | 2 | Normal calculation, terminated early at the roundoff error level estimate because this estimate exceeds the required accuracy (usually due to overly optimistic expectation by the user about attainable accuracy).                                        |
| 4 | 3 | The decay rate of the coefficients is too small. It may improve results to use <code>S2NLP</code> and increase <code>MTOP</code> .                                                                                                                         |
| 4 | 4 | The decay rate of the coefficients is too small. In addition, the roundoff error level is such that required accuracy cannot be reached.                                                                                                                   |

4 5 No error bounds are returned as the behavior of the coefficients does not enable reasonable prediction. Results are probably wrong. Check the value of SIGMA0. In this case, each of ERRVEC(J), J = 1, ..., 5, is set to -1.0.

3. The following are descriptions of the vectors ERRVEC and IFLOVC.

**ERRVEC** — Real vector of length eight.

ERRVEC(1) = Overall estimate of the pseudo error, ERRVEC(2) + ERRVEC(3) + ERRVEC(4). Pseudo error = absolute error / exp(sigma \* tvalue).

ERRVEC(2) = Estimate of the pseudo discretization error.

ERRVEC(3) = Estimate of the pseudo truncation error.

ERRVEC(4) = Estimate of the pseudo condition error on the basis of minimal noise levels in the function values.

ERRVEC(5) = K, the coefficient of the decay function for ACOEF, the coefficients of the Laguerre expansion.

ERRVEC(6) = R, the base of the decay function for ACOEF. Here  $\text{abs}(\text{ACOEFF}(J+1)) \cdot \text{LE.K/R}^{**J}$  for  $J \leq \text{MACT}/2$ , where MACT is the number of Laguerre coefficients actually computed.

ERRVEC(7) = ALPHA, the logarithm of the largest ACOEF.

ERRVEC(8) = BETA, the logarithm of the smallest nonzero ACOEF.

**IFLOVC** — Integer vector of length N containing the overflow/underflow indicators for FINV. For each I, the value of IFLOVC(I) signifies the following.

0 = Normal termination.

1 = The value of the inverse Laplace transform is found to be too large to be representable; FINV(I) is set to AMACH(6).

-1 = The value of the inverse Laplace transform is found to be too small to be representable; FINV(I) is set to 0.0.

2 = The value of the inverse Laplace transform is estimated to be too large, even before the series expansion, to be representable; FINV(I) is set to AMACH(6).

-2 = The value of the inverse Laplace transform is estimated to be too small, even before the series expansion, to be representable; FINV(I) is set to 0.0.

## Example

We invert the Laplace transform of the simple function  $(s - 1)^{-2}$  and print the computed answer, the true solution, and the difference at five different points. The correct inverse transform is  $xe^x$ .

```
USE SINLP_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER I, NOUT
REAL DIF(5), ERRVEC(8), EXP, FINV(5), FLOAT, RELERR, &
 SIGMA0, T(5), TRUE(5), EPSTOL
COMPLEX F
INTRINSIC EXP, FLOAT
EXTERNAL F
!
! Get output unit number
CALL UMACH (2, NOUT)
!
DO 10 I=1, 5
 T(I) = FLOAT(I) - 0.5
10 CONTINUE
SIGMA0 = 1.0E0
RELERR = 5.0E-4
EPSTOL = 1.0E-4
CALL SINLP (F, T, FINV, SIGMA0=SIGMA0, EPSTOL=RELERR)
!
! Evaluate the true solution and the
! difference
DO 20 I=1, 5
 TRUE(I) = T(I)*EXP(T(I))
 DIF(I) = TRUE(I) - FINV(I)
20 CONTINUE
!
WRITE (NOUT,99999) (T(I),FINV(I),TRUE(I),DIF(I),I=1,5)
99999 FORMAT (7X, 'T', 8X, 'FINV', 9X, 'TRUE', 9X, 'DIFF', /, &
 5(1X,E9.1,3X,1PE10.3,3X,1PE10.3,3X,1PE10.3,/))
END
!
COMPLEX FUNCTION F (S)
COMPLEX S
!
F = 1./(S-1.)**2
RETURN
END
```

## Output

| T       | FINV      | TRUE      | DIFF       |
|---------|-----------|-----------|------------|
| 0.5E+00 | 8.244E-01 | 8.244E-01 | -2.086E-06 |
| 1.5E+00 | 6.723E+00 | 6.723E+00 | -8.583E-06 |
| 2.5E+00 | 3.046E+01 | 3.046E+01 | 0.000E+00  |
| 3.5E+00 | 1.159E+02 | 1.159E+02 | 2.289E-05  |
| 4.5E+00 | 4.051E+02 | 4.051E+02 | -2.136E-04 |





# Chapter 7: Nonlinear Equations

---

## Routines

|                                                            |                       |      |
|------------------------------------------------------------|-----------------------|------|
| <b>7.1. Zeros of a Polynomial</b>                          |                       |      |
| Real coefficients using Laguerre method .....              | <a href="#">ZPLRC</a> | 1320 |
| Real coefficients using Jenkins-Traub method .....         | <a href="#">ZPORC</a> | 1322 |
| Complex coefficients .....                                 | <a href="#">ZPOCC</a> | 1324 |
| <b>7.2. Zero(s) of a Function</b>                          |                       |      |
| Zeros of a complex analytic function .....                 | <a href="#">ZANLY</a> | 1325 |
| Zero of a real univariate function .....                   | <a href="#">ZUNI</a>  | 1328 |
| Zero of a real function with sign changes .....            | <a href="#">ZBREN</a> | 1331 |
| Zeros of a real function .....                             | <a href="#">ZREAL</a> | 1334 |
| <b>7.3. Root of a System of Equations</b>                  |                       |      |
| Finite-difference Jacobian .....                           | <a href="#">NEQNF</a> | 1337 |
| Analytic Jacobian .....                                    | <a href="#">NEQNJ</a> | 1340 |
| Broyden's update and Finite-difference Jacobian .....      | <a href="#">NEQBF</a> | 1344 |
| Factored secant update with a user-supplied Jacobian ..... | <a href="#">NEQBJ</a> | 1350 |

---

## Usage Notes

### Zeros of a Polynomial

A polynomial function of degree  $n$  can be expressed as follows:

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_1 z + a_0$$

where  $a_n \neq 0$ .

There are three routines for zeros of a polynomial. The routines [ZPLRC](#) and [ZPORC](#) find zeros of the polynomial with real coefficients while the routine [ZPOCC](#) finds zeros of the polynomial with complex coefficients.

The Jenkins-Traub method is used for the routines [ZPORC](#) and [ZPOCC](#); whereas [ZPLRC](#) uses the Laguerre method. Both methods perform well in comparison with other methods. The Jenkins-Traub algorithm usually runs faster than the Laguerre method. Furthermore, the routine [ZANLY](#) in the next section can also be used for the complex polynomial.

## Zero(s) of a Function

The routines [ZANLY](#) and [ZREAL](#) use Müller's method to find the zeros of a complex analytic function and real zeros of a real function, respectively. The routine [ZBREN](#) finds a zero of a real function, using an algorithm that is a combination of interpolation and bisection. This algorithm requires the user to supply two points such that the function values at these two points have opposite sign. For functions where it is difficult to obtain two such points, [ZUNI](#) or [ZREAL](#) can be used.

## Root of System of Equations

A system of equations can be stated as follows:

$$f_i(x) = 0, \text{ for } i = 1, 2, \dots, n$$

where  $x \in \mathbf{R}^n$ .

The routines [NEQNF](#) and [NEQNJ](#) use a modified Powell hybrid method to find a zero of a system of nonlinear equations. The difference between these two routines is that the Jacobian is estimated by a finite-difference method in [NEQNF](#), whereas the user has to provide the Jacobian for [NEQNJ](#). It is advised that the Jacobian-checking routine, [CHJAC](#) (see [Chapter 8, Optimization](#)), be used to ensure the accuracy of the user-supplied Jacobian.

The routines [NEQBF](#) and [NEQBJ](#) use a secant method with Broyden's update to find a zero of a system of nonlinear equations. The difference between these two routines is that the Jacobian is estimated by a finite-difference method in [NEQBF](#); whereas the user has to provide the Jacobian for [NEQBJ](#). For more details, see Dennis and Schnabel (1983, Chapter 8).

---

## ZPLRC

Finds the zeros of a polynomial with real coefficients using Laguerre's method.

### Required Arguments

**COEFF** — Vector of length `NDEG + 1` containing the coefficients of the polynomial in increasing order by degree. (Input)

The polynomial is

$$\text{COEFF}(\text{NDEG} + 1) * Z^{**\text{NDEG}} + \text{COEFF}(\text{NDEG}) * Z^{**(\text{NDEG} - 1)} + \dots + \text{COEFF}(1).$$

**ROOT** — Complex vector of length `NDEG` containing the zeros of the polynomial. (Output)

### Optional Arguments

**NDEG** — Degree of the polynomial.  $1 \leq \text{NDEG} \leq 100$  (Input)

Default: `NDEG = size(COEFF,1) - 1`.

### FORTRAN 90 Interface

Generic:     `CALL ZPLRC (COEFF, ROOT [, ...])`

Specific: The specific interface names are `S_ZPLRC` and `D_ZPLRC`.

## FORTRAN 77 Interface

Single: `CALL ZPLRC (NDEG, COEFF, ROOT)`

Double: The double precision name is `DZPLRC`.

## Description

Routine `ZPLRC` computes the  $n$  zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_1 z + a_0$$

where the coefficients  $a_i$  for  $i = 0, 1, \dots, n$  are real and  $n$  is the degree of the polynomial.

The routine `ZPLRC` is a modification of B.T. Smith's routine `ZERPOL` (Smith 1967) that uses Laguerre's method. Laguerre's method is cubically convergent for isolated zeros and linearly convergent for multiple zeros. The maximum length of the step between successive iterates is restricted so that each new iterate lies inside a region about the previous iterate known to contain a zero of the polynomial. An iterate is accepted as a zero when the polynomial value at that iterate is smaller than a computed bound for the rounding error in the polynomial value at that iterate. The original polynomial is deflated after each real zero or pair of complex zeros is found. Subsequent zeros are found using the deflated polynomial.

## Comments

Informational errors

Type Code

- |   |   |                                                                                                                                                                     |
|---|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.       |
| 3 | 2 | Fewer than <code>NDEG</code> zeros were found. The <code>ROOT</code> vector will contain the value for machine infinity in the locations that do not contain zeros. |

## Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where  $z$  is a complex variable.

```
USE ZPLRC_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NDEG
PARAMETER (NDEG=3)
!
REAL COEFF(NDEG+1)
```

```

 COMPLEX ZERO(NDEG)
!
! Set values of COEFF
! COEFF = (-2.0 4.0 -3.0 1.0)
!
 DATA COEFF/-2.0, 4.0, -3.0, 1.0/
!
 CALL ZPLRC (COEFF, ZERO, NDEG)
!
 CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
 END

```

## Output

```

 The zeros found are
 1 2 3
(1.000, 1.000) (1.000,-1.000) (1.000, 0.000)

```

---

## ZPORC

Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.

### Required Arguments

*COEFF* — Vector of length  $NDEG + 1$  containing the coefficients of the polynomial in increasing order by degree. (Input)

The polynomial is

$$COEFF(NDEG + 1) * Z^{NDEG} + COEFF(NDEG) * Z^{(NDEG - 1)} + \dots + COEFF(1).$$

*ROOT* — Complex vector of length  $NDEG$  containing the zeros of the polynomial. (Output)

### Optional Arguments

*NDEG* — Degree of the polynomial.  $1 \leq NDEG \leq 100$  (Input)

Default:  $NDEG = \text{size}(COEFF, 1) - 1$ .

### FORTRAN 90 Interface

Generic: `CALL ZPORC (COEFF, ROOT [, ...])`

Specific: The specific interface names are `S_ZPORC` and `D_ZPORC`.

### FORTRAN 77 Interface

Single: `CALL ZPORC (NDEG, COEFF, ROOT)`

Double: The double precision name is `DZPORC`.

## Description

Routine ZPORC computes the  $n$  zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_1 z + a_0$$

where the coefficients  $a_i$  for  $i = 0, 1, \dots, n$  are real and  $n$  is the degree of the polynomial.

The routine ZPORC uses the Jenkins-Traub three-stage algorithm (Jenkins and Traub 1970; Jenkins 1975). The zeros are computed one at a time for real zeros or two at a time for complex conjugate pairs. As the zeros are found, the real zero or quadratic factor is removed by polynomial deflation.

## Comments

Informational errors

Type Code

|   |   |                                                                                                                                                               |
|---|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem. |
| 3 | 2 | Fewer than NDEG zeros were found. The ROOT vector will contain the value for machine infinity in the locations that do not contain zeros.                     |

## Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where  $z$  is a complex variable.

```
USE ZPORC_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NDEG
PARAMETER (NDEG=3)
!
REAL COEFF(NDEG+1)
COMPLEX ZERO(NDEG)
! Set values of COEFF
! COEFF = (-2.0 4.0 -3.0 1.0)
!
DATA COEFF/-2.0, 4.0, -3.0, 1.0/
!
CALL ZPORC (COEFF, ZERO)
!
CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
END
```



## Comments

Informational errors

Type Code

- |   |   |                                                                                                                                                               |
|---|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem. |
| 3 | 2 | Fewer than NDEG zeros were found. The ROOT vector will contain the value for machine infinity in the locations that do not contain zeros.                     |

## Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - (3 + 6i)z^2 - (8 - 12i)z + 10$$

where  $z$  is a complex variable.

```
USE ZPOCC_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NDEG
PARAMETER (NDEG=3)

COMPLEX COEFF(NDEG+1), ZERO(NDEG)
! Set values of COEFF
! COEFF = (10.0 + 0.0i)
! (-8.0 + 12.0i)
! (-3.0 - 6.0i)
! (1.0 + 0.0i)
!
DATA COEFF/(10.0,0.0), (-8.0,12.0), (-3.0,-6.0), (1.0,0.0)/
!
CALL ZPOCC (COEFF, ZERO)
!
CALL WRCRN ('The zeros found are', ZERO, 1, NDEG, 1)
!
END
```

## Output

```
 The zeros found are
 1 2 3
(1.000, 1.000) (1.000, 2.000) (1.000, 3.000)
```

---

# ZANLY

Finds the zeros of a univariate complex function using Müller's method.

## Required Arguments

**F** — User-supplied COMPLEX FUNCTION to compute the value of the function of which the zeros will be found. The form is  $F(Z)$ , where

**Z** — The complex value at which the function is evaluated. (Input)  
Z should not be changed by F.

**F** — The computed complex function value at the point Z. (Output)

F must be declared EXTERNAL in the calling program.

**Z** — A complex vector of length  $NKNOWN + NNEW$ . (Output)

$Z(1), \dots, Z(NKNOWN)$  contain the known zeros.  $Z(NKNOWN + 1), \dots, Z(NKNOWN + NNEW)$  contain the new zeros found by ZANLY. If ZINIT is not needed, ZINIT and Z can share the same storage locations.

## Optional Arguments

**ERRABS** — First stopping criterion. (Input)

Let  $FP(Z) = F(Z)/P$  where  $P = (Z - Z(1)) * (Z - Z(2)) * \dots * (Z - Z(K - 1))$   
and  $Z(1), \dots, Z(K - 1)$  are previously found zeros.

If  $(CABS(F(Z)) .LE. ERRABS .AND. CABS(FP(Z)) .LE. ERRABS)$ ,  
then Z is accepted as a zero.

Default:  $ERRABS = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Second stopping criterion is the relative error. (Input)

A zero is accepted if the difference in two successive approximations to this zero is within ERRREL. ERRREL must be less than 0.01; otherwise, 0.01 will be used.

Default:  $ERRREL = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**NKNOWN** — The number of previously known zeros, if any, that must be stored in

$ZINIT(1), \dots, ZINIT(NKNOWN)$  prior to entry to ZANLY. (Input)

NKNOWN must be set equal to zero if no zeros are known.

Default:  $NKNOWN = 0$ .

**NNEW** — The number of new zeros to be found by ZANLY. (Input)

Default:  $NNEW = 1$ .

**NGUESS** — The number of initial guesses provided. (Input)

These guesses must be stored in  $ZINIT(NKNOWN + 1), \dots, ZINIT(NKNOWN + NGUESS)$ .

NGUESS must be set equal to zero if no guesses are provided.

Default:  $NGUESS = 0$ .

**ITMAX** — The maximum allowable number of iterations per zero. (Input)

Default:  $ITMAX = 100$ .

**ZINIT** — A complex vector of length  $NKNOWN + NNEW$ . (Input)

$ZINIT(1), \dots, ZINIT(NKNOWN)$  must contain the known zeros.  $ZINIT(NKNOWN + 1), \dots,$

ZINIT(NKNOWN + NNEW) may, on user option, contain initial guesses for the NNEW new zeros that are to be computed. If the user does not provide an initial guess, zero is used.

**INFO** — An integer vector of length NKNOWN + NNEW. (Output)  
 INFO(J) contains the number of iterations used in finding the J-th zero when convergence was achieved. If convergence was not obtained in ITMAX iterations, INFO(J) will be greater than ITMAX.

### FORTRAN 90 Interface

Generic: CALL ZANLY (F, Z [, ...])

Specific: The specific interface names are S\_ZANLY and D\_ZANLY.

### FORTRAN 77 Interface

Single: CALL ZANLY (F, ERRABS, ERRREL, NKNOWN, NNEW, NGUESS, ZINIT, ITMAX, Z, INFO)

Double: The double precision name is DZANLY.

### Example

This example finds the zeros of the equation  $f(z) = z^3 + 5z^2 + 9z + 45$ , where  $z$  is a complex variable.

```

USE ZANLY_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER INFO(3), NGUESS, NNEW
COMPLEX F, Z(3), ZINIT(3)
EXTERNAL F

! Set the guessed zero values in ZINIT
!
! ZINIT = (1.0+1.0i 1.0+1.0i 1.0+1.0i)
DATA ZINIT/3*(1.0,1.0)/

! Set values for all input parameters
NNEW = 3
NGUESS = 3

! Find the zeros of F
CALL ZANLY (F, Z, NNEW=NNEW, NGUESS=NGUESS, &
 ZINIT=ZINIT, INFO=INFO)

! Print results
CALL WRCRN ('The zeros are', Z)
END

! External complex function
COMPLEX FUNCTION F (Z)
COMPLEX Z

F = Z**3 + 5.0*Z**2 + 9.0*Z + 45.0

```

```
RETURN
END
```

## Output

```
 The zeros are
 1 2 3
(0.000, 3.000) (0.000,-3.000) (-5.000, 0.000)
```

---

# ZUNI

Finds a zero of a real univariate function.

## Required Arguments

*F* — User-supplied function of which a zero will be found. The form is  $F(X [, \dots])$ , where:

### Function Return Value

*F* — The computed function value at the point  $x$ . (Output)

### Required Arguments

*X* — The point at which the function is evaluated. (Input)  
 $x$  should not be changed by *F*.

### Optional Arguments

*FCN\_DATA* — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function. For a detailed description of this argument see [FCN\\_DATA](#) below.

*F* must be declared `EXTERNAL` in the calling program.

*A* — See *B*. (Input/Output)

*B* — Two points at which the user-supplied function can be evaluated. (Input/Output)  
On input, if  $F(A)$  and  $F(B)$  are of opposite sign, the zero will be found in the interval  $[A, B]$  and on output *B* will contain the value of  $x$  at which  $F(x) = 0$ . If  $F(A) * F(B) > 0$ , and  $A \neq B$  then a search along the  $x$  number line is initiated for a point at which there is a sign change and  $|B - A|$  will be used in setting the step size for the initial search. If  $A = B$  on entry then the search is started as described in the description section below. On output, *B* is the abscissa at which  $|F(x)|$  had the smallest value. If  $F(B) \neq 0$  on output, *A* will contain the nearest abscissa to output *B* at which  $F(x)$  was evaluated and found to have the opposite sign from  $F(B)$ .

## Optional Arguments

**TOL**— Error tolerance. (Input)

If  $TOL > 0.0$ , the zero is to be isolated to an interval of length less than  $TOL$ .

If  $TOL < 0.0$ , an  $x$  is desired for which  $|F(x)|$  is  $\leq |TOL|$ .

If  $TOL = 0.0$ , the iteration continues until the zero of  $F(x)$  is isolated as accurately as possible.

Default:  $TOL = 0.0$ .

**MAXFN** — The number of function evaluations. (Input/Output)

On input, **MAXFN** specifies an upper bound on the number of function evaluations required for convergence. Set **MAXFN** to 0 if the number of function evaluations is to be unbounded.

On output, **MAXFN** will contain the actual number of function evaluations used.

Default:  $MAXFN = 0$  so the number of function evaluations is unbounded.

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied function.

The derived type, `s_fcn_data`, is defined as:

```
type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type
```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type. Note that if this optional argument is used then this argument must also be used in the user-supplied function. (Input/Output)

## FORTRAN 90 Interface

Generic: `CALL ZUNI (F, A, B [, ...])`

Specific: The specific interface names are `S_ZUNI` and `D_ZUNI`.

## Description

**ZUNI** is based on the JPL Library routine **SZERO**. The algorithm used is attributed to Dr. Fred T. Krogh, JPL, 1972. Tests have shown **ZUNI** to require fewer function evaluations, on average, than a number of other algorithms for finding a zero of a continuous function. Also, unlike **ZBREN** which restricts the user to supplying points **A** and **B** such that  $f(A)$  and  $f(B)$  are opposite in sign, **ZUNI** will accept any two points **A** and **B** and initiate a search on the number line for an  $x$  such that  $f(x) = 0$  when there is no sign difference between  $f(A)$  and  $f(B)$ . In either case, **B** is updated with a new value on each successive iteration. The algorithm description follows.

When  $f(A) \times f(B) > 0$  at the initial point, iterates for  $x$  are generated according to the formula  $x = x_{\min} + (x_{\min} - x_{\max}) \times \rho$ , where the subscript “min” is associated with the  $(f, x)$  pair that has the



```

!
 CALL UMACH (2, NOUT)
!
 CALL ZUNI (F, A, B, MAXFN=MAXFN)
!
 WRITE (NOUT,99999) B, MAXFN
99999 FORMAT (' The best approximation to the zero of F is equal to', &
 F5.1, '.', '/', ' The number of function evaluations', &
 ' required was ', I2, '.', '//)
!
 END
!
 REAL FUNCTION F (X)
 REAL X
!
 F = X*X + X - 2.0
 RETURN
 END

```

## Output

The best approximation to the zero of F is equal to -2.0.  
The number of function evaluations required was 10.

---

## ZBREN

Finds a zero of a real function that changes sign in a given interval.

### Required Arguments

**F** — User-supplied FUNCTION to compute the value of the function of which a zero will be found. The form is F(X), where

**X** — The point at which the function is evaluated. (Input)  
X should not be changed by F.

**F** — The computed function value at the point X. (Output)  
F must be declared EXTERNAL in the calling program.

**A** — See B. (Input/Output)

**B** — On input, the user must supply two points, A and B, such that F(A) and F(B) are opposite in sign. (Input/Output)  
On output, both A and B are altered. B will contain the best approximation to the zero of F.

### Optional Arguments

**ERRABS** — First stopping criterion. (Input)  
A zero, B, is accepted if ABS(F(B)) is less than or equal to ERRABS. ERRABS may be set

to zero.

Default:  $ERRABS = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Second stopping criterion is the relative error. (Input)

A zero is accepted if the change between two successive approximations to this zero is within **ERRREL**.

Default:  $ERRREL = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**MAXFN** — On input, **MAXFN** specifies an upper bound on the number of function evaluations required for convergence. (Input/Output)

On output, **MAXFN** will contain the actual number of function evaluations used.

Default:  $MAXFN = 100$ .

## FORTRAN 90 Interface

Generic: `CALL ZBREN (F, A, B [, ...])`

Specific: The specific interface names are `S_ZBREN` and `D_ZBREN`.

## FORTRAN 77 Interface

Single: `CALL ZBREN (F, ERRABS, ERRREL, A, B, MAXFN)`

Double: The double precision name is `DZBREN`.

## Description

The algorithm used by `ZBREN` is a combination of linear interpolation, inverse quadratic interpolation, and bisection. Convergence is usually superlinear and is never much slower than the rate for the bisection method. See Brent (1971) for a more detailed account of this algorithm.

## Comments

1. Informational error

Type Code

4 1 Failure to converge in **MAXFN** function evaluations.

2. On exit from `ZBREN` without any error message, **A** and **B** satisfy the following:

$$F(A)F(B) \leq 0.0$$

$$|F(B)| \leq |F(A)|, \text{ and}$$

$$\text{either } |F(B)| \leq ERRABS \text{ or}$$

$$|A - B| \leq \max(|B|, 0.1) * ERRREL.$$

The presence of 0.1 in the stopping criterion causes leading zeros to the right of the decimal point to be counted as significant digits. Scaling may be required in order to accurately determine a zero of small magnitude.



```

REAL X
!
F = X**2 + X - 2.0
RETURN
END

```

## Output

The best approximation to the zero of  $F$  is equal to  $-2.0$ .  
The number of function evaluations required was 12.

# ZREAL

Finds the real zeros of a real function using Müller's method.

## Required Arguments

**$F$**  — User-supplied FUNCTION to compute the value of the function of which a zero will be found. The form is  $F(X)$ , where

$X$  — The point at which the function is evaluated. (Input)  
 $X$  should not be changed by  $F$ .

$F$  — The computed function value at the point  $X$ . (Output)

$F$  must be declared EXTERNAL in the calling program.

**$X$**  — A vector of length NROOT. (Output)  
 $X$  contains the computed zeros.

## Optional Arguments

**ERRABS** — First stopping criterion. (Input)  
A zero  $X(I)$  is accepted if  $ABS(F(X(I))) < ERRABS$ .  
Default:  $ERRABS = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**ERRREL** — Second stopping criterion is the relative error. (Input)  
A zero  $X(I)$  is accepted if the relative change of two successive approximations to  $X(I)$  is less than  $ERRREL$ .  
Default:  $ERRREL = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**EPS** — See  $ETA$ . (Input)  
Default:  $EPS = 1.e-4$  for single precision and  $1.d-8$  for double precision.

**ETA** — Spread criteria for multiple zeros. (Input)  
If the zero  $X(I)$  has been computed and  $ABS(X(I) - X(J)) < EPS$ , where  $X(J)$  is a previously computed zero, then the computation is restarted with a guess equal to

$X(I) + \text{ETA}$ .  
Default:  $\text{ETA} = .01$ .

***NROOT*** — The number of zeros to be found by *ZREAL*. (Input)  
Default:  $\text{NROOT} = 1$ .

***ITMAX*** — The maximum allowable number of iterations per zero. (Input)  
Default:  $\text{ITMAX} = 100$ .

***XGUESS*** — A vector of length *NROOT*. (Input)  
*XGUESS* contains the initial guesses for the zeros.  
Default:  $\text{XGUESS} = 0.0$ .

***INFO*** — An integer vector of length *NROOT*. (Output)  
*INFO(J)* contains the number of iterations used in finding the *J*-th zero when convergence was achieved. If convergence was not obtained in *ITMAX* iterations, *INFO(J)* will be greater than *ITMAX*.

## **FORTRAN 90 Interface**

Generic:    `CALL ZREAL (F, X [, ...])`

Specific:    The specific interface names are `S_ZREAL` and `D_ZREAL`.

## **FORTRAN 77 Interface**

Single:    `CALL ZREAL (F, ERRABS, ERRREL, EPS, ETA, NROOT, ITMAX, XGUESS, X, INFO)`

Double:    The double precision name is `DZREAL`.

## **Description**

Routine *ZREAL* computes *n* real zeros of a real function *f*. Given a user-supplied function *f(x)* and an *n*-vector of initial guesses  $x_1, x_2, \dots, x_n$ , the routine uses Müller's method to locate *n* real zeros of *f*, that is, *n* real values of *x* for which  $f(x) = 0$ . The routine has two convergence criteria: the first requires that

$$\left| f(x_i^m) \right|$$

be less than *ERRABS*; the second requires that the relative change of any two successive approximations to an  $x_i$  be less than *ERRREL*. Here,

$$x_i^m$$

is the *m*-th approximation to  $x_i$ . Let *ERRABS* be  $\epsilon_1$ , and *ERRREL* be  $\epsilon_2$ . The criteria may be stated mathematically as follows:

Criterion 1:

$$\left| f(x_i^m) \right| < \varepsilon_1$$

Criterion 2:

$$\left| \frac{x_i^{m+1} - x_i^m}{x_i^m} \right| < \varepsilon_2$$

“Convergence” is the satisfaction of either criterion.

## Comments

1. Informational error

Type Code

3 1 Failure to converge within `ITMAX` iterations for at least one of the `NROOT` roots.

2. Routine `ZREAL` always returns the last approximation for zero `J` in `X(J)`. If the convergence criterion is satisfied, then `INFO(J)` is less than or equal to `ITMAX`. If the convergence criterion is not satisfied, then `INFO(J)` is set to `ITMAX + 1`.
3. The routine `ZREAL` assumes that there exist `NROOT` distinct real zeros for the function `F` and that they can be reached from the initial guesses supplied. The routine is designed so that convergence to any single zero cannot be obtained from two different initial guesses.
4. Scaling the `X` vector in the function `F` may be required, if any of the zeros are known to be less than one.

## Example

This example finds the real zeros of the second-degree polynomial

$$f(x) = x^2 + 2x - 6$$

with the initial guess (4.6, -193.3).

```

USE ZREAL_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NROOT
REAL EPS, ERRABS, ERRREL
PARAMETER (NROOT=2)
!
INTEGER INFO(NROOT)
REAL F, X(NROOT), XGUESS(NROOT)
EXTERNAL F
! Set values of initial guess

```

```

! XGUESS = (4.6 -193.3)
!
DATA XGUESS/4.6, -193.3/
!
EPS = 1.0E-5
ERRABS = 1.0E-5
ERRREL = 1.0E-5
!
! Find the zeros
CALL ZREAL (F, X, errabs=errabs, errrel=errrel, eps=eps, &
 nroot=nroot, xguess=xguess)
!
CALL WRRRN ('The zeros are', X, 1, NROOT, 1)
!
END
!
REAL FUNCTION F (X)
REAL X
!
F = X*X + 2.0*X - 6.0
RETURN
END

```

## Output

```

The zeros are
 1 2
1.646 -3.646

```

---

## NEQNF

Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

### Required Arguments

*FCN* — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is

```
CALL FCN (X, F, N), where
```

*X* — The point at which the functions are evaluated. (Input)  
*X* should not be changed by *FCN*.

*F* — The computed function values at the point *X*. (Output)

*N* — Length of *X* and *F*. (Input)

*FCN* must be declared EXTERNAL in the calling program.

*X* — A vector of length *N*. (Output)

*X* contains the best estimate of the root found by NEQNF.

## Optional Arguments

**ERRREL** — Stopping criterion. (Input)

The root is accepted if the relative error between two successive approximations to this root is less than **ERRREL**.

Default: **ERRREL** = 1.e-4 for single precision and 1.d-8 for double precision.

**N** — The number of equations to be solved and the number of unknowns. (Input)

Default: **N** = size (**X**,1).

**ITMAX** — The maximum allowable number of iterations. (Input)

The maximum number of calls to **FCN** is **ITMAX** \* (**N** + 1). Suggested value

**ITMAX** = 200.

Default: **ITMAX** = 200.

**XGUESS** — A vector of length **N**. (Input)

**XGUESS** contains the initial estimate of the root.

Default: **XGUESS** = 0.0.

**FNORM** — A scalar that has the value  $F(1)^2 + \dots + F(N)^2$  at the point **X**. (Output)

## FORTRAN 90 Interface

Generic:     CALL NEQNF (FCN, X [, ...])

Specific:    The specific interface names are S\_NEQNF and D\_NEQNF.

## FORTRAN 77 Interface

Single:       CALL NEQNF (FCN, ERRREL, N, ITMAX, XGUESS, X, FNORM)

Double:      The double precision name is DNEQNF.

## Description

Routine **NEQNF** is based on the **MINPACK** subroutine **HYBRD1**, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which uses a finite-difference approximation to the Jacobian and takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

Since a finite-difference method is used to estimate the Jacobian, for single precision calculation, the Jacobian may be so incorrect that the algorithm terminates far from a root. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, IMSL routine [NEQNJ](#) should be used instead.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of **N2QNF**/**DN2QNF**. The reference is:

```
CALL N2QNF (FCN, ERRREL, N, ITMAX, XGUESS, X, FNORM, FVEC,
 FJAC, R, QTF, WK)
```

The additional arguments are as follows:

**FVEC** — A vector of length  $N$ . **FVEC** contains the functions evaluated at the point  $x$ .

**FJAC** — An  $N$  by  $N$  matrix. **FJAC** contains the orthogonal matrix  $Q$  produced by the  $QR$  factorization of the final approximate Jacobian.

**R** — A vector of length  $N * (N + 1)/2$ . **R** contains the upper triangular matrix produced by the  $QR$  factorization of the final approximate Jacobian. **R** is stored row-wise.

**QTF** — A vector of length  $N$ . **QTF** contains the vector  $TRANS(Q) * FVEC$ .

**WK** — A work vector of length  $5 * N$ .

## 2. Informational errors

| Type | Code |                                                                                                      |
|------|------|------------------------------------------------------------------------------------------------------|
| 4    | 1    | The number of calls to <b>FCN</b> has exceeded $ITMAX * (N + 1)$ . A new initial guess may be tried. |
| 4    | 2    | <b>ERRREL</b> is too small. No further improvement in the approximate solution is possible.          |
| 4    | 3    | The iteration has not made good progress. A new initial guess may be tried.                          |

## Example

The following  $3 \times 3$  system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0$$

$$f_2(x) = e^{x_2-2} / x_1 + x_3^2 - 10 = 0$$

$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
USE NEQNF_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N
PARAMETER (N=3)
!
INTEGER K, NOUT
```

```

REAL FNORM, X(N), XGUESS(N)
EXTERNAL FCN
!
! Set values of initial guess
! XGUESS = (4.0 4.0 4.0)
!
!
DATA XGUESS/4.0, 4.0, 4.0/
!
!
CALL UMACH (2, NOUT)
!
! Find the solution
CALL NEQNF (FCN, X, xguess=xguess, fnorm=fnorm)
!
! Output
WRITE (NOUT,99999) (X(K),K=1,N), FNORM
99999 FORMAT (' The solution to the system is', /, ' X = (', 3F5.1, &
')', /, ' with FNORM =', F5.4, //)
!
END
!
! User-defined subroutine
SUBROUTINE FCN (X, F, N)
INTEGER N
REAL X(N), F(N)
!
REAL EXP, SIN
INTRINSIC EXP, SIN
!
F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
RETURN
END

```

## Output

```

The solution to the system is
X = (1.0 2.0 3.0)
with FNORM =.0000

```

---

## NEQNJ

Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.

### Required Arguments

*FCN* — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is

```
CALL FCN (X, F, N), where
```

X — The point at which the functions are evaluated. (Input)  
X should not be changed by FCN.

F — The computed function values at the point X. (Output)

$N$  – Length of  $X$ ,  $F$ . (Input)

$FCN$  must be declared `EXTERNAL` in the calling program.

**LSJAC** — User-supplied `SUBROUTINE` to evaluate the Jacobian at a point  $x$ . The usage is  
`CALL LSJAC (N, X, FJAC)`, where

$N$  – Length of  $X$ . (Input)

$X$  – The point at which the function is evaluated. (Input)  
 $X$  should not be changed by `LSJAC`.

$FJAC$  — The computed  $N$  by  $N$  Jacobian at the point  $x$ . (Output)

`LSJAC` must be declared `EXTERNAL` in the calling program.

$X$  — A vector of length  $N$ . (Output)  
 $X$  contains the best estimate of the root found by `NEQNJ`.

## Optional Arguments

**ERRREL** — Stopping criterion. (Input)  
The root is accepted if the relative error between two successive approximations to this root is less than `ERRREL`.  
Default: `ERRREL = 1.e-4` for single precision and `1.d-8` for double precision.

$N$  — The number of equations to be solved and the number of unknowns. (Input)  
Default: `N = size (X,1)`.

**ITMAX** — The maximum allowable number of iterations. (Input)  
Suggested value = 200.  
Default: `ITMAX = 200`.

**XGUESS** — A vector of length  $N$ . (Input)  
`XGUESS` contains the initial estimate of the root.  
Default: `XGUESS = 0.0`.

**FNORM** — A scalar that has the value  $F(1)^2 + \dots + F(N)^2$  at the point  $x$ . (Output)

## FORTRAN 90 Interface

Generic: `CALL NEQNJ (FCN, LSJAC, X [, ...])`

Specific: The specific interface names are `S_NEQNJ` and `D_NEQNJ`.

## FORTRAN 77 Interface

Single: `CALL NEQNJ (FCN, LSJAC, ERRREL, N, ITMAX, XGUESS, X, FNORM)`

Double: The double precision name is `DNEQNJ`.

## Description

Routine `NEQNJ` is based on the MINPACK subroutine `HYBRDJ`, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `N2QNJ/DN2QNJ`. The reference is:

```
CALL N2QNJ (FCN, LSJAC, ERRREL, N, ITMAX, XGUESS, X,
 FNORM, FVEC, FJAC, R, QTF, WK)
```

The additional arguments are as follows:

**FVEC** — A vector of length  $N$ . `FVEC` contains the functions evaluated at the point `X`.

**FJAC** — An  $N$  by  $N$  matrix. `FJAC` contains the orthogonal matrix `Q` produced by the QR factorization of the final approximate Jacobian.

**R** — A vector of length  $N * (N + 1) / 2$ . `R` contains the upper triangular matrix produced by the QR factorization of the final approximate Jacobian. `R` is stored row-wise.

**QTF** — A vector of length  $N$ . `QTF` contains the vector  $\text{TRANS}(Q) * \text{FVEC}$ .

**WK** — A work vector of length  $5 * N$ .

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                             |
|---|---|-------------------------------------------------------------------------------------------------------------|
| 4 | 1 | The number of calls to <code>FCN</code> has exceeded <code>ITMAX</code> . A new initial guess may be tried. |
|---|---|-------------------------------------------------------------------------------------------------------------|

|   |   |                                                                                                   |
|---|---|---------------------------------------------------------------------------------------------------|
| 4 | 2 | <code>ERRREL</code> is too small. No further improvement in the approximate solution is possible. |
|---|---|---------------------------------------------------------------------------------------------------|

|   |   |                                                                             |
|---|---|-----------------------------------------------------------------------------|
| 4 | 3 | The iteration has not made good progress. A new initial guess may be tried. |
|---|---|-----------------------------------------------------------------------------|

## Example

The following  $3 \times 3$  system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0$$

$$f_2(x) = e^{x_2-2} / x_1 + x_3^2 - 10 = 0$$

$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```

USE NEQNJ_INT
USE UMACH_INT

 IMPLICIT NONE
! Declare variables
 INTEGER N
 PARAMETER (N=3)
!
 INTEGER K, NOUT
 REAL FNORM, X(N), XGUESS(N)
 EXTERNAL FCN, LSJAC
! Set values of initial guess
! XGUESS = (4.0 4.0 4.0)
!
 DATA XGUESS/4.0, 4.0, 4.0/
!
!
 CALL UMACH (2, NOUT)
! Find the solution
 CALL NEQNJ (FCN, LSJAC, X, XGUESS=XGUESS, FNORM=FNORM)
! Output
 WRITE (NOUT,99999) (X(K),K=1,N), FNORM
99999 FORMAT (' The roots found are', /, ' X = (', 3F5.1, &
 ')', /, ' with FNORM = ',F5.4, //)
!
 END
! User-supplied subroutine
 SUBROUTINE FCN (X, F, N)
 INTEGER N
 REAL X(N), F(N)
!
 REAL EXP, SIN
 INTRINSIC EXP, SIN
!
 F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
 F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
 F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
 RETURN
 END
! User-supplied subroutine to
! compute Jacobian
 SUBROUTINE LSJAC (N, X, FJAC)
 INTEGER N
 REAL X(N), FJAC(N,N)
!
 REAL COS, EXP
 INTRINSIC COS, EXP
!

```

```

FJAC(1,1) = 1.0 + EXP(X(1)-1.0)
FJAC(1,2) = 2.0*(X(2)+X(3))
FJAC(1,3) = 2.0*(X(2)+X(3))
FJAC(2,1) = -EXP(X(2)-2.0)*(1.0/X(1)**2)
FJAC(2,2) = EXP(X(2)-2.0)*(1.0/X(1))
FJAC(2,3) = 2.0*X(3)
FJAC(3,1) = 0.0
FJAC(3,2) = COS(X(2)-2.0) + 2.0*X(2)
FJAC(3,3) = 1.0
RETURN
END

```

## Output

The roots found are  
X = ( 1.0 2.0 3.0)  
with FNORM =.0000

## NEQBF



Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.

### Required Arguments

*FCN* — User-supplied SUBROUTINE to evaluate the system of equations to be solved. The usage is

CALL FCN (N, X, F), where

N — Length of X and F. (Input)

X — The point at which the functions are evaluated. (Input)  
X should not be changed by FCN.

F — The computed function values at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Vector of length N containing the approximate solution. (Output)

### Optional Arguments

N — Dimension of the problem. (Input)  
Default: N = size (X,1).

**XGUESS** — Vector of length  $N$  containing initial guess of the root. (Input)

Default:  $XGUESS = 0.0$ .

**XSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the variables.

(Input)

$XSCALE$  is used mainly in scaling the distance between two points. In the absence of other information, set all entries to 1.0. If internal scaling is desired for  $XSCALE$ , set  $IPARAM$  (6) to 1.

Default:  $XSCALE = 1.0$ .

**FSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the functions.

(Input)

$FSCALE$  is used mainly in scaling the function residuals. In the absence of other information, set all entries to 1.0.

Default:  $FSCALE = 1.0$ .

**IPARAM** — Parameter vector of length 6. (Input/Output)

Set  $IPARAM$  (1) to zero for default values of  $IPARAM$  and  $RPARAM$ . See Comment 4.

Default:  $IPARAM = 0$ .

**RPARAM** — Parameter vector of length 5. (Input/Output)

See Comment 4.

**FVEC** — Vector of length  $N$  containing the values of the functions at the approximate solution. (Output)

## FORTRAN 90 Interface

Generic: `CALL NEQBF (FCN, X [, ...])`

Specific: The specific interface names are `S_NEQBF` and `D_NEQBF`.

## FORTRAN 77 Interface

Single: `CALL NEQBF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC)`

Double: The double precision name is `DNEQBF`.

## Description

Routine `NEQBF` uses a secant algorithm to solve a system of nonlinear equations, i.e.,

$$F(x) = 0$$

where  $F : \mathbf{R}^n \rightarrow \mathbf{R}^n$ , and  $x \in \mathbf{R}^n$ .

From a current point, the algorithm uses a double dogleg method to solve the following subproblem approximately:

$$\min_{s \in \mathbf{R}^n} \|F(x_c) + J(x_c)s\|_2$$

subject to  $\|s\|_2 \leq \delta_c$

to get a direction  $s_c$ , where  $F(x_c)$  and  $J(x_c)$  are the function values and the approximate Jacobian respectively evaluated at the current point  $x_c$ . Then, the function values at the point  $x_n = x_c + s_c$  are evaluated and used to decide whether the new point  $x_n$  should be accepted.

When the point  $x_n$  is rejected, this routine reduces the trust region  $\delta_c$  and goes back to solve the subproblem again. This procedure is repeated until a better point is found.

The algorithm terminates if the new point satisfies the stopping criterion. Otherwise,  $\delta_c$  is adjusted, and the approximate Jacobian is updated by Broyden's formula,

$$J_n = J_c + \frac{(y - J_c s_c) s_c^T}{s_c^T s_c}$$

where  $J_n = J(x_n)$ ,  $J_c = J(x_c)$ , and  $y = F(x_n) - F(x_c)$ . The algorithm then continues using the new point as the current point, i.e.  $x_c \leftarrow x_n$ .

For more details, see Dennis and Schnabel (1983, Chapter 8).

Since a finite-difference method is used to estimate the initial Jacobian, for single precision calculation, the Jacobian may be so incorrect that the algorithm terminates far from a root. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, IMSL routine [NEQBJ](#) should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of [N2QBF/DN2QBF](#). The reference is:

```
CALL N2QBF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM,
 RPARAM, X, FVEC, WK, LWK)
```

The additional arguments are as follows:

**WK** — A work vector of length `LWK`. On output `WK` contains the following information:

The third `N` locations contain the last step taken.

The fourth `N` locations contain the last Newton step.

The final `N2` locations contain an estimate of the Jacobian at the solution.

**LWK** — Length of `WK`, which must be at least  $2 * N^2 + 11 * N$ . (Input)

2. Informational errors

Type Code

- |   |   |                                                                                                                                                                                                                                                                   |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The last global step failed to decrease the 2-norm of $F(X)$ sufficiently; either the current point is close to a root of $F(X)$ and no more accuracy is possible, or the secant approximation to the Jacobian is inaccurate, or the step tolerance is too large. |
| 3 | 3 | The scaled distance between the last two steps is less than the step tolerance; the current point is probably an approximate root of $F(X)$ (unless <code>STEPTL</code> is too large).                                                                            |
| 3 | 4 | Maximum number of iterations exceeded.                                                                                                                                                                                                                            |
| 3 | 5 | Maximum number of function evaluations exceeded.                                                                                                                                                                                                                  |
| 3 | 7 | Five consecutive steps of length <code>STEPMX</code> have been taken; either the 2-norm of $F(X)$ asymptotes from above to a finite value in some direction or the maximum allowable step size <code>STEPMX</code> is too small.                                  |
3. The stopping criterion for `NEQBF` occurs when the scaled norm of the functions is less than the scaled function tolerance (`RPARAM(1)`).
4. If the default parameters are desired for `NEQBF`, then set `IPARAM(1)` to zero and call routine `NEQBF`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `NEQBF`:

```
CALL N4QBJ (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

**Note** that the call to `N4QBJ` will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

`IPARAM(4)` = Maximum number of function evaluations.  
Default: 400.

`IPARAM(5)` = Maximum number of Jacobian evaluations.  
Default: not used in `NEQBF`.

IPARAM(6) = Internal variable scaling flag.

If IPARAM(6) = 1, then the values of XSCALE are set internally.

Default: 0.

**RPARAM** — Real vector of length 5.

RPARAM(1) = Scaled function tolerance.

The scaled norm of the functions is computed as

$$\max_t (|f_i| * fs_i)$$

where  $f_i$  is the  $i$ -th component of the function vector  $F$ , and  $fs_i$  is the  $i$ -th component of  $FSCALE$ .

Default:

$$\sqrt{\varepsilon}$$

where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The scaled norm of the step between two points  $x$  and  $y$  is computed as

$$\max_t \left\{ \frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)} \right\}$$

where  $s_i$  is the  $i$ -th component of XSCALE.

Default:  $\varepsilon^{2/3}$ , where  $\varepsilon$  is the machine precision.

RPARAM(3) = False convergence tolerance.

Default: not used in NEQBF.

RPARAM(4) = Maximum allowable step size. (STEPMX)

Default:  $1000 * \max(\varepsilon_1, \varepsilon_2)$ , where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\varepsilon_2 = \|s\|_2$ ,  $s = XSCALE$ , and  $t = XGUESS$ .

RPARAM(5) = Size of initial trust region.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then DN4QBJ is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

### Example

The following  $3 \times 3$  system of nonlinear equations:

$$f_1(x) = x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0$$

$$f_2(x) = e^{x_2-2} / x_1 + x_3^2 - 10 = 0$$

$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```

USE NEQBF_INT
USE UMACH_INT

 IMPLICIT NONE
!
! Declare variables
 INTEGER N
 PARAMETER (N=3)
!
 INTEGER K, NOUT
 REAL X(N), XGUESS(N)
 EXTERNAL FCN
!
! Set values of initial guess
! XGUESS = (4.0 4.0 4.0)
!
 DATA XGUESS/3*4.0/
!
! Find the solution
 CALL NEQBF (FCN, X, XGUESS=XGUESS)
!
! Output
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) (X(K),K=1,N)
99999 FORMAT (' The solution to the system is', /, ' X = (', 3F8.3, &
 ')')
!
 END
!
! User-defined subroutine
 SUBROUTINE FCN (N, X, F)
 INTEGER N
 REAL X(N), F(N)
!
 REAL EXP, SIN
 INTRINSIC EXP, SIN
!
 F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
 F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
 F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
 RETURN
 END

```

## Output

The solution to the system is  
 $X = ( \quad 1.000 \quad 2.000 \quad 3.000 )$

---

## NEQBJ



Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.

### Required Arguments

**FCN** — User-supplied `SUBROUTINE` to evaluate the system of equations to be solved. The usage is

`CALL FCN (N, X, F)`, where

**N** — Length of **X** and **F**. (Input)

**X** — The point at which the functions are evaluated. (Input)  
**X** should not be changed by **FCN**.

**F** — The computed function values at the point **X**. (Output)

**FCN** must be declared `EXTERNAL` in the calling program.

**JAC** — User-supplied `SUBROUTINE` to evaluate the Jacobian at a point **X**. The usage is `CALL JAC (N, X, FJAC, LDFJAC)`, where

**N** — Length of **X**. (Input)

**X** — Vector of length **N** at which point the Jacobian is evaluated. (Input)  
**X** should not be changed by **JAC**.

**FJAC** — The computed **N** by **N** Jacobian at the point **X**. (Output)

**LDFJAC** — Leading dimension of **FJAC**. (Input)

**JAC** must be declared `EXTERNAL` in the calling program.

**X** — Vector of length **N** containing the approximate solution. (Output)

### Optional Arguments

**N** — Dimension of the problem. (Input)  
Default:  $N = \text{size}(X, 1)$ .

**XGUESS** — Vector of length  $N$  containing initial guess of the root. (Input)

Default:  $XGUESS = 0.0$ .

**XSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)

$XSCALE$  is used mainly in scaling the distance between two points. In the absence of other information, set all entries to 1.0. If internal scaling is desired for  $XSCALE$ , set  $IPARAM(6)$  to 1.

Default:  $XSCALE = 1.0$ .

**FSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the functions. (Input)

$FSCALE$  is used mainly in scaling the function residuals. In the absence of other information, set all entries to 1.0.

Default:  $FSCALE = 1.0$ .

**IPARAM** — Parameter vector of length 6. (Input/Output)

Set  $IPARAM(1)$  to zero for default values of  $IPARAM$  and  $RPARAM$ .

See Comment 4.

Default:  $IPARAM = 0$ .

**RPARAM** — Parameter vector of length 5. (Input/Output)

See Comment 4.

**FVEC** — Vector of length  $N$  containing the values of the functions at the approximate solution. (Output)

## FORTRAN 90 Interface

Generic: `CALL NEQBJ (FCN, JAC, X [, ...])`

Specific: The specific interface names are `S_NEQBJ` and `D_NEQBJ`.

## FORTRAN 77 Interface

Single: `CALL NEQBJ (FCN, JAC, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC)`

Double: The double precision name is `DNEQBJ`.

## Description

Routine `NEQBJ` uses a secant algorithm to solve a system of nonlinear equations, i. e.,

$$F(x) = 0$$

where  $F: \mathbf{R}^n \rightarrow \mathbf{R}^n$ , and  $x \in \mathbf{R}^n$ .

From a current point, the algorithm uses a double dogleg method to solve the following subproblem approximately:

$$\min_{s \in \mathbf{R}^n} \|F(x_c) + J(x_c)s\|_2$$

subject to  $\|s\|_2 \leq \delta_c$

to get a direction  $s_c$ , where  $F(x_c)$  and  $J(x_c)$  are the function values and the approximate Jacobian respectively evaluated at the current point  $x_c$ . Then, the function values at the point  $x_n = x_c + s_c$  are evaluated and used to decide whether the new point  $x_n$  should be accepted.

When the point  $x_n$  is rejected, this routine reduces the trust region  $\delta_c$  and goes back to solve the subproblem again. This procedure is repeated until a better point is found.

The algorithm terminates if the new point satisfies the stopping criterion. Otherwise,  $\delta_c$  is adjusted, and the approximate Jacobian is updated by Broyden's formula,

$$J_n = J_c + \frac{(y - J_c s_c) s_c^T}{s_c^T s_c}$$

where  $J_n = J(x_n)$ ,  $J_c = J(x_c)$ , and  $y = F(x_n) - F(x_c)$ . The algorithm then continues using the new point as the current point, i.e.  $x_c \leftarrow x_n$ .

For more details, see Dennis and Schnabel (1983, Chapter 8).

## Comments

1. Workspace may be explicitly provided, if desired, by use of N2QBJ/DN2QBJ. The reference is:

```
CALL N2QBJ (FCN, JAC, N, XGUESS, XSCALE, FSCALE, IPARAM,
 RPARAM, X, FVEC, WK, LWK)
```

The additional arguments are as follows:

**WK** — A work vector of length `LWK`. On output `WK` contains the following information: The third `N` locations contain the last step taken. The fourth `N` locations contain the last Newton step. The final `N2` locations contain an estimate of the Jacobian at the solution.

**LWK** — Length of `WK`, which must be at least  $2 * N^2 + 11 * N$ . (Input)

2. Informational errors

| Type | Code |                                                                                                                                                                                                                                                                   |
|------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | The last global step failed to decrease the 2-norm of $F(x)$ sufficiently; either the current point is close to a root of $F(x)$ and no more accuracy is possible, or the secant approximation to the Jacobian is inaccurate, or the step tolerance is too large. |
| 3    | 3    | The scaled distance between the last two steps is less than the step tolerance; the current point is probably an approximate root of $F(x)$ (unless <code>STEPTL</code> is too large).                                                                            |
| 3    | 4    | Maximum number of iterations exceeded.                                                                                                                                                                                                                            |

- 3    5        Maximum number of function evaluations exceeded.
- 3    7        Five consecutive steps of length `STEPMX` have been taken; either the 2-norm of  $F(X)$  asymptotes from above to a finite value in some direction or the maximum allowable stepsize `STEPMX` is too small.
3.    The stopping criterion for `NEQBJ` occurs when the scaled norm of the functions is less than the scaled function tolerance (`RPARAM(1)`).
4.    If the default parameters are desired for `NEQBJ`, then set `IPARAM(1)` to zero and call routine `NEQBJ`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `NEQBJ`:

```
CALL N4QBJ (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

**Note** that the call to `N4QBJ` will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

`IPARAM(4)` = Maximum number of function evaluations.  
Default: 400.

`IPARAM(5)` = Maximum number of Jacobian evaluations.  
Default: not used in `NEQBJ`.

`IPARAM(6)` = Internal variable scaling flag.  
If `IPARAM(6) = 1`, then the values of `XSCALE` are set internally.  
Default: 0.

***RPARAM*** — Real vector of length 5.

`RPARAM(1)` = Scaled function tolerance.  
The scaled norm of the functions is computed as

$$\max_t (|f_i| * fs_i)$$

where  $f_i$  is the  $i$ -th component of the function vector  $F$ , and  $f_i/s_i$  is the  $i$ -th component of  $FSCALE$ .

Default:

$$\sqrt{\epsilon}$$

where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The scaled norm of the step between two points  $x$  and  $y$  is computed as

$$\max_t \left\{ \frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)} \right\}$$

where  $s_i$  is the  $i$ -th component of  $XSCALE$ .

Default:  $\epsilon^{2/3}$ , where  $\epsilon$  is the machine precision.

RPARAM(3) = False convergence tolerance.

Default: not used in `NEQBJ`.

RPARAM(4) = Maximum allowable step size. (STEPMX)

Default:  $1000 * \max(\epsilon_1, \epsilon_2)$ , where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = XSCALE$ , and  $t = XGUESS$ .

RPARAM(5) = Size of initial trust region.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then `DN4QBJ` is called and `RPARAM` is declared double precision.

- Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

## Example

The following  $3 \times 3$  system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1-1} + (x_2 + x_3)^2 - 27 = 0$$

$$f_2(x) = e^{x_2-2} / x_1 + x_3^2 - 10 = 0$$

$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0$$

is solved with the initial guess (4.0, 4.0, 4.0).

```

USE NEQBJ_INT
USE UMACH_INT

 IMPLICIT NONE
! Declare variables
 INTEGER N
 PARAMETER (N=3)
!
 INTEGER K, NOUT
 REAL X(N), XGUESS(N)
 EXTERNAL FCN, JAC
! Set values of initial guess
! XGUESS = (4.0 4.0 4.0)
!
 DATA XGUESS/3*4.0/
! Find the solution
 CALL NEQBJ (FCN, JAC, X, XGUESS=XGUESS)
! Output
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) (X(K),K=1,N)
99999 FORMAT (' The solution to the system is', /, ' X = (', 3F8.3, &
 ')')
!
 END
! User-defined subroutine
 SUBROUTINE FCN (N, X, F)
 INTEGER N
 REAL X(N), F(N)
!
 REAL EXP, SIN
 INTRINSIC EXP, SIN
!
 F(1) = X(1) + EXP(X(1)-1.0) + (X(2)+X(3))*(X(2)+X(3)) - 27.0
 F(2) = EXP(X(2)-2.0)/X(1) + X(3)*X(3) - 10.0
 F(3) = X(3) + SIN(X(2)-2.0) + X(2)*X(2) - 7.0
 RETURN
 END
! User-supplied subroutine to
! compute Jacobian
 SUBROUTINE JAC (N, X, FJAC, LDFJAC)
 INTEGER N, LDFJAC
 REAL X(N), FJAC(LDFJAC,N)
!
 REAL COS, EXP
 INTRINSIC COS, EXP
!
 FJAC(1,1) = 1.0 + EXP(X(1)-1.0)
 FJAC(1,2) = 2.0*(X(2)+X(3))

```

```
FJAC(1,3) = 2.0*(X(2)+X(3))
FJAC(2,1) = -EXP(X(2)-2.0)*(1.0/X(1)**2)
FJAC(2,2) = EXP(X(2)-2.0)*(1.0/X(1))
FJAC(2,3) = 2.0*X(3)
FJAC(3,1) = 0.0
FJAC(3,2) = COS(X(2)-2.0) + 2.0*X(2)
FJAC(3,3) = 1.0
RETURN
END
```

## Output

The solution to the system is  
X = ( 1.000 2.000 3.000)

# Chapter 8: Optimization

---

## Routines

|                                                                                                       |                          |      |
|-------------------------------------------------------------------------------------------------------|--------------------------|------|
| <b>8.1. Unconstrained Minimization</b>                                                                |                          |      |
| 8.1.1 Univariate Function                                                                             |                          |      |
| Using function values only .....                                                                      | <a href="#">UVMIF</a>    | 1362 |
| Using function and first derivative values .....                                                      | <a href="#">UVMID</a>    | 1365 |
| Nonsmooth function .....                                                                              | <a href="#">UVMGS</a>    | 1369 |
| 8.1.2 Multivariate Function                                                                           |                          |      |
| Using finite-difference gradient .....                                                                | <a href="#">UMINF</a>    | 1372 |
| Using analytic gradient .....                                                                         | <a href="#">UMING</a>    | 1377 |
| Using finite-difference Hessian .....                                                                 | <a href="#">UMIDH</a>    | 1384 |
| Using analytic Hessian .....                                                                          | <a href="#">UMIAH</a>    | 1389 |
| Using conjugate gradient with finite-difference gradient.....                                         | <a href="#">UMCGF</a>    | 1395 |
| Using conjugate gradient with analytic gradient .....                                                 | <a href="#">UMCGG</a>    | 1399 |
| Nonsmooth function .....                                                                              | <a href="#">UMPOL</a>    | 1403 |
| 8.1.3 Nonlinear Least Squares                                                                         |                          |      |
| Using finite-difference Jacobian.....                                                                 | <a href="#">UNLSF</a>    | 1407 |
| Using analytic Jacobian .....                                                                         | <a href="#">UNLSJ</a>    | 1413 |
| <b>8.2. Minimization with Simple Bounds</b>                                                           |                          |      |
| Using finite-difference gradient .....                                                                | <a href="#">BCONF</a>    | 1420 |
| Using analytic gradient .....                                                                         | <a href="#">BCONG</a>    | 1427 |
| Using finite-difference Hessian .....                                                                 | <a href="#">BCODH</a>    | 1434 |
| Using analytic Hessian .....                                                                          | <a href="#">BCOAH</a>    | 1441 |
| Nonsmooth Function.....                                                                               | <a href="#">BCPOL</a>    | 1448 |
| Nonlinear least squares using finite-difference Jacobian ....                                         | <a href="#">BCLSF</a>    | 1452 |
| Nonlinear least squares using analytic Jacobian.....                                                  | <a href="#">BCLSJ</a>    | 1459 |
| Nonlinear least squares problem subject to bounds.....                                                | <a href="#">BCNLS</a>    | 1466 |
| <b>8.3. Linearly Constrained Minimization</b>                                                         |                          |      |
| Reads an MPS file containing a linear programming<br>problem or a quadratic programming problem ..... | <a href="#">READ_MPS</a> | 1475 |
| Deallocates the space allocated for the IMSL<br>derived type <code>s_MPS</code> .....                 | <a href="#">MPS_FREE</a> | 1485 |
| Dense linear programming .....                                                                        | <a href="#">DENSE_LP</a> | 1488 |
| Dense linear programming .....                                                                        | <a href="#">DLPRS</a>    | 1494 |
| Sparse linear programming .....                                                                       | <a href="#">SLPRS</a>    | 1497 |
| Solves a transportation problem .....                                                                 | <a href="#">TRAN</a>     | 1504 |

|             |                                                                |                       |      |
|-------------|----------------------------------------------------------------|-----------------------|------|
|             | Quadratic programming.....                                     | <a href="#">QPROG</a> | 1506 |
|             | General objective function with finite-difference gradient ... | <a href="#">LCONF</a> | 1510 |
|             | General objective function with analytic gradient.....         | <a href="#">LCONG</a> | 1516 |
| <b>8.4.</b> | <b>Nonlinearly Constrained Minimization</b>                    |                       |      |
|             | Using a sequential equality constrained QP method .....        | <a href="#">NNLPF</a> | 1522 |
|             | Using a sequential equality constrained QP method .....        | <a href="#">NNLPG</a> | 1528 |
| <b>8.5.</b> | <b>Service Routines</b>                                        |                       |      |
|             | Central-difference gradient.....                               | <a href="#">CDGRD</a> | 1536 |
|             | Forward-difference gradient .....                              | <a href="#">FDGRD</a> | 1538 |
|             | Forward-difference Hessian .....                               | <a href="#">FDHES</a> | 1541 |
|             | Forward-difference Hessian using analytic gradient .....       | <a href="#">GDHES</a> | 1543 |
|             | Divided-finite difference Jacobian .....                       | <a href="#">DDJAC</a> | 1546 |
|             | Forward-difference Jacobian.....                               | <a href="#">FDJAC</a> | 1555 |
|             | Check user-supplied gradient .....                             | <a href="#">CHGRD</a> | 1536 |
|             | Check user-supplied Hessian .....                              | <a href="#">CHHES</a> | 1561 |
|             | Check user-supplied Jacobian .....                             | <a href="#">CHJAC</a> | 1565 |
|             | Generate starting points .....                                 | <a href="#">GGUES</a> | 1569 |

---

## Usage Notes

### Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

where  $f: \mathbf{R}^n \rightarrow \mathbf{R}$  is at least continuous. The routines for unconstrained minimization are grouped into three categories: univariate functions (UV\*\*\*), multivariate functions (UM\*\*\*), and nonlinear least squares (UNLS\*).

For the univariate function routines, it is assumed that the function is unimodal within the specified interval. Otherwise, only a local minimum can be expected. For further discussion on unimodality, see Brent (1973).

A quasi-Newton method is used for the multivariate function routines [UMINF](#) and [UMING](#), whereas [UMIDH](#) and [UMIAH](#) use a modified Newton algorithm. The routines [UMCGF](#) and [UMCGG](#) make use of a conjugate gradient approach, and [UMPOL](#) uses a polytope method. For more details on these algorithms, see the documentation for the corresponding routines.

The nonlinear least squares routines use a modified Levenberg-Marquardt algorithm. If the nonlinear least squares problem is a nonlinear data-fitting problem, then software that is designed to deliver better statistical output may be useful; see IMSL (1991).

These routines are designed to find only a local minimum point. However, a function may have many local minima. It is often possible to obtain a better local solution by trying different initial points and intervals.

High precision arithmetic is recommended for the routines that use only function values. Also it is advised that the derivative-checking routines `CH***` be used to ensure the accuracy of the user-supplied derivative evaluation subroutines.

## Minimization with Simple Bounds

The minimization with simple bounds problem can be stated as follows:

$$\begin{aligned} \min_{x \in \mathbf{R}^n} f(x) \\ \text{subject to } l_i \leq x_i \leq u_i, \quad \text{for } i=1, 2, \dots, n \end{aligned}$$

where  $f: \mathbf{R}^n \rightarrow \mathbf{R}$ , and all the variables are not necessarily bounded.

The routines `BCO**` use the same algorithms as the routines `UMI**`, and the routines `BCLS*` are the corresponding routines of `UNLS*`. The only difference is that an active set strategy is used to ensure that each variable stays within its bounds. The routine `BCPOL` uses a function comparison method similar to the one used by `UMPOL`. Convergence for these polytope methods is not guaranteed; therefore, these routines should be used as a last alternative.

## Linearly Constrained Minimization

The linearly constrained minimization problem can be stated as follows:

$$\begin{aligned} \min_{x \in \mathbf{R}^n} f(x) \\ \text{subject to } Ax = b \end{aligned}$$

where  $f: \mathbf{R}^n \rightarrow \mathbf{R}$ ,  $A$  is an  $m \times n$  coefficient matrix, and  $b$  is a vector of length  $m$ . If  $f(x)$  is linear, then the problem is a linear programming problem; if  $f(x)$  is quadratic, the problem is a quadratic programming problem.

The routine `DLPRS` uses an active set strategy to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form. `SLPRS` uses the revised simplex method to solve large linear programming problems, which have sparse constraints matrices. `TRAN` solves a transportation problem, which is a very sparse linear programming application.

`QPROG` is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then `QPROG` modifies it to be positive definite. In this case, output should be interpreted with care.

The routines `LCONF` and `LCONG` use an iterative method to solve the linearly constrained problem with a general objective function. For a detailed description of the algorithm, see Powell (1988, 1989).

## Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\begin{aligned} \text{subject to } g_i(x) &= 0, & \text{for } i = 1, 2, \dots, m_1 \\ g_i(x) &\geq 0, & \text{for } i = m_1 + 1, \dots, m_1 \end{aligned}$$

where  $f: \mathbf{R}^n \rightarrow \mathbf{R}$  and  $g_i: \mathbf{R} \rightarrow \mathbf{R}$ , for  $i = 1, 2, \dots, m$

The routines [NNLPP](#) and [NNLPG](#) use a sequential equality constrained quadratic programming method. A more complete discussion of this algorithm can be found in the documentation.

## Selection of Routines

The following general guidelines are provided to aid in the selection of the appropriate routine.

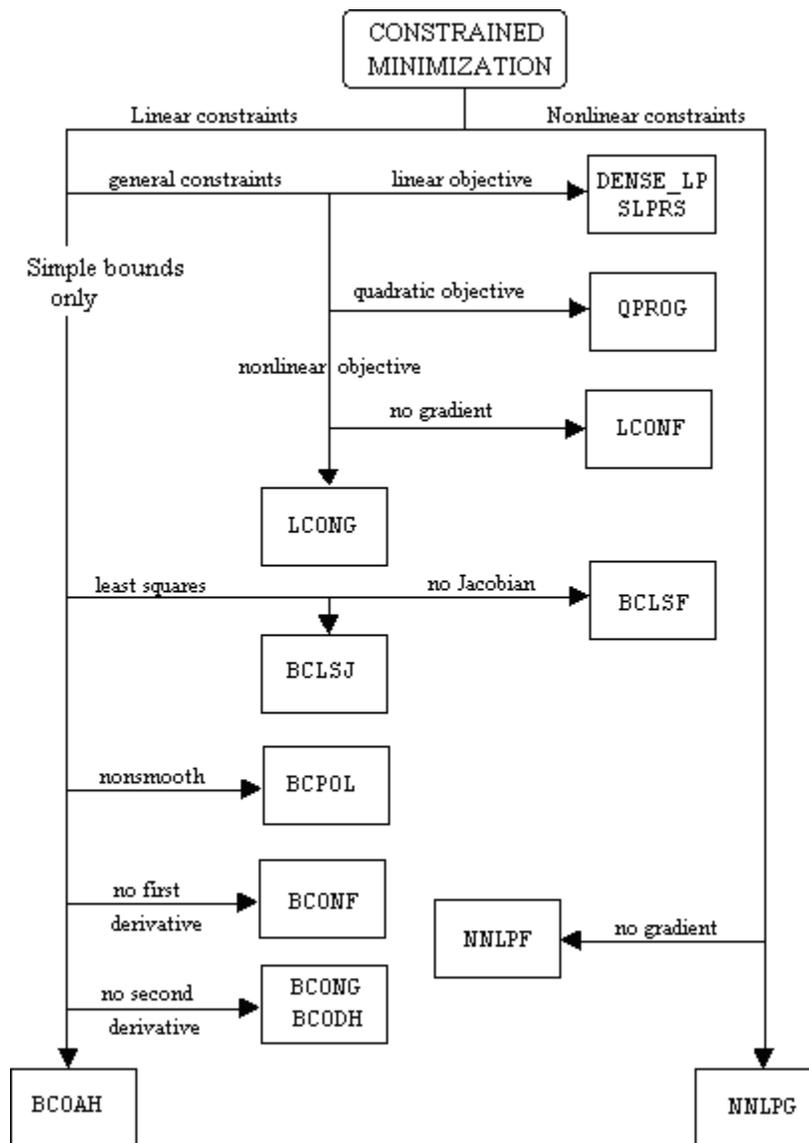
### Unconstrained Minimization

1. For the univariate case, use [UVMID](#) when the gradient is available, and use [UVMIF](#) when it is not. If discontinuities exist, then use [UVMGS](#).
2. For the multivariate case, use [UMCG\\*](#) when storage is a problem, and use [UMPOL](#) when the function is nonsmooth. Otherwise, use [UMI\\*\\*](#) depending on the availability of the gradient and the Hessian.
3. For least squares problems, use [UNLSJ](#) when the Jacobian is available, and use [UNLSF](#) when it is not.

### Minimization with Simple Bounds

1. Use [BCONF](#) when only function values are available. When first derivatives are available, use either [BCONG](#) or [BCODH](#). If first and second derivatives are available, then use [BCOAH](#).
2. For least squares, use [BCLSF](#) or [BCLSJ](#) depending on the availability of the Jacobian.
3. Use [BCPOL](#) for nonsmooth functions that could not be solved satisfactorily by the other routines.





## UVMIF

Finds the minimum point of a smooth function of a single variable using only function evaluations.

### Required Arguments

$F$  — User-supplied function to compute the value of the function to be minimized. The form is  $F(X)$ , where

$X$  — The point at which the function is evaluated. (Input)  
 $X$  should not be changed by  $F$ .

$F$  — The computed function value at the point  $X$ . (Output)

$F$  must be declared `EXTERNAL` in the calling program.

***XGUESS*** — An initial guess of the minimum point of  $F$ . (Input)

***BOUND*** — A positive number that limits the amount by which  $X$  may be changed from its initial value. (Input)

***X*** — The point at which a minimum value of  $F$  is found. (Output)

### Optional Arguments

***STEP*** — An order of magnitude estimate of the required change in  $X$ . (Input)  
Default: `STEP = 1.0`.

***XACC*** — The required absolute accuracy in the final value of  $X$ . (Input)  
On a normal return there are points on either side of  $X$  within a distance `XACC` at which  $F$  is no less than  $F(X)$ .  
Default: `XACC = 1.e-4`.

***MAXFN*** — Maximum number of function evaluations allowed. (Input)  
Default: `MAXFN = 1000`.

### FORTRAN 90 Interface

Generic: `CALL UVMIF (F, XGUESS, BOUND, X [, ...])`

Specific: The specific interface names are `S_UVMIF` and `D_UVMIF`.

### FORTRAN 77 Interface

Single: `CALL UVMIF (F, XGUESS, STEP, BOUND, XACC, MAXFN, X)`

Double: The double precision name is `DUVMIF`.

### Description

The routine `UVMIF` uses a safeguarded quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the routine `ZXLSE` written by M.J.D. Powell at the University of Cambridge.

The routine `UVMIF` finds the least value of a univariate function,  $f$ , that is specified by the function subroutine `F`. Other required data include an initial estimate of the solution, `XGUESS`, and a positive number `BOUND`. Let  $x_0 = \text{XGUESS}$  and  $b = \text{BOUND}$ , then  $x$  is restricted to the interval  $[x_0 - b, x_0 + b]$ . Usually, the algorithm begins the search by moving from  $x_0$  to  $x = x_0 + s$ , where

$s = \text{STEP}$  is also provided by the user and may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until  $x$  reaches one of the bounds  $x_0 \pm b$ . During this stage, the step length increases by a factor of between two and nine per function evaluation; the factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, we will have three points,  $x_1$ ,  $x_2$ , and  $x_3$ , with  $x_1 < x_2 < x_0$  and  $f(x_2) \leq f(x_1)$  and  $f(x_2) \leq f(x_3)$ . There are three main ingredients in the technique for choosing the new  $x$  from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter  $\epsilon$ , that depends on the closeness of  $f$  to a quadratic, and (iii) whether  $x_2$  is near the center of the range between  $x_1$  and  $x_3$  or is relatively close to an end of this range. In outline, the new value of  $x$  is as near as possible to the predicted minimum point, subject to being at least  $\epsilon$  from  $x_2$ , and subject to being in the longer interval between  $x_1$  and  $x_2$  or  $x_2$  and  $x_3$  when  $x_2$  is particularly close to  $x_1$  or  $x_3$ . There is some elaboration, however, when the distance between these points is close to the required accuracy; when the distance is close to the machine precision; or when  $\epsilon$  is relatively large.

The algorithm is intended to provide fast convergence when  $f$  has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as

$$f(x) = x + 1.001|x|$$

The algorithm can make  $\epsilon$  large automatically in the pathological cases. In this case, it is usual for a new value of  $x$  to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to  $f$  are dominated by computer rounding errors, which will almost certainly happen if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the routine claims to have achieved the required accuracy if it knows that there is a local minimum point within distance  $\delta$  of  $x$ , where  $\delta = \text{XACC}$ , even though the rounding errors in  $f$  may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high precision arithmetic is recommended.

## Comments

Informational errors

| Type | Code |                                                                               |
|------|------|-------------------------------------------------------------------------------|
| 3    | 1    | Computer rounding errors prevent further refinement of X.                     |
| 3    | 2    | The final value of X is at a bound. The minimum is probably beyond the bound. |
| 4    | 3    | The number of function evaluations has exceeded MAXFN.                        |

## Example

A minimum point of  $e^x - 5x$  is found.

```

USE UVMIF_INT
USE UMACH_INT

IMPLICIT NONE

! Declare variables
INTEGER MAXFN, NOUT
REAL BOUND, F, FX, STEP, X, XACC, XGUESS
EXTERNAL F

! Initialize variables
XGUESS = 0.0
XACC = 0.001
BOUND = 100.0
STEP = 0.1
MAXFN = 50

!
! Find minimum for F = EXP(X) - 5X
CALL UVMIF (F, XGUESS, BOUND, X, STEP=STEP, XACC=XACC, MAXFN=MAXFN)
FX = F(X)

! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, FX

!
99999 FORMAT (' The minimum is at ', 7X, F7.3, '//, ' The function ' &
, 'value is ', F7.3)

!
END

! Real function: F = EXP(X) - 5.0*X
REAL FUNCTION F (X)
REAL X

!
REAL EXP
INTRINSIC EXP

!
F = EXP(X) - 5.0E0*X

!
RETURN
END

```

## Output

The minimum is at 1.609

The function value is -3.047

---

## UVMID

Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.

### Required Arguments

*F* — User-supplied function to define the function to be minimized. The form is  $F(X)$ , where

*X* — The point at which the function is to be evaluated. (Input)

**F** — The computed value of the function at  $X$ . (Output)

$F$  must be declared `EXTERNAL` in the calling program.

**G** — User-supplied function to compute the derivative of the function. The form is  $G(X)$ , where

**X** — The point at which the derivative is to be computed. (Input)

**G** — The computed value of the derivative at  $X$ . (Output)

$G$  must be declared `EXTERNAL` in the calling program.

**A** —  $A$  is the lower endpoint of the interval in which the minimum point of  $F$  is to be located. (Input)

**B** —  $B$  is the upper endpoint of the interval in which the minimum point of  $F$  is to be located. (Input)

**X** — The point at which a minimum value of  $F$  is found. (Output)

### Optional Arguments

**XGUESS** — An initial guess of the minimum point of  $F$ . (Input)

Default:  $XGUESS = (a + b) / 2.0$ .

**ERRREL** — The required relative accuracy in the final value of  $X$ . (Input)

This is the first stopping criterion. On a normal return, the solution  $X$  is in an interval that contains a local minimum and is less than or equal to  $\text{MAX}(1.0, \text{ABS}(X)) * \text{ERRREL}$ . When the given **ERRREL** is less than machine epsilon,  $\text{SQRT}(\text{machine epsilon})$  is used as **ERRREL**.

Default:  $\text{ERRREL} = 1.e-4$ .

**GTOL** — The derivative tolerance used to decide if the current point is a local minimum.

(Input)

This is the second stopping criterion.  $X$  is returned as a solution when  $G_X$  is less than or equal to **GTOL**. **GTOL** should be nonnegative, otherwise zero would be used.

Default:  $\text{GTOL} = 1.e-4$ .

**MAXFN** — Maximum number of function evaluations allowed. (Input)

Default:  $\text{MAXFN} = 1000$ .

**FX** — The function value at point  $X$ . (Output)

**GX** — The derivative value at point  $X$ . (Output)

### FORTRAN 90 Interface

Generic: `CALL UVMID (F, G, A, B, X [, ...])`

Specific: The specific interface names are S\_UVMID and D\_UVMID.

## FORTRAN 77 Interface

Single: CALL UVMID (F, G, XGUESS, ERRREL, GTOL, MAXFN, A, B, X, FX, GX)

Double: The double precision name is DUVMID.

## Description

The routine UVMID uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the routine terminates with a solution. Otherwise, the point with least function value will be used as the starting point.

From the starting point, say  $x_c$ , the function value  $f_c = f(x_c)$ , the derivative value  $g_c = g(x_c)$ , and a new point  $x_n$  defined by  $x_n = x_c - g_c$  are computed. The function  $f_n = f(x_n)$ , and the derivative  $g_n = g(x_n)$  are then evaluated. If either  $f_n \geq f_c$  or  $g_n$  has the opposite sign of  $g_c$ , then there exists a minimum point between  $x_c$  and  $x_n$ ; and an initial interval is obtained. Otherwise, since  $x_c$  is kept as the point that has lowest function value, an interchange between  $x_n$  and  $x_c$  is performed. The secant method is then used to get a new point

$$x_s = x_c - g_c \left( \frac{g_n - g_c}{x_n - x_c} \right)$$

Let  $x_n \leftarrow x_s$  and repeat this process until an interval containing a minimum is found or one of the convergence criteria is satisfied. The convergence criteria are as follows:

Criterion 1:

$$|x_c - x_n| \leq \epsilon_c$$

Criterion 2:

$$|g_c| \leq \epsilon_g$$

where  $\epsilon_c = \max\{1.0, |x_c|\}\epsilon$ ,  $\epsilon$  is a relative error tolerance and  $\epsilon_g$  is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. Function and derivative are then evaluated at that point; and accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval reduces by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this procedure is repeated until one of the stopping criteria is met.

## Comments

Informational errors

Type Code

|   |   |                                                                                       |
|---|---|---------------------------------------------------------------------------------------|
| 3 | 1 | The final value of X is at the lower bound. The minimum is probably beyond the bound. |
|---|---|---------------------------------------------------------------------------------------|

- 3 2 The final value of X is at the upper bound. The minimum is probably beyond the bound.
- 4 3 The maximum number of function evaluations has been exceeded.

### Example

A minimum point of  $e^x - 5x$  is found.

```

USE UVMID_INT
USE UMACH_INT

IMPLICIT NONE

! Declare variables
INTEGER MAXFN, NOUT
REAL A, B, ERRREL, F, FX, G, GTOL, GX, X, XGUESS, FTOL
EXTERNAL F, G

! Initialize variables
XGUESS = 0.0

! Set ERRREL to zero in order
! to use SQRT(machine epsilon)
! as relative error
ERRREL = 0.0
GTOL = 0.0
A = -10.0
B = 10.0
MAXFN = 50

!
! Find minimum for F = EXP(X) - 5X
CALL UVMID (F, G, A, B, X, XGUESS=XGUESS, ERRREL=ERRREL, &
 GTOL=FTOL, MAXFN=MAXFN, FX=FX, GX=GX)

! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, FX, GX

!
99999 FORMAT (' The minimum is at ', 7X, F7.3, '//, ' The function ' &
 , 'value is ', F7.3, '//, ' The derivative is ', F7.3)
!
END

! Real function: F = EXP(X) - 5.0*X
REAL FUNCTION F (X)
REAL X

!
REAL EXP
INTRINSIC EXP

!
F = EXP(X) - 5.0E0*X

!
RETURN
END

!
REAL FUNCTION G (X)
REAL X

!
REAL EXP

```

```

 INTRINSIC EXP
!
 G = EXP(X) - 5.0E0
 RETURN
 END

```

### Output

```

The minimum is at 1.609

The function value is -3.047

The derivative is -0.001

```

## UVMGS

Finds the minimum point of a nonsmooth function of a single variable.

### Required Arguments

**F** — User-supplied function to compute the value of the function to be minimized. The form is  $F(X)$ , where

**X** — The point at which the function is evaluated. (Input)  
**X** should not be changed by **F**.

**F** — The computed function value at the point **X**. (Output)

**F** must be declared **EXTERNAL** in the calling program.

**A** — On input, **A** is the lower endpoint of the interval in which the minimum of **F** is to be located. On output, **A** is the lower endpoint of the interval in which the minimum of **F** is located. (Input/Output)

**B** — On input, **B** is the upper endpoint of the interval in which the maximum of **F** is to be located. On output, **B** is the upper endpoint of the interval in which the minimum of **F** is located. (Input/Output)

**XMIN** — The approximate minimum point of the function **F** on the original interval (**A**, **B**). (Output)

### Optional Arguments

**TOL** — The allowable length of the final subinterval containing the minimum point. (Input)  
 Default: **TOL** = 1.e-4.

### FORTRAN 90 Interface

Generic:    `CALL UVMGS (F, A, B, XMIN [, ...])`

Specific: The specific interface names are S\_UVMGS and D\_UVMGS.

## FORTRAN 77 Interface

Single: CALL UVMGS (F, A, B, TOL, XMIN)

Double: The double precision name is DUVMGS.

## Description

The routine UVMGS uses the *golden section search* technique to compute to the desired accuracy the independent variable value that minimizes a unimodal function of one independent variable, where a known finite interval contains the minimum.

Let  $\tau = \text{TOL}$ . The number of iterations required to compute the minimizing value to accuracy  $\tau$  is the greatest integer less than or equal to

$$\frac{\ln(\tau/(b-a))}{\ln(1-c)} + 1$$

where  $a$  and  $b$  define the interval and

$$c = (3 - \sqrt{5})/2$$

The first two test points are  $v_1$  and  $v_2$  that are defined as

$$v_1 = a + c(b-a), \text{ and } v_2 = b - c(b-a)$$

If  $f(v_1) < f(v_2)$ , then the minimizing value is in the interval  $(a, v_2)$ . In this case,  $b \leftarrow v_2$ ,  $v_2 \leftarrow v_1$ , and  $v_1 \leftarrow a + c(b-a)$ . If  $f(v_1) \geq f(v_2)$ , the minimizing value is in  $(v_1, b)$ . In this case,  $a \leftarrow v_1$ ,  $v_1 \leftarrow v_2$ , and  $v_2 \leftarrow b - c(b-a)$ .

The algorithm continues in an analogous manner where only one new test point is computed at each step. This process continues until the desired accuracy  $\tau$  is achieved. XMIN is set to the point producing the minimum value for the current iteration.

Mathematically, the algorithm always produces the minimizing value to the desired accuracy; however, numerical problems may be encountered. If  $f$  is too flat in part of the region of interest, the function may appear to be constant to the computer in that region. Error code 2 indicates that this problem has occurred. The user may rectify the problem by relaxing the requirement on  $\tau$ , modifying (scaling, etc.) the form of  $f$  or executing the program in a higher precision.

## Comments

1. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |  |                                   |
|---|--|-----------------------------------|
| 3 |  | TOL is too small to be satisfied. |
|---|--|-----------------------------------|

|   |   |                                                          |
|---|---|----------------------------------------------------------|
| 4 | 2 | Due to rounding errors F does not appear to be unimodal. |
|---|---|----------------------------------------------------------|

2. On exit from UVMGS without any error messages, the following conditions hold:  
 $(B-A) \leq \text{TOL}$ .  
 $A \leq \text{XMIN}$  and  $\text{XMIN} \leq B$   
 $F(\text{XMIN}) \leq F(A)$  and  $F(\text{XMIN}) \leq F(B)$
  
3. On exit from UVMGS with error code 2, the following conditions hold:  
 $A \leq \text{XMIN}$  and  $\text{XMIN} \leq B$   
 $F(\text{XMIN}) \geq F(A)$  and  $F(\text{XMIN}) \geq F(B)$  (only one equality can hold).  
 Further analysis of the function  $F$  is necessary in order to determine whether it is not unimodal in the mathematical sense or whether it appears to be not unimodal to the routine due to rounding errors in which case the  $A$ ,  $B$ , and  $\text{XMIN}$  returned may be acceptable.

### Example

A minimum point of  $3x^2 - 2x + 4$  is found.

```

USE UVMGS_INT
USE UMACH_INT

 IMPLICIT NONE
! Specification of variables
 INTEGER NOUT
 REAL A, B, FCN, FMIN, TOL, XMIN
 EXTERNAL FCN
! Initialize variables
 A = 0.0E0
 B = 5.0E0
 TOL = 1.0E-3
! Minimize FCN
 CALL UVMGS (FCN, A, B, XMIN, TOL=TOL)
 FMIN = FCN(XMIN)
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) XMIN, FMIN, A, B
99999 FORMAT (' The minimum is at ', F5.3, '//, ' The ', &
 'function value is ', F5.3, '//, ' The final ', &
 'interval is (', F6.4, ', ', F6.4, ')', /)
!
 END
!
! REAL FUNCTION: F = 3*X**2 - 2*X + 4
 REAL FUNCTION FCN (X)
 REAL X
!
 FCN = 3.0E0*X*X - 2.0E0*X + 4.0E0
!
 RETURN
 END

```

### Output

The minimum is at 0.333

The function value is 3.667

The final interval is (0.3331,0.3340)

---

## UMINF

Minimizes a function of  $N$  variables using a quasi-Newton method and a finite-difference gradient.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is

CALL FCN (N, X, F), where

$N$  — Length of  $X$ . (Input)

$X$  — The point at which the function is evaluated. (Input)  
 $X$  should not be changed by FCN.

$F$  — The computed function value at the point  $X$ . (Output)

FCN must be declared EXTERNAL in the calling program.

$X$  — Vector of length  $N$  containing the computed solution. (Output)

### Optional Arguments

$N$  — Dimension of the problem. (Input)

Default:  $N = \text{SIZE}(X,1)$ .

**XGUESS** — Vector of length  $N$  containing an initial guess of the computed solution. (Input)

Default:  $XGUESS = 0.0$ .

**XSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the variables.

(Input)

XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.

Default:  $XSCALE = 1.0$ .

**FSCALE** — Scalar containing the function scaling. (Input)

FSCALE is used mainly in scaling the gradient. In the absence of other information, set FSCALE to 1.0.

Default:  $FSCALE = 1.0$ .

**IPARAM** — Parameter vector of length 7. (Input/Output)

Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.

Default:  $IPARAM = 0$ .

**RPARAM** — Parameter vector of length 7.(Input/Output)

See Comment 4.

*FVALUE* — Scalar containing the value of the function at the computed solution. (Output)

### **FORTRAN 90 Interface**

Generic:    CALL UMINF (FCN, X [, ...])

Specific:    The specific interface names are S\_UMINF and D\_UMINF.

### **FORTRAN 77 Interface**

Single:     CALL UMINF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X,  
                  FVALUE)

Double:     The double precision name is DUMINF.

### **Description**

The routine UMINF uses a quasi-Newton method to find the minimum of a function  $f(x)$  of  $n$  variables. Only function values are required. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point  $x_c$ , the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where  $B$  is a positive definite approximation of the Hessian and  $g_c$  is the gradient evaluated at  $x_c$ . A line search is then used to find a new point

$$x_n = x_c + \lambda d, \quad \lambda > 0$$

such that

$$f(x_n) \leq f(x_c) + \alpha g_c^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality condition  $\|g(x)\| = \epsilon$  is checked where  $\epsilon$  is a gradient tolerance.

When optimality is not achieved,  $B$  is updated according to the BFGS formula

$$B \leftarrow B - \frac{B s s^T B}{s^T B s} + \frac{y y^T}{y^T s}$$

where  $s = x_n - x_c$  and  $y = g_n - g_c$ . Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

Since a finite-difference method is used to estimate the gradient, for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, IMSL routine [UMING](#) should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `U2INF/DU2INF`. The reference is:

```
CALL U2INF (FCN, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X,
FVALUE, WK)
```

The additional argument is:

**WK** — Work vector of length  $N(N + 8)$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Cholesky factorization of a BFGS approximation to the Hessian at the solution.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4 | 5 | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 3 | 8 | The last global step failed to locate a lower point than the current $x$ value.                                                                                                                       |

3. The first stopping criterion for `UMINF` occurs when the infinity norm of the scaled gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `UMINF` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).
4. If the default parameters are desired for `UMINF`, then set `IPARAM(1)` to zero and call the routine `UMINF`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UMINF`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

Note that the call to `U4INF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter.  
If IPARAM(6) = 0, the Hessian is initialized to the identity matrix;  
otherwise, it is initialized to a diagonal matrix containing

$$\max(|f(t)|, f_s) * s_i^2$$

on the diagonal where  $t = \text{XGUESS}$ ,  $f_s = \text{FSCALE}$ , and  $s = \text{XSCALE}$ .  
Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: Not used in UMINF.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  
 $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: Not used in UMINF.

RPARAM(4) = Absolute function tolerance

Default: Not used in UMINF.

RPARAM(5) = False convergence tolerance.

Default: Not used in UMINF.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}, \epsilon_2 = \|s\|_2, s = \text{XSCALE}, \text{ and } t = \text{XGUESS}$$

RPARAM(7) = Size of initial trust region radius.

Default: Not used in UMINF.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

## Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized.

```
USE UMINF_INT
USE U4INF_INT
USE UMACH_INT

IMPLICIT NONE
```

```

 INTEGER N
 PARAMETER (N=2)
!
 INTEGER IPARAM(7), L, NOUT
 REAL F, RPARAM(7), X(N), XGUESS(N), &
 XSCALE(N)
 EXTERNAL ROSBRK
!
 DATA XGUESS/-1.2E0, 1.0E0/
!
! Relax gradient tolerance stopping
! criterion
 CALL U4INF (IPARAM, RPARAM)
 RPARAM(1) = 10.0E0*RPARAM(1)
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
 CALL UMINF (ROSBRK, X, XGUESS=XGUESS, IPARAM=IPARAM, RPARAM=RPARAM, &
 FVALUE=F)
!
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
 'value is ', F8.3, '//, ' The number of iterations is ', &
 10X, I3, '/', ' The number of function evaluations is ', &
 I3, '/', ' The number of gradient evaluations is ', I3)
!
 END
!
 SUBROUTINE ROSBRK (N, X, F)
 INTEGER N
 REAL X(N), F
!
 F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
 RETURN
 END

```

## Output

```

The solution is 1.000 1.000

The function value is 0.000

The number of iterations is 15
The number of function evaluations is 40
The number of gradient evaluations is 19

```

---

## UMING

Minimizes a function of  $N$  variables using a quasi-Newton method and a user-supplied gradient.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Length of X and G. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by GRAD.

G — The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

## Optional Arguments

N — Dimension of the problem. (Input)  
Default: N = SIZE (X,1).

**XGUESS** — Vector of length N containing the initial guess of the minimum. (Input)  
Default: XGUESS = 0.0.

**XSCALE** — Vector of length N containing the diagonal scaling matrix for the variables.  
(Input)  
XSCALE is used mainly in scaling the gradient and the distance between two points. In  
the absence of other information, set all entries to 1.0.  
Default: XSCALE = 1.0.

**FSCALE** — Scalar containing the function scaling. (Input)  
FSCALE is used mainly in scaling the gradient. In the absence of other information, set  
FSCALE to 1.0.  
Default: FSCALE = 1.0.

**IPARAM** — Parameter vector of length 7. (Input/Output)  
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.  
Default: IPARAM = 0.

**RPARAM** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

**FVALUE** — Scalar containing the value of the function at the computed solution. (Output)

### **FORTRAN 90 Interface**

Generic: CALL UMING (FCN, GRAD, X [, ...])

Specific: The specific interface names are S\_UMING and D\_UMING.

### **FORTRAN 77 Interface**

Single: CALL UMING (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X,  
FVALUE)

Double: The double precision name is DUMING.

### **Description**

The routine UMING uses a quasi-Newton method to find the minimum of a function  $f(x)$  of  $n$  variables. Function values and first derivatives are required. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point  $x_c$ , the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where  $B$  is a positive definite approximation of the Hessian and  $g_c$  is the gradient evaluated at  $x_c$ . A line search is then used to find a new point

$$x_n = x_c + \lambda d, \quad \lambda > 0$$

such that

$$f(x_n) \leq f(x_c) + \alpha g_c^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality condition  $\|g(x)\| = \epsilon$  is checked where  $\epsilon$  is a gradient tolerance.

When optimality is not achieved,  $B$  is updated according to the BFGS formula

$$B \leftarrow B - \frac{B s s^T B}{s^T B s} + \frac{y y^T}{y^T s}$$

where  $s = x_n - x_c$  and  $y = g_n - g_c$ . Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `U2ING/DU2ING`. The reference is:

```
CALL U2ING (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM,
 RPARAM, X, FVALUE, WK)
```

The additional argument is

**WK** — Work vector of length  $N * (N + 8)$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Cholesky factorization of a `BFGS` approximation to the Hessian at the solution.

2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4 | 5 | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 3 | 8 | The last global step failed to locate a lower point than the current $x$ value.                                                                                                                       |

3. The first stopping criterion for `UMING` occurs when the infinity norm of the scaled gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `UMING` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).
4. If the default parameters are desired for `UMING`, then set `IPARAM(1)` to zero and call routine `UMING`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UMING`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter  
If IPARAM(6) = 0, the Hessian is initialized to the identity matrix;  
otherwise, it is initialized to a diagonal matrix containing

$$\max(|f(t)|, f_s) * s_i^2$$

on the diagonal where  $t = \text{XGUESS}$ ,  $f_s = \text{FSCALE}$ , and  $s = \text{XSCALE}$ .  
Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: Not used in UMING.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: Not used in UMING.

RPARAM(4) = Absolute function tolerance.

Default: Not used in UMING.

RPARAM(5) = False convergence tolerance.

Default: Not used in UMING.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: Not used in UMING.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

## Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized. Default values for parameters are used.

```

USE UMING_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
!
INTEGER IPARAM(7), L, NOUT
REAL F, X(N), XGUESS(N)
EXTERNAL ROSBRK, ROSGRD
!
DATA XGUESS/-1.2E0, 1.0E0/
!
IPARAM(1) = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL UMING (ROSBRK, ROSGRD, X, XGUESS=XGUESS, IPARAM=IPARAM, FVALUE=F)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
' value is ', F8.3, '//, ' The number of iterations is ', &
10X, I3, '/', ' The number of function evaluations is ', &
I3, '/', ' The number of gradient evaluations is ', I3)
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
RETURN
END
!
SUBROUTINE ROSGRD (N, X, G)
INTEGER N
REAL X(N), G(N)
!
G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
RETURN
END

```

## Output

```

The solution is 1.000 1.000

The function value is 0.000

The number of iterations is 18
The number of function evaluations is 31
The number of gradient evaluations is 22

```

---

## UMIDH

Minimizes a function of  $N$  variables using a modified Newton method and a finite-difference Hessian.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Length of X and G. (Input)

X — The point at which the gradient is evaluated. (Input)  
X should not be changed by GRAD.

G — The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

### Optional Arguments

N — Dimension of the problem. (Input)  
Default: N = SIZE (X,1).

**XGUESS** — Vector of length N containing initial guess. (Input)  
Default: XGUESS = 0.0.

**XSCALE** — Vector of length N containing the diagonal scaling matrix for the variables. (Input)  
XSCALE is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.  
Default: XSCALE = 1.0.

**FSCALE** — Scalar containing the function scaling. (Input)  
FSCALE is used mainly in scaling the gradient. In the absence of other information, set

FSCALE to 1.0.  
Default: FSCALE = 1.0.

**IPARAM** — Parameter vector of length 7. (Input/Output)  
Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.  
Default: IPARAM = 0.

**RPARAM** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

**FVALUE** — Scalar containing the value of the function at the computed solution. (Output)

## FORTRAN 90 Interface

Generic: CALL UMIDH (FCN, GRAD, X [, ...])

Specific: The specific interface names are S\_UMIDH and D\_UMIDH.

## FORTRAN 77 Interface

Single: CALL UMIDH (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X,  
FVALUE)

Double: The double precision name is DUMIDH.

## Description

The routine UMIDH uses a modified Newton method to find the minimum of a function  $f(x)$  of  $n$  variables. First derivatives must be provided by the user. The algorithm computes an optimal locally constrained step (Gay 1981) with a trust region restriction on the step. It handles the case that the Hessian is indefinite and provides a way to deal with negative curvature. For more details, see Dennis and Schnabel (1983, Appendix A) and Gay (1983).

Since a finite-difference method is used to estimate the Hessian for some single precision calculations, an inaccurate estimate of the Hessian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Hessian can be easily provided, IMSL routine [UMIAH](#) should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of U2IDH/DU2IDH. The reference is:

```
CALL U2IDH (FCN, GRAD, N, XGUESS, XSCALE, FSCALE, IPARAM,
 RPARAM, X, FVALUE, WK)
```

The additional argument is:

**WK** — Work vector of length  $N * (N + 9)$ . WK contains the following information on output: The second  $N$  locations contain the last step

taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Hessian at the approximate solution.

2. Informational errors

| Type | Code |                                                                                                                                                                                                       |
|------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 4    | 2    | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4    | 5    | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2    | 7    | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 4    | 7    | Maximum number of Hessian evaluations exceeded.                                                                                                                                                       |
| 3    | 8    | The last global step failed to locate a lower point than the current $x$ value.                                                                                                                       |

3. The first stopping criterion for `UMIDH` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `UMIDH` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).

4. If the default parameters are desired for `UMIDH`, then set `IPARAM(1)` to zero and call routine `UMIDH`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UMIDH`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4INF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 7.

`IPARAM(1)` = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter  
Default: Not used in UMIDH.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: 100

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)  
The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .  
Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.  
Default:  $\max(10^{-10}, \epsilon^{2/3})$ ,  $\max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.  
Default: Not used in UMIDH.

RPARAM(5) = False convergence tolerance.  
Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.  
Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.  
Default: Based on initial scaled Cauchy step.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized. Default values for parameters are used.

```
USE UMIDH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
!
INTEGER IPARAM(7), L, NOUT
REAL F, X(N), XGUESS(N)
EXTERNAL ROSBRK, ROSGRD
!
DATA XGUESS/-1.2E0, 1.0E0/
!
IPARAM(1) = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL UMIDH (ROSBRK, ROSGRD, X, XGUESS=XGUESS, IPARAM=IPARAM, FVALUE=F)
! Print results
CALL UMACH (2, NOUT)
```

```

WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
'value is ', F8.3, '//, ' The number of iterations is ', &
10X, I3, '/', ' The number of function evaluations is ', &
I3, '/', ' The number of gradient evaluations is ', I3, '/', &
' The number of Hessian evaluations is ', I3)
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
RETURN
END
!
SUBROUTINE ROSGRD (N, X, G)
INTEGER N
REAL X(N), G(N)
!
G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
RETURN
END

```

## Output

```

The solution is 1.000 1.000
The function value is 0.000
The number of iterations is 21
The number of function evaluations is 30
The number of gradient evaluations is 22
The number of Hessian evaluations is 21

```

---

## UMIAH

Minimizes a function of  $N$  variables using a modified Newton method and a user-supplied Hessian.

### Required Arguments

*FCN* — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL *FCN* ( $N, X, F$ ), where

$N$  — Length of  $X$ . (Input)

$X$  — Vector of length  $N$  at which point the function is evaluated. (Input)  
 $X$  should not be changed by *FCN*.

F – The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N – Length of X and G. (Input)

X – Vector of length N at which point the gradient is evaluated. (Input)  
X should not be changed by GRAD.

G – The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

**HESS** — User-supplied subroutine to compute the Hessian at the point X. The usage is  
CALL HESS (N, X, H, LDH), where

N – Length of X. (Input)

X – Vector of length N at which point the Hessian is evaluated. (Input)  
X should not be changed by HESS.

H – The Hessian evaluated at the point X. (Output)

LDH – Leading dimension of H exactly as specified in the dimension statement  
of the calling program. (Input)

HESS must be declared EXTERNAL in the calling program.

X — Vector of length N containing the computed solution. (Output)

## Optional Arguments

N — Dimension of the problem. (Input)  
Default: N = SIZE (X,1).

**XGUESS** — Vector of length N containing initial guess. (Input)  
Default: XGUESS = 0.0.

**XSCALE** — Vector of length N containing the diagonal scaling matrix for the variables.  
(Input)  
XSCALE is used mainly in scaling the gradient and the distance between two points. In  
the absence of other information, set all entries to 1.0.  
Default: XSCALE = 1.0.

**FSCALE** — Scalar containing the function scaling. (Input)  
FSCALE is used mainly in scaling the gradient. In the absence of other information, set  
FSCALE to 1.0.  
Default: FSCALE = 1.0.

**IPARAM** — Parameter vector of length 7. (Input/Output)  
Set `IPARAM(1)` to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.  
Default: `IPARAM = 0`.

**RPARAM** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

**FVALUE** — Scalar containing the value of the function at the computed solution. (Output)

## FORTRAN 90 Interface

Generic: `CALL UMIAH (FCN, GRAD, HESS, X, [, ...])`

Specific: The specific interface names are `S_UMIAH` and `D_UMIAH`.

## FORTRAN 77 Interface

Single: `CALL UMIAH (FCN, GRAD, HESS, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)`

Double: The double precision name is `DUMIAH`.

## Description

The routine `UMIAH` uses a modified Newton method to find the minimum of a function  $f(x)$  of  $n$  variables. First and second derivatives must be provided by the user. The algorithm computes an optimal locally constrained step (Gay 1981) with a trust region restriction on the step. This algorithm handles the case where the Hessian is indefinite and provides a way to deal with negative curvature. For more details, see Dennis and Schnabel (1983, Appendix A) and Gay (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `U2IAH/DU2IAH`. The reference is:

```
CALL U2IAH (FCN, GRAD, HESS, N, XGUESS, XSCALE, FSCALE,
 IPARAM, RPARAM, X, FVALUE, WK)
```

The additional argument is:

**WK** — Work vector of length  $N * (N + 9)$ . `WK` contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Hessian at the approximate solution.

2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 4 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4 | 5 | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 4 | 7 | Maximum number of Hessian evaluations exceeded.                                                                                                                                                       |
| 3 | 8 | The last global step failed to locate a lower point than the current <code>x</code> value.                                                                                                            |

- The first stopping criterion for `UMIAH` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `UMIAH` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).
- If the default parameters are desired for `UMIAH`, then set `IPARAM(1)` to zero and call the routine `UMIAH`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UMIAH`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4INF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 7.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter  
Default: Not used in UMIH.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: 100.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = XSCALE$ , and  $f_s = FSCALE$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)  
The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = XSCALE$ .  
Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.  
Default:  $\max(10^{-10}, \epsilon^{2/3})$ ,  $\max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.  
Default: Not used in UMIH.

RPARAM(5) = False convergence tolerance.  
Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = XSCALE$ , and  $t = XGUESS$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called, and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized. Default values for parameters are used.

```
USE UMIAH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
!
INTEGER IPARAM(7), L, NOUT
REAL F, FSCALE, RPARAM(7), X(N), &
 XGUESS(N), XSCALE(N)
EXTERNAL ROSBRK, ROSGRD, ROSHES
!
DATA XGUESS/-1.2E0, 1.0E0/, XSCALE/1.0E0, 1.0E0/, FSCALE/1.0E0/
!
IPARAM(1) = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL UMIAH (ROSBRK, ROSGRD, ROSHES, X, XGUESS=XGUESS, IPARAM=IPARAM, &
 FVALUE=F)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
 'value is ', F8.3, '//, ' The number of iterations is ', &
 10X, I3, '/', ' The number of function evaluations is ', &
```

```

 I3, /, ' The number of gradient evaluations is ', I3, /, &
 ' The number of Hessian evaluations is ', I3)
!
 END
!
 SUBROUTINE ROSBRK (N, X, F)
 INTEGER N
 REAL X(N), F
!
 F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
 RETURN
 END
!
 SUBROUTINE ROSGRD (N, X, G)
 INTEGER N
 REAL X(N), G(N)
!
 G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
 G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
 RETURN
 END
!
 SUBROUTINE ROSHES (N, X, H, LDH)
 INTEGER N, LDH
 REAL X(N), H(LDH,N)
!
 H(1,1) = -4.0E2*X(2) + 1.2E3*X(1)*X(1) + 2.0E0
 H(2,1) = -4.0E2*X(1)
 H(1,2) = H(2,1)
 H(2,2) = 2.0E2
!
 RETURN
 END

```

## Output

The solution is            1.000    1.000

The function value is    0.000

The number of iterations is            21

The number of function evaluations is 31

The number of gradient evaluations is 22

The number of Hessian evaluations is 21

---

## UMCGF

Minimizes a function of  $N$  variables using a conjugate gradient algorithm and a finite-difference gradient.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is

CALL FCN (N, X, F), where

N — Length of X. (Input)

X — The point at which the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**DFPRED** — A rough estimate of the expected reduction in the function. (Input)

DFPRED is used to determine the size of the initial change to X.

X — Vector of length N containing the computed solution. (Output)

## Optional Arguments

N — Dimension of the problem. (Input)

Default: N = SIZE (X,1).

**XGUESS** — Vector of length N containing the initial guess of the minimum. (Input)

Default: XGUESS = 0.0.

**XSCALE** — Vector of length N containing the diagonal scaling matrix for the variables.

(Input)

Default: XSCALE = 1.0.

**GRADTL** — Convergence criterion. (Input)

The calculation ends when the sum of squares of the components of G is less than

GRADTL.

Default: GRADTL = 1.e-4.

**MAXFN** — Maximum number of function evaluations. (Input)

If MAXFN is set to zero, then no restriction on the number of function evaluations is set.

Default: MAXFN = 0.

G — Vector of length N containing the components of the gradient at the final parameter

estimates. (Output)

**FVALUE** — Scalar containing the value of the function at the computed solution. (Output)

## FORTRAN 90 Interface

Generic: CALL UMGF (FCN, DFPRED, X [, ...])

Specific: The specific interface names are `S_UMCGF` and `D_UMCGF`.

## FORTRAN 77 Interface

Single: `CALL UMCGF (FCN, N, XGUESS, XSCALE, GRADTL, MAXFN, DFPRED, X, G, FVALUE)`

Double: The double precision name is `DUMCGF`.

## Description

The routine `UMCGF` uses a conjugate gradient method to find the minimum of a function  $f(x)$  of  $n$  variables. Only function values are required.

The routine is based on the version of the conjugate gradient algorithm described in Powell (1977). The main advantage of the conjugate gradient technique is that it provides a fast rate of convergence without the storage of any matrices. Therefore, it is particularly suitable for unconstrained minimization calculations where the number of variables is so large that matrices of dimension  $n$  cannot be stored in the main memory of the computer. For smaller problems, however, a routine such as routine `UMINF`, is usually more efficient because each iteration makes use of additional information from previous iterations.

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine `UMCGG` should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `U2CGF/DU2CGF`. The reference is:

```
CALL U2CGF (FCN, N, XGUESS, XSCALE, GRADTL, MAXFN, DFPRED,
 X, G, FVALUE, S, RSS, RSG, GINIT, XOPT, GOPT)
```

The additional arguments are as follows:

*S* — Vector of length  $N$  used for the search direction in each iteration.

*RSS* — Vector of length  $N$  containing conjugacy information.

*RSG* — Vector of length  $N$  containing conjugacy information.

*GINIT* — Vector of length  $N$  containing the gradient values at the start of an iteration.

*XOPT* — Vector of length  $N$  containing the parameter values that yield the least calculated value for `FVALUE`.

*GOPT* — Vector of length  $N$  containing the gradient values that yield the least calculated value for `FVALUE`.

2. Informational errors

Type Code

|   |   |                                                                                                    |
|---|---|----------------------------------------------------------------------------------------------------|
| 4 | 1 | The line search of an integration was abandoned. This error may be caused by an error in gradient. |
| 4 | 2 | The calculation cannot continue because the search is uphill.                                      |
| 4 | 3 | The iteration was terminated because MAXFN was exceeded.                                           |
| 3 | 4 | The calculation was terminated because two consecutive iterations failed to reduce the function.   |

3. Because of the close relation between the conjugate-gradient method and the method of steepest descent, it is very helpful to choose the scale of the variables in a way that balances the magnitudes of the components of a typical gradient vector. It can be particularly inefficient if a few components of the gradient are much larger than the rest.

4. If the value of the parameter GRADTL in the argument list of the routine is set to zero, then the subroutine will continue its calculation until it stops reducing the objective function. In this case, the usual behavior is that changes in the objective function become dominated by computer rounding errors before precision is lost in the gradient vector. Therefore, because the point of view has been taken that the user requires the least possible value of the function, a value of the objective function that is small due to computer rounding errors can prevent further progress. Hence, the precision in the final values of the variables may be only about half the number of significant digits in the computer arithmetic, but the least value of FVALUE is usually found to be quite accurate.

## Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized and the solution is printed.

```
USE UMCGF_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER N
PARAMETER (N=2)
!
INTEGER I, MAXFN, NOUT
REAL DFPRED, FVALUE, G(N), GRADTL, X(N), XGUESS(N)
EXTERNAL ROSBRK
!
DATA XGUESS/-1.2E0, 1.0E0/
!
```

```

DFPRED = 0.2
GRADTL = 1.0E-6
MAXFN = 100
!
! Minimize the Rosenbrock function
CALL UMGF (ROSBRK, DFPRED, X, xguess=xguess, gradtl=gradtl, &
 g=g, fvalue=fvalue)
!
! Print the results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (X(I),I=1,N), FVALUE, (G(I),I=1,N)
99999 FORMAT (' The solution is ', 2F8.3, '//, ' The function ', &
 'evaluated at the solution is ', F8.3, '//, ' The ', &
 'gradient is ', 2F8.3, '/')
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
RETURN
END

```

## Output

```

The solution is 0.999 0.998

The function evaluated at the solution is 0.000

The gradient is -0.001 0.000

```

---

## UMCGG

Minimizes a function of  $N$  variables using a conjugate gradient algorithm and a user-supplied gradient.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
 CALL FCN (N, X, F), where

$N$  — Length of  $X$ . (Input)

$X$  — The point at which the function is evaluated. (Input)  
 $X$  should not be changed by FCN.

$F$  — The computed function value at the point  $X$ . (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point  $X$ . The usage is  
 CALL GRAD (N, X, G), where

$N$  — Length of  $X$  and  $G$ . (Input)

$X$  — The point at which the gradient is evaluated. (Input)  
 $X$  should not be changed by  $GRAD$ .

$G$  — The gradient evaluated at the point  $X$ . (Output)

$GRAD$  must be declared `EXTERNAL` in the calling program.

***DFPRED*** — A rough estimate of the expected reduction in the function. (Input)  
 $DFPRED$  is used to determine the size of the initial change to  $X$ .

$X$  — Vector of length  $N$  containing the computed solution. (Output)

### Optional Arguments

$N$  — Dimension of the problem. (Input)  
Default:  $N = \text{SIZE}(X,1)$ .

***XGUESS*** — Vector of length  $N$  containing the initial guess of the minimum. (Input)  
Default:  $XGUESS = 0.0$ .

***GRADTL*** — Convergence criterion. (Input)  
The calculation ends when the sum of squares of the components of  $G$  is less than  $GRADTL$ .  
Default:  $GRADTL = 1.e-4$ .

***MAXFN*** — Maximum number of function evaluations. (Input)  
Default:  $MAXFN = 100$ .

$G$  — Vector of length  $N$  containing the components of the gradient at the final parameter estimates. (Output)

***FVALUE*** — Scalar containing the value of the function at the computed solution. (Output)

### FORTRAN 90 Interface

Generic: `CALL UMCGG (FCN, GRAD, DFPRED, X [, ...])`

Specific: The specific interface names are `S_UMCGG` and `D_UMCGG`.

### FORTRAN 77 Interface

Single: `CALL UMCGG (FCN, GRAD, N, XGUESS, GRADTL, MAXFN, DFPRED, X, G, FVALUE)`

Double: The double precision name is `DUMCGG`.

## Description

The routine `UMCGG` uses a conjugate gradient method to find the minimum of a function  $f(x)$  of  $n$  variables. Function values and first derivatives are required.

The routine is based on the version of the conjugate gradient algorithm described in Powell (1977). The main advantage of the conjugate gradient technique is that it provides a fast rate of convergence without the storage of any matrices. Therefore, it is particularly suitable for unconstrained minimization calculations where the number of variables is so large that matrices of dimension  $n$  cannot be stored in the main memory of the computer. For smaller problems, however, a subroutine such as IMSL routine `UMING`, is usually more efficient because each iteration makes use of additional information from previous iterations.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `U2CGG/DU2CGG`. The reference is:

```
CALL U2CGG (FCN, GRAD, N, XGUESS, GRADTL, MAXFN, DFPRED,
 X, G, FVALUE, S, RSS, RSG, GINIT, XOPT, GOPT)
```

The additional arguments are as follows:

**S** — Vector of length  $N$  used for the search direction in each iteration.

**RSS** — Vector of length  $N$  containing conjugacy information.

**RSG** — Vector of length  $N$  containing conjugacy information.

**GINIT** — Vector of length  $N$  containing the gradient values at the start on an iteration.

**XOPT** — Vector of length  $N$  containing the parameter values which yield the least calculated value for `FVALUE`.

**GOPT** — Vector of length  $N$  containing the gradient values which yield the least calculated value for `FVALUE`.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                    |
|---|---|----------------------------------------------------------------------------------------------------|
| 4 | 1 | The line search of an integration was abandoned. This error may be caused by an error in gradient. |
| 4 | 2 | The calculation cannot continue because the search is uphill.                                      |
| 4 | 3 | The iteration was terminated because <code>MAXFN</code> was exceeded.                              |
| 3 | 4 | The calculation was terminated because two consecutive iterations failed to reduce the function.   |

3. The routine includes no thorough checks on the part of the user program that calculates the derivatives of the objective function. Therefore, because derivative calculation is a

frequent source of error, the user should verify independently the correctness of the derivatives that are given to the routine.

4. Because of the close relation between the conjugate-gradient method and the method of steepest descent, it is very helpful to choose the scale of the variables in a way that balances the magnitudes of the components of a typical gradient vector. It can be particularly inefficient if a few components of the gradient are much larger than the rest.
5. If the value of the parameter GRADTL in the argument list of the routine is set to zero, then the subroutine will continue its calculation until it stops reducing the objective function. In this case, the usual behavior is that changes in the objective function become dominated by computer rounding errors before precision is lost in the gradient vector. Therefore, because the point of view has been taken that the user requires the least possible value of the function, a value of the objective function that is small due to computer rounding errors can prevent further progress. Hence, the precision in the final values of the variables may be only about half the number of significant digits in the computer arithmetic, but the least value of FVALUE is usually found to be quite accurate.

### Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized and the solution is printed.

```

USE UMCGG_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER N
PARAMETER (N=2)
!
INTEGER I, NOUT
REAL DFPRED, FVALUE, G(N), GRADTL, X(N), &
 XGUESS(N)
EXTERNAL ROSBRK, ROSGRD
!
DATA XGUESS/-1.2E0, 1.0E0/
!
DFPRED = 0.2
GRADTL = 1.0E-7
! Minimize the Rosenbrock function
CALL UMCGG (ROSBRK, ROSGRD, DFPRED, X, xguess=xguess, &
 gradtl=gradtl, g=g, fvalue=fvalue)
! Print the results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (X(I),I=1,N), FVALUE, (G(I),I=1,N)

```

```

99999 FORMAT (' The solution is ', 2F8.3, '//, ' The function ', &
 'evaluated at the solution is ', F8.3, '//, ' The ', &
 'gradient is ', 2F8.3, '/')
!
 END
!
 SUBROUTINE ROSBRK (N, X, F)
 INTEGER N
 REAL X(N), F
!
 F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
 RETURN
 END
!
 SUBROUTINE ROSGRD (N, X, G)
 INTEGER N
 REAL X(N), G(N)
!
 G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
 G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
 RETURN
 END
!
 SUBROUTINE ROSBRK (N, X, F)
 INTEGER N
 REAL X(N), F
!
 F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
 RETURN
 END
!
 SUBROUTINE ROSGRD (N, X, G)
 INTEGER N
 REAL X(N), G(N)
!
 G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
 G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
 RETURN
 END

```

## Output

```

The solution is 1.000 1.000

The function evaluated at the solution is 0.000

The gradient is 0.000 -0.000

```

---

## UMPOL

Minimizes a function of N variables using a direct search polytope algorithm.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

X — Real vector of length N containing the best estimate of the minimum found. (Output)

## Optional Arguments

N — Dimension of the problem. (Input)  
Default: N = SIZE (X,1).

**XGUESS** — Real vector of length N which contains an initial guess to the minimum. (Input)  
Default: XGUESS = 0.0.

S — On input, real scalar containing the length of each side of the initial simplex.  
(Input/Output)  
If no reasonable information about S is known, S could be set to a number less than or equal to zero and UMPOL will generate the starting simplex from the initial guess with a random number generator. On output, the average distance from the vertices to the centroid that is taken to be the solution; see Comment 4.  
Default: S = 0.0.

**FTOL** — First convergence criterion. (Input)  
The algorithm stops when a relative error in the function values is less than FTOL, i.e. when  $(F(\text{worst}) - F(\text{best})) < FTOL * (1 + ABS(F(\text{best})))$  where F(worst) and F(best) are the function values of the current worst and best points, respectively. Second convergence criterion. The algorithm stops when the standard deviation of the function values at the N + 1 current points is less than FTOL. If the subroutine terminates prematurely, try again with a smaller value for FTOL.  
Default: FTOL = 1.e-7.

**MAXFCN** — On input, maximum allowed number of function evaluations. (Input/ Output)  
On output, actual number of function evaluations needed.  
Default: MAXFCN = 200.

**FVALUE** — Function value at the computed solution. (Output)

## FORTRAN 90 Interface

Generic:    CALL UMPOL (FCN, X [, ...])

Specific:   The specific interface names are S\_UMPOL and D\_UMPOL.

## FORTRAN 77 Interface

Single:     CALL UMPOL (FCN, N, XGUESS, S, FTOL, MAXFCN, X, FVALUE)

Double:     The double precision name is DUMPOL.

## Description

The routine UMPOL uses the polytope algorithm to find a minimum point of a function  $f(x)$  of  $n$  variables. The polytope method is based on function comparison; no smoothness is assumed. It starts with  $n + 1$  points  $x_1, x_2, \dots, x_{n+1}$ . At each iteration, a new point is generated to replace the worst point  $x_j$ , which has the largest function value among these  $n + 1$  points. The new point is constructed by the following formula:

$$x_k = c + \alpha(c - x_j)$$

where

$$c = \frac{1}{n} \sum_{i \neq j} x_i$$

and  $\alpha$  ( $\alpha > 0$ ) is the *reflection coefficient*.

When  $x_k$  is a best point, that is  $f(x_k) \leq f(x_i)$  for  $i = 1, \dots, n + 1$ , an expansion point is computed  $x_e = c + \beta(x_k - c)$  where  $\beta$  ( $\beta > 1$ ) is called the *expansion coefficient*. If the new point is a worst point, then the polytope would be contracted to get a better new point. If the contraction step is unsuccessful, the polytope is shrunk by moving the vertices halfway toward current best point. This procedure is repeated until one of the following stopping criteria is satisfied:

Criterion 1:

$$f_{best} - f_{worst} \leq \epsilon_f (1 + |f_{best}|)$$

Criterion 2:

$$\sum_{i=1}^{n+1} \left( f_i - \frac{\sum_{j=1}^{n+1} f_j}{n+1} \right)^2 \leq \epsilon_f$$

where  $f_i = f(x_i)$ ,  $f_j = f(x_j)$ , and  $\epsilon_f$  is a given tolerance. For a complete description, see Nelder and Mead (1965) or Gill et al. (1981).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of U2POL/DU2POL. The reference is:

```
CALL U2POL (FCN, N, XGUESS, S, FTOL, MAXFCN, X, FVALUE,
 WK)
```

The additional argument is:

**WK** — Real work vector of length  $N^2 + 5 * N + 1$ .

2. Informational error

Type Code

4 1 Maximum number of function evaluations exceeded.

3. Since `UMPOL` uses only function value information at each step to determine a new approximate minimum, it could be quite inefficient on smooth problems compared to other methods such as those implemented in routine `UMINF` that takes into account derivative information at each iteration. Hence, routine `UMPOL` should only be used as a last resort. Briefly, a set of  $N + 1$  points in an  $N$ -dimensional space is called a simplex. The minimization process iterates by replacing the point with the largest function value by a new point with a smaller function value. The iteration continues until all the points cluster sufficiently close to a minimum.

4. The value returned in `S` is useful for assessing the flatness of the function near the computed minimum. The larger its value for a given value of `FTOL`, the flatter the function tends to be in the neighborhood of the returned point.

### Example

The function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

is minimized and the solution is printed.

```
USE UMPOL_INT
USE UMACH_INT

IMPLICIT NONE
! Variable declarations
INTEGER N
PARAMETER (N=2)
!
INTEGER K, NOUT
REAL FTOL, FVALUE, S, X(N), XGUESS(N)
EXTERNAL FCN
!
! Initializations
! XGUESS = (-1.2, 1.0)
!
DATA XGUESS/-1.2, 1.0/
!
FTOL = 1.0E-10
```

```

S = 1.0
!
CALL UMPOL (FCN, X, xguess=xguess, s=s, ftol=ftol,&
 fvalue=fvalue)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (X(K),K=1,N), FVALUE
99999 FORMAT (' The best estimate for the minimum value of the', /, &
 ' function is X = (' , 2(2X,F4.2), ')', /, ' with ', &
 'function value FVALUE = ', E12.6)
!
END
!
 External function to be minimized
SUBROUTINE FCN (N, X, F)
INTEGER N
REAL X(N), F
!
F = 100.0*(X(1)*X(1)-X(2))**2 + (1.0-X(1))**2
RETURN
END

```

## Output

The best estimate for the minimum value of the  
function is X = ( 1.00 1.00)  
with function value FVALUE = 0.502496E-10

# UNLSF



Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

## Required Arguments

*FCN* — User-supplied subroutine to evaluate the function that defines the least-squares problem. The usage is

CALL FCN (M, N, X, F), where

M — Length of F. (Input)

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — Vector of length M containing the function values at X. (Output)

FCN must be declared EXTERNAL in the calling program.

M — Number of functions. (Input)

*X* — Vector of length *N* containing the approximate solution. (Output)

### Optional Arguments

*N* — Number of variables. *N* must be less than or equal to *M*. (Input)  
Default: *N* = SIZE (*X*,1).

*XGUESS* — Vector of length *N* containing the initial guess. (Input)  
Default: *XGUESS* = 0.0.

*XSCALE* — Vector of length *N* containing the diagonal scaling matrix for the variables. (Input)  
*XSCALE* is used mainly in scaling the gradient and the distance between two points. By default, the values for *XSCALE* are set internally. See *IPARAM*(6) in Comment 4.  
Default: *XSCALE* = 1.0.

*FSCALE* — Vector of length *M* containing the diagonal scaling matrix for the functions. (Input)  
*FSCALE* is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.  
Default: *FSCALE* = 1.0.

*IPARAM* — Parameter vector of length 6. (Input/Output)  
Set *IPARAM*(1) to zero for default values of *IPARAM* and *RPARAM*. See Comment 4.  
Default: *IPARAM* = 0.

*RPARAM* — Parameter vector of length 7. (Input/Output)  
See Comment 4.

*FVEC* — Vector of length *M* containing the residuals at the approximate solution. (Output)

*FJAC* — *M* by *N* matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

*LDFJAC* — Leading dimension of *FJAC* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDFJAC* = SIZE (*FJAC*,1).

### FORTRAN 90 Interface

Generic:     CALL UNLSF (FCN, M, X [, ...])

Specific:    The specific interface names are *S\_UNLSF* and *D\_UNLSF*.

## FORTRAN 77 Interface

Single:      CALL UNLSF (FCN, M, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC)

Double:      The double precision name is DUNLSF.

## Description

The routine UNLSF is based on the MINPACK routine LMDIF by Moré et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2$$

where  $m \geq n$ ,  $F: \mathbf{R}^n \rightarrow \mathbf{R}^m$ , and  $f_i(x)$  is the  $i$ -th component function of  $F(x)$ . From a current point, the algorithm uses the trust region approach:

$$\min_{x_n \in \mathbf{R}^n} \|F(x_c) + J(x_c)(x_n - x_c)\|_2$$

subject to  $\|x_n - x_c\|_2 \leq \delta c$

to get a new point  $x_n$ , which is computed as

$$x_n = x_c - \left( J(x_c)^T J(x_c) + \mu_c I \right)^{-1} J(x_c)^T F(x_c)$$

where  $\mu_c = 0$  if  $\delta c \geq \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$  and  $\mu_c > 0$  otherwise.  $F(x_c)$  and  $J(x_c)$  are the function values and the Jacobian evaluated at the current point  $x_c$ . This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

Since a finite-difference method is used to estimate the Jacobian for some single precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, routine UNLSJ should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of U2LSF/DU2LSF. The reference is:

```
CALL U2LSF (FCN, M, N, XGUESS, XSCALE, FSCALE, IPARAM,
 RPARAM, X, FVEC, FJAC, LDFJAC, WK, IWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length  $9 * N + 3 * M - 1$ . WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Gauss-Newton

step. The fourth  $N$  locations contain an estimate of the gradient at the solution.

**IWK** — Integer work vector of length  $N$  containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

| Type | Code |                                                                                                                                                                                          |
|------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                 |
| 3    | 2    | The iterates appear to be converging to a noncritical point.                                                                                                                             |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                                                                   |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                                                                         |
| 3    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                                                                     |
| 2    | 7    | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big. |

3. The first stopping criterion for UNLSF occurs when the norm of the function is less than the absolute function tolerance (RPARAM(4)). The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance (RPARAM(1)). The third stopping criterion for UNLSF occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).

4. If the default parameters are desired for UNLSF, then set IPARAM(1) to zero and call the routine UNLSF. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling UNLSF:

```
CALL U4LSF (IPARAM, RPARAM)
```

Set nondefault values for desired IPARAM, RPARAM elements.

---

Note that the call to U4LSF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 6.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.  
Default: Not used in UNLSF.

IPARAM(6) = Internal variable scaling flag.  
If IPARAM(6) = 1, then the values for XSCALE are set internally.  
Default: 1.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\|F(x)\|_2^2}$$

where

$$g_i = \left( J(x)^T F(x) \right)_i * (f_s)_i^2$$

$J(x)$  is the Jacobian,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)  
The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\varepsilon^{2/3}$  where  $\varepsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.  
Default:  $\max(10^{-10}, \varepsilon^{2/3}), \max(10^{-20}, \varepsilon^{2/3})$  in double where  $\varepsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default:  $\max(10^{-20}, \epsilon^2), \max(10^{-40}, \epsilon^2)$  in double where  $\epsilon$  is the machine precision.

RPARAM(5) = False convergence tolerance.

Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then DU4LSF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “[Error Handling](#)” in the Introduction.

## Example

The nonlinear least squares problem

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^2 f_i(x)^2$$

where

$$f_1(x) = 10(x_2 - x_1^2) \text{ and } f_2(x) = (1 - x_1)$$

is solved. RPARAM(4) is changed to a non-default value.

```
USE UMACH_INT
USE U4LSF_INT
USE UNLSF_INT

IMPLICIT NONE
! Declaration of variables
INTEGER LDFJAC, M, N
PARAMETER (LDFJAC=2, M=2, N=2)
!
INTEGER IPARAM(6), NOUT
REAL FVEC(M), RPARAM(7), X(N), XGUESS(N)
```

```

EXTERNAL ROSBCK
!
! Compute the least squares for the
! Rosenbrock function.
DATA XGUESS/-1.2E0, 1.0E0/
!
! Relax the first stopping criterion by
! calling U4LSF and scaling the
! absolute function tolerance by 10.
CALL U4LSF (IPARAM, RPARAM)
RPARAM(4) = 10.0E0*RPARAM(4)
!
CALL UNLSF (ROSBCK, M, X, xguess=xguess, iparam=iparam, rparam=rparam, &
 fvec=fvec)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT (' The solution is ', 2F9.4, '//, ' The function ', &
 'evaluated at the solution is ', /, 18X, 2F9.4, '//, &
 ' The number of iterations is ', 10X, I3, '//, ' The ', &
 'number of function evaluations is ', I3, /)
END
!
SUBROUTINE ROSBCK (M, N, X, F)
INTEGER M, N
REAL X(N), F(M)
!
F(1) = 10.0E0*(X(2)-X(1)*X(1))
F(2) = 1.0E0 - X(1)
RETURN
END

```

## Output

```

The solution is 1.0000 1.0000

The function evaluated at the solution is
0.0000 0.0000

The number of iterations is 24
The number of function evaluations is 33

```

---

## UNLSJ



Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function which defines the least-squares problem. The usage is

CALL FCN (M, N, X, F), where

M — Length of F. (Input)

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — Vector of length M containing the function values at X. (Output)

FCN must be declared EXTERNAL in the calling program.

**JAC** — User-supplied subroutine to evaluate the Jacobian at a point X. The usage is

CALL JAC (M, N, X, FJAC, LDFJAC), where

M — Length of F. (Input)

N — Length of X. (Input)

X — Vector of length N at which point the Jacobian is evaluated. (Input)  
X should not be changed by JAC.

FJAC — The computed M by N Jacobian at the point X. (Output)

LDFJAC — Leading dimension of FJAC. (Input)

JAC must be declared EXTERNAL in the calling program.

**M** — Number of functions. (Input)

**X** — Vector of length N containing the approximate solution. (Output)

## Optional Arguments

**N** — Number of variables. N must be less than or equal to M. (Input)

Default: N = SIZE (X,1).

**XGUESS** — Vector of length N containing the initial guess. (Input)

Default: XGUESS = 0.0.

**XSCALE** — Vector of length N containing the diagonal scaling matrix for the variables. (Input)

XSCALE is used mainly in scaling the gradient and the distance between two points. By default, the values for XSCALE are set internally. See IPARAM(6) in Comment 4.

Default: XSCALE = 1.0.

**FSCALE** — Vector of length  $M$  containing the diagonal scaling matrix for the functions.  
(Input)

FSCALE is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.

Default: FSCALE = 1.0.

**IPARAM** — Parameter vector of length 6. (Input/Output)

Set IPARAM(1) to zero for default values of IPARAM and RPARAM. See Comment 4.

Default: IPARAM = 0.

**RPARAM** — Parameter vector of length 7. (Input/Output)

See Comment 4.

**FVEC** — Vector of length  $M$  containing the residuals at the approximate solution. (Output)

**FJAC** —  $M$  by  $N$  matrix containing a finite-difference approximate Jacobian at the approximate solution. (Output)

**LDFJAC** — Leading dimension of FJAC exactly as specified in the dimension statement of the calling program. (Input)

Default: LDFJAC = SIZE (FJAC,1).

## FORTRAN 90 Interface

Generic: CALL UNLSJ (FCN, JAC, M, X [, ...])

Specific: The specific interface names are S\_UNLSJ and D\_UNLSJ.

## FORTRAN 77 Interface

Single: CALL UNLSJ (FCN, JAC, M, N, XGUESS, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC)

Double: The double precision name is DUNLSJ.

## Description

The routine UNLSJ is based on the MINPACK routine LMDER by Moré et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2$$

where  $m \geq n$ ,  $F: \mathbf{R}^n \rightarrow \mathbf{R}^m$ , and  $f_i(x)$  is the  $i$ -th component function of  $F(x)$ . From a current point, the algorithm uses the trust region approach:

$$\min_{x_n \in \mathbf{R}^n} \|F(x_c) + J(x_c)(x_n - x_c)\|_2$$

subject to  $\|x_n - x_c\|_2 \leq \delta_c$

to get a new point  $x_n$ , which is computed as

$$x_n = x_c - \left( J(x_c)^T J(x_c) + \mu_c I \right)^{-1} J(x_c)^T F(x_c)$$

where

$$\mu_c = 0 \text{ if } \delta_c \geq \left\| \left( J(x_c)^T J(x_c) \right)^{-1} J(x_c)^T F(x_c) \right\|_2$$

and  $\mu_c > 0$  otherwise.  $F(x_c)$  and  $J(x_c)$  are the function values and the Jacobian evaluated at the current point  $x_c$ . This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt(1963), or Dennis and Schnabel (1983, Chapter 10).

## Comments

1. Workspace may be explicitly provided, if desired, by use of U2LSJ/DU2LSJ. The reference is:

```
CALL U2LSJ (FCN, JAC, M, N, XGUESS, XSCALE, FSCALE,
 IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $9 * N + 3 * M - 1$ . WK contains the following information on output: The second N locations contain the last step taken. The third N locations contain the last Gauss-Newton step. The fourth N locations contain an estimate of the gradient at the solution.

**IWK** — Work vector of length N containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

| Type | Code |                                                                                                                                          |
|------|------|------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance. |
| 3    | 2    | The iterates appear to be converging to a noncritical point.                                                                             |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                   |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                         |
| 4    | 5    | Maximum number of Jacobian evaluations exceeded.                                                                                         |
| 3    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                     |

2 7 Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or `STEPTL` is too big.

3. The first stopping criterion for `UNLSJ` occurs when the norm of the function is less than the absolute function tolerance (`RPARAM(4)`). The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance (`RPARAM(1)`). The third stopping criterion for `UNLSJ` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).
4. If the default parameters are desired for `UNLSJ`, then set `IPARAM(1)` to zero and call the routine `UNLSJ`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `UNLSJ`:

```
CALL U4LSF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4LSF` will set `IPARAM` and `RPARAM` to their default values, so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

`IPARAM(4)` = Maximum number of function evaluations.  
Default: 400.

`IPARAM(5)` = Maximum number of Jacobian evaluations.  
Default: 100.

`IPARAM(6)` = Internal variable scaling flag.  
If `IPARAM(6) = 1`, then the values for `XSCALE` are set internally.  
Default: 1.

***RPARAM*** — Real vector of length 7.

`RPARAM(1)` = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\|F(x)\|_2^2}$$

where

$$g_i = \left( J(x)^T F(x) \right)_i * (f_s)_i^2$$

$J(x)$  is the Jacobian,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .

Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default:  $\max(10^{-10}, \epsilon^{2/3}), \max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default:  $\max(10^{-20}, \epsilon^2), \max(10^{-40}, \epsilon^2)$  in double where  $\epsilon$  is the machine precision.

RPARAM(5) = False convergence tolerance.

Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then DU4LSF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The nonlinear least-squares problem

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^2 f_i(x)^2$$

where

$$f_1(x) = 10(x_2 - x_1^2) \text{ and } f_2(x) = (1 - x_1)$$

is solved; default values for parameters are used.

```
USE UNLSJ_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER LDFJAC, M, N
PARAMETER (LDFJAC=2, M=2, N=2)
!
INTEGER IPARAM(6), NOUT
REAL FVEC(M), X(N), XGUESS(N)
EXTERNAL ROSBCK, ROSJAC
! Compute the least squares for the
! Rosenbrock function.
DATA XGUESS/-1.2E0, 1.0E0/
IPARAM(1) = 0
!
CALL UNLSJ (ROSBCK, ROSJAC, M, X, XGUESS=XGUESS, &
 IPARAM=IPARAM, FVEC=FVEC)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4), IPARAM(5)
!
99999 FORMAT (' The solution is ', 2F9.4, '//, ' The function ', &
 'evaluated at the solution is ', /, 18X, 2F9.4, '//, &
 ' The number of iterations is ', 10X, I3, '//, ' The ', &
 'number of function evaluations is ', I3, '//, ' The ', &
 'number of Jacobian evaluations is ', I3, /)
END
!
SUBROUTINE ROSBCK (M, N, X, F)
INTEGER M, N
REAL X(N), F(M)
```

```

!
F(1) = 10.0E0*(X(2)-X(1)*X(1))
F(2) = 1.0E0 - X(1)
RETURN
END
!
SUBROUTINE ROSJAC (M, N, X, FJAC, LDFJAC)
INTEGER M, N, LDFJAC
REAL X(N), FJAC(LDFJAC,N)
!
FJAC(1,1) = -20.0E0*X(1)
FJAC(2,1) = -1.0E0
FJAC(1,2) = 10.0E0
FJAC(2,2) = 0.0E0
RETURN
END

```

## Output

The solution is     1.0000    1.0000

The function evaluated at the solution is  
0.0000    0.0000

The number of iterations is                    23  
The number of function evaluations is        32  
The number of Jacobian evaluations is        24

---

## BCONF

Minimizes a function of  $N$  variables subject to bounds on the variables using a quasi-Newton method and a finite-difference gradient.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

**IBTYPE    Action**

0            User will supply all the bounds.

- 1 All variables are nonnegative.
- 2 All variables are nonpositive.
- 3 User supplies only the bounds on 1st variable, all other variables will have the same bounds.

**XLB** — Vector of length  $N$  containing the lower bounds on variables. (Input, if  $IBTYPE = 0$ ; output, if  $IBTYPE = 1$  or  $2$ ; input/output, if  $IBTYPE = 3$ )

**XUB** — Vector of length  $N$  containing the upper bounds on variables. (Input, if  $IBTYPE = 0$ ; output, if  $IBTYPE = 1$  or  $2$ ; input/output, if  $IBTYPE = 3$ )

**X** — Vector of length  $N$  containing the computed solution. (Output)

### Optional Arguments

**$N$**  — Dimension of the problem. (Input)  
Default:  $N = \text{SIZE}(X,1)$ .

**XGUESS** — Vector of length  $N$  containing an initial guess of the computed solution. (Input)  
Default:  $XGUESS = 0.0$ .

**XSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)  
 $XSCALE$  is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.  
Default:  $XSCALE = 1.0$ .

**FSCALE** — Scalar containing the function scaling. (Input)  
 $FSCALE$  is used mainly in scaling the gradient. In the absence of other information, set  $FSCALE$  to 1.0.  
Default:  $FSCALE = 1.0$ .

**IPARAM** — Parameter vector of length 7. (Input/Output)  
Set  $IPARAM(1)$  to zero for default values of  $IPARAM$  and  $RPARAM$ . See Comment 4.  
Default:  $IPARAM = 0$ .

**RPARAM** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

**FVALUE** — Scalar containing the value of the function at the computed solution. (Output)

### FORTRAN 90 Interface

Generic: `CALL BCONF (FCN, IBTYPE, XLB, XUB, X [, ...])`

Specific: The specific interface names are `S_BCONF` and `D_BCONF`.

## FORTRAN 77 Interface

Single:      CALL BCONF (FCN, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double:     The double precision name is DBCONF.

## Description

The routine BCONF uses a quasi-Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to  $l \leq x \leq u$

From a given starting point  $x^c$ , an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -B^{-1} g^c$$

where  $B$  is a positive definite approximation of the Hessian and  $g^c$  is the gradient evaluated at  $x^c$ ; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point  $x^n$ ,

$$x^n = x^c + \lambda d, \quad \lambda \in (0, 1]$$

such that

$$f(x^n) \leq f(x^c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \leq \varepsilon, \quad l_i < x_i < u_i$$

$$g(x_i) < 0, \quad x_i = u_i$$

$$g(x_i) > 0, \quad x_i = l_i$$

are checked, where  $\varepsilon$  is a gradient tolerance. When optimality is not achieved,  $B$  is updated according to the BFGS formula:

$$B \leftarrow B - \frac{B s s^T B}{s^T B s} + \frac{y y^T}{y^T s}$$

where  $s = x^n - x^c$  and  $y = g^n - g^c$ . Another search direction is then computed to begin the next iteration.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more

details on the quasi-Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine `BCONG` should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2ONF/DB2ONF`. The reference is:

```
CALL B2ONF (FCN, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
 FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length  $N * (2 * N + 8)$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain a `BFGS` approximation to the Hessian at the solution. Only the lower triangular portion of the matrix is stored in **WK**. The values returned in the upper triangle should be ignored.

**IWK** — Work vector of length  $N$  stored in column order.

2. Informational errors

| Type | Code |                                                                                                                                                                                                       |
|------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 4    | 2    | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4    | 5    | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2    | 7    | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 3    | 8    | The last global step failed to locate a lower point than the current $x$ value.                                                                                                                       |

3. The first stopping criterion for `BCONF` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `BCONF`

occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).

4. If the default parameters are desired for `BCONF`, then set `IPARAM(1)` to zero and call the routine `BCONF`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `BCONF`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4INF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 7.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

`IPARAM(4)` = Maximum number of function evaluations.  
Default: 400.

`IPARAM(5)` = Maximum number of gradient evaluations.  
Default: 400.

`IPARAM(6)` = Hessian initialization parameter.  
If `IPARAM(6) = 0`, the Hessian is initialized to the identity matrix;  
otherwise,  
it is initialized to a diagonal matrix containing

$$\max(|f(t)|, f_s) * s_i^2$$

on the diagonal where  $t = \text{XGUESS}$ ,  $f_s = \text{FSCALE}$ , and  $s = \text{XSCALE}$ .  
Default: 0.

`IPARAM(7)` = Maximum number of Hessian evaluations.  
Default: Not used in `BCONF`.

***RPARAM*** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.

The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .

Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\varepsilon^{2/3}$  where  $\varepsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: Not used in BCONF.

RPARAM(4) = Absolute function tolerance.

Default: Not used in BCONF.

RPARAM(5) = False convergence tolerance.

Default: Not used in BCONF.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\varepsilon_1, \varepsilon_2)$  where

$$\varepsilon_1 \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\varepsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called and RPARAM is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The problem

$$\begin{aligned} \min f(x) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{subject to } &-2 \leq x_1 \leq 0.5 \\ &-1 \leq x_2 \leq 2 \end{aligned}$$

is solved with an initial guess (-1.2, 1.0) and default values for parameters.

```

USE BCONF_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
!
INTEGER IPARAM(7), ITP, L, NOUT
REAL F, FSCALE, RPARAM(7), X(N), XGUESS(N), &
 XLB(N), XSCALE(N), XUB(N)
EXTERNAL ROSBRK
!
DATA XGUESS/-1.2E0, 1.0E0/
DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
! All the bounds are provided
ITP = 0
!
! Default parameters are used
IPARAM(1) = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL BCONF (ROSBRK, ITP, XLB, XUB, X, XGUESS=XGUESS, &
 iparam=iparam, FVALUE=F)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
 'value is ', F8.3, '//, ' The number of iterations is ', &
 10X, I3, '/', ' The number of function evaluations is ', &
 I3, '/', ' The number of gradient evaluations is ', I3)
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!

```

```
RETURN
END
```

## Output

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 24
The number of function evaluations is 34
The number of gradient evaluations is 26
```

---

## BCONG

Minimizes a function of  $N$  variables subject to bounds on the variables using a quasi-Newton method and a user-supplied gradient.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Length of X and G. (Input)

X — Vector of length N at which point the gradient is evaluated. (Input)  
X should not be changed by GRAD.

G — The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| IBTYPE | Action                           |
|--------|----------------------------------|
| 1      | User will supply all the bounds. |
| 2      | All variables are nonnegative.   |
| 3      | All variables are nonpositive.   |

4

User supplies only the bounds on 1st variable, all other variables will have the same bounds.

***XLB*** — Vector of length  $N$  containing the lower bounds on variables. (Input, if `IBTYPE = 0`; output, if `IBTYPE = 1` or `2`; input/output, if `IBTYPE = 3`)

***XUB*** — Vector of length  $N$  containing the upper bounds on variables. (Input, if `IBTYPE = 0`; output, if `IBTYPE = 1` or `2`; input/output, if `IBTYPE = 3`)

***X*** — Vector of length  $N$  containing the computed solution. (Output)

### Optional Arguments

***N*** — Dimension of the problem. (Input)  
Default: `N = SIZE (X,1)`.

***XGUESS*** — Vector of length  $N$  containing the initial guess of the minimum. (Input)  
Default: `XGUESS = 0.0`.

***XSCALE*** — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)  
`XSCALE` is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.  
Default: `XSCALE = 1.0`.

***FSCALE*** — Scalar containing the function scaling. (Input)  
`FSCALE` is used mainly in scaling the gradient. In the absence of other information, set `FSCALE` to 1.0.  
Default: `FSCALE = 1.0`.

***IPARAM*** — Parameter vector of length 7. (Input/Output)  
Set `IPARAM(1)` to zero for default values of `IPARAM` and `RPARAM`. See Comment 4.  
Default: `IPARAM = 0`.

***RPARAM*** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

***FVALUE*** — Scalar containing the value of the function at the computed solution. (Output)

### FORTRAN 90 Interface

Generic:     `CALL BCONG (FCN, GRAD, IBTYPE, XLB, XUB, X [, ...])`

Specific:    The specific interface names are `S_BCONG` and `D_BCONG`.

## FORTRAN 77 Interface

Single:      CALL BCONG (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE,  
                  IPARAM, RPARAM, X, FVALUE)

Double:      The double precision name is DBCONG.

## Description

The routine BCONG uses a quasi-Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to  $l \leq x \leq u$

From a given starting point  $x^c$ , an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -B^{-1} g^c$$

where  $B$  is a positive definite approximation of the Hessian and  $g^c$  is the gradient evaluated at  $x^c$ ; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point  $x^n$ ,

$$x^n = x^c + \lambda d, \quad \lambda \in (0, 1]$$

such that

$$f(x^n) \leq f(x^c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \leq \epsilon, \quad l_i < x_i < u_i$$

$$g(x_i) < 0, \quad x_i = u_i$$

$$g(x_i) > 0, \quad x_i = l_i$$

are checked, where  $\epsilon$  is a gradient tolerance. When optimality is not achieved,  $B$  is updated according to the BFGS formula:

$$B \leftarrow B - \frac{B s s^T B}{s^T B s} + \frac{y y^T}{y^T s}$$

where  $s = x^n - x^c$  and  $y = g^n - g^c$ . Another search direction is then computed to begin the next iteration.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the quasi-Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2ONG/DB2ONG`. The reference is:

```
CALL B2ONG (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE,
 IPARAM, RPARAM, X, FVALUE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length  $N * (2 * N + 8)$ . `WK` contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain a `BFGS` approximation to the Hessian at the solution. Only the lower triangular portion of the matrix is stored in `WK`. The values returned in the upper triangle should be ignored.

**IWK** — Work vector of length  $N$  stored in column order.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 4 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4 | 5 | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 3 | 8 | The last global step failed to locate a lower point than the current <code>X</code> value.                                                                                                            |

3. The first stopping criterion for `BCONG` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `BCONG` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).
4. If the default parameters are desired for `BCONG`, then set `IPARAM(1)` to zero and call the routine `BCONG`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `BCONG`:

CALL U4INF (IPARAM, RPARAM)

Set nondefault values for desired IPARAM, RPARAM elements.

---

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter.  
If IPARAM (6) = 0, the Hessian is initialized to the identity matrix;  
otherwise, it is initialized to a diagonal matrix containing

$$\max(|f(t)|, f_s) * s_i^2$$

on the diagonal where  $t = \text{XGUESS}$ ,  $f_s = \text{FSCALE}$ , and  $s = \text{XSCALE}$ .  
Default: 0.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: Not used in BCONG.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default: Not used in BCONG.

RPARAM(4) = Absolute function tolerance.

Default: Not used in BCONG.

RPARAM(5) = False convergence tolerance.

Default: Not used in BCONG.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then `DU4INF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

## Example

The problem

$$\min f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

$$\text{subject to } -2 \leq x_1 \leq 0.5$$

$$-1 \leq x_2 \leq 2$$

is solved with an initial guess  $(-1.2, 1.0)$ , and default values for parameters.

```

USE BCONG_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
!
INTEGER IPARAM(7), ITP, L, NOUT
REAL F, X(N), XGUESS(N), XLB(N), XUB(N)
EXTERNAL ROSBRK, ROSGRD
!
DATA XGUESS/-1.2E0, 1.0E0/
DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
! All the bounds are provided
ITP = 0
! Default parameters are used
IPARAM(1) = 0
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL BCONG (ROSBRK, ROSGRD, ITP, XLB, XUB, X, XGUESS=XGUESS, &
IPARAM=IPARAM, FVALUE=F)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
'value is ', F8.3, '//, ' The number of iterations is ', &
10X, I3, '/', ' The number of function evaluations is ', &
I3, '/', ' The number of gradient evaluations is ', I3)
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
RETURN
END
!
SUBROUTINE ROSGRD (N, X, G)
INTEGER N
REAL X(N), G(N)
!
G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
G(2) = 2.0E2*(X(2)-X(1)*X(1))
!

```

```
RETURN
END
```

## Output

```
The solution is 0.500 0.250

The function value is 0.250

The number of iterations is 22
The number of function evaluations is 32
The number of gradient evaluations is 23
```

---

## BCODH

Minimizes a function of  $N$  variables subject to bounds on the variables using a modified Newton method and a finite-difference Hessian.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Length of X and G. (Input)

X — Vector of length N at which point the gradient is evaluated. (Input)  
X should not be changed by GRAD.

G — The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| IBTYPE | Action |
|--------|--------|
|--------|--------|

|   |                                  |
|---|----------------------------------|
| 0 | User will supply all the bounds. |
|---|----------------------------------|

|   |                                |
|---|--------------------------------|
| 1 | All variables are nonnegative. |
|---|--------------------------------|

- 2 All variables are nonpositive.
- 3 User supplies only the bounds on 1st variable, all other variables will have the same bounds.

*XLB* — Vector of length *N* containing the lower bounds on the variables. (Input)

*XUB* — Vector of length *N* containing the upper bounds on the variables. (Input)

*X* — Vector of length *N* containing the computed solution. (Output)

### Optional Arguments

*N* — Dimension of the problem. (Input)  
Default: *N* = SIZE (*X*,1).

*XGUESS* — Vector of length *N* containing the initial guess of the minimum. (Input)  
Default: *XGUESS* = 0.0.

*XSCALE* — Vector of length *N* containing the diagonal scaling matrix for the variables. (Input)  
*XSCALE* is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.  
Default: *XSCALE* = 1.0.

*FSCALE* — Scalar containing the function scaling. (Input)  
*FSCALE* is used mainly in scaling the gradient. In the absence of other information, set *FSCALE* to 1.0.  
Default: *FSCALE* = 1.0.

*IPARAM* — Parameter vector of length 7. (Input/Output)  
Set *IPARAM*(1) to zero for default values of *IPARAM* and *RPARAM*. See Comment 4.  
Default: *IPARAM* = 0.

*RPARAM* — Parameter vector of length 7. (Input/Output)  
See Comment 4.

*FVALUE* — Scalar containing the value of the function at the computed solution. (Output)

### FORTRAN 90 Interface

Generic: CALL BCODH (FCN, GRAD, IBTYPE, XLB, XUB, X [, ...])

Specific: The specific interface names are S\_BCODH and D\_BCODH.

### FORTRAN 77 Interface

Single: CALL BCODH (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)

Double: The double precision name is `DBCODH`.

## Description

The routine `BCODH` uses a modified Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as

$$\begin{aligned} \min_{x \in \mathbf{R}^n} f(x) \\ \text{subject to } l \leq x \leq u \end{aligned}$$

From a given starting point  $x^c$ , an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -H^{-1} g^c$$

where  $H$  is the Hessian and  $g^c$  is the gradient evaluated at  $x^c$ ; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point  $x^n$ ,

$$x^n = x^c + \lambda d, \quad \lambda \in (0, 1]$$

such that

$$f(x^n) \leq f(x^c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\|g(x_i)\| \leq \epsilon, \quad l_i < x_i < u_i$$

$$g(x_i) < 0, \quad x_i = u_i$$

$$g(x_i) > 0, \quad x_i = l_i$$

are checked where  $\epsilon$  is a gradient tolerance. When optimality is not achieved, another search direction is computed to begin the next iteration. This process is repeated until the optimality criterion is met.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the modified Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Hessian for some single precision calculations, an inaccurate estimate of the Hessian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Hessian can be easily provided, routine `BCOAH` should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of B2ODH/DB2ODH. The reference is:

```
CALL B2ODH (FCN, GRAD, N, XGUESS, IBTYPE, XLB, XUB,
 XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Real work vector of length  $N * (N + 8)$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Hessian at the approximate solution.

**IWK** — Integer work vector of length  $N$ .

2. Informational errors

| Type | Code |                                                                                                                                                                                          |
|------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                 |
| 4    | 2    | The iterates appear to be converging to a noncritical point.                                                                                                                             |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                                                                   |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                                                                         |
| 4    | 5    | Maximum number of gradient evaluations exceeded.                                                                                                                                         |
| 4    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                                                                     |
| 2    | 7    | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or STEPTL is too big. |
| 4    | 7    | Maximum number of Hessian evaluations exceeded.                                                                                                                                          |

3. The first stopping criterion for B2ODH occurs when the norm of the gradient is less than the given gradient tolerance (RPARAM(1)). The second stopping criterion for B2ODH occurs when the scaled distance between the last two steps is less than the step tolerance (RPARAM(2)).
4. If the default parameters are desired for B2ODH, then set IPARAM(1) to zero and call the routine B2ODH. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM; then the following steps should be taken before calling B2ODH:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired IPARAM, RPARAM elements.

---

Note that the call to U4INF will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter.  
Default: Not used in BCODH.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: 100.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)  
The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default:  $\max(10^{-10}, \epsilon^{2/3})$ ,  $\max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in BCODH.

RPARAM(5) = False convergence tolerance.

Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then DU4INF is called and RPARAM is declared double precision.

- Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to "Error Handling" in the Introduction.

### Example

The problem

$$\begin{aligned} \min f(x) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{subject to } &-2 \leq x_1 \leq 0.5 \\ &-1 \leq x_2 \leq 2 \end{aligned}$$

is solved with an initial guess  $(-1.2, 1.0)$ , and default values for parameters.

```
USE BCODH_INT
USE UMACH_INT
```

```

 IMPLICIT NONE
 INTEGER N
 PARAMETER (N=2)
!
 INTEGER IP, IPARAM(7), L, NOUT
 REAL F, X(N), XGUESS(N), XLB(N), XUB(N)
 EXTERNAL ROSBRK, ROSGRD
!
 DATA XGUESS/-1.2E0, 1.0E0/
 DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
 IPARAM(1) = 0
 IP = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
 CALL BCODH (ROSBRK, ROSGRD, IP, XLB, XUB, X, XGUESS=XGUESS, &
 IPARAM=IPARAM, FVALUE=F)
!
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
 'value is ', F8.3, '//, ' The number of iterations is ', &
 10X, I3, '/', ' The number of function evaluations is ', &
 I3, '/', ' The number of gradient evaluations is ', I3)
!
 END
!
 SUBROUTINE ROSBRK (N, X, F)
 INTEGER N
 REAL X(N), F
!
 F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
 RETURN
 END
 SUBROUTINE ROSGRD (N, X, G)
 INTEGER N
 REAL X(N), G(N)
!
 G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
 G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
 RETURN
 END

```

## Output

```

The solution is 0.500 0.250

The function value is 0.250

The number of iterations is 17
The number of function evaluations is 26
The number of gradient evaluations is 18

```

---

# BCOAH

Minimizes a function of  $N$  variables subject to bounds on the variables using a modified Newton method and a user-supplied Hessian.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Length of X. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Length of X and G. (Input)

X — Vector of length N at which point the gradient is evaluated. (Input)  
X should not be changed by GRAD.

G — The gradient evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

**HESS** — User-supplied subroutine to compute the Hessian at the point X. The usage is  
CALL HESS (N, X, H, LDH), where

N — Length of X. (Input)

X — Vector of length N at which point the Hessian is evaluated. (Input)  
X should not be changed by HESS.

H — The Hessian evaluated at the point X. (Output)

LDH — Leading dimension of H exactly as specified in the dimension statement of the calling program. (Input)

HESS must be declared EXTERNAL in the calling program.

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| IBYTE | Action                           |
|-------|----------------------------------|
| 1     | User will supply all the bounds. |
| 2     | All variables are nonnegative.   |

User supplies only the bounds on 1st variable, all other variables will have the same bounds.

***XLB*** — Vector of length  $N$  containing the lower bounds on the variables. (Input)

***XUB*** — Vector of length  $N$  containing the upper bounds on the variables. (Input)

***X*** — Vector of length  $N$  containing the computed solution. (Output)

### Optional Arguments

***N*** — Dimension of the problem. (Input)

Default:  $N = \text{SIZE}(X,1)$ .

***XGUESS*** — Vector of length  $N$  containing the initial guess. (Input)

Default:  $XGUESS = 0.0$ .

***XSCALE*** — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)

*XSCALE* is used mainly in scaling the gradient and the distance between two points. In the absence of other information, set all entries to 1.0.

Default:  $XSCALE = 1.0$ .

***FSCALE*** — Scalar containing the function scaling. (Input)

*FSCALE* is used mainly in scaling the gradient. In the absence of other information, set *FSCALE* to 1.0.

Default:  $FSCALE = 1.0$ .

***IPARAM*** — Parameter vector of length 7. (Input/Output)

Set *IPARAM*(1) to zero for default values of *IPARAM* and *RPARAM*. See Comment 4.

Default:  $IPARAM = 0$ .

***RPARAM*** — Parameter vector of length 7. (Input/Output)

See Comment 4.

***FVALUE*** — Scalar containing the value of the function at the computed solution. (Output)

### FORTRAN 90 Interface

Generic: `CALL BCOAH (FCN, GRAD, HESS, IBTYPE, XLB, XUB, X [, ...])`

Specific: The specific interface names are `S_BCOAH` and `D_BCOAH`.

### FORTRAN 77 Interface

Single: `CALL BCOAH (FCN, GRAD, HESS, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE)`

Double: The double precision name is DBCOAH.

## Description

The routine BCOAH uses a modified Newton method and an active set strategy to solve minimization problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to  $l \leq x \leq u$

From a given starting point  $x^c$ , an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -H^{-1} g^c$$

where  $H$  is the Hessian and  $g^c$  is the gradient evaluated at  $x^c$ ; both are computed with respect to the free variables. The search direction for the variables in IA is set to zero. A line search is used to find a new point  $x^n$ ,

$$x^n = x^c + \lambda d, \quad \lambda \in (0, 1]$$

such that

$$f(x^n) \leq f(x^c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality conditions

$$\begin{aligned} \|g(x_i)\| &\leq \varepsilon, \quad l_i < x_i < u_i \\ g(x_i) &< 0, \quad x_i = u_i \\ g(x_i) &> 0, \quad x_i = l_i \end{aligned}$$

are checked where  $\varepsilon$  is a gradient tolerance. When optimality is not achieved, another search direction is computed to begin the next iteration. This process is repeated until the optimality criterion is met.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more details on the modified Newton method and line search, see Dennis and Schnabel (1983). For more detailed information on active set strategy, see Gill and Murray (1976).

## Comments

1. Workspace may be explicitly provided, if desired, by use of B2OAH/DB2OAH. The reference is:

```
CALL B2OAH (FCN, GRAD, HESS, N, XGUESS, IBTYPE, XLB, XUB,
 XSCALE, FSCALE, IPARAM, RPARAM, X, FVALUE, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $N * (N + 8)$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution. The final  $N^2$  locations contain the Hessian at the approximate solution.

**IWK** — Work vector of length  $N$ .

2. Informational errors

Type Code

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 4 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 4 | 5 | Maximum number of gradient evaluations exceeded.                                                                                                                                                      |
| 4 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |
| 4 | 7 | Maximum number of Hessian evaluations exceeded.                                                                                                                                                       |
| 3 | 8 | The last global step failed to locate a lower point than the current $x$ value.                                                                                                                       |

3. The first stopping criterion for `BCOAH` occurs when the norm of the gradient is less than the given gradient tolerance (`RPARAM(1)`). The second stopping criterion for `BCOAH` occurs when the scaled distance between the last two steps is less than the step tolerance (`RPARAM(2)`).

4. If the default parameters are desired for `BCOAH`, then set `IPARAM(1)` to zero and call the routine `BCOAH`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `BCOAH`:

```
CALL U4INF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4INF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 7.

IPARAM(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of gradient evaluations.  
Default: 400.

IPARAM(6) = Hessian initialization parameter.  
Default: Not used in BCOAH.

IPARAM(7) = Maximum number of Hessian evaluations.  
Default: 100.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.

The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where  $g = \nabla f(x)$ ,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = XSCALE$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default:  $\max(10^{-10}, \epsilon^{2/3}), \max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default: Not used in BCOAH.

RPARAM(5) = False convergence tolerance.

Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = XSCALE$ , and  $t = XGUESS$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is required, then `DU4INF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The problem

$$\begin{aligned} \min f(x) &= 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{subject to } &-2 \leq x_1 \leq 0.5 \\ &-1 \leq x_2 \leq 2 \end{aligned}$$

is solved with an initial guess  $(-1.2, 1.0)$ , and default values for parameters.

```
USE BCOAH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=2)
```

```

!
INTEGER IP, IPARAM(7), L, NOUT
REAL F, X(N), XGUESS(N), XLB(N), XUB(N)
EXTERNAL ROSBRK, ROSGRD, ROSHES
!
DATA XGUESS/-1.2E0, 1.0E0/
DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
IPARAM(1) = 0
IP = 0
!
! Minimize Rosenbrock function using
! initial guesses of -1.2 and 1.0
CALL BCOAH (ROSBRK, ROSGRD, ROSHES, IP, XLB, XUB, X, &
 XGUESS=XGUESS, IPARAM=IPARAM, FVALUE=F)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, F, (IPARAM(L),L=3,5), IPARAM(7)
!
99999 FORMAT (' The solution is ', 6X, 2F8.3, '//, ' The function ', &
 'value is ', F8.3, '//, ' The number of iterations is ', &
 10X, I3, '/', ' The number of function evaluations is ', &
 I3, '/', ' The number of gradient evaluations is ', I3, '/', &
 ' The number of Hessian evaluations is ', I3)
!
END
!
SUBROUTINE ROSBRK (N, X, F)
INTEGER N
REAL X(N), F
!
F = 1.0E2*(X(2)-X(1)*X(1))**2 + (1.0E0-X(1))**2
!
RETURN
END
!
SUBROUTINE ROSGRD (N, X, G)
INTEGER N
REAL X(N), G(N)
!
G(1) = -4.0E2*(X(2)-X(1)*X(1))*X(1) - 2.0E0*(1.0E0-X(1))
G(2) = 2.0E2*(X(2)-X(1)*X(1))
!
RETURN
END
!
SUBROUTINE ROSHES (N, X, H, LDH)
INTEGER N, LDH
REAL X(N), H(LDH,N)
!
H(1,1) = -4.0E2*X(2) + 1.2E3*X(1)*X(1) + 2.0E0
H(2,1) = -4.0E2*X(1)
H(1,2) = H(2,1)
H(2,2) = 2.0E2
!
RETURN
END

```

## Output

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 18
The number of function evaluations is 29
The number of gradient evaluations is 19
The number of Hessian evaluations is 18
```

---

## BCPOL

Minimizes a function of  $N$  variables subject to bounds on the variables using a direct search complex algorithm.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

**N** — Length of  $X$ . (Input)

**X** — Vector of length  $N$  at which point the function is evaluated. (Input)  
 $X$  should not be changed by FCN.

**F** — The computed function value at the point  $X$ . (Output)

FCN must be declared EXTERNAL in the calling program.

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| <b>IBTYPE</b> | <b>Action</b> |
|---------------|---------------|
|---------------|---------------|

|   |                                  |
|---|----------------------------------|
| 0 | User will supply all the bounds. |
|---|----------------------------------|

|   |                                |
|---|--------------------------------|
| 1 | All variables are nonnegative. |
|---|--------------------------------|

|   |                                |
|---|--------------------------------|
| 2 | All variables are nonpositive. |
|---|--------------------------------|

|   |                                                                                                      |
|---|------------------------------------------------------------------------------------------------------|
| 3 | User supplies only the bounds on the first, variable. All other variables will have the same bounds. |
|---|------------------------------------------------------------------------------------------------------|

**XLB** — Vector of length  $N$  containing the lower bounds on the variables. (Input, if  
IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

**XUB** — Vector of length  $N$  containing the upper bounds on the variables. (Input, if  
IBTYPE = 0; output, if IBTYPE = 1 or 2; input/output, if IBTYPE = 3)

**X** — Real vector of length  $N$  containing the best estimate of the minimum found. (Output)

## Optional Arguments

***N*** — The number of variables. (Input)  
Default:  $N = \text{SIZE}(X\text{GUESS},1)$ .

***XGUESS*** — Real vector of length  $N$  that contains an initial guess to the minimum. (Input)  
Default:  $X\text{GUESS} = 0.0$ .

***FTOL*** — First convergence criterion. (Input)  
The algorithm stops when a relative error in the function values is less than  $FTOL$ , i.e. when  $(F(\text{worst}) - F(\text{best})) < FTOL * (1 + \text{ABS}(F(\text{best})))$  where  $F(\text{worst})$  and  $F(\text{best})$  are the function values of the current worst and best point, respectively. Second convergence criterion. The algorithm stops when the standard deviation of the function values at the  $2 * N$  current points is less than  $FTOL$ . If the subroutine terminates prematurely, try again with a smaller value  $FTOL$ .  
Default:  $FTOL = 1.0\text{e-}4$  for single and  $1.0\text{d-}8$  for double precision.

***MAXFCN*** — On input, maximum allowed number of function evaluations. (Input/ Output)  
On output, actual number of function evaluations needed.  
Default:  $\text{MAXFCN} = 300$ .

***FVALUE*** — Function value at the computed solution. (Output)

## FORTRAN 90 Interface

Generic: `CALL BCPOL (FCN, IBTYPE, XLB, XUB, X [, ...])`

Specific: The specific interface names are `S_BCPOL` and `D_BCPOL`.

## FORTRAN 77 Interface

Single: `CALL BCPOL (FCN, N, XGUESS, IBTYPE, XLB, XUB, FTOL, MAXFCN, X, FVALUE)`

Double: The double precision name is `DBC POL`.

## Description

The routine `BCPOL` uses the complex method to find a minimum point of a function of  $n$  variables. The method is based on function comparison; no smoothness is assumed. It starts with  $2n$  points  $x_1, x_2, \dots, x_{2n}$ . At each iteration, a new point is generated to replace the worst point  $x_j$ , which has the largest function value among these  $2n$  points. The new point is constructed by the following formula:

$$x_k = c + \alpha(c - x_j)$$

where

$$c = \frac{1}{2n-1} \sum_{i \neq j} x_i$$

and  $\alpha$  ( $\alpha > 0$ ) is the *reflection coefficient*.

When  $x_k$  is a best point, that is, when  $f(x_k) \leq f(x_i)$  for  $i = 1, \dots, 2n$ , an expansion point is computed  $x_e = c + \beta(x_k - c)$ , where  $\beta(\beta > 1)$  is called the *expansion coefficient*. If the new point is a worst point, then the complex would be contracted to get a better new point. If the contraction step is unsuccessful, the complex is shrunk by moving the vertices halfway toward the current best point. Whenever the new point generated is beyond the bound, it will be set to the bound. This procedure is repeated until one of the following stopping criteria is satisfied:

Criterion 1:

$$f_{best} - f_{worst} \leq \epsilon_f(1. + |f_{best}|)$$

Criterion 2:

$$\sum_{i=1}^{2n} \left( f_i - \frac{\sum_{j=1}^{2n} f_j}{2n} \right)^2 \leq \epsilon_f$$

where  $f_i = f(x_i)$ ,  $f_j = f(x_j)$ , and  $\epsilon_f$  is a given tolerance. For a complete description, see Nelder and Mead (1965) or Gill et al. (1981).

## Comments

1. Workspace may be explicitly provided, if desired, by use of B2POL/DB2POL. The reference is:

```
CALL B2POL (FCN, N, XGUESS, IBTYPE, XLB, XUB, FTOL,
 MAXFCN, X, FVALUE, WK)
```

The additional argument is:

**WK** — Real work vector of length  $2 * N**2 + 5 * N$

2. Informational error

Type Code

3 1 The maximum number of function evaluations is exceeded.

3. Since BCPOL uses only function-value information at each step to determine a new approximate minimum, it could be quite inefficient on smooth problems compared to other methods such as those implemented in routine BCONF, which takes into account derivative information at each iteration. Hence, routine BCPOL should only be used as a last resort. Briefly, a set of  $2 * N$  points in an  $N$ -dimensional space is called a complex. The minimization process iterates by replacing the point with the largest function value by a new point with a smaller function value. The iteration continues until all the points cluster sufficiently close to a minimum.

## Example

The problem

$$\min f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

subject to  $-2 \leq x_1 \leq 0.5$

$-1 \leq x_2 \leq 2$

is solved with an initial guess  $(-1.2, 1.0)$ , and the solution is printed.

```

USE BCPOL_INT
USE UMACH_INT

IMPLICIT NONE
! Variable declarations
INTEGER N
PARAMETER (N=2)

!
INTEGER IBTYPE, K, NOUT
REAL FTOL, FVALUE, X(N), XGUESS(N), XLB(N), XUB(N)
EXTERNAL FCN

!
! Initializations
! XGUESS = (-1.2, 1.0)
! XLB = (-2.0, -1.0)
! XUB = (0.5, 2.0)
DATA XGUESS/-1.2, 1.0/, XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/

!
FTOL = 1.0E-5
IBTYPE = 0

!
CALL BCPOL (FCN, IBTYPE, XLB, XUB, X, xguess=xguess, ftol=ftol, &
 fvalue=fvalue)

!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (X(K),K=1,N), FVALUE
99999 FORMAT (' The best estimate for the minimum value of the', /, &
 ' function is X = (' , 2(2X,F4.2), ')', /, ' with ', &
 'function value FVALUE = ', E12.6)

!
END

! External function to be minimized
SUBROUTINE FCN (N, X, F)
INTEGER N
REAL X(N), F

!
F = 100.0*(X(2)-X(1)*X(1))**2 + (1.0-X(1))**2
RETURN
END

```

## Output

The best estimate for the minimum value of the  
function is X = ( 0.50 0.25)  
with function value FVALUE = 0.250002E+00

---

# BCLSF



Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

## Required Arguments

*FCN* — User-supplied subroutine to evaluate the function to be minimized. The usage is  
`CALL FCN (M, N, X, F)`, where

*M* — Length of *F*. (Input)

*N* — Length of *X*. (Input)

*X* — The point at which the function is evaluated. (Input)  
*X* should not be changed by *FCN*.

*F* — The computed function at the point *X*. (Output)

*FCN* must be declared `EXTERNAL` in the calling program.

*M* — Number of functions. (Input)

*IBTYPE* — Scalar indicating the types of bounds on variables. (Input)

| <b>IBTYPE</b> | <b>Action</b> |
|---------------|---------------|
|---------------|---------------|

|   |                                  |
|---|----------------------------------|
| 0 | User will supply all the bounds. |
|---|----------------------------------|

|   |                                |
|---|--------------------------------|
| 1 | All variables are nonnegative. |
|---|--------------------------------|

|   |                                |
|---|--------------------------------|
| 2 | All variables are nonpositive. |
|---|--------------------------------|

|   |                                                                                               |
|---|-----------------------------------------------------------------------------------------------|
| 3 | User supplies only the bounds on 1st variable, all other variables will have the same bounds. |
|---|-----------------------------------------------------------------------------------------------|

*XLB* — Vector of length *N* containing the lower bounds on variables. (Input, if *IBTYPE* = 0; output, if *IBTYPE* = 1 or 2; input/output, if *IBTYPE* = 3)

*XUB* — Vector of length *N* containing the upper bounds on variables. (Input, if *IBTYPE* = 0; output, if *IBTYPE* = 1 or 2; input/output, if *IBTYPE* = 3)

*X* — Vector of length *N* containing the approximate solution. (Output)

## Optional Arguments

*N* — Number of variables. (Input)

*N* must be less than or equal to *M*.

Default: *N* = SIZE (*X*,1).

*XGUESS* — Vector of length *N* containing the initial guess. (Input)

Default: *XGUESS* = 0.0.

*XSCALE* — Vector of length *N* containing the diagonal scaling matrix for the variables.

(Input)

*XSCALE* is used mainly in scaling the gradient and the distance between two points. By default, the values for *XSCALE* are set internally. See *IPARAM*(6) in Comment 4.

*FSCALE* — Vector of length *M* containing the diagonal scaling matrix for the functions.

(Input)

*FSCALE* is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.

Default: *FSCALE* = 1.0.

*IPARAM* — Parameter vector of length 6. (Input/Output)

Set *IPARAM*(1) to zero for default values of *IPARAM* and *RPARAM*. See Comment 4.

Default: *IPARAM* = 0.

*RPARAM* — Parameter vector of length 7. (Input/Output)

See Comment 4.

*FVEC* — Vector of length *M* containing the residuals at the approximate solution. (Output)

*FJAC* — *M* by *N* matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

*LDFJAC* — Leading dimension of *FJAC* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDFJAC* = SIZE (*FJAC* ,1).

## FORTRAN 90 Interface

Generic: CALL BCLSF (FCN, M, IBTYPE, XLB, XUB, X [, ...])

Specific: The specific interface names are S\_BCLSF and D\_BCLSF.

## FORTRAN 77 Interface

Single: CALL BCLSF (FCN, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC)

Double: The double precision name is DBCLSF.

## Description

The routine `BCLSF` uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2$$

subject to  $l \leq x \leq u$

where  $m \geq n$ ,  $F: \mathbf{R}^n \rightarrow \mathbf{R}^m$ , and  $f_i(x)$  is the  $i$ -th component function of  $F(x)$ . From a given starting point, an active set `IA`, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where  $\mu$  is the Levenberg-Marquardt parameter,  $F = F(x)$ , and  $J$  is the Jacobian with respect to the free variables. The search direction for the variables in `IA` is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$\begin{aligned} \|g(x_i)\| &\leq \epsilon, \quad l_i < x_i < u_i \\ g(x_i) &< 0, \quad x_i = u_i \\ g(x_i) &> 0, \quad x_i = l_i \end{aligned}$$

where  $\epsilon$  is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in `IA`, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of `IA`. For more detail on the Levenberg-Marquardt method, see Levenberg (1944), or Marquardt (1963). For more detailed information on active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Jacobian for some single precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, routine `BCLSJ` should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LSF/DB2LSF`. The reference is:

```
CALL B2LSF (FCN, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE,
 FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC, WK,
 IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $11 * N + 3 * M - 1$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Gauss-Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution.

**IWK** — Work vector of length  $2 * N$  containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

| Type | Code |                                                                                                                                                                                                 |
|------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3    | 1    | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                        |
| 3    | 2    | The iterates appear to be converging to a noncritical point.                                                                                                                                    |
| 4    | 3    | Maximum number of iterations exceeded.                                                                                                                                                          |
| 4    | 4    | Maximum number of function evaluations exceeded.                                                                                                                                                |
| 3    | 6    | Five consecutive steps have been taken with the maximum step length.                                                                                                                            |
| 2    | 7    | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <b>STEPTL</b> is too big. |

3. The first stopping criterion for **BCLSF** occurs when the norm of the function is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance. The third stopping criterion for **BCLSF** occurs when the scaled distance between the last two steps is less than the step tolerance.

4. If the default parameters are desired for **BCLSF**, then set **IPARAM**(1) to zero and call the routine **BCLSF**. Otherwise, if any nondefault parameters are desired for **IPARAM** or **RPARAM**, then the following steps should be taken before calling **BCLSF**:

```
CALL U4LSF (IPARAM, RPARAM)
```

Set nondefault values for desired **IPARAM**, **RPARAM** elements.

---

Note that the call to **U4LSF** will set **IPARAM** and **RPARAM** to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

**IPARAM** — Integer vector of length 6.

**IPARAM**(1) = Initialization flag.

IPARAM(2) = Number of good digits in the function.  
Default: Machine dependent.

IPARAM(3) = Maximum number of iterations.  
Default: 100.

IPARAM(4) = Maximum number of function evaluations.  
Default: 400.

IPARAM(5) = Maximum number of Jacobian evaluations.  
Default: 100.

IPARAM(6) = Internal variable scaling flag.  
If IPARAM(6) = 1, then the values for XSCALE are set internally.  
Default: 1.

**RPARAM** — Real vector of length 7.

RPARAM(1) = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\|F(x)\|_2^2}$$

where

$$g_i = \left( J(x)^T F(x) \right)_i * (f_s)_i^2$$

$J(x)$  is the Jacobian,  $s = \text{XSCALE}$ , and  $f_s = \text{FSCALE}$ .  
Default:

$$\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$$

in double where  $\varepsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)  
The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\varepsilon^{2/3}$  where  $\varepsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default:  $\max(10 \epsilon^{2/3}, \max(10^{-20}, \epsilon^{2/3}))$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default:  $\max(10^{-20} \epsilon^2), \max(10^{-40}, \epsilon^2)$  in double where  $\epsilon$  is the machine precision.

RPARAM(5) = False convergence tolerance.

Default:  $100 \epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step size.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then `DU4LSF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to “Error Handling” in the Introduction.

## Example

The nonlinear least squares problem

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^2 f_i(x)^2$$

subject to  $-2 \leq x_1 \leq 0.5$

$$-1 \leq x_2 \leq 2$$

where

$$f_1(x) = 10(x_2 - x_1^2) \text{ and } f_2(x) = (1 - x_1)$$

is solved with an initial guess  $(-1.2, 1.0)$  and default values for parameters.

```
USE BCLSF_INT
```

```
USE UMACH_INT
```

```
IMPLICIT NONE
```

```
!
```

```
Declaration of variables
```

```

 INTEGER M, N
 PARAMETER (M=2, N=2)
!
 INTEGER IPARAM(7), ITP, NOUT
 REAL FSCALE(M), FVEC(M), X(N), XGUESS(N), XLB(N), XS(N), XUB(N)
 EXTERNAL ROSBCK
!
! Compute the least squares for the
! Rosenbrock function.
 DATA XGUESS/-1.2E0, 1.0E0/
 DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
!
! All the bounds are provided
 ITP = 0
!
! Default parameters are used
 IPARAM(1) = 0
!
 CALL BCLSF (ROSBCK, M, ITP, XLB, XUB, X, xguess=xguess, &
 iparam=iparam, fvec=fvec)
!
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT (' The solution is ', 2F9.4, '//, ' The function ', &
 'evaluated at the solution is ',/, 18X, 2F9.4, '//, &
 ' The number of iterations is ', 10X, I3, '//, ' The ', &
 'number of function evaluations is ', I3, '/')
 END
!
 SUBROUTINE ROSBCK (M, N, X, F)
 INTEGER M, N
 REAL X(N), F(M)
!
 F(1) = 1.0E1*(X(2)-X(1)*X(1))
 F(2) = 1.0E0 - X(1)
 RETURN
 END

```

## Output

```

The solution is 0.5000 0.2500

The function evaluated at the solution is
0.0000 0.5000

The number of iterations is 15
The number of function evaluations is 20

```

---

## BCLSJ



Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
`CALL FCN (M, N, X, F)`, where

**M** — Length of **F**. (Input)

**N** — Length of **X**. (Input)

**X** — The point at which the function is evaluated. (Input)  
**X** should not be changed by **FCN**.

**F** — The computed function at the point **X**. (Output)

**FCN** must be declared **EXTERNAL** in the calling program.

**JAC** — User-supplied subroutine to evaluate the Jacobian at a point **X**. The usage is  
`CALL JAC (M, N, X, FJAC, LDFJAC)`, where

**M** — Length of **F**. (Input)

**N** — Length of **X**. (Input)

**X** — The point at which the function is evaluated. (Input)  
**X** should not be changed by **FCN**.

**FJAC** — The computed **M** by **N** Jacobian at the point **X**. (Output)

**LDFJAC** — Leading dimension of **FJAC**. (Input)

**JAC** must be declared **EXTERNAL** in the calling program.

**M** — Number of functions. (Input)

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| <b>IBTYPE</b> | <b>Action</b>                    |
|---------------|----------------------------------|
| 0             | User will supply all the bounds. |
| 1             | All variables are nonnegative.   |
| 2             | All variables are nonpositive.   |

3

User supplies only the bounds on 1st variable, all other variables will have the same bounds.

***XLB*** — Vector of length  $N$  containing the lower bounds on variables. (Input, if  $IBTYPE = 0$ ; output, if  $IBTYPE = 1$  or  $2$ ; input/output, if  $IBTYPE = 3$ )

***XUB*** — Vector of length  $N$  containing the upper bounds on variables. (Input, if  $IBTYPE = 0$ ; output, if  $IBTYPE = 1$  or  $2$ ; input/output, if  $IBTYPE = 3$ )

***X*** — Vector of length  $N$  containing the approximate solution. (Output)

### Optional Arguments

***N*** — Number of variables. (Input)  
 $N$  must be less than or equal to  $M$ .  
Default:  $N = \text{SIZE}(X,1)$ .

***XGUESS*** — Vector of length  $N$  containing the initial guess. (Input)  
Default:  $XGUESS = 0.0$ .

***XSCALE*** — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)  
 $XSCALE$  is used mainly in scaling the gradient and the distance between two points. By default, the values for  $XSCALE$  are set internally. See  $IPARAM(6)$  in Comment 4.

***FSCALE*** — Vector of length  $M$  containing the diagonal scaling matrix for the functions. (Input)  
 $FSCALE$  is used mainly in scaling the gradient. In the absence of other information, set all entries to 1.0.  
Default:  $FSCALE = 1.0$ .

***IPARAM*** — Parameter vector of length 6. (Input/Output)  
Set  $IPARAM(1)$  to zero for default values of  $IPARAM$  and  $RPARAM$ . See Comment 4.  
Default:  $IPARAM = 0$ .

***RPARAM*** — Parameter vector of length 7. (Input/Output)  
See Comment 4.

***FVEC*** — Vector of length  $M$  containing the residuals at the approximate solution. (Output)

***FJAC*** —  $M$  by  $N$  matrix containing a finite difference approximate Jacobian at the approximate solution. (Output)

***LDFJAC*** — Leading dimension of  $FJAC$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDFJAC \text{ SIZE} = (FJAC,1)$ .

## FORTRAN 90 Interface

Generic: CALL BCLSJ (FCN, JAC, M, IBTYPE, XLB, XUB, X [, ...])

Specific: The specific interface names are S\_BCLSJ and D\_BCLSJ.

## FORTRAN 77 Interface

Single: CALL BCLSJ (FCN, JAC, M, N, XGUESS, IBTYPE, XLB, XUB, XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC, LDFJAC)

Double: The double precision name is DBCLSJ.

## Description

The routine BCLSJ uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^m f_i(x)^2$$

subject to  $l \leq x \leq u$

where  $m \geq n$ ,  $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$ , and  $f_i(x)$  is the  $i$ -th component function of  $F(x)$ . From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where  $\mu$  is the Levenberg-Marquardt parameter,  $F = F(x)$ , and  $J$  is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$\|g(x_i)\| \leq \varepsilon, \quad l_t < x_t < u_t$$

$$g(x_t) < 0, \quad x_t = u_t$$

$$g(x_t) > 0, \quad x_t = l_t$$

where  $\varepsilon$  is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detailed information on active set strategy, see Gill and Murray (1976).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2LSJ/DB2LSJ`. The reference is:

```
CALL B2LSJ (FCN, JAC, M, N, XGUESS, IBTYPE, XLB, XUB,
 XSCALE, FSCALE, IPARAM, RPARAM, X, FVEC, FJAC,
 LDFJAC, WK, IWK)
```

The additional arguments are as follows:

**WK** — Work vector of length  $11 * N + 3 * M - 1$ . **WK** contains the following information on output: The second  $N$  locations contain the last step taken. The third  $N$  locations contain the last Gauss-Newton step. The fourth  $N$  locations contain an estimate of the gradient at the solution.

**IWK** — Work vector of length  $2 * N$  containing the permutations used in the QR factorization of the Jacobian at the solution.

2. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                                                                                       |
|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.                                                              |
| 3 | 2 | The iterates appear to be converging to a noncritical point.                                                                                                                                          |
| 4 | 3 | Maximum number of iterations exceeded.                                                                                                                                                                |
| 4 | 4 | Maximum number of function evaluations exceeded.                                                                                                                                                      |
| 3 | 6 | Five consecutive steps have been taken with the maximum step length.                                                                                                                                  |
| 4 | 5 | Maximum number of Jacobian evaluations exceeded.                                                                                                                                                      |
| 2 | 7 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or <code>STEPTL</code> is too big. |

3. The first stopping criterion for `BCLSJ` occurs when the norm of the function is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance. The third stopping criterion for `BCLSJ` occurs when the scaled distance between the last two steps is less than the step tolerance.
4. If the default parameters are desired for `BCLSJ`, then set `IPARAM(1)` to zero and call the routine `BCLSJ`. Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, then the following steps should be taken before calling `BCLSJ`:

```
CALL U4LSF (IPARAM, RPARAM)
```

Set nondefault values for desired `IPARAM`, `RPARAM` elements.

---

Note that the call to `U4LSF` will set `IPARAM` and `RPARAM` to their default values so only nondefault values need to be set above.

---

The following is a list of the parameters and the default values:

***IPARAM*** — Integer vector of length 6.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = Number of good digits in the function.  
Default: Machine dependent.

`IPARAM(3)` = Maximum number of iterations.  
Default: 100.

`IPARAM(4)` = Maximum number of function evaluations.  
Default: 400.

`IPARAM(5)` = Maximum number of Jacobian evaluations.  
Default: 100.

`IPARAM(6)` = Internal variable scaling flag.

If `IPARAM(6) = 1`, then the values for `XSCALE` are set internally.  
Default: 1.

***RPARAM*** — Real vector of length 7.

`RPARAM(1)` = Scaled gradient tolerance.  
The  $i$ -th component of the scaled gradient at  $x$  is calculated as

$$\frac{|g_i| * \max(|x_i|, 1/s_i)}{\|F(x)\|_2^2}$$

where

$$g_i = \left( J(x)^T F(x) \right)_i * (f_s)_i^2$$

$J(x)$  is the Jacobian,  $s = XSCALE$ , and  $f_s = FSCALE$ .  
Default:

$$\sqrt{\epsilon}, \sqrt[3]{\epsilon}$$

in double where  $\epsilon$  is the machine precision.

RPARAM(2) = Scaled step tolerance. (STEPTL)

The  $i$ -th component of the scaled step between two points  $x$  and  $y$  is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where  $s = \text{XSCALE}$ .

Default:  $\epsilon^{2/3}$  where  $\epsilon$  is the machine precision.

RPARAM(3) = Relative function tolerance.

Default:  $\max(10^{-10}, \epsilon^{2/3}), \max(10^{-20}, \epsilon^{2/3})$  in double where  $\epsilon$  is the machine precision.

RPARAM(4) = Absolute function tolerance.

Default:  $\max(10^{-20}, \epsilon^2), \max(10^{-40}, \epsilon^2)$  in double where  $\epsilon$  is the machine precision.

RPARAM(5) = False convergence tolerance.

Default:  $100\epsilon$  where  $\epsilon$  is the machine precision.

RPARAM(6) = Maximum allowable step SIZE.

Default:  $1000 \max(\epsilon_1, \epsilon_2)$  where

$$\epsilon_1 = \sqrt{\sum_{i=1}^n (s_i t_i)^2}$$

$\epsilon_2 = \|s\|_2$ ,  $s = \text{XSCALE}$ , and  $t = \text{XGUESS}$ .

RPARAM(7) = Size of initial trust region radius.

Default: based on the initial scaled Cauchy step.

If double precision is desired, then `DU4LSF` is called and `RPARAM` is declared double precision.

5. Users wishing to override the default print/stop attributes associated with error messages issued by this routine are referred to `ERROR HANDLING` in the Introduction.

## Example

The nonlinear least squares problem

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^2 f_i(x)^2$$

subject to  $-2 \leq x_1 \leq 0.5$

$-1 \leq x_2 \leq 2$

where

$$f_1(x) = 10(x_2 - x_1^2) \text{ and } f_2(x) = (1 - x_1)$$

is solved with an initial guess  $(-1.2, 1.0)$  and default values for parameters.

```
USE BCLSJ_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER LDFJAC, M, N
PARAMETER (LDFJAC=2, M=2, N=2)
!
INTEGER IPARAM(7), ITP, NOUT
REAL FVEC(M), RPARAM(7), X(N), XGUESS(N), XLB(N), XUB(N)
EXTERNAL ROSBCK, ROSJAC
! Compute the least squares for the
! Rosenbrock function.
DATA XGUESS/-1.2E0, 1.0E0/
DATA XLB/-2.0E0, -1.0E0/, XUB/0.5E0, 2.0E0/
! All the bounds are provided
ITP = 0
! Default parameters are used
IPARAM(1) = 0
!
CALL BCLSJ (ROSBCK,ROSJAC,M,ITP, XLB,XUB,X,XGUESS=XGUESS, &
 IPARAM=IPARAM, FVEC=FVEC)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) X, FVEC, IPARAM(3), IPARAM(4)
!
99999 FORMAT (' The solution is ', 2F9.4, '//, ' The function ', &
 'evaluated at the solution is ',/, 18X, 2F9.4, '//, &
 ' The number of iterations is ', 10X, I3, '/', ' The ', &
 'number of function evaluations is ', I3, /)
END
!
SUBROUTINE ROSBCK (M, N, X, F)
INTEGER M, N
REAL X(N), F(M)
!
F(1) = 1.0E1*(X(2)-X(1)*X(1))
F(2) = 1.0E0 - X(1)
RETURN
END
!
SUBROUTINE ROSJAC (M, N, X, FJAC, LDFJAC)
INTEGER M, N, LDFJAC
REAL X(N), FJAC(LDFJAC,N)
!
FJAC(1,1) = -20.0E0*X(1)
FJAC(2,1) = -1.0E0
```

```
FJAC(1,2) = 10.0E0
FJAC(2,2) = 0.0E0
RETURN
END
```

## Output

The solution is      0.5000    0.2500

The function evaluated at the solution is  
0.0000    0.5000

The number of iterations is                    13  
The number of function evaluations is    21

---

## BCNLS



Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.

### Required Arguments

*FCN* — User-supplied subroutine to evaluate the function to be minimized. The usage is  
`CALL FCN (M, N, X, F)`, where

*M* — Number of functions. (Input)

*N* — Number of variables. (Input)

*X* — Array of length *N* containing the point at which the function will be evaluated. (Input)

*F* — Array of length *M* containing the computed function at the point *x*. (Output)

The routine *FCN* must be declared `EXTERNAL` in the calling program.

*M* — Number of functions. (Input)

*C* — *MCON* × *N* matrix containing the coefficients of the *MCON* general linear constraints. (Input)

*BL* — Vector of length *MCON* containing the lower limit of the general constraints. (Input).

*BU* — Vector of length *MCON* containing the upper limit of the general constraints. (Input).

**IRTYPE** — Vector of length *MCON* indicating the types of general constraints in the matrix *C*. (Input)

Let  $R(I) = C(I, 1)*X(1) + \dots + C(I, N)*X(N)$ . Then the value of *IRTYPE(I)* signifies the following:

| <b>IRTYPE(I)</b> | <b>I-th CONSTRAINT</b>       |
|------------------|------------------------------|
| 0                | $BL(I) .EQ. R(I) .EQ. BU(I)$ |
| 1                | $R(I) .LE. BU(I)$            |
| 2                | $R(I) .GE. BL(I)$            |
| 3                | $BL(I) .LE. R(I) .LE. BU(I)$ |

**XLB** — Vector of length *N* containing the lower bounds on variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)

**XUB** — Vector of length *N* containing the upper bounds on variables; if there is no upper bound on a variable, then -1.0E30 should be set as the upper bound. (Input)

**X** — Vector of length *N* containing the approximate solution. (Output)

### Optional Arguments

**N** — Number of variables. (Input)  
Default:  $N = \text{SIZE}(C, 2)$ .

**MCON** — The number of general linear constraints for the system, not including simple bounds. (Input)  
Default:  $MCON = \text{SIZE}(C, 1)$ .

**LDC** — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)  
*LDC* must be at least *MCON*.  
Default:  $LDC = \text{SIZE}(C, 1)$ .

**XGUESS** — Vector of length *N* containing the initial guess. (Input)  
Default:  $XGUESS = 0.0$ .

**RNORM** — The Euclidean length of components of the function  $f(x)$  after the approximate solution has been found. (Output).

**ISTAT** — Scalar indicating further information about the approximate solution *x*. (Output)  
See the Comments section for a description of the tolerances and the vectors *IPARAM* and *RPARAM*.

| <b>ISTAT</b> | <b>Meaning</b>                                                                                                                                                  |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1            | The function $f(x)$ has a length less than $TOLF = RPARAM(1)$ . This is the expected value for <i>ISTAT</i> when an actual zero value of $f(x)$ is anticipated. |

- |   |                                                                                                                                                                                                                                                                                                 |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2 | The function $f(x)$ has reached a local minimum. This is the expected value for <code>ISTAT</code> when a nonzero value of $f(x)$ is anticipated.                                                                                                                                               |
| 3 | A small change (absolute) was noted for the vector $x$ . A full model problem step was taken. The condition for <code>ISTAT</code> = 2 may also be satisfied, so that a minimum has been found. However, this test is made before the test for <code>ISTAT</code> = 2.                          |
| 4 | A small change (relative) was noted for the vector $x$ . A full model problem step was taken. The condition for <code>ISTAT</code> = 2 may also be satisfied, so that a minimum has been found. However, this test is made before the test for <code>ISTAT</code> = 2.                          |
| 5 | The number of terms in the quadratic model is being restricted by the amount of storage allowed for that purpose. It is suggested, but not required, that additional storage be given for the quadratic model parameters. This is accessed through the vector <i>IPARAM</i> , documented below. |
| 6 | Return for evaluation of function and Jacobian if reverse communication is desired. See the Comments below.                                                                                                                                                                                     |

### **FORTRAN 90 Interface**

Generic:    CALL BCNLS (FCN, M, C, BL, BU, IRTYPE, XLB, XUB, X [, ...] )

Specific:    The specific interface names are `S_BCNLS` and `D_BCNLS`.

### **FORTRAN 77 Interface**

Single:     CALL BCNLS (FCN, M, N, MCON, C, LDC, BL, BU, IRTYPE, XLB, XUB, XGUESS, X, RNORM, ISTAT)

Double:     The double precision name is `DBCNLS`.

### **Description**

The routine `BCNLS` solves the nonlinear least squares problem

$$\min \sum_{i=1}^m f_i(x)^2$$

subject to

$$b_l \leq Cx \leq b_u$$

$$x_l \leq x \leq x_u$$

BCNLS is based on the routine `DQED` by R.J. Hanson and F.T. Krogh. The section of BCNLS that approximates, using finite differences, the Jacobian of  $f(x)$  is a modification of `JACBF` by D.E. Salane.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `B2NLS/DB2NLS`. The reference is:

```
CALL B2NLS (FCN, M, N, MCON, C, LDC, BL, BU, IRTYPE, XLB,
 XUB, XGUESS, X, RNORM, ISTAT, IPARAM, RPARAM, JAC, F,
 FJ, LDFJ, IWORK, LIWORK, WORK, LWORK)
```

The additional arguments are as follows:

**IPARAM** — Integer vector of length six used to change certain default attributes of BCNLS. (Input).

If the default parameters are desired for `B2NLS`, set `IPARAM(1)` to zero.

Otherwise, if any nondefault parameters are desired for `IPARAM` or `RPARAM`, the following steps should be taken before calling `B2NLS`:

```
CALL B7NLS (IPARAM, RPARAM)
```

Set nondefault values for `IPARAM` and `RPARAM`.

If double precision is being used, `DB7NLS` should be called instead. Following is a list of parameters and the default values.

`IPARAM(1)` = Initialization flag.

`IPARAM(2)` = `ITMAX`, the maximum number of iterations allowed.  
Default: 75

`IPARAM(3)` = a flag that suppresses the use of the quadratic model in the inner loop. If set to one, then the quadratic model is never used. Otherwise use the quadratic model where appropriate. This option decreases the amount of workspace as well as the computing overhead required. A user may wish to determine if the application really requires the use of the quadratic model.  
Default: 0

`IPARAM(4)` = `NTERMS`, one more than the maximum number of terms used in the quadratic model.  
Default: 5

`IPARAM(5)` = `RCSTAT`, a flag that determines whether forward or reverse communication is used. If set to zero, forward communication through functions `FCN` and `JAC` is used. If set to one, reverse communication is

used, and the dummy routines B10LS/DB10LS and B11LS/DB11LS may be used in place of FCN and JAC, respectively. When BCNLS returns with ISTAT = 6, arrays F and FJ are filled with  $f(x)$  and the Jacobian of  $f(x)$ , respectively. BCNLS is then called again.  
Default: 0

IPARAM(6) = a flag that determines whether the analytic Jacobian, as supplied in JAC, is used, or if a finite difference approximation is computed. If set to zero, JAC is not accessed and finite differences are used. If set to one, JAC is used to compute the Jacobian.  
Default: 0

**RPARAM** — Real vector of length 7 used to change certain default attributes of BCNLS. (Input)

For the description of RPARAM, we make the following definitions:

FC current value of the length of  $f(x)$

FB best value of length of  $f(x)$

FL value of length of  $f(x)$  at the previous step

PV predicted value of length of  $f(x)$ , after the step is taken, using the approximating model  $\epsilon$  machine epsilon = amach(4).

The conditions  $|FB - PV| \leq \text{TOLSNR} * FB$  and  $|FC - PV| \leq \text{TOLP} * FB$  and  $|FC - FL| \leq \text{TOLSNR} * FB$  together with taking a full model step, must be satisfied before the condition ISTAT = 2 is returned. (Decreasing any of the values for TOLF, TOLD, TOLX, TOLSNR, or TOLP will likely increase the number of iterations required for convergence.)

RPARAM(1) = TOLF, tolerance used for stopping when  $FC \leq \text{TOLF}$ .

Default :  $\min(1.E-5, \sqrt{\epsilon})$

RPARAM(2) = TOLX, tolerance for stopping when change to  $x$  values has length less than or equal to  $\text{TOLX} * \text{length of } x \text{ values}$ .

Default :  $\min(1.E-5, \sqrt{\epsilon})$

RPARAM(3) = TOLD, tolerance for stopping when change to  $x$  values has length less than or equal to TOLD.

Default :  $\min(1.E-5, \sqrt{\epsilon})$

RPARAM(4) = TOLSNR, tolerance used in stopping condition ISTAT = 2.

Default: 1.E-5

RPARAM(5) = TOLP, tolerance used in stopping condition ISTAT = 2.

Default: 1.E-5

RPARAM(6) = TOLUSE, tolerance used to avoid values of  $x$  in the quadratic model's interpolation of previous points. Decreasing this value may

result in more terms being included in the quadratic model.

Default:  $\sqrt{\epsilon}$

RPARAM(7) = COND, largest condition number to allow when solving for the quadratic model coefficients. Increasing this value may result in more terms being included in the quadratic model.

Default: 30

**JAC** — User-supplied subroutine to evaluate the Jacobian. The usage is

CALL JAC(M, N, X, FJAC, LDFJAC), where

M — Number of functions. (Input)

N — Number of variables. (Input)

X — Array of length N containing the point at which the Jacobian will be evaluated. (Input)

FJAC — The computed  $M \times N$  Jacobian at the point X. (Output)

LDFJAC — Leading dimension of the array FJAC. (Input)

The routine JAC must be declared EXTERNAL in the calling program.

**F** — Real vector of length N used to pass  $f(x)$  if reverse communication (IPARAM(4)) is enabled. This array must be allocated regardless of the setting of (IPARAM(4)). (Input)

**FJ** — Real array of size  $M \times N$ . It is used to store the Jacobian matrix of  $f(x)$  if reverse communication (IPARAM(4)) is enabled. This array must be allocated regardless of the setting of (IPARAM(4)). (Input)  
Specifically,

$$FJ(i, j) = \frac{\partial f_i}{\partial x_j}$$

**LDFJ** — Leading dimension of FJ exactly as specified in the dimension statement of the calling program. (Input)

**IWORK** — Integer work vector of length LIWORK.

**LIWORK** — Length of work vector IWORK. LIWORK must be at least  $5MCON + 12N + 47 + \text{MAX}(M, N)$

**WORK** — Real work vector of length LWORK

**LWORK** — Length of work vector WORK. LWORK must be at least  $41N + 6M + 11MCON + (M + MCON)(N + 1) + NA(NA + 7) + 8 \text{MAX}(M, N) + 99$ . Where  $NA = MCON + 2N + 6$ .

## 2. Informational errors

Type Code

- |   |   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 1 | The function $f(x)$ has reached a value that may be a local minimum. However, the bounds on the trust region defining the size of the step are being hit at each step. Thus, the situation is suspect. (Situations of this type can occur when the solution is at infinity at some of the components of the unknowns, $x$ ).                                                                                                                                       |
| 3 | 2 | The model problem solver has noted a value for the linear or quadratic model problem residual vector length that is greater than or equal to the current value of the function, i.e. the Euclidean length of $f(x)$ . This situation probably means that the evaluation of $f(x)$ has more uncertainty or noise than is possible to account for in the tolerances used to not a local minimum. The value of $x$ is suspect, but a minimum has probably been found. |
| 3 | 3 | More than <code>ITMAX</code> iterations were taken to obtain the solution. The value obtained for $x$ is suspect, although it is the best set of $x$ values that occurred in the entire computation. The value of <code>ITMAX</code> can be increased though the <code>IPARAM</code> vector.                                                                                                                                                                       |

### Example 1

This example finds the four variables  $x_1, x_2, x_3, x_4$  that are in the model function

$$h(t) = x_1 e^{x_2 t} + x_3 e^{x_4 t}$$

There are values of  $h(t)$  at five values of  $t$ .

$$h(0.05) = 2.206$$

$$h(0.1) = 1.994$$

$$h(0.4) = 1.35$$

$$h(0.5) = 1.216$$

$$h(1.0) = 0.7358$$

There are also the constraints that  $x_2, x_4 \leq 0$ ,  $x_1, x_3 \geq 0$ , and  $x_2$  and  $x_4$  must be separated by at least 0.05. Nothing more about the values of the parameters is known so the initial guess is 0.

```

USE BCNLS_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER MCON, N
PARAMETER (MCON=1, N=4)
!
! SPECIFICATIONS FOR PARAMETERS
INTEGER LDC, M
PARAMETER (M=5, LDC=MCON)
!
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER IRTYPE(MCON), NOUT
REAL BL(MCON), C(MCON,N), RNORM, X(N), XLB(N), &
XUB(N)
!
! SPECIFICATIONS FOR SUBROUTINES
```

```

! SPECIFICATIONS FOR FUNCTIONS
EXTERNAL FCN
!
CALL UMACH (2, NOUT)
! Define the separation between x(2)
! and x(4)
C(1,1) = 0.0
C(1,2) = 1.0
C(1,3) = 0.0
C(1,4) = -1.0
BL(1) = 0.05
IRTYPE(1) = 2
! Set lower bounds on variables
XLB(1) = 0.0
XLB(2) = 1.0E30
XLB(3) = 0.0
XLB(4) = 1.0E30
! Set upper bounds on variables
XUB(1) = -1.0E30
XUB(2) = 0.0
XUB(3) = -1.0E30
XUB(4) = 0.0
!
CALL BCNLS (FCN, M, C, BL, BL, IRTYPE, XLB, XUB, X, RNORM=RNORM)
CALL WRRRN ('X', X, 1, N, 1)
WRITE (NOUT,99999) RNORM
99999 FORMAT (/, 'rnorm = ', E10.5)
END
!
SUBROUTINE FCN (M, N, X, F)
! SPECIFICATIONS FOR ARGUMENTS
INTEGER M, N
REAL X(*), F(*)
! SPECIFICATIONS FOR LOCAL VARIABLES
INTEGER I
! SPECIFICATIONS FOR SAVE VARIABLES
REAL H(5), T(5)
SAVE H, T
! SPECIFICATIONS FOR INTRINSICS
INTRINSIC EXP
REAL EXP
!
DATA T/0.05, 0.1, 0.4, 0.5, 1.0/
DATA H/2.206, 1.994, 1.35, 1.216, 0.7358/
!
DO 10 I=1, M
 F(I) = X(1)*EXP(X(2)*T(I)) + X(3)*EXP(X(4)*T(I)) - H(I)
10 CONTINUE
RETURN
END

```

## Output

```

 X
 1 2 3 4
1.999 -1.000 0.500 -9.954
rnorm = .42425E-03

```

## Additional Examples

### Example 2

This example solves the same problem as the last example, but reverse communication is used to evaluate  $f(x)$  and the Jacobian of  $f(x)$ . The use of the quadratic model is turned off.

```
USE B2NLS_INT
USE UMACH_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER LDC, LDFJ, M, MCON, N
PARAMETER (M=5, MCON=1, N=4, LDC=MCON, LDFJ=M)
!
! Specifications for local variables
INTEGER I, IPARAM(6), IRTYPE(MCON), ISTAT, IWORK(1000), &
LIWORK, LWORK, NOUT
REAL BL(MCON), C(MCON,N), F(M), FJ(M,N), RNORM, RPARAM(7), &
WORK(1000), X(N), XGUESS(N), XLB(N), XUB(N)
REAL H(5), T(5)
SAVE H, T
INTRINSIC EXP
REAL EXP
!
! Specifications for subroutines
EXTERNAL B7NLS
!
! Specifications for functions
EXTERNAL B10LS, B11LS
!
DATA T/0.05, 0.1, 0.4, 0.5, 1.0/
DATA H/2.206, 1.994, 1.35, 1.216, 0.7358/
!
CALL UMACH (2, NOUT)
!
! Define the separation between x(2)
! and x(4)
C(1,1) = 0.0
C(1,2) = 1.0
C(1,3) = 0.0
C(1,4) = -1.0
BL(1) = 0.05
IRTYPE(1) = 2
!
! Set lower bounds on variables
XLB(1) = 0.0
XLB(2) = 1.0E30
XLB(3) = 0.0
XLB(4) = 1.0E30
!
! Set upper bounds on variables
XUB(1) = -1.0E30
XUB(2) = 0.0
XUB(3) = -1.0E30
XUB(4) = 0.0
!
! Set initial guess to 0.0
XGUESS = 0.0E0
!
! Call B7NLS to set default parameters
CALL B7NLS (IPARAM, RPARAM)
!
! Suppress the use of the quadratic
```

```

! model, evaluate functions and
! Jacobian by reverse communication
IPARAM(3) = 1
IPARAM(5) = 1
IPARAM(6) = 1
LWORK = 1000
LIWORK = 1000
!
! Specify dummy routines for FCN
! and JAC since we are using reverse
! communication
10 CONTINUE
CALL B2NLS (B10LS, M, N, MCON, C, LDC, BL, BL, IRTYPE, XLB, &
 XUB, XGUESS, X, RNORM, ISTAT, IPARAM, RPARAM, &
 B11LS, F, FJ, LDFJ, IWORK, LIWORK, WORK, LWORK)
!
! Evaluate functions if the routine
! returns with ISTAT = 6
IF (ISTAT .EQ. 6) THEN
DO 20 I=1, M
FJ(I,1) = EXP(X(2)*T(I))
FJ(I,2) = T(I)*X(1)*FJ(I,1)
FJ(I,3) = EXP(X(4)*T(I))
FJ(I,4) = T(I)*X(3)*FJ(I,3)
F(I) = X(1)*FJ(I,1) + X(3)*FJ(I,3) - H(I)
20 CONTINUE
GO TO 10
END IF
!
CALL WRRRN ('X', X, 1, N, 1)
WRITE (NOUT,99999) RNORM
99999 FORMAT (/, 'rnorm = ', E10.5)
END

```

### Output

```

 X
 1 2 3 4
1.999 -1.000 0.500 -9.954
rnorm = .42413E-03

```

---

## READ\_MPS

This subroutine reads an MPS file containing a linear programming problem or a quadratic programming problem.

### Required Arguments

**FILENAME** — Character string containing the name of the MPS file to be read. (Input)

**MPS** — A structure of IMSL defined derived type `s_MPS` containing the data read from the MPS file. (Output)

The IMSL defined derived type `s_MPS` consists of the following components:

| <b>Component</b>                                                   | <b>Description</b>                                                                                                                                                                                  |
|--------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>character</i> , allocatable :: filename                         | Name of the MPS file.                                                                                                                                                                               |
| <i>character</i> (len=8) name                                      | Name of the problem.                                                                                                                                                                                |
| <i>integer</i> nrows                                               | Number of rows in the constraint matrix.                                                                                                                                                            |
| <i>integer</i> ncolumns                                            | Number of columns in the constraint matrix. This is also the number of variables.                                                                                                                   |
| <i>integer</i> nonzeros                                            | Number of non-zeros in the constraint matrix.                                                                                                                                                       |
| <i>integer</i> nhessian                                            | Number of non-zeros in the Hessian matrix. If zero, then there is no Hessian matrix.                                                                                                                |
| <i>integer</i> ninteger                                            | Number of variables required to be integer. This includes binary variables.                                                                                                                         |
| <i>integer</i> nbinary                                             | Number of variables required to be binary (0 or 1).                                                                                                                                                 |
| <i>real</i> (kind(1e0)), allocatable :: objective (:)              | A real array of length ncolumns containing the objective vector.                                                                                                                                    |
| <i>type</i> (s_SparseMatrixElement), allocatable :: constraint (:) | A derived type array of length nonzeros and of type s_SparseMatrixElement containing the sparse matrix representation of the constraint matrix. See below for details.                              |
| <i>type</i> (s_SparseMatrixElement), allocatable :: hessian (:)    | A derived type array of length nhessian and of type s_SparseMatrixElement containing the sparse matrix representation of the Hessian matrix. If nhessian is zero, then this field is not allocated. |
| <i>real</i> (kind(1e0)), allocatable :: lower_range (:)            | A real array of length nrows containing the lower constraint bounds. If a constraint is unbounded below, the corresponding entry in lower_range is set to negative infinity, defined below.         |
| <i>real</i> (kind(1e0)), allocatable :: upper_range (:)            | A real array of length nrows containing the upper constraint bounds. If a constraint is unbounded above, the corresponding entry in upper_range is set to positive infinity, defined below.         |
| <i>real</i> (kind(1e0)), allocatable :: lower_bound (:)            | A real array of length ncolumns containing the lower variable bounds. If a variable is unbounded below, the corresponding entry in lower_bound is set                                               |

| Component                                                     | Description                                                                                                                                                                                                                        |
|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                               | to <code>negative_infinity</code> , defined below.                                                                                                                                                                                 |
| <code>real (kind(1e0)), allocatable :: upper_bound(:)</code>  | A real array of length <code>ncolumns</code> containing the upper variable bounds. If a variable is unbounded above, the corresponding entry in <code>upper_bound</code> is set to <code>positive_infinity</code> , defined below. |
| <code>integer, allocatable :: variable_type(:)</code>         | An integer array of length <code>ncolumns</code> containing the type of each variable. Variable types are:                                                                                                                         |
|                                                               | 0      Continuous                                                                                                                                                                                                                  |
|                                                               | 1      Integer                                                                                                                                                                                                                     |
|                                                               | 2      Binary (0 or 1)                                                                                                                                                                                                             |
|                                                               | 3      Semicontinuous                                                                                                                                                                                                              |
| <code>character (len=8) name_objective</code>                 | Name of the set in ROWS used for the objective row.                                                                                                                                                                                |
| <code>character (len=8) name_rhs</code>                       | Name of the RHS set used.                                                                                                                                                                                                          |
| <code>character (len=8) name_ranges</code>                    | Name of the RANGES set used or the empty string if no RANGES section in the file.                                                                                                                                                  |
| <code>character (len=8) name_bounds</code>                    | Name of the BOUNDS set used or the empty string if no BOUNDS section in the file.                                                                                                                                                  |
| <code>character (len=8), allocatable :: name_row(:)</code>    | Array of length <code>nrows</code> containing the row names. The name of the <i>i</i> -th constraint row is <code>name_row(i)</code> .                                                                                             |
| <code>character (len=8), allocatable :: name_column(:)</code> | Array of length <code>ncolumns</code> containing the column names. The name of the <i>i</i> -th column and variable is <code>name_column(i)</code> .                                                                               |
| <code>real (kind(1e0)) positive_infinity</code>               | Value used for a constraint or bound upper limit when the constraint or bound is unbounded above. This can be set using an optional argument. Default is 1.0e+30.                                                                  |
| <code>real (kind(1e0)) negative_infinity</code>               | Value used for a constraint or bound lower limit when the constraint or bound is unbounded below. This can be set using an optional argument. Default is -1.0e+30.                                                                 |

This derived type stores the constraint and Hessian matrices in a simple sparse matrix format of derived type `s_SparseMatrixElement` defined in the interface module `mp_types`. `s_SparseMatrixElement` consists of three components; a row index, a column index, and a value. For each non-zero element in the constraint and Hessian matrices an element of derived type `s_SparseMatrixElement` is stored. The following code fragment expands the sparse

constraint matrix of the derived type `s_SparseMatrixElement` contained in `mps`, a derived type of type `s_MPS`, into a dense matrix:

```
! allocate a matrix
integer nr = mps%nrows
integer nc = mps%ncolumns
real (kind(1e0)), allocatable :: matrix(:, :)
allocate(matrix(nr,nc))

matrix = 0.0e0
! expand the sparse matrix
do k = 1, mps%nonzeros
 i = mps%constraint(k)%row
 j = mps%constraint(k)%column
 matrix(i,j) = mps%constraint(k)%value
end do
```

The IMSL derived type `d_MPS` is the double precision counterpart to `s_MPS`. The IMSL derived type `d_SparseMatrixElement` is the double precision counterpart to `s_SparseMatrixElement`.

To release the space allocated for this derived type use the following statement:

```
call mps_free(mps)
```

## Optional Arguments

***NUNIT***— The unit number for reading an MPS file opened by the user. If `NUNIT` is not used, this subroutine opens the file indicated by `FILENAME` for reading and then closes it after reading. (Input)  
By default, 7 is used.

***OBJ***— Character string of length 8 containing the name of the objective function set to be used. (Input)  
An MPS file can contain multiple objective function sets.  
By default, the first objective function set in the MPS file is used. This name is case sensitive.

***RHS***— Character string of length 8 containing the name of the `RHS` set to be used. (Input)  
An MPS file can contain multiple `RHS` sets.  
By default, the first `RHS` set in the MPS file is used. This name is case sensitive.

***RANGES***— Character string of length 8 containing the name of the `RANGES` set to be used. (Input)  
An MPS file can contain multiple `RANGES` sets.  
By default, the first `RANGES` set in the MPS file is used. This name is case sensitive.

***BOUNDS***— Character string of length 8 containing the name of the `BOUNDS` set to be used. (Input)  
An MPS file can contain multiple `BOUNDS` sets.  
By default, the first `BOUNDS` set in the MPS file is used. This name is case sensitive.

**POS\_INF** — Value used for a constraint or bound upper limit when the constraint or bound is unbounded above. (Input)  
 Default: 1.0e+30.

**NEG\_INF** — Value used for a constraint or bound lower limit when the constraint or bound is unbounded below. (Input)  
 Default: -1.0e+30.

## FORTRAN 90 Interface

Generic: `CALL READ_MPS (FILENAME, MPS [, ...])`

Specific: The specific interface names are `S_READ_MPS` and `D_READ_MPS`.

## Description

An MPS file defines a linear or quadratic programming problem.

A linear programming problem is assumed to have the form:

$$\begin{aligned} \min_x \quad & c^T x \\ & b_l \leq Ax \leq b_u \\ & x_l \leq x \leq x_u \end{aligned}$$

A quadratic programming problem is assumed to have the form:

$$\begin{aligned} \min_x \quad & \frac{1}{2} x^T Qx + c^T x \\ & b_l \leq Ax \leq b_u \\ & x_l \leq x \leq x_u \end{aligned}$$

The following table maps this notation into the components in the derived type returned by `READ_MPS`:

|       |             |
|-------|-------------|
| $C$   | Objective   |
| $A$   | Constraint  |
| $Q$   | Hessian     |
| $b_l$ | lower_range |
| $b_u$ | upper_range |
| $x_l$ | lower_bound |
| $x_u$ | upper_bound |

If the MPS file specifies an equality constraint or bound, the corresponding lower and upper values in the returned derived type will be exactly equal.

The problem formulation assumes that the constraints and bounds are two-sided. If a particular constraint or bound has no lower limit, then the corresponding component of the derived type is set to  $-1.0e+30$ . If the upper limit is missing, then the corresponding component of the derived type is set to  $+1.0e+30$ .

## MPS File Format

There is some variability in the MPS format. This section describes the MPS format accepted by this reader.

An MPS file consists of a number of sections. Each section begins with a name in column 1. With the exception of the NAME section, the rest of this line is ignored. Lines with a '\*' or '\$' in column 1 are considered comment lines and are ignored.

The body of each section consists of lines divided into fields, as follows:

| Field Number | Columns | Contents  |
|--------------|---------|-----------|
| 1            | 2-3     | Indicator |
| 2            | 5-12    | Name      |
| 3            | 15-22   | Name      |
| 4            | 25-36   | Value     |
| 5            | 40-47   | Name      |
| 6            | 50-61   | Value     |

The format limits MPS names to 8 characters and values to 12 characters. The names in fields 2, 3 and 5 are case sensitive. Leading and trailing blanks are ignored, but internal spaces are significant.

The sections in an MPS file are as follows.

- NAME
- ROWS
- COLUMNS
- RHS
- RANGES (optional)
- BOUNDS (optional)
- QUADRATIC (optional)
- ENDDATA

Sections must occur in the above order.

MPS keywords, section names and indicator values, are case insensitive. Row, column and set names are case sensitive.

## NAME Section

The NAME section contains a single line. A problem name can occur anywhere on the line after NAME and before column 62. The problem name is truncated to 8 characters.

## ROWS Section

The ROWS section defines the name and type for each row. Field 1 contains the row type and field 2 contains the row name. Row type values are not case sensitive. Row names are case sensitive. The following row types are allowed:

| Row Type | Meaning                           |
|----------|-----------------------------------|
| E        | Equality Constraint.              |
| L        | Less than or equal constraint     |
| G        | Greater than or equal constraint. |
| N        | Objective or a free row.          |

## COLUMNS Section

The COLUMNS section defines the nonzero entries in the objective and the constraint matrix. The row names here must have been defined in the ROWS section.

| Field | Contents                                                              |
|-------|-----------------------------------------------------------------------|
| 2     | Column name.                                                          |
| 3     | Row name.                                                             |
| 4     | Value for the entry whose row and column are given by fields 3 and 2. |
| 5     | Row name.                                                             |
| 6     | Value for the entry whose row and column are given by fields 5 and 2. |

**NOTE:** Fields 5 and 6 are optional.

The COLUMNS section can also contain markers. These are indicated by the name 'MARKER' (with the quotes) in field 3 and the marker type in field 4 or 5.

Marker type 'INTORG' (with the quotes) begins an integer group. The marker type 'INTEND' (with the quotes) ends this group. The variables corresponding to the columns defined within this group are required to be integer.

## RHS Section

The RHS section defines the right-hand side of the constraints. An MPS file can contain more than one RHS set, distinguished by the RHS set name. The row names here must be defined in the ROWS section.

| Field | Contents                                                           |
|-------|--------------------------------------------------------------------|
| 2     | RHS set name.                                                      |
| 3     | Row name.                                                          |
| 4     | Value for the entry whose set and row are given by fields 2 and 3. |
| 5     | Row name.                                                          |
| 6     | Value for the entry whose set and row are given by fields 2 and 5. |

**NOTE:** Fields 5 and 6 are optional.

## RANGES Section

The optional RANGES section defines two-sided constraints. An MPS file can contain more than one range set, distinguished by the range set name. The row names here must have been defined in the ROWS section.

| Field | Contents                                                           |
|-------|--------------------------------------------------------------------|
| 2     | Range set name.                                                    |
| 3     | Row name.                                                          |
| 4     | Value for the entry whose set and row are given by fields 2 and 3. |
| 5     | Row name.                                                          |
| 6     | Value for the entry whose set and row are given by fields 2 and 5. |

**NOTE:** Fields 5 and 6 are optional.

Ranges change one-sided constraints, defined in the RHS section, into two-sided constraints. The two-sided constraint for row  $i$  depends on the range value,  $r_i$ , defined in this section. The right-hand side value,  $b_i$ , is defined in the RHS section. The two-sided constraints for row  $i$  are given in the following table:

| Row Type | Lower Constraint     | Upper Constraint     |
|----------|----------------------|----------------------|
| G        | $b_i$                | $b_i +  r_i $        |
| L        | $b_i -  r_i $        | $b_i$                |
| E        | $b_i + \min(0, r_i)$ | $b_i + \max(0, r_i)$ |

## BOUNDS Section

The optional BOUNDS section defines bounds on the variables. By default, the bounds are  $0 \leq x_i \leq \infty$ . The bounds can also be used to indicate that a variable must be an integer.

More than one bound can be set for a single variable. For example, to set  $2 \leq x_i \leq 6$  use a LO bound with value 2 to set  $2 \leq x_i$  and a UP bound with value 6 to add the condition  $x_i \leq 6$ .

An MPS file can contain more than one bounds set, distinguished by the bound set name.

| Field | Contents                                                              |
|-------|-----------------------------------------------------------------------|
| 1     | Bounds type.                                                          |
| 2     | Bounds set name.                                                      |
| 3     | Column name                                                           |
| 4     | Value for the entry whose set and column are given by fields 2 and 3. |
| 5     | Column name.                                                          |
| 6     | Value for the entry whose set and column are given by fields 2 and 5. |

**NOTE:** Fields 5 and 6 are optional.

The bound types are as follows. Here  $b_i$  are the bound values defined in this section, the  $x_i$  are the variables, and  $I$  is the set of integers.

| Bounded Type | Definition                                 | Formula                        |
|--------------|--------------------------------------------|--------------------------------|
| LO           | Lower bound                                | $b_j \leq x_i$                 |
| UP           | Upper bound                                | $x_i \leq b_i$                 |
| FX           | Fixed variable                             | $x_i = b_i$                    |
| FR           | Free variable                              | $-\infty \leq x_i \leq \infty$ |
| MI           | Lower bound is minus infinity              | $-\infty \leq x_i$             |
| PL           | Upper bound is positive infinity           | $x_i \leq \infty$              |
| BV           | Binary variable (variable must be 0 or 1). | $x_i \in \{0,1\}$              |

| Bounded Type | Definition              | Formula                           |
|--------------|-------------------------|-----------------------------------|
| UI           | Upper bound and integer | $x_i \leq b_i$ and<br>$x_i \in I$ |
| LI           | Lower bound and integer | $b_i \leq x_i$ and<br>$x_i \in I$ |
| SC           | Semicontinuous          | 0 or $b_i \leq x_i$               |

The bound type names are not case sensitive.

If the bound type is UP or UI and  $b_j < 0$  then the lower bound is set to  $-\infty$ .

## QUADRATIC Section

The optional QUADRATIC section defines the Hessian for quadratic programming problems. The names HESSIAN, QUADS, QUADOBJ, QSECTION, and QMATRIX are also recognized as beginning the QUADRATIC section.

| Field | Contents                                         |
|-------|--------------------------------------------------|
| 2     | Column name.                                     |
| 3     | Column name.                                     |
| 4     | Value for the entry specified by fields 2 and 3. |
| 5     | Column name.                                     |
| 6     | Value for the entry specified by fields 2 and 5. |

**NOTE:** Fields 5 and 6 are optional.

## ENDATA Section

The ENDATA section ends the MPS file.

## Comments

Informational errors

| Type | Code |                                  |
|------|------|----------------------------------|
| 3    | 5    | No objective coefficients found. |
| 3    | 6    | No RHS values found.             |
| 3    | 8    | No range values found.           |

| Type | Code |                            |
|------|------|----------------------------|
| 3    | 9    | No bounds found.           |
| 4    | 3    | Missing section title.     |
| 4    | 4    | Error reading input file.  |
| 4    | 7    | Invalid number.            |
| 4    | 11   | Unexpected section header. |
| 4    | 12   | Unknown row type.          |
| 4    | 13   | Out-of-order marker.       |
| 4    | 14   | Unknown marker type.       |
| 4    | 15   | Unknown column name.       |
| 4    | 16   | Unknown bound type.        |
| 4    | 17   | Unknown row name.          |
| 4    | 18   | Unexpected section name.   |

### Example 1

```

use read_mps_int
implicit none

TYPE(S_MPS) mps
CALL read_mps ('test.mps', mps)
End

```

### Additional Examples

#### Example 2

See [Example 2](#) of `DENSE_LP`.

---

## MPS\_FREE

Deallocates the space allocated for the IMSL derived type `s_MPS`. This routine is usually used in conjunction with [READ\\_MPS](#).

### Required Arguments

**MPS** — A structure of IMSL defined derived type `s_MPS` containing the data read from the MPS file. (Input/Output)  
The allocated components of `s_MPS` will be deallocated on output.

The IMSL defined derived type `s_MPS` consists of the following components:

| Component                                  | Description           |
|--------------------------------------------|-----------------------|
| <i>character</i> , allocatable :: filename | Name of the MPS file. |

| <b>Component</b>                                                   | <b>Description</b>                                                                                                                                                                                  |
|--------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>character</i> (len=8) name                                      | Name of the problem.                                                                                                                                                                                |
| <i>integer</i> nrows                                               | Number of rows in the constraint matrix.                                                                                                                                                            |
| <i>integer</i> ncolumns                                            | Number of columns in the constraint matrix. This is also the number of variables.                                                                                                                   |
| <i>integer</i> nonzeros                                            | Number of non-zeros in the constraint matrix.                                                                                                                                                       |
| <i>integer</i> nhessian                                            | Number of non-zeros in the Hessian matrix. If zero, then there is no Hessian matrix.                                                                                                                |
| <i>integer</i> ninteger                                            | Number of variables required to be integer. This includes binary variables.                                                                                                                         |
| <i>integer</i> nbinary                                             | Number of variables required to be binary (0 or 1).                                                                                                                                                 |
| <i>real</i> (kind(1e0)), allocatable :: objective (:)              | A real array of length ncolumns containing the objective vector.                                                                                                                                    |
| <i>type</i> (s_SparseMatrixElement), allocatable :: constraint (:) | A derived type array of length nonzeros and of type s_SparseMatrixElement containing the sparse matrix representation of the constraint matrix. See below for details.                              |
| <i>type</i> (s_SparseMatrixElement), allocatable :: hessian (:)    | A derived type array of length nhessian and of type s_SparseMatrixElement containing the sparse matrix representation of the Hessian matrix. If nhessian is zero, then this field is not allocated. |
| <i>real</i> (kind(1e0)), allocatable :: lower_range (:)            | A real array of length nrows containing the lower constraint bounds. If a constraint is unbounded below, the corresponding entry in lower_range is set to negative infinity, defined below.         |
| <i>real</i> (kind(1e0)), allocatable :: upper_range (:)            | A real array of length nrows containing the upper constraint bounds. If a constraint is unbounded above, the corresponding entry in upper_range is set to positive infinity, defined below.         |
| <i>real</i> (kind(1e0)), allocatable :: lower_bound (:)            | A real array of length ncolumns containing the lower variable bounds. If a variable is unbounded below, the corresponding entry in lower_bound is set to negative infinity, defined below.          |

| Component                                               | Description                                                                                                                                                                                                                        |                 |
|---------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| <i>real</i> (kind(1e0)), allocatable :: upper_bound(:)  | A real array of length <code>ncolumns</code> containing the upper variable bounds. If a variable is unbounded above, the corresponding entry in <code>upper_bound</code> is set to <code>positive_infinity</code> , defined below. |                 |
| <i>integer</i> , allocatable :: variable_type(:)        | An integer array of length <code>ncolumns</code> containing the type of each variable. Variable types are:                                                                                                                         |                 |
|                                                         | 0                                                                                                                                                                                                                                  | Continuous      |
|                                                         | 1                                                                                                                                                                                                                                  | Integer         |
|                                                         | 2                                                                                                                                                                                                                                  | Binary (0 or 1) |
|                                                         | 3                                                                                                                                                                                                                                  | Semicontinuous  |
| <i>character</i> (len=8) name_objective                 | Name of the set in ROWS used for the objective row.                                                                                                                                                                                |                 |
| <i>character</i> (len=8) name_rhs                       | Name of the RHS set used.                                                                                                                                                                                                          |                 |
| <i>character</i> (len=8) name_ranges                    | Name of the RANGES set used or the empty string if no RANGES section in the file.                                                                                                                                                  |                 |
| <i>character</i> (len=8) name_bounds                    | Name of the BOUNDS set used or the empty string if no BOUNDS section in the file.                                                                                                                                                  |                 |
| <i>character</i> (len=8), allocatable :: name_row(:)    | Array of length <code>nrows</code> containing the row names. The name of the <i>i</i> -th constraint row is <code>name_row(i)</code> .                                                                                             |                 |
| <i>character</i> (len=8), allocatable :: name_column(:) | Array of length <code>ncolumns</code> containing the column names. The name of the <i>i</i> -th column and variable is <code>name_column(i)</code> .                                                                               |                 |
| <i>real</i> (kind(1e0)) positive_infinity               | Value used for a constraint or bound upper limit when the constraint or bound is unbounded above. This can be set using an optional argument. Default is 1.0e+30.                                                                  |                 |
| <i>real</i> (kind(1e0)) negative_infinity               | Value used for a constraint or bound lower limit when the constraint or bound is unbounded below. This can be set using an optional argument. Default is -1.0e+30.                                                                 |                 |

This derived type stores the constraint and Hessian matrices in a simple sparse matrix format of derived type `s_SparseMatrixElement` defined in the interface module `mp_types`. `s_SparseMatrixElement` consists of three components; a row index, a column index, and a value. For each non-zero element in the constraint and Hessian matrices an element of derived type `s_SparseMatrixElement` is stored. The following code fragment expands the sparse constraint matrix of the derived type `s_SparseMatrixElement` contained in `mps`, a derived type of type `s_MPS`, into a dense matrix:

```

! allocate a matrix
integer nr = mps%nrows
integer nc = mps%ncolumns
real (kind(1e0)), allocatable :: matrix(:, :)
allocate(matrix(nr,nc))

matrix = 0.0e0
! expand the sparse matrix
do k = 1, mps%nonzeros
 i = mps%constraint(k)%row
 j = mps%constraint(k)%column
 matrix(i,j) = mps%constraint(k)%value
end do

```

The IMSL derived type `d_MPS` is the double precision counterpart to `s_MPS`. The IMSL derived type `d_SparseMatrixElement` is the double precision counterpart to `s_SparseMatrixElement`.

### **FORTRAN 90 Interface**

Generic:     CALL `MPS_FREE` (`MPS`)

Specific:    The specific interface names are `S_MPS_FREE` and `D_MPS_FREE`.

### **Description**

This subroutine simply issues deallocate statements for each of the arrays allocated in the IMSL derived type `s_MPS` defined above. It is supplied as a convenience utility to the user of `READ_MPS`.

### **Example**

In the following example, the space that had been allocated to accommodate the IMSL derived type `S_MPS` is deallocated with a call to `MPS_FREE` after a call to `READ_MPS` was made.

```

use read_mps_int
use mps_free_int
implicit none

TYPE(S_MPS) mps
CALL read_mps ('test.mps', mps)
.
.
.
call mps_free (mps)
end

```

---

## **DENSE\_LP**

Solves a linear programming problem.

---

**NOTE:** `DENSE_LP` is available in double precision only.

---

## Required Arguments

**A** —  $M$  by  $NVAR$  matrix containing the coefficients of the  $M$  constraints. (Input)

**BL** — Vector of length  $M$  containing the lower limit of the general constraints; if there is no lower limit on the  $I$ -th constraint, then  $BL(I)$  is not referenced. (Input)

**BU** — Vector of length  $M$  containing the upper limit of the general constraints; if there is no upper limit on the  $I$ -th constraint, then  $BU(I)$  is not referenced; if there are no range constraints,  $BL$  and  $BU$  can share the same storage locations. (Input)

**C** — Vector of length  $NVAR$  containing the coefficients of the objective function. (Input)

**IRTYPE** — Vector of length  $M$  indicating the types of general constraints in the matrix  $A$ . (Input)

Let  $R(I) = A(I, 1) * XSOL(1) + \dots + A(I, NVAR) * XSOL(NVAR)$ . Then, the value of  $IRTYPE(I)$  signifies the following:

| <b>Irtype [I]</b> | <b>I-th Constraint</b>       |
|-------------------|------------------------------|
| 0                 | $BL(I) = R(I) = BU(I)$       |
| 1                 | $R(I) \leq BU(I)$            |
| 2                 | $R(I) \geq BL(I)$            |
| 3                 | $BL(I) \leq R(I) \leq BU(I)$ |
| 4                 | Ignore this constraint       |

**OBJ** — Value of the objective function. (Output)

**XSOL** — Vector of length  $NVAR$  containing the primal solution. (Output)

**DSOL** — Vector of length  $M$  containing the dual solution. (Output)

## Optional Arguments

**M** — Number of constraints. (Input)  
Default:  $M = \text{SIZE}(A, 1)$ .

**NVAR** — Number of variables. (Input)  
Default:  $NVAR = \text{SIZE}(A, 2)$ .

**LDA** — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
 $LDA$  must be at least  $M$ .  
Default:  $LDA = \text{SIZE}(A, 1)$ .

**XLB** — Vector of length  $NVAR$  containing the lower bound on the variables; if there is no lower bound on a variable, then 1.0D30 should be set as the lower bound. (Input)  
Default:  $XLB = 0.0D0$ .

**XUB** — Vector of length `NVAR` containing the upper bound on the variables; if there is no upper bound on a variable, then `-1.0D30` should be set as the upper bound. (Input)  
 Default: No upperbound enforced.

**ITREF** — The type if iterative refinement used. (Input)

| <b>ITREF</b> | <b>Refinement</b>                                        |
|--------------|----------------------------------------------------------|
| 0            | No refinement                                            |
| 1            | Iterative refinement                                     |
| 2            | Use extended refinement. Iterate until no more progress. |

Default: `ITREF = 0`.

**ITERS** — Number of iterations. (Output)

**IERR** — Status flag indicating which warning conditions were set upon completion. (Output)

| <b>IERR</b> | <b>Status</b>                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\geq 0$    | Solution found. <b>IERR</b> = 0 indicates there are no warning conditions. If the solution was found with warning conditions <b>IERR</b> is incremented by the number given below.                                                                                                               |
| 1           | 1 is added to the value returned if there are multiple solutions giving essentially the same minimum.                                                                                                                                                                                            |
| 2           | 2 is added to the value returned if there were some constraints discarded because they were too linearly dependent on other active constraints.                                                                                                                                                  |
| 4           | 4 is added to the value returned if the constraints were not satisfied. $L_1$ minimization was applied to all (including bounds on simple variables) but the equalities, to approximate violated non-equalities as well as possible. If a feasible solution is possible then refinement may help |
| 8           | 8 is added to the value returned if the algorithm appears to be cycling. Using refinement may help.                                                                                                                                                                                              |

### **FORTRAN 90 Interface**

Generic: `CALL DENSE_LP (A, BL, BU, C, IRTYPE, OBJ, XSOL, DSOL [, ...])`

Specific: The specific interface name is `D_DENSE_LP`. This subroutine is available in double precision only.

## Description

The routine `DENSE_LP` solves the linear programming problem

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} c^T x \\ & \text{subject to } b_l \leq A_x \leq b_u \\ & \quad x_l \leq x \leq x_u \end{aligned}$$

where  $c$  is the objective coefficient vector,  $A$  is the coefficient matrix, and the vectors  $b_l$ ,  $b_u$ ,  $x_l$  and  $x_u$  are the lower and upper bounds on the constraints and the variables, respectively.

`DENSE_LP` uses an active set strategy.

Refer to the following paper for further information: Krogh, Fred, T. (2005), *An Algorithm for Linear Programming*, <http://mathalacarte.com/fkrogh/pub/lp.pdf>, Tujunga, CA.

## Comments

1. Informational errors

| Type | Code |                                                                                                       |
|------|------|-------------------------------------------------------------------------------------------------------|
| 1    | 1    | Multiple solutions giving essentially the same solution exist.                                        |
| 3    | 1    | Some constraints were discarded because they were too linearly dependent on other active constraints. |
| 3    | 2    | All constraints are not satisfied.                                                                    |
| 3    | 3    | The algorithm appears to be cycling.                                                                  |
| 4    | 1    | The problem appears vacuous.                                                                          |
| 4    | 2    | The problem is unbounded.                                                                             |
| 4    | 3    | An acceptable pivot could not be found.                                                               |
| 4    | 4    | The constraint bounds are inconsistent.                                                               |
| 4    | 5    | The variable bounds are inconsistent.                                                                 |

## Example 1

The linear programming problem in the standard form

$$\begin{aligned} & \min f(x) = -x_1 - 3x_2 \\ & \text{subject to } \begin{array}{rcl} x_1 + x_2 + x_3 & & = 1.5 \\ x_1 + x_2 & -x_4 & = 0.5 \\ x_1 & & +x_5 = 1.0 \\ & x_2 & +x_6 = 1.0 \\ & & x_i \geq 0, \text{ for } i = 1, \dots, 6 \end{array} \end{aligned}$$

is solved.

```
USE UMACH_INT
USE WRRRN_INT
USE DENSE_LP_INT
IMPLICIT NONE
INTEGER NOUT, M, NVAR
PARAMETER (M=4, NVAR=6)
DOUBLE PRECISION A(M, NVAR), B(M), C(NVAR), XSOL(NVAR), &
 DSOL(M), BL(M), BU(M), OBJ
INTEGER IRTYPE(M)
DATA A/1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, -1, &
 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1/
DATA B/1.5, 0.5, 1.0, 1.0/
DATA C/-1.0, -3.0, 0.0, 0.0, 0.0, 0.0/
DATA BL/1.5, 0.5, 1.0, 1.0/
DATA BU/M*-1.D30/
DATA IRTYPE/M*0/

CALL UMACH(2, NOUT)
! Solve the LP problem
CALL DENSE_LP (A, BL, BU, C, IRTYPE, OBJ, XSOL, DSOL)

WRITE(NOUT, 99999) OBJ
CALL WRRRN('Solution', XSOL, 1, NVAR, 1)
99999 FORMAT (' Objective', F9.4)
END
```

## Output

Objective -3.5000

|  |       | Solution |       |       |       |       |  |
|--|-------|----------|-------|-------|-------|-------|--|
|  | 1     | 2        | 3     | 4     | 5     | 6     |  |
|  | 0.500 | 1.000    | 0.000 | 1.000 | 0.500 | 0.000 |  |

## Additional Examples

### Example 2

This example demonstrates how `READ_MPS` can be used together with `DENSE_LP` to solve a linear programming problem defined in an MPS file. The MPS file used in this example is an *uncompressed* version of the file 'afiro', available from <http://www.netlib.org/lp/data/>.

```
USE UMACH_INT
USE WRRRN_INT
USE READ_MPS_INT
USE DENSE_LP_INT
IMPLICIT NONE
REAL(KIND(1D0)) OBJ
REAL(KIND(1D0)), ALLOCATABLE :: XSOL(:)
REAL(KIND(1D0)), ALLOCATABLE :: DSOL(:)
REAL(KIND(1D0)), ALLOCATABLE :: A(:, :)
INTEGER, ALLOCATABLE :: IRTYPE(:)
```

```

TYPE(D_MPS) PROBLEM
CHARACTER NAME*256
INTEGER I, J, K, NOUT

CALL UMACH(2, NOUT)

! READ LP PROBLEM FROM THE MPS FILE.
NAME = 'afiro'
CALL READ_MPS (NAME, PROBLEM)
ALLOCATE (A(PROBLEM%NROWS, PROBLEM%NCOLUMNS))
ALLOCATE (IRTYPE(PROBLEM%NROWS))
ALLOCATE (XSOL(PROBLEM%NCOLUMNS))
ALLOCATE (DSOL(PROBLEM%NROWS))
A = 0
IRTYPE = 3
! FILL DENSE A
DO K = 1, PROBLEM%NONZEROS
 I = PROBLEM%CONSTRAINT(K)%ROW
 J = PROBLEM%CONSTRAINT(K)%COLUMN
 A(I, J) = PROBLEM%CONSTRAINT(K)%VALUE
ENDDO
! CALL THE LP SOLVER
CALL DENSE_LP (A, PROBLEM%LOWER_RANGE, PROBLEM%UPPER_RANGE, &
 PROBLEM%OBJECTIVE, IRTYPE, OBJ, XSOL, DSOL, &
 XLB=PROBLEM%LOWER_BOUND, XUB=PROBLEM%UPPER_BOUND)
WRITE(NOUT, 99999) OBJ
CALL WRRRN('Solution', XSOL, 1, PROBLEM%NROWS, 1)

DEALLOCATE(A)
DEALLOCATE(IRTYPE)
DEALLOCATE(XSOL)
DEALLOCATE(DSOL)
99999 FORMAT('Objective: ', E16.7)
END

```

## Output

Objective: -0.4647531E+03

|       |      |      |      |      |     |     |     |     |     | Solution |     |      |      |      |       |       |      |     |       |
|-------|------|------|------|------|-----|-----|-----|-----|-----|----------|-----|------|------|------|-------|-------|------|-----|-------|
| 1     | 2    | 3    | 4    | 5    | 6   | 7   | 8   | 9   | 10  | 11       | 12  | 13   | 14   | 15   | 16    | 17    | 18   | 19  | 20    |
| 80.0  | 25.5 | 54.5 | 84.8 | 57.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0      | 0.0 | 18.2 | 39.7 | 61.3 | 500.0 | 475.9 | 24.1 | 0.0 | 215.0 |
| 21    | 22   | 23   | 24   | 25   | 26  | 27  |     |     |     |          |     |      |      |      |       |       |      |     |       |
| 363.9 | 0.0  | 0.0  | 0.0  | 0.0  | 0.0 | 0.0 |     |     |     |          |     |      |      |      |       |       |      |     |       |

---

# DLPRS



Solves a linear programming problem via the revised simplex algorithm.

## Required Arguments

**A** —  $M$  by  $NVAR$  matrix containing the coefficients of the  $M$  constraints. (Input)

**BL** — Vector of length  $M$  containing the lower limit of the general constraints; if there is no lower limit on the  $I$ -th constraint, then  $BL(I)$  is not referenced. (Input)

**BU** — Vector of length  $M$  containing the upper limit of the general constraints; if there is no upper limit on the  $I$ -th constraint, then  $BU(I)$  is not referenced; if there are no range constraints,  $BL$  and  $BU$  can share the same storage locations. (Input)

**C** — Vector of length  $NVAR$  containing the coefficients of the objective function. (Input)

**IRTYPE** — Vector of length  $M$  indicating the types of general constraints in the matrix  $A$ . (Input)

Let  $R(I) = A(I, 1) * XSOL(1) + \dots + A(I, NVAR) * XSOL(NVAR)$ . Then, the value of  $IRTYPE(I)$  signifies the following:

| <b>IRTYPE(I)</b> | <b>I-th Constraint</b>       |
|------------------|------------------------------|
| 0                | $BL(I) .EQ. R(I) .EQ. BU(I)$ |
| 1                | $R(I) .LE. BU(I)$            |
| 2                | $R(I) .GE. BL(I)$            |
| 3                | $BL(I) .LE. R(I) .LE. BU(I)$ |

**OBJ** — Value of the objective function. (Output)

**XSOL** — Vector of length  $NVAR$  containing the primal solution. (Output)

**DSOL** — Vector of length  $M$  containing the dual solution. (Output)

## Optional Arguments

**M** — Number of constraints. (Input)  
Default:  $M = \text{SIZE}(A,1)$ .

**NVAR** — Number of variables. (Input)

Default: `NVAR = SIZE (A,2)`.

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)

LDA must be at least **M**.

Default: `LDA = SIZE (A,1)`.

**XLB** — Vector of length **NVAR** containing the lower bound on the variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)

Default: `XLB = 0.0`.

**XUB** — Vector of length **NVAR** containing the upper bound on the variables; if there is no upper bound on a variable, then -1.0E30 should be set as the upper bound. (Input)

Default: `XUB = 3.4e38` for single precision and `1.79d + 308` for double precision.

## FORTRAN 90 Interface

Generic: `CALL DLPRS (A, BL, BU, C, IRTYPE, OBJ, XSOL, DSOL [, ...])`

Specific: The specific interface names are `S_DLPRS` and `D_DLPRS`.

## FORTRAN 77 Interface

Single: `CALL DLPRS (M, NVAR, A, LDA, BL, BU, C, IRTYPE, XLB, XUB, OBJ, XSOL, DSOL)`

Double: The double precision name is `DDLPRS`.

## Description

The routine `DLPRS` uses a revised simplex method to solve linear programming problems, i.e., problems of the form

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} c^T x \\ & \text{subject to } b_l \leq Ax \leq b_u \\ & \quad x_l \leq x \leq x_u \end{aligned}$$

where  $c$  is the objective coefficient vector,  $A$  is the coefficient matrix, and the vectors  $b_l$ ,  $b_u$ ,  $x_l$  and  $x_u$  are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `D2PRS/DD2PRS`. The reference is:

```
CALL D2PRS (M, NVAR, A, LDA, BL, BU, C, IRTYPE, XLB, XUB,
 OBJ, XSOL, DSOL, AWK, LDAWK, WK, IWK)
```

The additional arguments are as follows:

**AWK** — Real work array of dimension 1 by 1. (AWK is not used in the new implementation of the revised simplex algorithm. It is retained merely for calling sequence consistency.)

**LDAWK** — Leading dimension of AWK exactly as specified in the dimension statement of the calling program. LDAWK should be 1. (LDAWK is not used in the new implementation of the revised simplex algorithm. It is retained merely for calling sequence consistency.)

**WK** — Real work vector of length  $M * (M + 28)$ .

**IWK** — Integer work vector of length  $29 * M + 3 * NVAR$ .

## 2. Informational errors

| Type | Code |                                                                                |
|------|------|--------------------------------------------------------------------------------|
| 3    | 1    | The problem is unbounded.                                                      |
| 4    | 2    | Maximum number of iterations exceeded.                                         |
| 3    | 3    | The problem is infeasible.                                                     |
| 4    | 4    | Moved to a vertex that is poorly conditioned; using double precision may help. |
| 4    | 5    | The bounds are inconsistent.                                                   |

## Example

A linear programming problem is solved.

```
USE DLPRS_INT
USE UMACH_INT
USE SSCAL_INT

IMPLICIT NONE
INTEGER LDA, M, NVAR
PARAMETER (M=2, NVAR=2, LDA=M)
!
! M = number of constraints
! NVAR = number of variables
!
INTEGER I, IRTYPE(M), NOUT
REAL A(LDA, NVAR), B(M), C(NVAR), DSOL(M), OBJ, XLB(NVAR), &
 XSOL(NVAR), XUB(NVAR)
!
! Set values for the following problem
!
! Max 1.0*XSOL(1) + 3.0*XSOL(2)
```



**BL** — Vector of length *M* containing the lower limit of the general constraints; if there is no lower limit on the *I*-th constraint, then *BL(I)* is not referenced. (Input)

**BU** — Vector of length *M* containing the upper lower limit of the general constraints; if there is no upper limit on the *I*-th constraint, then *BU(I)* is not referenced. (Input)

**C** — Vector of length *NVAR* containing the coefficients of the objective function. (Input)

**IRTYPE** — Vector of length *M* indicating the types of general constraints in the matrix *A*. (Input)

Let  $R(I) = A(I, 1)*XSOL(1) + \dots + A(I, NVAR)*XSOL(NVAR)$

| <b>IRTYPE(I)</b> | <b>I-th CONSTRAINT</b>       |
|------------------|------------------------------|
| 0                | $BL(I) = R(I) = BU(I)$       |
| 1                | $R(I) \leq BU(I)$            |
| 2                | $R(I) \geq BL(I)$            |
| 3                | $BL(I) \leq R(I) \leq BU(I)$ |

**OBJ** — Value of the objective function. (Output)

**XSOL** — Vector of length *NVAR* containing the primal solution. (Output)

**DSOL** — Vector of length *M* containing the dual solution. (Output)

### Optional Arguments

**M** — Number of constraints. (Input)  
Default:  $M = \text{SIZE}(\text{IRTYPE}, 1)$ .

**NVAR** — Number of variables. (Input)  
Default:  $NVAR = \text{SIZE}(C, 1)$ .

**NZ** — Number of nonzero coefficients in the matrix *A*. (Input)  
Default:  $NZ = \text{SIZE}(A, 1)$ .

**XLB** — Vector of length *NVAR* containing the lower bound on the variables; if there is no lower bound on a variable, then 1.0E30 should be set as the lower bound. (Input)  
Default:  $XLB = 0.0$ .

**XUB** — Vector of length *NVAR* containing the upper bound on the variables; if there is no upper bound on a variable, then -1.0E30 should be set as the upper bound. (Input)  
Default:  $XLB = 3.4e38$  for single precision and  $1.79d + 308$  for double precision.

### FORTRAN 90 Interface

Generic:     CALL SLPRS (A, IROW, JCOL, BL, BU, C, IRTYPE, OBJ, XSOL, DSOL [, ...])

Specific: The specific interface names are S\_SLPRS and D\_SLPRS.

## FORTRAN 77 Interface

Single: CALL SLPRS (M, NVAR, NZ, A, IROW, JCOL, BL, BU, C, IRTYPE, XLB, XUB, OBJ, XSOL, DSOL)

Double: The double precision name is DSLPRS.

## Description

This subroutine solves problems of the form

$$\min c^T x$$

subject to

$$b_l \leq Ax \leq b_u,$$

$$x_l \leq x \leq x_u$$

where  $c$  is the objective coefficient vector,  $A$  is the coefficient matrix, and the vectors  $b_l$ ,  $b_u$ ,  $x_l$ , and  $x_u$  are the lower and upper bounds on the constraints and the variables, respectively. SLPRS is designed to take advantage of sparsity in  $A$ . The routine is based on DPLO by Hanson and Hiebert.

## Comments

Workspace may be explicitly provided, if desired, by use of S2PRS/DS2PRS. The reference is:

```
CALL S2PRS (M, NVAR, NZ, A, IROW, JCOL, BL, BU, C, IRTYPE,
 XLB, XUB, OBJ, XSOL, DSOL, IPARAM, RPARAM, COLSCL,
 ROWSCL, WORK, LW, IWORK, LIW)
```

The additional arguments are as follows:

**IPARAM** — Integer parameter vector of length 12. If the default parameters are desired for SLPRS, then set IPARAM(1) to zero and call the routine SLPRS. Otherwise, if any nondefault parameters are desired for IPARAM or RPARAM, then the following steps should be taken before calling SLPRS:

```
CALL S5PRS (IPARAM, RPARAM)
```

Set nondefault values for IPARAM and RPARAM.

---

Note that the call to S5PRS will set IPARAM and RPARAM to their default values so only nondefault values need to be set above.

---

IPARAM(1) = 0 indicates that a minimization problem is solved. If set to 1, a maximization problem is solved.

Default: 0

IPARAM(2) = switch indicating the maximum number of iterations to be taken before returning to the user. If set to zero, the maximum number of iterations taken is set to  $3*(NVAR+M)$ . If positive, that value is used as the iteration limit.  
Default: IPARAM(2) = 0

IPARAM(3) = indicator for choosing how columns are selected to enter the basis. If set to zero, the routine uses the steepest edge pricing strategy which is the best local move. If set to one, the minimum reduced cost pricing strategy is used. The steepest edge pricing strategy generally uses fewer iterations than the minimum reduced cost pricing, but each iteration costs more in terms of the amount of calculation performed. However, this is very problem-dependent.  
Default: IPARAM(3) = 0

IPARAM(4) = MXITBR, the number of iterations between recalculating the error in the primal solution is used to monitor the error in solving the linear system. This is an expensive calculation and every tenth iteration is generally enough.  
Default: IPARAM(4) = 10

IPARAM(5) = NPP, the number of negative reduced costs (at most) to be found at each iteration of choosing a variable to enter the basis. If set to zero,  $NPP = NVARS$  will be used, implying that all of the reduced costs are computed at each such step. This "Partial pricing" may increase the total number of iterations required. However, it decreases the number of calculation required at each iteration. The effect on overall efficiency is very problem-dependent. If set to some positive number, that value is used as NPP.  
Default: IPARAM(5) = 0

IPARAM(6) = IREDFQ, the number of steps between basis matrix redecompositions. Redecompositions also occur whenever the linear systems for the primal and dual systems have lost half their working precision.  
Default: IPARAM(6) = 50

IPARAM(7) = LAMAT, the length of the portion of WORK that is allocated to sparse matrix storage and decomposition. LAMAT must be greater than  $NZ + NVARS + 4$ .  
Default: LAMAT = NZ + NVARS + 5

IPARAM(8) = LBM, then length of the portion of IWORK that is allocated to sparse matrix storage and decomposition. LBM must be positive.  
Default: LBM = 8\*M

IPARAM(9) = switch indicating that partial results should be saved after the maximum number of iterations, IPARAM(2), or at the optimum. If

$IPARAM(9)$  is not zero, data essential to continuing the calculation is saved to a file, attached to unit number  $IPARAM(9)$ . The data saved includes all the information about the sparse matrix  $A$  and information about the current basis. If  $IPARAM(9)$  is set to zero, partial results are not saved. It is the responsibility of the calling program to open the output file.

$IPARAM(10)$  = switch indicating that partial results have been computed and stored on unit number  $IPARAM(10)$ , if greater than zero. If  $IPARAM(10)$  is zero, a new problem is started.  
Default:  $IPARAM(10) = 0$

$IPARAM(11)$  = switch indicating that the user supplies scale factors for the columns of the matrix  $A$ . If  $IPARAM(11) = 0$ ,  $SLPRS$  computes the scale factors as the reciprocals of the max norm of each column. If  $IPARAM(11)$  is set to one, element  $I$  of the vector  $COLSCL$  is used as the scale factor for column  $I$  of the matrix  $A$ . The scaling is implicit, so no input data is actually changed.  
Default:  $IPARAM(11) = 0$

$IPARAM(12)$  = switch indicating that the user supplied scale factors for the rows of the matrix  $A$ . If  $IPARAM(12)$  is set to zero, no row scaling is one. If  $IPARAM(12)$  is set to 1, element  $I$  of the vector  $ROWSCL$  is used as the scale factor for row  $I$  of the matrix  $A$ . The scaling is implicit, so no input data is actually changed.  
Default:  $IPARAM(12) = 0$

***RPARAM*** — Real parameter vector of length 7.

$RPARAM(1) = COSTSC$ , a scale factor for the vector of costs. Normally  $SLPRS$  computes this scale factor to be the reciprocal of the max norm if the vector costs after the column scaling has been applied. If  $RPARAM(1)$  is zero,  $SLPRS$  compute  $COSTSC$ .  
Default:  $RPARAM(1) = 0.0$

$RPARAM(2) = ASMALL$ , the smallest magnitude of nonzero entries in the matrix  $A$ . If  $RPARAM(2)$  is nonzero, checking is done to ensure that all elements of  $A$  are at least as large as  $RPARAM(2)$ . Otherwise, no checking is done.  
Default:  $RPARAM(2) = 0.0$

$RPARAM(3) = ABIG$ , the largest magnitude of nonzero entries in the matrix  $A$ . If  $RPARAM(3)$  is nonzero, checking is done to ensure that all elements of  $A$  are no larger than  $RPARAM(3)$ . Otherwise, no checking is done.  
Default:  $RPARAM(3) = 0.0$

$RPARAM(4) = TOLLS$ , the relative tolerance used in checking if the residuals are feasible.  $RPARAM(4)$  is nonzero, that value is used as  $TOLLS$ ,

otherwise the default value is used.  
Default:  $TOLLS = 1000.0 * amach(4)$

$RPARAM(5) = PHI$ , the scaling factor used to scale the reduced cost error estimates. In some environments, it may be necessary to reset  $PHI$  to the range  $[0.01, 0.1]$ , particularly on machines with short word length and working precision when solving a large problem. If  $RPARAM(5)$  is nonzero, that value is used as  $PHI$ , otherwise the default value is used.  
Default:  $PHI = 1.0$

$RPARAM(6) = TOLABS$ , an absolute error test on feasibility. Normally a relative test is used with  $TOLLS$  (see  $RPARAM(4)$ ). If this test fails, an absolute test will be applied using the value  $TOLABS$ .  
Default:  $TOLABS = 0.0$

$RPARAM(7) =$  pivot tolerance of the underlying sparse factorization routine. If  $RPARAM(7)$  is set to zero, the default pivot tolerance is used, otherwise, the  $RPARAM(7)$  is used.  
Default:  $RPARAM(7) = 0.1$

**COLSCL** — Array of length  $NVARS$  containing column scale factors for the matrix  $A$ . (Input).  
 $COLSCL$  is not used if  $IPARAM(11)$  is set to zero.

**ROWSCL** — Array of length  $M$  containing row scale factors for the matrix  $A$ . (Input)  
 $ROWSCL$  is not used if  $IPARAM(12)$  is set to zero.

**WORK** — Work array of length  $LW$ .

**LW** — Length of real work array.  $LW$  must be at least  
 $2 + 2NZ + 9NVAR + 27M + \max(NZ + NVAR + 8, 4NVAR + 7)$ .

**IWORK** — Integer work array of length  $LIW$ .

**LIW** — Length of integer work array.  $LIW$  must be at least  
 $1 + 3NVAR + 41M + \max(NZ + NVAR + 8, 4NVAR + 7)$ .

## Example

Solve a linear programming problem, with

$$A = \begin{bmatrix} 0 & 0.5 & & & \\ & 1 & 0.5 & & \\ & & 1 & \ddots & \\ & & & \ddots & 0.5 \\ & & & & 1 \end{bmatrix}$$

defined in sparse coordinate format.

```

USE SLPRS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER M, NVAR
PARAMETER (M=200, NVAR=200)
!
! Specifications for local variables
INTEGER INDEX, IROW(3*M), J, JCOL(3*M), NOUT, NZ
REAL A(3*M), DSOL(M), OBJ, XSOL(NVAR)
INTEGER IRTYPE(M)
REAL B(M), C(NVAR), XL(NVAR), XU(NVAR)
!
! Specifications for subroutines
DATA B/199*1.7, 1.0/
DATA C/-1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0, &
-10.0, 190*-1.0/
DATA XL/200*0.1/
DATA XU/200*2.0/
DATA IRTYPE/200*1/
!
CALL UMACH (2, NOUT)
!
! Define A
INDEX = 1
DO 10 J=2, M
!
! Superdiagonal element
IROW(INDEX) = J - 1
JCOL(INDEX) = J
A(INDEX) = 0.5
!
! Diagonal element
IROW(INDEX+1) = J
JCOL(INDEX+1) = J
A(INDEX+1) = 1.0
INDEX = INDEX + 2
10 CONTINUE
NZ = INDEX - 1
!
!
XL(4) = 0.2
CALL SLPRS (A, IROW, JCOL, B, B, C, IRTYPE, OBJ, XSOL, DSOL, &
NZ=NZ, XLB=XL, XUB=XU)
!
WRITE (NOUT,99999) OBJ
!
99999 FORMAT (/, 'The value of the objective function is ', E12.6)
!
END

```

## Output

The value of the objective function is  $-.280971\text{E}+03$

---

# TRAN

Solves a transportation problem.

## Required Arguments

*WCAP* — Array of size *NW* containing the source (warehouse) capacities. (Input)

*SREQ* — Array of size *NS* containing the sink (store) requirements. (Input)

*COST* — Array of size *NW* by *NS* containing the cost matrix. (Input)  
*COST* (*I*, *J*) is the per unit cost to ship from source *I* to sink *J*.

*X* — Array of size *NW* by *NS* containing the optimal routing. (Output)  
*X* (*I*, *J*) units should be shipped from source *I* to sink *J*.

*CMIN* — Total cost of the optimal routing. (Output)

## Optional Arguments

*NW* — Number of sources. (Input)  
Default: *NW* = *SIZE* (*WCAP*, 1).

*NS* — Number of sinks. (Input)  
Default: *NS* = *SIZE* (*SREQ*, 1).

*MAXITN* — Upper bound on the number of simplex steps. (Input)  
Default: *MAXITN* = 0, means no limit.

*DUAL* — Array of size *NW* + *NS* containing the dual solution. (Output)

## FORTRAN 90 Interface

Generic:     CALL TRAN (*WCAP*, *SREQ*, *COST*, *X*, *CMIN* [, ...])

Specific:    The specific interface names are *S\_TRAN* and *D\_TRAN*.

## Description

Routine *TRAN* solves the transportation problem.

Minimize

$$\sum_{i=1}^{NW} \sum_{j=1}^{NS} C_{ij} X_{ij}$$

subject to the constraints

$$\sum_{j=1}^{NS} X_{ij} \leq W_i \quad \text{for } i=1, \dots, NW$$

and

$$\sum_{i=1}^{NW} X_{ij} = S_j \quad \text{for } j=1, \dots, NS$$

and

$$X_{ij} \geq 0$$

where  $C = \text{COST}$ ,  $X = x$ ,  $W = \text{WCAP}$  and  $S = \text{SREQ}$ .

The revised simplex method is used to solve a very sparse linear programming problem with  $NW + NS$  constraints and  $NW * NS$  variables. If  $NW = NS = k$ , the work per iteration is  $O(k^2)$ , compared with  $O(k^3)$  when a dense simplex algorithm is used. For more details, see Sewell (2005).

$\text{DUAL}(I)$  gives the decrease in total cost per unit increase in  $\text{WCAP}(I)$ , for small increases, and  $-\text{DUAL}(NW+J)$  gives the increase in total cost per unit increase in  $\text{SREQ}(J)$ .

## Comments

Informational errors

Type Code

3 1 There is insufficient source capacity. The total source capacity is less than the total sink needs, so `TRAN` will return a solution which minimizes the cost to distribute everything in the sources, but does not fill all the sink needs.

4 2 The maximum number of iterations has been exceeded.

## Example

In this example, there are two warehouses with capacities 40 and 20, and 3 stores, which need 25, 10 and 22 units, respectively.

```

USE TRAN_INT
IMPLICIT NONE
INTEGER, PARAMETER :: NW=2, NS=3
INTEGER :: I, J, NOUT
REAL :: X(NW,NS), COST(NW,NS), CMIN
!
 WAREHOUSE CAPACITIES

```

```

REAL :: WCAP(NW) = (/40, 20/)
!
! STORE REQUIREMENTS
REAL :: SREQ(NS) = (/25, 10, 22/)
!
! COSTS
DATA COST/550,350,300,300,400,100/
!
CALL UMACH(2, NOUT)
!
! SOLVE TRANSPORTATION PROBLEM
!
CALL TRAN(WCAP, SREQ, COST, X, CMIN)
!
! PRINT RESULTS
WRITE(NOUT, 99995) CMIN
DO I=1, NW
 DO J=1, NS
 WRITE (NOUT, 99996) X(I,J),I,J
 END DO
END DO
99995 FORMAT (' Minimum cost is ',F10.2)
99996 FORMAT (' Ship ',F5.2,' units from warehouse ',I2, &
' to store ',I2)
END

```

## Output

```

Minimum cost is 19550.00
Ship 25.00 units from warehouse 1 to store 1
Ship 10.00 units from warehouse 1 to store 2
Ship 2.00 units from warehouse 1 to store 3
Ship 0.00 units from warehouse 2 to store 1
Ship 0.00 units from warehouse 2 to store 2
Ship 20.00 units from warehouse 2 to store 3

```

---

## QPROG

Solves a quadratic programming problem subject to linear equality/inequality constraints.

### Required Arguments

**NEQ** — The number of linear equality constraints. (Input)

**A** — NCON by NVAR matrix. (Input)

The matrix contains the equality constraints in the first NEQ rows followed by the inequality constraints.

**B** — Vector of length NCON containing right-hand sides of the linear constraints. (Input)

**G** — Vector of length NVAR containing the coefficients of the linear term of the objective function. (Input)

**H** — NVAR by NVAR matrix containing the Hessian matrix of the objective function. (Input)  
H should be symmetric positive definite; if H is not positive definite, the algorithm

attempts to solve the QP problem with  $H$  replaced by a  $H + \text{DIAGNL} * I$  such that  $H + \text{DIAGNL} * I$  is positive definite. See Comment 3.

**SOL** — Vector of length *NVAR* containing solution. (Output)

### Optional Arguments

**NVAR** — The number of variables. (Input)

Default: *NVAR* = *SIZE* (*A*,2).

**NCON** — The number of linear constraints. (Input)

Default: *NCON* = *SIZE* (*A*,1).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

**LDH** — Leading dimension of *H* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDH* = *SIZE* (*H*,1).

**DIAGNL** — Scalar equal to the multiple of the identity matrix added to *H* to give a positive definite matrix. (Output)

**NACT** — Final number of active constraints. (Output)

**IACT** — Vector of length *NVAR* containing the indices of the final active constraints in the first *NACT* positions. (Output)

**ALAMDA** — Vector of length *NVAR* containing the Lagrange multiplier estimates of the final active constraints in the first *NACT* positions. (Output)

**MAXITN** — This number is the maximum number of iterations allowed. (Input)

If *MAXITN* is set to 0 the iteration count is unbounded.

Default: *MAXITN* = 100000.

**SMALL** — This constant is used in the determination of the positive definiteness of the Hessian *H*. (Input)

*SMALL* is also used for the convergence criteria of a constraint violation.

Default: *SMALL* = 10.0 \* machine precision for single precision and 1000.0\*machine precision for double precision.

### FORTRAN 90 Interface

Generic:     CALL QPROG (NEQ, A, B, G, H, SOL [, ...])

Specific:    The specific interface names are S\_QPROG and D\_QPROG.

## FORTRAN 77 Interface

Single:      CALL QPROG (NVAR, NCON, NEQ, A, LDA, B, G, H, LDH, DIAGNL, SOL, NACT,  
                  IACT, ALAMDA)

Double:      The double precision name is DQPROG.

## Description

The routine QPROG is based on M.J.D. Powell's implementation of the Goldfarb and Idnani (1983) dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints, i.e., problems of the form

$$\begin{aligned} \min_{x \in \mathbf{R}^n} \quad & g^T x + \frac{1}{2} x^T H x \\ \text{subject to} \quad & A_1 x = b_1 \\ & A_2 x \geq b_2 \end{aligned}$$

given the vectors  $b_1$ ,  $b_2$ , and  $g$  and the matrices  $H$ ,  $A_1$ , and  $A_2$ .  $H$  is required to be positive definite. In this case, a unique  $x$  solves the problem or the constraints are inconsistent. If  $H$  is not positive definite, a positive definite perturbation of  $H$  is used in place of  $H$ . For more details, see Powell (1983, 1985).

## Comments

1.    Workspace may be explicitly provided, if desired, by use of Q2ROG/DQ2ROG. The reference is:

```
CALL Q2ROG (NVAR, NCON, NEQ, A, LDA, B, G, H, LDH, DIAGNL,
 SOL, NACT, IACT, ALAMDA, WK)
```

The additional argument is:

**WK** — Work vector of length  $(3 * NVAR**2 + 11 * NVAR)/2 + NCON$ .

2.    Informational errors

Type   Code

3      1      Due to the effect of computer rounding error, a change in the variables fail to improve the objective function value; usually the solution is close to optimum.

4      2      The system of equations is inconsistent. There is no solution.

3.    If a perturbation of  $H$ ,  $H + \text{DIAGNL} * I$ , was used in the QP problem, then  $H + \text{DIAGNL} * I$  should also be used in the definition of the Lagrange multipliers.

## Example

The quadratic programming problem

$$\begin{aligned} \min f(x) &= x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1 \\ \text{subject to } &x_1 + x_2 + x_3 + x_4 + x_5 = 5 \\ &x_3 - 2x_4 - 2x_5 = -3 \end{aligned}$$

is solved.

```
USE QPROG_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDH, NCON, NEQ, NVAR
PARAMETER (NCON=2, NEQ=2, NVAR=5, LDA=NCON, LDH=NVAR)
!
INTEGER K, NACT, NOUT
REAL A(LDA,NVAR), ALAMDA(NVAR), B(NCON), G(NVAR), &
H(LDH,LDH), SOL(NVAR)
!
! Set values of A, B, G and H.
! A = (1.0 1.0 1.0 1.0 1.0)
! (0.0 0.0 1.0 -2.0 -2.0)
!
! B = (5.0 -3.0)
!
! G = (-2.0 0.0 0.0 0.0 0.0)
!
! H = (2.0 0.0 0.0 0.0 0.0)
! (0.0 2.0 -2.0 0.0 0.0)
! (0.0 -2.0 2.0 0.0 0.0)
! (0.0 0.0 0.0 2.0 -2.0)
! (0.0 0.0 0.0 -2.0 2.0)
!
DATA A/1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, -2.0, 1.0, -2.0/
DATA B/5.0, -3.0/
DATA G/-2.0, 4*0.0/
DATA H/2.0, 5*0.0, 2.0, -2.0, 3*0.0, -2.0, 2.0, 5*0.0, 2.0, &
-2.0, 3*0.0, -2.0, 2.0/
!
CALL QPROG (NEQ, A, B, G, H, SOL)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (SOL(K),K=1,NVAR)
99999 FORMAT (' The solution vector is', /, ' SOL = (', 5F6.1, &
')')
!
END
```

## Output

```
The solution vector is
SOL = (1.0 1.0 1.0 1.0 1.0)
```

---

# LCONF

Minimizes a general objective function subject to linear equality/inequality constraints.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
`CALL FCN (N, X, F)`, where

**N** — Value of **NVAR**. (Input)

**X** — Vector of length **N** at which point the function is evaluated. (Input)  
**X** should not be changed by **FCN**.

**F** — The computed function value at the point **X**. (Output)

**FCN** must be declared **EXTERNAL** in the calling program.

**NEQ** — The number of linear equality constraints. (Input)

**A** — **NCON** by **NVAR** matrix. (Input)

The matrix contains the equality constraint gradients in the first **NEQ** rows, followed by the inequality constraint gradients.

**B** — Vector of length **NCON** containing right-hand sides of the linear constraints. (Input)

Specifically, the constraints on the variables  $X(I)$ ,  $I = 1, \dots, \text{NVAR}$  are

$A(K, 1) * X(1) + \dots + A(K, \text{NVAR}) * X(\text{NVAR}) .EQ. B(K)$ ,  $K = 1, \dots,$

$\text{NEQ} . A(K, 1) * X(1) + \dots + A(K, \text{NVAR}) * X(\text{NVAR}) .LE. B(K)$ ,  $K = \text{NEQ} + 1, \dots,$

**NCON**. Note that the data that define the equality constraints come before the data of the inequalities.

**XLB** — Vector of length **NVAR** containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set

$XLB(I) = XUB(I)$  to freeze the  $I$ -th variable. (Input)

Specifically, these simple bounds are  $XLB(I) .LE. X(I)$ ,  $I = 1, \dots, \text{NVAR}$ .

**XUB** — Vector of length **NVAR** containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. (Input)

Specifically, these simple bounds are  $X(I) .LE. XUB(I)$ ,  $I = 1, \dots, \text{NVAR}$ .

**SOL** — Vector of length **NVAR** containing solution. (Output)

## Optional Arguments

**NVAR** — The number of variables. (Input)

Default:  $\text{NVAR} = \text{SIZE}(A, 2)$ .

**NCON** — The number of linear constraints (excluding simple bounds). (Input)

Default:  $\text{NCON} = \text{SIZE}(A, 1)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

**XGUESS** — Vector of length **NVAR** containing the initial guess of the minimum. (Input)  
Default: `XGUESS = 0.0`.

**ACC** — The nonnegative tolerance on the first order conditions at the calculated solution. (Input)  
Default: `ACC = 1.e-4` for single precision and `1.d-8` for double precision.

**MAXFCN** — On input, maximum number of function evaluations allowed. (Input/ Output)  
On output, actual number of function evaluations needed.  
Default: `MAXFCN = 400`.

**OBJ** — Value of the objective function. (Output)

**NACT** — Final number of active constraints. (Output)

**IACT** — Vector containing the indices of the final active constraints in the first **NACT** positions. (Output)  
Its length must be at least `NCON + 2 * NVAR`.

**ALAMDA** — Vector of length **NVAR** containing the Lagrange multiplier estimates of the final active constraints in the first **NACT** positions. (Output)

## FORTRAN 90 Interface

Generic: `CALL LCONF (FCN, NEQ, A, B, XLB, XUB, SOL [, ...])`

Specific: The specific interface names are `S_LCONF` and `D_LCONF`.

## FORTRAN 77 Interface

Single: `CALL LCONF (FCN, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB, XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT, ALAMDA)`

Double: The double precision name is `DLCONF`.

## Description

The routine `LCONF` is based on M.J.D. Powell's `TOLMIN`, which solves linearly constrained optimization problems, i.e., problems of the form

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} f(x) \\ & \text{subject to } A_1 x = b_1 \\ & \quad \quad \quad A_2 x \leq b_2 \end{aligned}$$

$$x_l \leq x \leq x_u$$

given the vectors  $b_1$ ,  $b_2$ ,  $x_l$  and  $x_u$  and the matrices  $A_1$ , and  $A_2$ .

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise  $x^0$ , the initial guess provided by the user, to satisfy

$$A_1 x = b_1$$

Next,  $x^0$  is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible  $x^k$ , let  $J_k$  be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let  $I_k$  be the set of indices of active constraints. The following quadratic programming problem

$$\min f(x^k) + d^T \nabla f(x^k) + \frac{1}{2} d^T B^k d$$

$$\text{subject to } a_j d = 0 \quad j \in I_k$$

$$a_j d \leq 0 \quad j \in J_k$$

is solved to get  $(d^k, \lambda^k)$  where  $a_j$  is a row vector representing either a constraint in  $A_1$  or  $A_2$  or a bound constraint on  $x$ . In the latter case, the  $a_j = e_i$  for the bound constraint  $x_i \leq (x_u)_i$  and  $a_j = -e_i$  for the constraint  $-x_i \leq (-x_l)_i$ . Here,  $e_i$  is a vector with a 1 as the  $i$ -th component, and zeroes elsewhere.  $\lambda^k$  are the Lagrange multipliers, and  $B^k$  is a positive definite approximation to the second derivative  $\nabla^2 f(x^k)$ .

After the search direction  $d^k$  is obtained, a line search is performed to locate a better point. The new point  $x^{k+1} = x^k + \alpha^k d^k$  has to satisfy the conditions

$$f(x^k + \alpha^k d^k) \leq f(x^k) + 0.1 \alpha^k (d^k)^T \nabla f(x^k)$$

and

$$(d^k)^T \nabla f(x^k + \alpha^k d^k) \geq 0.7 (d^k)^T \nabla f(x^k)$$

The main idea in forming the set  $J_k$  is that, if any of the inequality constraints restricts the step-length  $\alpha^k$ , then its index is not in  $J_k$ . Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation,  $B^k$ , is updated by the BFGS formula, if the condition

$$(d^k)^T \nabla f(x^k + \alpha^k d^k) - \nabla f(x^k) > 0$$

holds. Let  $x^k \leftarrow x^{k+1}$ , and start another iteration.

The iteration repeats until the stopping criterion

$$\left\| \nabla f(x^k) - A^k \lambda^k \right\|_2 \leq \tau$$

is satisfied; here,  $\tau$  is a user-supplied tolerance. For more details, see Powell (1988, 1989).

Since a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, whenever the exact gradient can be easily provided, routine `LCONF` should be used instead.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2ONF/DL2ONF`. The reference is:

```
CALL L2ONF (FCN, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB,
 XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT, ALAMDA,
 IPRINT, INFO, WK)
```

The additional arguments are as follows:

***IPRINT*** — Print option (see Comment 3). (Input)

***INFO*** — Informational flag (see Comment 3). (Output)

***WK*** — Real work vector of length  $NVAR**2 + 11 * NVAR + NCON$ .

2. Informational Errors

| Type | Code | Description                                                                                                                                                            |
|------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4    | 4    | The equality constraints are inconsistent.                                                                                                                             |
| 4    | 5    | The equality constraints and the bounds on the variables are found to be inconsistent.                                                                                 |
| 4    | 6    | No vector $x$ satisfies all of the constraints. In particular, the current active constraints prevent any change in $x$ that reduces the sum of constraint violations. |
| 4    | 7    | Maximum number of function evaluations exceeded.                                                                                                                       |
| 4    | 9    | The variables are determined by the equality constraints.                                                                                                              |

3. The following are descriptions of the arguments `IPRINT` and `INFO`:

***IPRINT*** — This argument must be set by the user to specify the frequency of printing during the execution of the routine `LCONF`. There is no printed output if `IPRINT` = 0. Otherwise, after ensuring feasibility, information is given every `IABS(IPRINT)` iterations and whenever a parameter called `TOL` is reduced. The printing provides the values of  $x(\cdot)$ ,  $F(\cdot)$  and  $G(\cdot) = \text{GRAD}(F)$  if `IPRINT` is positive. If `IPRINT` is negative, this information is augmented by the current values of `IACT(K)`  $K = 1, \dots, NACT, `PAR(K)`  $K = 1, \dots, NACT and `RESKT(I)`  $I =$$$

1, ..., N. The reason for returning to the calling program is also displayed when IPRINT is nonzero.

**INFO** — On exit from L2ONE, INFO will have one of the following integer values to indicate the reason for leaving the routine:

INFO = 1 SOL is feasible, and the condition that depends on ACC is satisfied.

INFO = 2 SOL is feasible, and rounding errors are preventing further progress.

INFO = 3 SOL is feasible, but the objective function fails to decrease although a decrease is predicted by the current gradient vector.

INFO = 4 In this case, the calculation cannot begin because LDA is less than NCON or because the lower bound on a variable is greater than the upper bound.

INFO = 5 This value indicates that the equality constraints are inconsistent. These constraints include any components of X(.) that are frozen by setting XL(I) = XU(I).

INFO = 6 In this case there is an error return because the equality constraints and the bounds on the variables are found to be inconsistent.

INFO = 7 This value indicates that there is no vector of variables that satisfies all of the constraints. Specifically, when this return or an INFO = 6 return occurs, the current active constraints (whose indices are IACT(K), K = 1, ..., NACT) prevent any change in X(.) that reduces the sum of constraint violations. Bounds are only included in this sum if INFO = 6.

INFO = 8 Maximum number of function evaluations exceeded.

INFO = 9 The variables are determined by the equality constraints.

### Example

The problem from Schittkowski (1987)

$$\begin{aligned} \min f(x) &= -x_1x_2x_3 \\ \text{subject to} \quad &-x_1 - 2x_2 - 2x_3 \leq 0 \\ &x_1 + 2x_2 + 2x_3 \leq 72 \\ &0 \leq x_1 \leq 20 \\ &0 \leq x_2 \leq 11 \\ &0 \leq x_3 \leq 42 \end{aligned}$$

is solved with an initial guess  $x_1 = 10$ ,  $x_2 = 10$  and  $x_3 = 10$ .

```

USE LCONF_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER NCON, NEQ, NVAR
PARAMETER (NCON=2, NEQ=0, NVAR=3)
!
INTEGER MAXFCN, NOUT
REAL A(NCON,NVAR), ACC, B(NCON), OBJ, &
SOL(NVAR), XGUESS(NVAR), XLB(NVAR), XUB(NVAR)
EXTERNAL FCN
!
! Set values for the following problem.
!
! Min -X(1)*X(2)*X(3)
!
! -X(1) - 2*X(2) - 2*X(3) .LE. 0
! X(1) + 2*X(2) + 2*X(3) .LE. 72
!
! 0 .LE. X(1) .LE. 20
! 0 .LE. X(2) .LE. 11
! 0 .LE. X(3) .LE. 42
!
DATA A/-1.0, 1.0, -2.0, 2.0, -2.0, 2.0/, B/0.0, 72.0/
DATA XLB/3*0.0/, XUB/20.0, 11.0, 42.0/, XGUESS/3*10.0/
DATA ACC/0.0/, MAXFCN/400/
!
CALL UMACH (2, NOUT)
!
CALL LCONF (FCN, NEQ, A, B, XLB, XUB, SOL, XGUESS=XGUESS, &
MAXFCN=MAXFCN, ACC=ACC, OBJ=OBJ)
!
WRITE (NOUT,99998) 'Solution:'
WRITE (NOUT,99999) SOL
WRITE (NOUT,99998) 'Function value at solution:'
WRITE (NOUT,99999) OBJ
WRITE (NOUT,99998) 'Number of function evaluations:', MAXFCN
STOP
99998 FORMAT (//, ' ', A, I4)
99999 FORMAT (1X, 5F16.6)
END
!
SUBROUTINE FCN (N, X, F)
INTEGER N
REAL X(*), F
!
F = -X(1)*X(2)*X(3)
RETURN
END

```

## Output

```

Solution:
20.000000 11.000000 15.000000

```

Function value at solution:  
-3300.000000

Number of function evaluations: 5

---

## LCONG

Minimizes a general objective function subject to linear equality/inequality constraints.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

N — Value of NVAR. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — User-supplied subroutine to compute the gradient at the point X. The usage is  
CALL GRAD (N, X, G), where

N — Value of NVAR. (Input)

X — Vector of length N at which point the function is evaluated. (Input)  
X should not be changed by GRAD.

G — Vector of length N containing the values of the gradient of the objective function evaluated at the point X. (Output)

GRAD must be declared EXTERNAL in the calling program.

**NEQ** — The number of linear equality constraints. (Input)

**A** — NCON by NVAR matrix. (Input)

The matrix contains the equality constraint gradients in the first NEQ rows, followed by the inequality constraint gradients.

**B** — Vector of length NCON containing right-hand sides of the linear constraints. (Input)

Specifically, the constraints on the variables  $X(I)$ ,  $I = 1, \dots, \text{NVAR}$  are

$A(K, 1) * X(1) + \dots + A(K, \text{NVAR}) * X(\text{NVAR}) \text{ .EQ. } B(K)$ ,  $K = 1, \dots,$

$\text{NEQ} \text{ .A. } A(K, 1) * X(1) + \dots + A(K, \text{NVAR}) * X(\text{NVAR}) \text{ .LE. } B(K)$ ,  $K = \text{NEQ} + 1, \dots, \text{NCON}$ .

Note that the data that define the equality constraints come before the data of the inequalities.

***XLB*** — Vector of length *NVAR* containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set  $XLB(I) = XUB(I)$  to freeze the *I*-th variable. (Input)  
Specifically, these simple bounds are  $XLB(I) \leq X(I)$ ,  $I = 1, \dots, NVAR$ .

***XUB*** — Vector of length *NVAR* containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. (Input)  
Specifically, these simple bounds are  $X(I) \leq XUB(I)$ ,  $I = 1, \dots, NVAR$ .

***SOL*** — Vector of length *NVAR* containing solution. (Output)

### Optional Arguments

***NVAR*** — The number of variables. (Input)  
Default:  $NVAR = SIZE(A,2)$ .

***NCON*** — The number of linear constraints (excluding simple bounds). (Input)  
Default:  $NCON = SIZE(A,1)$ .

***LDA*** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = SIZE(A,1)$ .

***XGUESS*** — Vector of length *NVAR* containing the initial guess of the minimum. (Input)  
Default:  $XGUESS = 0.0$ .

***ACC*** — The nonnegative tolerance on the first order conditions at the calculated solution. (Input)  
Default:  $ACC = 1.e-4$  for single precision and  $1.d-8$  for double precision.

***MAXFCN*** — On input, maximum number of function evaluations allowed.(Input/ Output)  
On output, actual number of function evaluations needed.  
Default:  $MAXFCN = 400$ .

***OBJ*** — Value of the objective function. (Output)

***NACT*** — Final number of active constraints. (Output)

***IACT*** — Vector containing the indices of the final active constraints in the first *NACT* positions. (Output)  
Its length must be at least  $NCON + 2 * NVAR$ .

***ALAMDA*** — Vector of length *NVAR* containing the Lagrange multiplier estimates of the final active constraints in the first *NACT* positions. (Output)

### FORTRAN 90 Interface

Generic:     CALL LCONG (FCN, GRAD, NEQ, A, B, XLB, XUB, SOL [, ...])

Specific: The specific interface names are S\_LCONG and D\_LCONG.

## FORTRAN 77 Interface

Single: CALL LCONG (FCN, GRAD, NVAR, NCON, NEQ, A, LDA, B, XLB, XUB, XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT, ALAMDA)

Double: The double precision name is DLCONG.

## Description

The routine LCONG is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} f(x) \\ & \text{subject to} \quad A_1 x = b_1 \\ & \quad \quad \quad A_2 x \leq b_2 \\ & \quad \quad \quad x_l \leq x \leq x_u \end{aligned}$$

given the vectors  $b_1$ ,  $b_2$ ,  $x_l$  and  $x_u$  and the matrices  $A_1$ , and  $A_2$ .

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise  $x^0$ , the initial guess provided by the user, to satisfy

$$A_1 x = b_1$$

Next,  $x^0$  is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible  $x_k$ , let  $J_k$  be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let  $I_k$  be the set of indices of active constraints. The following quadratic programming problem

$$\begin{aligned} & \min f(x^k) + d^T \nabla f(x^k) + \frac{1}{2} d^T B^k d \\ & \text{subject to} \quad a_j d = 0 \quad j \in I_k \\ & \quad \quad \quad a_j d \leq 0 \quad j \in J_k \end{aligned}$$

is solved to get  $(d^k, \lambda^k)$  where  $a_j$  is a row vector representing either a constraint in  $A_1$  or  $A_2$  or a bound constraint on  $x$ . In the latter case, the  $a_j = e_i$  for the bound constraint  $x_i \leq (x_u)_i$  and  $a_j = -e_i$  for the constraint  $-x_i \leq (-x_l)_i$ . Here,  $e_i$  is a vector with a 1 as the  $i$ -th component, and zeroes elsewhere.  $\lambda^k$  are the Lagrange multipliers, and  $B^k$  is a positive definite approximation to the second derivative  $\nabla^2 f(x^k)$ .

After the search direction  $d^k$  is obtained, a line search is performed to locate a better point. The new point  $x^{k+1} = x^k + \alpha^k d^k$  has to satisfy the conditions

$$f(x^k + \alpha^k d^k) \leq f(x^k) + 0.1 \alpha^k (d^k)^T \nabla f(x^k)$$

and

$$(d^k)^T \nabla f(x^k + \alpha^k d^k) \geq 0.7 (d^k)^T \nabla f(x^k)$$

The main idea in forming the set  $J_k$  is that, if any of the inequality constraints restricts the step-length  $\alpha^k$ , then its index is not in  $J_k$ . Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation,  $B^k$ , is updated by the BFGS formula, if the condition

$$(d^k)^T \nabla f(x^k + \alpha^k d^k) - \nabla f(x^k) > 0$$

holds. Let  $x^k \leftarrow x^{k+1}$ , and start another iteration.

The iteration repeats until the stopping criterion

$$\|\nabla f(x^k) - A^k \lambda^k\|_2 \leq \tau$$

is satisfied; here,  $\tau$  is a user-supplied tolerance. For more details, see Powell (1988, 1989).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `L2ONG/DL2ONG`. The reference is:

```
CALL L2ONG (FCN, GRAD, NVAR, NCON, NEQ, A, LDA, B, XLB,
 XUB, XGUESS, ACC, MAXFCN, SOL, OBJ, NACT, IACT,
 ALAMDA, IPRINT, INFO, WK)
```

The additional arguments are as follows:

**IPRINT** — Print option (see Comment 3). (Input)

**INFO** — Informational flag (see Comment 3). (Output)

**WK** — Real work vector of length  $\text{NVAR}^*2 + 11 * \text{NVAR} + \text{NCON}$ .

2. Informational errors

Type Code

4 4 The equality constraints are inconsistent.

4 5 The equality constraints and the bounds on the variables are found to be inconsistent.

- 4    6            No vector  $x$  satisfies all of the constraints. In particular, the current active constraints prevent any change in  $x$  that reduces the sum of constraint violations.
- 4    7            Maximum number of function evaluations exceeded.
- 4    9            The variables are determined by the equality constraints.

3. The following are descriptions of the arguments `IPRINT` and `INFO`:

***IPRINT*** — This argument must be set by the user to specify the frequency of printing during the execution of the routine `LCONG`. There is no printed output if `IPRINT = 0`. Otherwise, after ensuring feasibility, information is given every `IABS(IPRINT)` iterations and whenever a parameter called `TOL` is reduced. The printing provides the values of  $x(\cdot)$ ,  $F(\cdot)$  and  $G(\cdot) = \text{GRAD}(F)$  if `IPRINT` is positive. If `IPRINT` is negative, this information is augmented by the current values of `IACT(K)`  $K = 1, \dots, \text{NACT}$ , `PAR(K)`  $K = 1, \dots, \text{NACT}$  and `RESKT(I)`  $I = 1, \dots, N$ . The reason for returning to the calling program is also displayed when `IPRINT` is nonzero.

***INFO*** — On exit from `L2ONG`, `INFO` will have one of the following integer values to indicate the reason for leaving the routine:

`INFO = 1` SOL is feasible and the condition that depends on `ACC` is satisfied.

`INFO = 2` SOL is feasible and rounding errors are preventing further progress.

`INFO = 3` SOL is feasible but the objective function fails to decrease although a decrease is predicted by the current gradient vector.

`INFO = 4` In this case, the calculation cannot begin because `LDA` is less than `NCON` or because the lower bound on a variable is greater than the upper bound.

`INFO = 5` This value indicates that the equality constraints are inconsistent. These constraints include any components of  $x(\cdot)$  that are frozen by setting `XL(I) = XU(I)`.

`INFO = 6` In this case, there is an error return because the equality constraints and the bounds on the variables are found to be inconsistent.

`INFO = 7` This value indicates that there is no vector of variables that satisfies all of the constraints. Specifically, when this return or an

`INFO = 6` return occurs, the current active constraints (whose indices are `IACT(K)`,  $K = 1, \dots, \text{NACT}$ ) prevent any change in  $x(\cdot)$  that reduces the sum of constraint violations, where only bounds are included in this sum if `INFO = 6`.

INFO = 8 Maximum number of function evaluations exceeded.

INFO = 9 The variables are determined by the equality constraints.

## Example

The problem from Schittkowski (1987)

$$\begin{aligned} \min f(x) &= -x_1 x_2 x_3 \\ \text{subject to} \quad &-x_1 - 2x_2 - 2x_3 \leq 0 \\ &x_1 + 2x_2 + 2x_3 \leq 72 \\ &0 \leq x_1 \leq 20 \\ &0 \leq x_2 \leq 11 \\ &0 \leq x_3 \leq 42 \end{aligned}$$

is solved with an initial guess  $x_1 = 10$ ,  $x_2 = 10$  and  $x_3 = 10$ .

```
USE LCONG_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER NCON, NEQ, NVAR
PARAMETER (NCON=2, NEQ=0, NVAR=3)
!
INTEGER MAXFCN, NOUT
REAL A(NCON,NVAR), ACC, B(NCON), OBJ, &
SOL(NVAR), XGUESS(NVAR), XLB(NVAR), XUB(NVAR)
EXTERNAL FCN, GRAD
!
! Set values for the following problem.
!
! Min -X(1)*X(2)*X(3)
!
! -X(1) - 2*X(2) - 2*X(3) .LE. 0
! X(1) + 2*X(2) + 2*X(3) .LE. 72
!
! 0 .LE. X(1) .LE. 20
! 0 .LE. X(2) .LE. 11
! 0 .LE. X(3) .LE. 42
!
DATA A/-1.0, 1.0, -2.0, 2.0, -2.0, 2.0/, B/0.0, 72.0/
DATA XLB/3*0.0/, XUB/20.0, 11.0, 42.0/, XGUESS/3*10.0/
DATA ACC/0.0/, MAXFCN/400/
!
CALL UMACH (2, NOUT)
!
CALL LCONG (FCN, GRAD, NEQ, A, B, XLB, XUB, SOL, XGUESS=XGUESS, &
ACC=ACC, MAXFCN=MAXFCN, OBJ=OBJ)
!
WRITE (NOUT,99998) 'Solution:'
WRITE (NOUT,99999) SOL
WRITE (NOUT,99998) 'Function value at solution:'
```

```

 WRITE (NOUT,99999) OBJ
 WRITE (NOUT,99998) 'Number of function evaluations:', MAXFCN
 STOP
99998 FORMAT (//, ' ', A, I4)
99999 FORMAT (1X, 5F16.6)
 END
!
 SUBROUTINE FCN (N, X, F)
 INTEGER N
 REAL X(*), F
!
 F = -X(1)*X(2)*X(3)
 RETURN
 END
!
 SUBROUTINE GRAD (N, X, G)
 INTEGER N
 REAL X(*), G(*)
!
 G(1) = -X(2)*X(3)
 G(2) = -X(1)*X(3)
 G(3) = -X(1)*X(2)
 RETURN
 END

```

## Output

```

Solution:
20.000000 11.000000 15.000000

Function value at solution:
-3300.000000

Number of function evaluations: 5

```

---

## NNLPF

Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the objective function and constraints at a given point. The internal usage is  
 CALL FCN (X, IACT, RESULT, IERR), where

**X** — The point at which the objective function or constraint is evaluated.  
 (Input)

**IACT** — Integer indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If IACT is zero, then an objective function evaluation is requested. If IACT is nonzero then the value if IACT indicates the index of the constraint to evaluate. IACT = 1 to

$ME$  for equality constraints and  $I_{ACT} = ME + 1$  to  $M$  for inequality constraints. (Input)

**RESULT** – If  $I_{ACT}$  is zero, then **RESULT** is the computed function value at the point  $X$ . If  $I_{ACT}$  is nonzero, then **RESULT** is the computed constraint value at the point  $X$ . (Output)

**IERR** – Logical variable. On input **IERR** is set to `.FALSE.` If an error or other undesirable condition occurs during evaluation, then **IERR** should be set to `.TRUE.` Setting **IERR** to `.TRUE.` will result in the step size being reduced and the step being tried again. (If **IERR** is set to `.TRUE.` for **XGUESS**, then an error is issued.)

The routine **FCN** must be use-associated in a user module that uses `NNLPPF_INT`, or else declared `EXTERNAL` in the calling program. If **FCN** is a separately compiled routine, not in a module, then it must be declared `EXTERNAL`.

**M** — Total number of constraints. (Input)

**ME** — Number of equality constraints. (Input)

**IBTYPE** — Scalar indicating the types of bounds on variables. (Input)

| <b>IBTYPE</b> | <b>Action</b>                                                                                 |
|---------------|-----------------------------------------------------------------------------------------------|
| 0             | User will supply all the bounds.                                                              |
| 1             | All variables are nonnegative.                                                                |
| 2             | All variables are nonpositive.                                                                |
| 3             | User supplies only the bounds on 1st variable; all other variables will have the same bounds. |

**XLB** — Vector of length  $N$  containing the lower bounds on variables. (Input, if **IBTYPE** = 0; output, if **IBTYPE** = 1 or 2; input/output, if **IBTYPE** = 3)  
If there is no lower bound for a variable, then the corresponding **XLB** value should be set to `-Huge(X(1))`.

**XUB** — Vector of length  $N$  containing the upper bounds on variables. (Input, if **IBTYPE** = 0; output, if **IBTYPE** = 1 or 2; input/output, if **IBTYPE** = 3).  
If there is no upper bound for a variable, then the corresponding **XUB** value should be set to `Huge(X(1))`.

**X** — Vector of length  $N$  containing the computed solution. (Output)

## Optional Arguments

**N** — Number of variables. (Input)  
Default:  $N = \text{SIZE}(X)$ .

**XGUESS** — Vector of length  $N$  containing an initial guess of the solution. (Input)

Default:  $XGUESS = x$ , (with the smallest value of  $\|x\|_2$ ) that satisfies the bounds.

**XSCALE** — Vector of length  $N$  setting the internal scaling of the variables. The initial value given and the objective function and gradient evaluations however are always in the original unscaled variables. The first internal variable is obtained by dividing values  $X(I)$  by  $XSCALE(I)$ . (Input)

In the absence of other information, set all entries to 1.0.

Default:  $XSCALE(:) = 1.0$ .

**IPRINT** — Parameter indicating the desired output level. (Input)

**IPRINT    Action**

0            No output printed.

1            One line of intermediate results is printed in each iteration.

2            Lines of intermediate results summarizing the most important data for each step are printed.

3            Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking and etc are printed

4            Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated and etc are printed.

Default:  $IPRINT = 0$ .

**MAXITN** — Maximum number of iterations allowed. (Input)

Default:  $MAXITN = 200$ .

**EPSDIF** — Relative precision in gradients. (Input)

Default:  $EPSDIF = \text{epsilon}(1)$

**TAU0** — A universal bound describing how much the unscaled penalty-term may deviate from zero. (Input)

NNLPF assumes that within the region described by

$$\sum_{i=1}^{M_e} |g_i(x)| - \sum_{i=M_e+1}^M \min(0, g_i(x)) \leq \text{TAU0}$$

all functions may be evaluated safely. The initial guess, however, may violate these requirements. In that case an initial feasibility improvement phase is run by NNLPF until such a point is found. A small  $TAU0$  diminishes the efficiency of NNLPF, because

the iterates then will follow the boundary of the feasible set closely. Conversely, a large `TAU0` may degrade the reliability of the code.

Default `TAU0 = 1.E0`

**DELO** — In the initial phase of minimization a constraint is considered binding if

$$\frac{g_i(x)}{\max(1, \|\nabla g_i(x)\|)} \leq_{\text{DELO}} \quad i = M_e + 1, \dots, M$$

Good values are between .01 and 1.0. If `DELO` is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if `DELO` is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well-scaled problems `DELO=1.0` is reasonable. (Input)

Default: `DELO = .5*TAU0`

**EPSFCN** – Relative precision of the function evaluation routine. (Input)

Default: `EPSFCN = epsilon(1)`

**IDTYPE** – Type of numerical differentiation to be used. (Input)

Default: `IDTYPE = 1`

| <b>IDTYPE</b> | <b>Action</b>                                                                                                                                                                         |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1             | Use a forward difference quotient with discretization stepsize $0.1(\text{EPSFCN}^{1/2})$ componentwise relative.                                                                     |
| 2             | Use the symmetric difference quotient with discretization stepsize $0.1(\text{EPSFCN}^{1/3})$ componentwise relative                                                                  |
| 3             | Use the sixth order approximation computing a Richardson extrapolation of three symmetric difference quotient values. This uses a discretization stepsize $0.01(\text{EPSFCN}^{1/7})$ |

**TAUBND** – Amount by which bounds may be violated during numerical differentiation.

Bounds are violated by `TAUBND` (at most) only if a variable is on a bound and finite differences are taken for gradient evaluations. (Input)

Default: `TAUBND = 1.E0`

**SMALLW** — Scalar containing the error allowed in the multipliers. For example, a negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than `SMALLW`. (Input)

Default: `SMALLW = exp(2*log(epsilon(x(1)/3)))`

**DELMIN** — Scalar which defines allowable constraint violations of the final accepted result.

Constraints are satisfied if  $|g_i(x)| \leq_{\text{DELMIN}}$ , and  $g_j(x) \geq (-\text{DELMIN})$  respectively.

(Input)

Default:  $\text{DELMIN} = \min(\text{DEL0}/10, \max(\text{EPSDIF}, \min(\text{DEL0}/10, \max(1.E-6*\text{DEL0}, \text{SMALLW}))))$

**SCFMAX** — Scalar containing the bound for the internal automatic scaling of the objective function. (Input)  
Default:  $\text{SCFMAX} = 1.0\text{E}4$

**FVALUE** — Scalar containing the value of the objective function at the computed solution. (Output)

**LGMULT** — Vector of length  $M$  containing the Lagrange multiplier estimates of the constraints. (Output)

**CONSTRES** — Vector of length  $M$  containing the constraint residuals. (Output)

## FORTRAN 90 Interface

Generic: `CALL>NNLPF (FCN, M, ME, IBTYPE, XLB, XUB, X [, ...] )`

Specific: The specific interface names are `S>NNLPF` and `D>NNLPF`.

## Description

The routine `NNLPF` provides an interface to a licensed version of subroutine `DONLP2`, a FORTRAN code developed by Peter Spellucci (1998). It uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the “working sets”). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armijijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: *An SQP method for general nonlinear programs using only equality constrained subproblems*. Math. Prog. 82, (1998), 413-448.

P. Spellucci: *A new technique for inconsistent problems in the SQP method*. Math. Meth. of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} f(x) \\ \text{subject to} & \quad g_j(x) = 0, \text{ for } j = 1, \dots, m_e \\ & \quad g_j(x) \geq 0, \text{ for } j = m_e + 1, \dots, m \\ & \quad x_l \leq x \leq x_u \end{aligned}$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, `NNLPF` allows for several

parameters of the algorithm to be adjusted to account for specific characteristics of problems. The DONLP2 Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the performance of the algorithm. The DONLP2 Users Guide is available in the “help” subdirectory of the main IMSL product installation directory. In addition, the following are a number of guidelines to consider when using NNLPF.

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See optional argument XGUESS.
- Gradient approximation methods can have an effect on the success of NNLPF. Selecting a higher order approximation method may be necessary for some problems. See optional argument IDTYPE.
- If a two sided constraint  $l_i \leq g_i(x) \leq u_i$  is transformed into two constraints  $g_{2i}(x) \geq 0$  and  $g_{2i+1}(x) \geq 0$ , then choose  $DELO < \frac{1}{2}(u_i - l_i) / \max\{1, \|\nabla g_i(x)\|\}$ , or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument DELO.
- The parameter IERR provided in the interface to the user supplied function FCN can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting IERR to .TRUE. and returning without performing the evaluation will avoid the exception. NNLPF will then reduce the stepsize and try the step again. Note, if IERR is set to .TRUE. for the initial guess, then an error is issued.

## Comments

### 1. Informational errors

| Type | Code |                                                                                 |
|------|------|---------------------------------------------------------------------------------|
| 4    | 1    | Constraint evaluation returns an error with current point.                      |
| 4    | 2    | Objective evaluation returns an error with current point.                       |
| 4    | 3    | Working set is singular in dual extended QP.                                    |
| 4    | 4    | QP problem is seemingly infeasible.                                             |
| 4    | 5    | A stationary point located or termination criteria too strong.                  |
| 4    | 8    | Maximum number of iterations exceeded.                                          |
| 4    | 9    | Stationary point not feasible.                                                  |
| 4    | 10   | Very slow primal progress.                                                      |
| 4    | 11   | The problem is singular.                                                        |
| 4    | 12   | Matrix of gradients of binding constraints is singular or very ill-conditioned. |
| 4    | 13   | Small changes in the penalty function.                                          |

## Example

The problem

$$\begin{aligned} \min F(x) &= (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to } g_1(x) &= x_1 - 2x_2 + 1 = 0 \\ g_2(x) &= -x_1^2/4 - x_2^2 + 1 \geq 0 \end{aligned}$$

is solved.

```

USE>NNLPF_INT
USE>WRRRN_INT

IMPLICIT>NONE
INTEGER>IBTYPE, M, ME
PARAMETER>(IBTYPE=0, M=2, ME=1)
!
REAL(KIND(1E0)) FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
EXTERNAL FCN
!
XLB = -HUGE(X(1))
XUB = HUGE(X(1))
!
CALL>NNLPF(FCN, M, ME, IBTYPE, XLB, XUB, X)
!
CALL>WRRRN('The solution is', X)
END

SUBROUTINE FCN(X, IACT, RESULT, IERR)
INTEGER>IACT
REAL(KIND(1E0)) X(*), RESULT
LOGICAL>IERR
!
SELECT CASE(IACT)
CASE(0)
RESULT = (X(1)-2.0E0)**2 + (X(2)-1.0E0)**2
CASE(1)
RESULT = X(1) - 2.0E0*X(2) + 1.0E0
CASE(2)
RESULT = -(X(1)**2)/4.0E0 - X(2)**2 + 1.0E0
END SELECT
RETURN
END

```

## Output

```

The solution is
1 0.8229
2 0.9114

```

---

## NNLPG

Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method with user supplied gradients.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the objective function and constraints at a given point. The internal usage is

CALL FCN (X, IACT, RESULT, IERR), where

**X** — The point at which the objective function or constraint is evaluated.  
(Input)

**IACT** — Integer indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If IACT is zero, then an objective function evaluation is requested. If IACT is nonzero then the value if IACT indicates the index of the constraint to evaluate. IACT = 1 to ME for equality constraints and IACT = ME + 1 to M for inequality constraints. (Input)

**RESULT** — If IACT is zero, then RESULT is the computed objective function value at the point X. If IACT is nonzero, then RESULT is the computed constraint value at the point X. (Output)

**IERR** — Logical variable. On input IERR is set to .FALSE. If an error or other undesirable condition occurs during evaluation, then IERR should be set to .TRUE. Setting IERR to .TRUE. will result in the step size being reduced and the step being tried again. (If IERR is set to .TRUE. for XGUESS, then an error is issued.)

The routine FCN must be use-associated in a user module that uses>NNLPG\_INT, or else declared EXTERNAL in the calling program. If FCN is a separately compiled routine, not in a module, then it must be declared EXTERNAL.

**GRAD** — User-supplied subroutine to evaluate the gradients at a given point. The usage is

CALL GRAD (X, IACT, RESULT), where

**X** — The point at which the gradient of the objective function or gradient of a constraint is evaluated. (Input)

**IACT** — Integer indicating whether evaluation of the function gradient is requested or evaluation of a constraint gradient is requested. If IACT is zero, then an objective function gradient evaluation is requested. If IACT is nonzero then the value if IACT indicates the index of the constraint gradient to evaluate. (Input)  
IACT = 1 to ME for equality constraints and IACT = ME + 1 to M for inequality constraints.

**RESULT** — If IACT is zero, then RESULT is the computed gradient of the objective function at the point X. If IACT is nonzero, then RESULT is the computed gradient of the requested constraint value at the point X. (Output)

The routine GRAD must be use-associated in a user module that uses>NNLPG\_INT, or else declared EXTERNAL in the calling program. If GRAD is a separately compiled routine, not in a module, then it must be declared EXTERNAL.

***M*** — Total number of constraints. (Input)

***ME*** — Number of equality constraints. (Input)

***IBTYPE*** — Scalar indicating the types of bounds on variables. (Input)

| <b>IBTYPE</b> | <b>Action</b>                                                                                 |
|---------------|-----------------------------------------------------------------------------------------------|
| 0             | User will supply all the bounds.                                                              |
| 1             | All variables are nonnegative.                                                                |
| 2             | All variables are nonpositive.                                                                |
| 3             | User supplies only the bounds on 1st variable, all other variables will have the same bounds. |

***XLB*** — Vector of length *N* containing the lower bounds on the variables. (Input, if *IBTYPE* = 0; output, if *IBTYPE* = 1 or 2; input/output, if *IBTYPE* = 3) If there is no lower bound on a variable, then the corresponding *XLB* value should be set to  $-\text{huge}(x(1))$ .

***XUB*** — Vector of length *N* containing the upper bounds on the variables. (Input, if *IBTYPE* = 0; output, if *IBTYPE* = 1 or 2; input/output, if *IBTYPE* = 3) If there is no upper bound on a variable, then the corresponding *XUB* value should be set to  $\text{huge}(x(1))$ .

***X*** — Vector of length *N* containing the computed solution. (Output)

### Optional Arguments

***N*** — Number of variables. (Input)  
Default:  $N = \text{SIZE}(X)$ .

***IPRINT*** — Parameter indicating the desired output level. (Input)

| <b>IPRINT</b> | <b>Action</b>                                                                                                                                                            |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0             | No output printed.                                                                                                                                                       |
| 1             | One line of intermediate results is printed in each iteration.                                                                                                           |
| 2             | Lines of intermediate results summarizing the most important data for each step are printed.                                                                             |
| 3             | Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking and etc are printed |

4 Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated and etc are printed.

Default: `IPRINT = 0`.

**MAXITN** — Maximum number of iterations allowed. (Input)

Default: `MAXITN = 200`.

**XGUESS** — Vector of length `N` containing an initial guess of the solution. (Input)

Default: `XGUESS = x`, (with the smallest value of  $\|x\|_2$ ) that satisfies the bounds.

**TAU0** — A universal bound describing how much the unscaled penalty-term may deviate from zero. (Input)

`NNLPG` assumes that within the region described by

$$\sum_{i=1}^{M_e} |g_i(x)| - \sum_{i=M_e+1}^M \min(0, g_i(x)) \leq \text{TAU0}$$

all functions may be evaluated safely. The initial guess however, may violate these requirements. In that case an initial feasibility improvement phase is run by `NNLPG` until such a point is found. A small `TAU0` diminishes the efficiency of `NNLPG`, because the iterates then will follow the boundary of the feasible set closely. Conversely, a large `TAU0` may degrade the reliability of the code.

Default: `TAU0 = 1.E0`

**DELO** — In the initial phase of minimization a constraint is considered binding if

$$\frac{g_i(x)}{\max(1, \|\nabla g_i(x)\|)} \leq \text{DELO} \quad i = M_e + 1, \dots, M$$

Good values are between .01 and 1.0. If `DELO` is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if `DELO` is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well-scaled problems `DELO=1.0` is reasonable. (Input)

Default: `DELO = .5*TAU0`

**SMALLW** — Scalar containing the error allowed in the multipliers. For example, a negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than `SMALLW`. (Input)

Default: `SMALLW = exp(2*log(epsilon(x(1)/3)))`

**DELMIN** — Scalar which defines allowable constraint violations of the final accepted result. Constraints are satisfied if  $|g_i(x)| \leq \text{DELMIN}$ , and  $g_j(x) \geq (-\text{DELMIN})$  respectively.

(Input)

Default:  $\text{DELMIN} = \min(\text{DELO}/10, \max(1.E-6*\text{DELO}, \text{SMALLW}))$

**SCFMAX** — Scalar containing the bound for the internal automatic scaling of the objective function. (Input)

Default:  $\text{SCFMAX} = 1.0E4$

**FVALUE** — Scalar containing the value of the objective function at the computed solution. (Output)

**LGMULT** — Vector of length  $M$  containing the Lagrange multiplier estimates of the constraints. (Output)

**CONSTRES** — Vector of length  $M$  containing the constraint residuals. (Output)

## FORTRAN 90 Interface

Generic: `CALL>NNLPG (FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X [, ...])`

Specific: The specific interface names are `S>NNLPG` and `D>NNLPG`.

## Description

The routine `NNLPG` provides an interface to a licensed version of subroutine `DONLP2`, a FORTRAN code developed by Peter Spellucci (1998). It uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the “working sets”). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armijijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: *An SQP method for general nonlinear programs using only equality constrained subproblems*. Math. Prog. 82, (1998), 413-448.

P. Spellucci: *A new technique for inconsistent problems in the SQP method*. Math. Meth. of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

$$\begin{aligned} & \min_{x \in \mathbf{R}^n} f(x) \\ \text{subject to} & \quad g_j(x) = 0, \text{ for } j = 1, \dots, m_e \\ & \quad g_j(x) \geq 0, \text{ for } j = m_e + 1, \dots, m \\ & \quad x_l \leq x \leq x_u \end{aligned}$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, `NNLPG` allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The `DONLP2` Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the performance of the algorithm. The `DONLP2` Users Guide is available in the “*help*” subdirectory of the main IMSL product installation directory. In addition, the following are a number of guidelines to consider when using `NNLPG`.

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See optional argument `XGUESS`.
- If a two sided constraint  $l_i \leq g_i(x) \leq u_i$  is transformed into two constraints  $g_{2i}(x) \geq 0$  and  $g_{2i+1}(x) \geq 0$ , then choose `DELO`  $< \frac{1}{2}(u_i - l_i) / \max\{1, \|\nabla g_i(x)\|\}$ , or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument `DELO`.
- The parameter `IERR` provided in the interface to the user supplied function `FCN` can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting `IERR` to `.TRUE.` and returning without performing the evaluation will avoid the exception. `NNLPG` will then reduce the stepsize and try the step again. Note, if `IERR` is set to `.TRUE.` for the initial guess, then an error is issued.

## Comments

### 1. Informational errors

| Type | Code |                                                                                 |
|------|------|---------------------------------------------------------------------------------|
| 4    | 1    | Constraint evaluation returns an error with current point.                      |
| 4    | 2    | Objective evaluation returns an error with current point.                       |
| 4    | 3    | Working set is singular in dual extended QP.                                    |
| 4    | 4    | QP problem is seemingly infeasible.                                             |
| 4    | 5    | A stationary point located or termination criteria too strong.                  |
| 4    | 8    | Maximum number of iterations exceeded.                                          |
| 4    | 9    | Stationary point not feasible.                                                  |
| 4    | 10   | Very slow primal progress.                                                      |
| 4    | 11   | The problem is singular.                                                        |
| 4    | 12   | Matrix of gradients of binding constraints is singular or very ill-conditioned. |
| 4    | 13   | Small changes in the penalty function.                                          |

## Example 1

The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to  $g_1(x) = x_1 - 2x_2 + 1 = 0$

$$g_2(x) = -x_1^2/4 - x_2^2 + 1 \geq 0$$

is solved.

```

USE>NNLPG_INT
USE>WRRRN_INT

IMPLICIT>NONE
INTEGER>IBTYPE, M, ME
PARAMETER>(IBTYPE=0, M=2, ME=1)
!
REAL(KIND(1E0))>FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
EXTERNAL>FCN, GRAD
!
XLB>=-HUGE(X(1))
XUB>=HUGE(X(1))
!
CALL>NNLPG(FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X)
!
CALL>WRRRN('The solution is', X)
END

SUBROUTINE>FCN(X, IACT, RESULT, IERR)
INTEGER>IACT
REAL(KIND(1E0))>X(*), RESULT
LOGICAL>IERR
!
SELECT>CASE(IACT)
CASE(0)
 RESULT>=(X(1)-2.0E0)**2 + (X(2)-1.0E0)**2
CASE(1)
 RESULT>=X(1) - 2.0E0*X(2) + 1.0E0
CASE(2)
 RESULT>=-(X(1)**2)/4.0E0 - X(2)**2 + 1.0E0
END>SELECT
RETURN
END

SUBROUTINE>GRAD(X, IACT, RESULT)
INTEGER>IACT
REAL(KIND(1E0))>X(*), RESULT(*)
!
SELECT>CASE(IACT)
CASE(0)
 RESULT>(1) = 2.0E0*(X(1)-2.0E0)
 RESULT>(2) = 2.0E0*(X(2)-1.0E0)
CASE(1)
 RESULT>(1) = 1.0E0
 RESULT>(2) = -2.0E0
CASE(2)
 RESULT>(1) = -0.5E0*X(1)

```

```

 RESULT (2) = -2.0E0*X(2)
 END SELECT
 RETURN
END

```

## Output

```

The solution is
1 0.8229
2 0.9114

```

## Additional Examples

### Example 2

The same problem from Example 1 is solved, but here we use central differences to compute the gradient of the first constraint. This example demonstrates how `NNLPG` can be used in cases when analytic gradients are known for only a portion of the constraints and/or objective function. The subroutine `CDGRD` is used to compute an approximation to the gradient of the first constraint.

```

 USE>NNLPG_INT
 USE>CDGRD_INT
 USE>WRRRN_INT

 IMPLICIT>NONE
 INTEGER>IBTYPE, M, ME
 PARAMETER>(IBTYPE=0, M=2, ME=1)
!
 REAL(KIND(1E0))>FVALUE, X(2), XGUESS(2), XLB(2), XUB(2)
 EXTERNAL>FCN, GRAD
!
 XLB>=-HUGE(X(1))
 XUB>=HUGE(X(1))
!
 CALL>NNLPG(FCN, GRAD, M, ME, IBTYPE, XLB, XUB, X)
!
 CALL>WRRRN('The solution is', X)
 END

 SUBROUTINE>FCN(X, IACT, RESULT, IERR)
 INTEGER>IACT
 REAL(KIND(1E0))>X(2), RESULT
 LOGICAL>IERR
 EXTERNAL>CONSTR1
!
 SELECT>CASE(IACT)
 CASE(0)
 RESULT=(X(1)-2.0E0)**2+(X(2)-1.0E0)**2
 CASE(1)
 CALL>CONSTR1(2, X, RESULT)
 CASE(2)
 RESULT=-X(1)**2/4.0E0-X(2)**2+1.0E0
 END>SELECT
 RETURN
 END

```

```

SUBROUTINE GRAD (X, IACT, RESULT)
USE CDGRD_INT
INTEGER IACT
REAL(KIND(1E0)) X(2), RESULT(2)
EXTERNAL CONSTR1
!
SELECT CASE (IACT)
CASE(0)
 RESULT (1) = 2.0E0*(X(1)-2.0E0)
 RESULT (2) = 2.0E0*(X(2)-1.0E0)
CASE(1)
 CALL CDGRD(CONSTR1, X, RESULT)
CASE(2)
 RESULT (1) = -0.5E0*X(1)
 RESULT (2) = -2.0E0*X(2)
END SELECT
RETURN
END

SUBROUTINE CONSTR1 (N, X, RESULT)
INTEGER N
REAL(KIND(1E0)) X(*), RESULT
RESULT = X(1) - 2.0E0*X(2) + 1.0E0
RETURN
END

```

## Output

```

The solution is
1 0.8229
2 0.9114

```

---

## CDGRD

Approximates the gradient using central differences.

### Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is

```
CALL FCN (N, X, F), where
```

**N** — Length of *X*. (Input)

**X** — The point at which the function is evaluated. (Input)  
*X* should not be changed by *FCN*.

**F** — The computed function value at the point *X*. (Output)

*FCN* must be declared `EXTERNAL` in the calling program.

**XC** — Vector of length *N* containing the point at which the gradient is to be estimated.  
 (Input)

**GC** — Vector of length  $N$  containing the estimated gradient at  $XC$ . (Output)

### Optional Arguments

**$N$**  — Dimension of the problem. (Input)  
Default:  $N = \text{SIZE}(XC,1)$ .

**$XSCALE$**  — Vector of length  $N$  containing the diagonal scaling matrix for the variables. (Input)  
In the absence of other information, set all entries to 1.0.  
Default:  $XSCALE = 1.0$ .

**$EPSFCN$**  — Estimate for the relative noise in the function. (Input)  
 $EPSFCN$  must be less than or equal to 0.1. In the absence of other information, set  $EPSFCN$  to 0.0.  
Default:  $EPSFCN = 0.0$ .

### FORTRAN 90 Interface

Generic:    CALL CDGRD (FCN, XC, GC [, ...])

Specific:   The specific interface names are S\_CDGRD and D\_CDGRD.

### FORTRAN 77 Interface

Single:     CALL CDGRD (FCN, N, XC, XSCALE, EPSFCN, GC)

Double:     The double precision name is DCDGRD.

### Description

The routine CDGRD uses the following finite-difference formula to estimate the gradient of a function of  $n$  variables at  $x$ :

$$\frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i} \quad \text{for } i = 1, \dots, n$$

where

$$h_i = \varepsilon^{1/3} \max\{|x_i|, 1/s_j\} \text{sign}(x_i),$$

$\varepsilon$  is the machine epsilon,  $s_j$  is the scaling factor of the  $i$ -th variable, and  $e_i$  is the  $i$ -th unit vector. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

## Comments

This is Description A5.6.4, Dennis and Schnabel, 1983, page 323.

## Example

In this example, the gradient of  $f(x) = x_1 - x_1x_2 - 2$  is estimated by the finite-difference method at the point (1.0, 1.0).

```
USE CDGRD_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER I, N, NOUT
PARAMETER (N=2)
REAL EPSFCN, GC(N), XC(N)
EXTERNAL FCN

! Initialization.
DATA XC/2*1.0E0/
! Set function noise.
EPSFCN = 0.01
!
CALL CDGRD (FCN, XC, GC, EPSFCN=EPSFCN)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (GC(I),I=1,N)
99999 FORMAT (' The gradient is', 2F8.2, /)
!
END
!
SUBROUTINE FCN (N, X, F)
INTEGER N
REAL X(N), F
!
F = X(1) - X(1)*X(2) - 2.0E0
!
RETURN
END
```

## Output

```
The gradient is 0.00 -1.00
```

---

# FDGRD

Approximates the gradient using forward differences.

## Required Arguments

*FCN* — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (N, X, F), where

*N* – Length of *X*. (Input)

*X* – The point at which the function is evaluated. (Input)  
X should not be changed by *FCN*.

*F* – The computed function value at the point *X*. (Output)

*FCN* must be declared `EXTERNAL` in the calling program.

*XC* — Vector of length *N* containing the point at which the gradient is to be estimated. (Input)

*FC* — Scalar containing the value of the function at *XC*. (Input)

*GC* — Vector of length *N* containing the estimated gradient at *XC*. (Output)

### Optional Arguments

*N* — Dimension of the problem. (Input)

Default: `N = SIZE (XC,1)`.

*XSCALE* — Vector of length *N* containing the diagonal scaling matrix for the variables. (Input)

In the absence of other information, set all entries to 1.0.

Default: `XSCALE = 1.0`.

*EPSFCN* — Estimate of the relative noise in the function. (Input)

*EPSFCN* must be less than or equal to 0.1. In the absence of other information, set *EPSFCN* to 0.0.

Default: `EPSFCN = 0.0`.

### FORTRAN 90 Interface

Generic: `CALL FDGRD (FCN, XC, FC, GC [, ...])`

Specific: The specific interface names are `S_FDGRD` and `D_FDGRD`.

### FORTRAN 77 Interface

Single: `CALL FDGRD (FCN, N, XC, XSCALE, FC, EPSFCN GC)`

Double: The double precision name is `DFDGRD`.

### Description

The routine `FDGRD` uses the following finite-difference formula to estimate the gradient of a function of *n* variables at *x*:

$$\frac{f(x + h_i e_i) - f(x)}{h_i} \quad \text{for } i = 1, \dots, n$$

where  $h_i = \epsilon^{1/2} \max\{|x_i|, 1/s_i\} \text{sign}(x_i)$ ,  $\epsilon$  is the machine epsilon,  $e_i$  is the  $i$ -th unit vector, and  $s_i$  is the scaling factor of the  $i$ -th variable. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended. When accuracy of the gradient is important, IMSL routine [CDGRD](#) should be used.

## Comments

This is Description A5.6.3, Dennis and Schnabel, 1983, page 322.

## Example

In this example, the gradient of  $f(x) = x_1 - x_1 x_2 - 2$  is estimated by the finite-difference method at the point (1.0, 1.0).

```

USE FDGRD_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER I, N, NOUT
PARAMETER (N=2)
REAL EPSFCN, FC, GC(N), XC(N)
EXTERNAL FCN
! Initialization.
DATA XC/2*1.0E0/
! Set function noise.
EPSFCN = 0.01
! Get function value at current
! point.
CALL FCN (N, XC, FC)
!
CALL FDGRD (FCN, XC, FC, GC, EPSFCN=EPSFCN)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (GC(I),I=1,N)
99999 FORMAT (' The gradient is', 2F8.2, /)
!
END
!
SUBROUTINE FCN (N, X, F)
INTEGER N
REAL X(N), F
!
F = X(1) - X(1)*X(2) - 2.0E0
!
RETURN
END

```

## Output

The gradient is      0.00    -1.00

---

# FDHES

Approximates the Hessian using forward differences and function values.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function to be minimized. The usage is

CALL FCN (N, X, F), where

**N** — Length of **X**. (Input)

**X** — The point at which the function is evaluated. (Input)  
**X** should not be changed by **FCN**.

**F** — The computed function value at the point **X**. (Output)

**FCN** must be declared **EXTERNAL** in the calling program.

**XC** — Vector of length **N** containing the point at which the Hessian is to be approximated. (Input)

**FC** — Function value at **XC**. (Input)

**H** — **N** by **N** matrix containing the finite difference approximation to the Hessian in the lower triangle. (Output)

## Optional Arguments

**N** — Dimension of the problem. (Input)  
Default: **N** = **SIZE** (**XC**,1).

**XSCALE** — Vector of length **N** containing the diagonal scaling matrix for the variables. (Input)  
In the absence of other information, set all entries to 1.0.  
Default: **XSCALE** = 1.0.

**EPSFCN** — Estimate of the relative noise in the function. (Input)  
**EPSFCN** must be less than or equal to 0.1. In the absence of other information, set **EPSFCN** to 0.0.  
Default: **EPSFCN** = 0.0.

**LDH** — Row dimension of **H** exactly as specified in the dimension statement of the calling program. (Input)  
Default: **LDH** = **SIZE** (**H**,1).

## FORTRAN 90 Interface

Generic:     CALL FDHES (FCN, XC, FC, H [, ...])

Specific:    The specific interface names are S\_FDHEES and D\_FDHEES.

## FORTRAN 77 Interface

Single:     CALL FDHES (FCN, N, XC, XSCALE, FC, EPSFCN, H, LDH)

Double:     The double precision name is DFDHEES.

## Description

The routine FDHES uses the following finite-difference formula to estimate the Hessian matrix of function  $f$  at  $x$ :

$$\frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

Where

$$h_i = \varepsilon^{1/3} \max\{|x_i|, 1/s_i\} \operatorname{sign}(x_i), h_j = \varepsilon^{1/3} \max\{|x_j|, 1/s_j\} \operatorname{sign}(x_j),$$

$\varepsilon$  is the machine epsilon or user-supplied estimate of the relative noise,  $s_i$  and  $s_j$  are the scaling factors of the  $i$ -th and  $j$ -th variables, and  $e_i$  and  $e_j$  are the  $i$ -th and  $j$ -th unit vectors, respectively. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of F2HES/DF2HES. The reference is:

```
CALL F2HES (FCN, N, XC, XSCALE, FC, EPSFCN, H, LDH, WK1,
 WK2)
```

The additional arguments are as follows:

**WK1** — Real work vector of length N.

**WK2** — Real work vector of length N.

2.    This is Description A5.6.2 from Dennis and Schnabel, 1983; page 321.

## Example

The Hessian is estimated for the following function at (1, -1)

$$f(x) = x_1^2 - x_1x_2 - 2$$

```
USE FDHES_INT
USE UMACH_INT

IMPLICIT NONE
! Declaration of variables
INTEGER N, LDHES, NOUT
PARAMETER (N=2, LDHES=2)
REAL XC(N), FVALUE, HES(LDHES,N), EPSFCN
EXTERNAL FCN

! Initialization
DATA XC/1.0E0,-1.0E0/
! Set function noise
EPSFCN = 0.001
! Evaluate the function at
! current point
CALL FCN (N, XC, FVALUE)
! Get Hessian forward difference
! approximation
CALL FDHES (FCN, XC, FVALUE, HES, EPSFCN=EPSFCN)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) ((HES(I,J),J=1,I),I=1,N)
99999 FORMAT (' The lower triangle of the Hessian is', /,&
5X,F10.2,/,5X,2F10.2,/)
!
END
!
SUBROUTINE FCN (N, X, F)
! SPECIFICATIONS FOR ARGUMENTS
INTEGER N
REAL X(N), F
!
F = X(1)*X(1) - X(2) - 2.0E0
!
RETURN
END
```

## Output

```
The lower triangle of the Hessian is
 2.00
-1.00 0.00
```

---

# GDHES

Approximates the Hessian using forward differences and a user-supplied gradient.

## Required Arguments

**GRAD** — User-supplied subroutine to compute the gradient at the point  $x$ . The usage is

CALL GRAD (N, X, G) , where

$N$  — Length of  $X$  and  $G$ . (Input)

$X$  — The point at which the gradient is evaluated. (Input)  
 $X$  should not be changed by GRAD.

$G$  — The gradient evaluated at the point  $x$ . (Output)

GRAD must be declared EXTERNAL in the calling program.

**XC** — Vector of length  $N$  containing the point at which the Hessian is to be estimated.  
(Input)

**GC** — Vector of length  $N$  containing the gradient of the function at  $x_C$ . (Input)

**H** —  $N$  by  $N$  matrix containing the finite-difference approximation to the Hessian in the lower triangular part and diagonal. (Output)

## Optional Arguments

$N$  — Dimension of the problem. (Input)

Default:  $N = \text{SIZE}(XC,1)$ .

**XSCALE** — Vector of length  $N$  containing the diagonal scaling matrix for the variables.  
(Input)

In the absence of other information, set all entries to 1.0.

Default:  $XSCALE = 1.0$ .

**EPSFCN** — Estimate of the relative noise in the function. (Input)

EPSFCN must be less than or equal to 0.1. In the absence of other information, set  
EPSFCN to 0.0.

Default:  $EPSFCN = 0.0$ .

**LDH** — Leading dimension of  $H$  exactly as specified in the dimension statement of the calling  
program. (Input)

Default:  $LDH = \text{SIZE}(H,1)$ .

## FORTRAN 90 Interface

Generic: CALL GDHES (GRAD, XC, GC, H [, ...])

Specific: The specific interface names are S\_GDHES and D\_GDHES.

## FORTRAN 77 Interface

Single:      CALL GDHES (GRAD, N, XC, XSCALE, GC, EPSFCN, H, LDH)

Double:     The double precision name is DGDHES.

## Description

The routine GDHES uses the following finite-difference formula to estimate the Hessian matrix of function  $F$  at  $x$ :

$$\frac{g(x + h_j e_j) - g(x)}{h_j}$$

where

$$h_j = \varepsilon^{1/3} \max\{|x_j|, 1/s_j\} \text{sign}(x_j),$$

$\varepsilon$  is the machine epsilon,  $s_j$  is the scaling factor of the  $j$ -th variable,  $g$  is the analytic gradient of  $F$  at  $x$ , and  $e_j$  is the  $j$ -th unit vector. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

## Comments

1.    Workspace may be explicitly provided, if desired, by use of G2HES/DG2HES. The reference is:

```
CALL G2HES (GRAD, N, XC, XSCALE, GC, EPSFCN, H, LDH, WK)
```

The additional argument is

**WK** — Work vector of length N.

2.    This is Description A5.6.1, Dennis and Schnabel, 1983; page 320.

## Example

The Hessian is estimated by the finite-difference method at point (1.0, 1.0) from the following gradient functions:

$$g_1 = 2x_1x_2 - 2$$

$$g_2 = x_1x_1 + 1$$

```
USE GDHES_INT
USE UMACH_INT
```

```

 IMPLICIT NONE
! Declaration of variables
 INTEGER N, LDHES, NOUT
 PARAMETER (N=2, LDHES=2)
 REAL XC(N), GC(N), HES(LDHES,N)
 EXTERNAL GRAD
!
 DATA XC/2*1.0E0/
! Set function noise
! Evaluate the gradient at the
! current point
 CALL GRAD (N, XC, GC)
! Get Hessian forward-difference
! approximation
 CALL GDHES (GRAD, XC, GC, HES)
!
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) ((HES(I,J),J=1,N),I=1,N)
99999 FORMAT (' THE HESSIAN IS', /, 2(5X,2F10.2,/,/))
!
 END
!
 SUBROUTINE GRAD (N, X, G)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER N
 REAL X(N), G(N)
!
 G(1) = 2.0E0*X(1)*X(2) - 2.0E0
 G(2) = X(1)*X(1) + 1.0E0
!
 RETURN
 END

```

## Output

```

THE HESSIAN IS
2.00 2.00
2.00 0.00

```

---

## DDJAC

Approximates the Jacobian of  $m$  functions in  $n$  unknowns using divided differences.

### Required Arguments

*FCN* — User-supplied subroutine to evaluate functions. The usage is  
 CALL FCN (INDX, Y, F[, ...]) where

#### Required Arguments

INDX — Index of the variable whose derivative is to be computed. (Input)  
 DDJAC will set this argument to the index of the variable whose derivative  
 is being computed. In those cases where there is a mix of finite

differencing taking place along with additional analytic terms being computed, (see `METHOD = 2`), `DDJAC` will make two calls to `FCN` each time a new function evaluation is needed, once with `INDX` positive and a second time with `INDX` negative.

**Y** — Array containing the point at which the function is to be computed. (Input)

**F** — Array of length *M*, where *M* is the number of functions to be evaluated at point *Y*, containing the function values of the equations at point *Y*. (Output)

Normally, the user will return the values of the functions evaluated at point *Y* in *F*. However, when the function can be broken into two parts, a part which is known analytically and a part to be differenced, `FCN` will be called by `DDJAC` once with `INDX` positive for the portion to be differenced and again with `INDX` negative for the portion which is known analytically. In the case where `METHOD=2` has been chosen, `FCN` must be written to handle the known analytic portion separate from the part to be differenced. (See [Example 4](#) for an example where `METHOD=2` is used.)

### Optional Arguments

`FCN_DATA` — A derived type, `s_fcn_data`, which may be used to pass additional integer or floating point information to or from the user-supplied subroutine. For a detailed description of this argument see `FCN_DATA` below. (Input/Output)

`FCN` must be declared `EXTERNAL` in the calling program.

**Y** — Array of length *N* containing the point at which the Jacobian is to be evaluated. (Input)

**F** — Array of length *M* containing the function values of the equations at point *Y*. (Output)

**FJAC** — Two dimensional array of which the first *M* by *N* subarray contains the estimated Jacobian. (Input/Output)

On input the user may set entries of columns that are to be accumulated to initial values (See the optional argument `METHOD`). On final output, `FJAC` will contain the estimated Jacobian.

### Optional Arguments

**M** — The number of equations. (Input)  
Default: `M = SIZE (F)`.

**N** — The number of variables. (Input)  
Default: `N = SIZE (Y)`.

**YSCALE** — Array of length *N* containing the diagonal scaling matrix for the variables. (Input)  
`YSCALE` can also be used to provide appropriate signs for the increments.  
Default: `YSCALE = 1.0`.

**METHOD** — Array of length  $N$  containing the methods used to compute the derivatives. (Input)

$METHOD(i)$  is the method to be used for the  $i$ -th variable.  $METHOD(i)$  can be one of the values in the following table:

| Value | Description                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | Indicates one-sided differences.                                                                                                              |
| 1     | Indicates central differences.                                                                                                                |
| 2     | Indicates the accumulation of the result from whatever type of differences have been specified previously into initial values of the Jacobian |
| 3     | Indicates a variable is to be skipped                                                                                                         |

Default: One-sided differences are used for all variables.

**FACTOR** — Array of length  $N$  containing the percentage factor for differencing. (Input)

For each divided difference for variable  $j$  the increment used is  $del$ . The value of  $del$  is computed as follows: First define  $\sigma = \text{sign}(YSCALE(j))$ . If the user has set the elements of array  $YSCALE$  to non-default values, then define  $y_a = /YSCALE(j)/$ . Otherwise,  $y_a = /Y(j)/$  and  $\sigma = 1$ . Finally, compute  $del = \sigma y_a \text{FACTOR}(j)$ . By changing the sign of  $YSCALE(j)$ , the difference  $del$  can have any desired orientation, such as staying within bounds on variable  $j$ . For central differences, a reduced factor is used for  $del$  that normally results in relative errors as small as machine precision to the  $2/3$  power. The elements of  $FACTOR$  must be such that machine precision to the  $3/4$  power  $\leq \text{FACTOR}(j) \leq 0.1$

Default: All elements of  $FACTOR$  are set to  $\text{sqrt}(\text{machine precision})$ .

**ISTATUS** — Array of length 10 which contains status information that might prove useful to the user wanting to gain better control over the differencing parameters. (Output)

This information can often be ignored. The following table describes the diagnostic information which is returned in each of the entries of  $ISTATUS$ :

| <i>index</i> | Description                                                                                                                                                                                                                                                                                                                  |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1            | The number of times a function evaluation was computed.                                                                                                                                                                                                                                                                      |
| 2            | The number of columns in which three attempts were made to increase a percentage factor for differencing (i.e. a component in the $FACTOR$ array) but the computed $del$ remained unacceptably small relative to $Y[j]$ or $YSCALE[j]$ . In such cases the percentage factor is set to the square root of machine precision. |
| 3            | The number of columns in which the computed $del$ was zero to machine precision because $Y[j]$ or $YSCALE[j]$ was zero. In such cases $del$ is set to the square root of machine precision.                                                                                                                                  |

| <i>index</i> | Description                                                                                                                                                                                                                                                                                                                                                                     |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4            | The number of Jacobian columns which had to be recomputed because the largest difference formed in the column was close to zero relative to scale, where $scale = \max( f_i(y) ,  f_i(y + del \times e_j) )$ and <i>i</i> denotes the row index of the largest difference in the column currently being processed. <i>index</i> = 10 gives the last column where this occurred. |
| 5            | The number of columns whose largest difference is close to zero relative to scale after the column has been recomputed.                                                                                                                                                                                                                                                         |
| 6            | The number of times scale information was not available for use in the roundoff and truncation error tests. This occurs when $\min( f_i(y) ,  f_i(y + del \times e_j) ) = 0$ Where <i>i</i> is the index of the largest difference for the column currently being processed.                                                                                                    |
| 7            | The number of times the increment for differencing ( <i>del</i> ) was computed and had to be increased because $(YSCALE[j] + del) - YSCALE[j]$ was too small relative to $Y[j]$ or $YSCALE[j]$ .                                                                                                                                                                                |
| 8            | The number of times a component of the <code>FACTOR</code> array was reduced because changes in function values were large and excess truncation error was suspected. <i>index</i> = 9 gives the last column in which this occurred.                                                                                                                                            |
| 9            | The index of the last column where the corresponding component of the <code>FACTOR</code> array had to be reduced because excessive truncation error was suspected.                                                                                                                                                                                                             |
| 10           | The index of the last column where the difference was small and the column had to be recomputed with an adjusted increment (see <i>index</i> = 4). The largest derivative in this column may be inaccurate due to excessive roundoff error.                                                                                                                                     |

**FCN\_DATA** — A derived type, `s_fcn_data`, which may be used to pass additional information to/from the user-supplied subroutine. (Input/Output)

The derived type, `s_fcn_data`, is defined as:

```

type s_fcn_data
 real(kind(1e0)), pointer, dimension(:) :: rdata
 integer, pointer, dimension(:) :: idata
end type

```

in module `mp_types`. The double precision counterpart to `s_fcn_data` is named `d_fcn_data`. The user must include a `use mp_types` statement in the calling program to define this derived type.

## FORTRAN 90 Interface

Generic:     CALL DDJAC (FCN, Y, F, FJAC [, ...])

Specific:    The specific interface names are S\_DDJAC and D\_DDJAC.

## Description

Computes the Jacobian matrix for a function  $f(y)$  with  $m$  components in  $n$  independent variables. DDJAC uses divided finite differences to compute the Jacobian. This subroutine is designed for use in numerical methods for solving nonlinear problems where a Jacobian is evaluated repeatedly at neighboring arguments. For example this occurs in a Gauss-Newton method for solving non-linear least squares problems or a non-linear optimization method.

DDJAC is suited for applications where the Jacobian is a dense matrix. All cases  $m < n$ ,  $m = n$ , or  $m > n$  are allowed. Both one-sided and central divided differences can be used.

The design allows for computation of derivatives in a variety of contexts. Note that a gradient should be considered as the special case with  $m = 1$ ,  $n \geq 1$ . A derivative of a single function of one variable is the case  $m = 1$ ,  $n = 1$ . Any non-linear solving routine that optionally requests a Jacobian or gradient can use DDJAC. This should be considered if there are special properties or scaling issues associated with  $f(y)$ . Use the argument METHOD to specify different differencing options for numerical differentiation. These can be combined with some analytic subexpressions or other known relationships.

The divided differences are computed using values of the independent variables at the initial point  $y_j = y$ , and differenced points  $y_e = y + del \times e_j$ . Here the  $e_j$ ,  $j = 1, \dots, n$ , are the unit coordinate vectors.

The value for each difference  $del$  depends on the variable  $j$ , the differencing method, and the scaling for that variable. This difference is computed internally. See FACTOR for computational details. The evaluation of  $f(y_e)$  is normally done by the user-provided argument FCN, using the values  $y_e$ . The index  $j$ , values  $y_e$ , and output F are arguments to FCN.

The computational kernel of DDJAC performs the following steps:

1. Evaluates the equations at the point Y using FCN.
2. Computes the Jacobian.
3. Computes the difference at  $y_e$ .

There are four examples provided which illustrate various ways to use DDJAC. A discussion of the expected errors for the difference methods is found in *A First Course in Numerical Analysis*, Anthony Ralston, McGraw-Hill, NY, (1965).

## Example 1

In this example, the Jacobian matrix of

$$\begin{aligned}f_1(x) &= x_1 x_2 - 2 \\f_2(x) &= x_1 - x_1 x_2 + 1\end{aligned}$$

is estimated by the finite-difference method at the point (1.0, 1.0).

```

USE DDJAC_INT
USE WRRRN_INT
IMPLICIT NONE

INTEGER, PARAMETER :: N=2, M=2
REAL FJAC(M,N), Y(N), F(M)
EXTERNAL FCN

DATA Y/2*1.0/
! Get Jacobian one-sided difference
! approximation
CALL DDJAC (FCN, Y, F, FJAC)
CALL WRRRN ("The Jacobian is:", FJAC)
END

SUBROUTINE FCN (INDX, Y, F)
INTEGER INDX
REAL Y(*), F(*)

F(1) = Y(1)*Y(2) - 2.0
F(2) = Y(1) - Y(1)*Y(2) + 1.0

RETURN
END

```

## Output

```

The Jacobian is:
 1 2
1 1.000 1.000
2 0.000 -1.000

```

## Example 2

A simple use of DDJAC is shown. The gradient of the function  $f(y_1, y_2) = a \exp(by_1) + cy_1y_2^2$ .

is required for values  $a = 2.5e6$ ,  $b = 3.4$ ,  $c = 4.5$ ,  $y_1 = 2.1$ ,  $y_2 = 3.2$ .

The analytic gradient of this function is:

$$\text{grad}(f) = [a \exp(by_1) + cy_2^2, 2cy_1cy_2]$$

```

USE DDJAC_INT
USE UMACH_INT
IMPLICIT NONE

INTEGER, PARAMETER :: N=2, M=1
INTEGER J, NOUT
REAL FJAC(M,N), Y(N), F(M), SCALE(N)
EXTERNAL FCN

```

```

DATA Y/2.1, 3.2/ SCALE/1.0, 8000.0/
!
! Get Gradient one-sided difference
! approximation
CALL DDJAC (FCN, Y, F, FJAC, YSCALE=SCALE)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (FJAC(1,J),J=1,N)
99999 FORMAT (' The Numerical Gradient is (' , 2e15.4,')')
END

SUBROUTINE FCN (INDX, Y, F)
INTEGER INDX
REAL A, B, C, Y(*), F(*)

A = 2500000.
B = 3.4
C = 4.5

F(1) = A * EXP (B * Y(1)) + C * Y(1) * Y(2) * Y(2)

RETURN
END

```

## Output

The Numerical Gradient is ( 0.1073E+11 0.9268E+02 )

## Example 3

This example uses the same data as in [Example 2](#). Here we assume that the second component of the gradient is analytically known. Therefore only the first gradient component needs numerical approximation. The input values of array METHOD specify that numerical differentiation with respect to  $y_2$  is skipped.

```

USE DDJAC_INT
USE UMACH_INT
IMPLICIT NONE

INTEGER, PARAMETER :: N=2, M=1
INTEGER J, NOUT, METHOD(2)
REAL FJAC(M,N), Y(N), F(M), SCALE(N)
EXTERNAL FCN

DATA Y/2.1, 3.2/ SCALE/1.0, 8000.0/
!
! Initialize second component
! of Jacobian since it is
! known analytically and can be
! skipped
FJAC(1,2) = 2.0 * 4.5 * Y(1) * Y(2)
!
! Set METHOD to skip the second
! component
METHOD(1) = 0

```

```

METHOD(2) = 3
! Get Gradient approximation

CALL DDJAC (FCN, Y, F, FJAC, YSCALE=SCALE, METHOD=METHOD)
! Print results

CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (FJAC(1,J),J=1,N)
99999 FORMAT (' The Numerical Gradient is (' , 2e15.4, ')')
END

SUBROUTINE FCN (INDX, Y, F)
INTEGER INDX
REAL A, B, C, Y(*), F(*)

A = 2500000.
B = 3.4
C = 4.5

F(1) = A * EXP (B * Y(1)) + C * Y(1) * Y(2) * Y(2)

RETURN
END

```

## Output

```
The Numerical Gradient is (0.1073E+11 0.6048E+02)
```

## Example 4

This example uses the same data as in [Example 2](#). An alternate examination of the function

$$f(y_1, y_2) = a \exp(by_1) + cy_1 y_2^2$$

shows that the first term on the right-hand side need be evaluated just when computing the first partial. The additive term  $cy_1 y_2^2$  occurs when computing the partial with respect to  $y_1$ . Also the first term does not depend on the second variable. Thus the first term can be left out of the function evaluation when computing the partial with respect to  $y_2$ , potentially avoiding cancellation errors. The input values of array METHOD allow DDJAC to use these facts and obtain greater accuracy using a minimum number of computations of the exponential function

```

USE DDJAC_INT
USE UMACH_INT
USE MP_TYPES
IMPLICIT NONE

INTEGER, PARAMETER :: N=2, M=1
INTEGER J, NOUT, METHOD(2)
REAL FJAC(M,N), Y(N), F(M), SCALE(N)
REAL, TARGET :: RDATA(3)
TYPE(S_FCN_DATA) USER_DATA
EXTERNAL FCN

DATA Y/2.1, 3.2/ SCALE/1.0, 8000.0/

```

```

! Set up to pass some extra
! information to the function
RDATA(1) = 2500000.0
RDATA(2) = 3.4
RDATA(3) = 4.5
USER_DATA%RDATA => RDATA
! Initialize first component
! of function since it is
! known
FJAC(1,1) = 4.5 * Y(2) * Y(2)
! Set METHOD to accumulate for
! part of the first partial,
! one-sided differences for
! the second
METHOD(1) = 2
METHOD(2) = 0
! Get Gradient approximation
CALL DDJAC (FCN, Y, F, FJAC, YSCALE=SCALE, METHOD=METHOD, &
 FCN_DATA=USER_DATA)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) (FJAC(1,J),J=1,N)
99999 FORMAT (' The Numerical Gradient is (' , 2e15.4, ')')
END

SUBROUTINE FCN (INDX, Y, F, FCN_DATA)
USE MP_TYPES
IMPLICIT NONE

INTEGER INDX
REAL A, B, C, Y(*), F(*)
TYPE(S_FCN_DATA) FCN_DATA

A = FCN_DATA%RDATA(1)
B = FCN_DATA%RDATA(2)
C = FCN_DATA%RDATA(3)
! Handle both the differenced
! part and the part that is
! known analytically for each
! dependent variable
SELECT CASE (INDX)
CASE (1)
 F(1)=A*EXP(B*Y(1))

CASE (-1)
 F(1)= C*Y(2)**2

CASE (2)
 F(1) = C*Y(1)*Y(2)**2

CASE (-2)
 F(1)=0
END SELECT

```

```
RETURN
END
```

## Output

The Numerical Gradient is ( 0.1073E+11 0.6046E+02 )

---

# FDJAC

Approximates the Jacobian of  $M$  functions in  $N$  unknowns using forward differences.

## Required Arguments

*FCN* — User-supplied subroutine to evaluate the function to be minimized. The usage is  
CALL FCN (M, N, X, F), where

*M* — Length of *F*. (Input)

*N* — Length of *X*. (Input)

*X* — The point at which the function is evaluated. (Input)  
*X* should not be changed by *FCN*.

*F* — The computed function at the point *X*. (Output)

*FCN* must be declared EXTERNAL in the calling program.

*XC* — Vector of length *N* containing the point at which the gradient is to be estimated.  
(Input)

*FC* — Vector of length *M* containing the function values at *XC*. (Input)

*FJAC* —  $M$  by  $N$  matrix containing the estimated Jacobian at *XC*. (Output)

## Optional Arguments

*M* — The number of functions. (Input)  
Default:  $M = \text{SIZE}(FC,1)$ .

*N* — The number of variables. (Input)  
Default:  $N = \text{SIZE}(XC,1)$ .

*XSCALE* — Vector of length *N* containing the diagonal scaling matrix for the variables.  
(Input)  
In the absence of other information, set all entries to 1.0.  
Default:  $XSCALE = 1.0$ .

*EPSFCN* — Estimate for the relative noise in the function. (Input)  
*EPSFCN* must be less than or equal to 0.1. In the absence of other information, set

EPSFCN to 0.0.  
Default: EPSFCN = 0.0.

**LDFJAC** — Leading dimension of FJAC exactly as specified in the dimension statement of the calling program. (Input)  
Default: LDFJAC = SIZE (FJAC,1).

### **FORTRAN 90 Interface**

Generic: CALL FDJAC (FCN, XC, FC, FJAC [, ...])

Specific: The specific interface names are S\_FDJAC and D\_FDJAC.

### **FORTRAN 77 Interface**

Single: CALL FDJAC (FCN, M, N, XC, XSCALE, FC, EPSFCN, FJAC, LDFJAC)

Double: The double precision name is DFDJAC.

### **Description**

The routine FDJAC uses the following finite-difference formula to estimate the Jacobian matrix of function  $f$  at  $x$ :

$$\frac{f(x + h_j e_j) - f(x)}{h_j}$$

where  $e_j$  is the  $j$ -th unit vector,  $h_j = \epsilon^{1/2} \max\{|x_j|, 1/s_j\} \text{sign}(x_j)$ ,  $\epsilon$  is the machine epsilon, and  $s_j$  is the scaling factor of the  $j$ -th variable. For more details, see Dennis and Schnabel (1983).

Since the finite-difference method has truncation error, cancellation error, and rounding error, users should be aware of possible poor performance. When possible, high precision arithmetic is recommended.

### **Comments**

1. Workspace may be explicitly provided, if desired, by use of F2JAC/DF2JAC. The reference is:

```
CALL F2JAC (FCN, M, N, XC, XSCALE, FC, EPSFCN, FJAC,
LDFJAC, WK)
```

The additional argument is:

**WK** — Work vector of length M.

2. This is Description A5.4.1, Dennis and Schnabel, 1983, page 314.

## Example

In this example, the Jacobian matrix of

$$f_1(x) = x_1x_2 - 2$$

$$f_2(x) = x_1 - x_1x_2 + 1$$

is estimated by the finite-difference method at the point (1.0, 1.0).

```
 USE FDJAC_INT
 USE UMACH_INT

 IMPLICIT NONE
! Declaration of variables
 INTEGER N, M, LDFJAC, NOUT
 PARAMETER (N=2, M=2, LDFJAC=2)
 REAL FJAC(LDFJAC,N), XC(N), FC(M), EPSFCN
 EXTERNAL FCN
!
 DATA XC/2*1.0E0/
! Set function noise
 EPSFCN = 0.01
! Evaluate the function at the
! current point
 CALL FCN (M, N, XC, FC)
! Get Jacobian forward-difference
! approximation
 CALL FDJAC (FCN, XC, FC, FJAC, EPSFCN=EPFSCN)
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99999) ((FJAC(I,J),J=1,N),I=1,M)
99999 FORMAT (' The Jacobian is', /, 2(5X,2F10.2,/,/))
!
 END
!
 SUBROUTINE FCN (M, N, X, F)
! SPECIFICATIONS FOR ARGUMENTS
 INTEGER M, N
 REAL X(N), F(M)
!
 F(1) = X(1)*X(2) - 2.0E0
 F(2) = X(1) - X(1)*X(2) + 1.0E0
!
 RETURN
 END
```

## Output

```
The Jacobian is
1.00 1.00
0.00 -1.00
```

---

# CHGRD

Checks a user-supplied gradient of a function.

## Required Arguments

**FCN** — User-supplied subroutine to evaluate the function of which the gradient will be checked. The usage is

CALL FCN (N, X, F), where

N — Length of X. (Input)

X — The point at which the function is evaluated. (Input)  
X should not be changed by FCN.

F — The computed function value at the point X. (Output)

FCN must be declared EXTERNAL in the calling program.

**GRAD** — Vector of length N containing the estimated gradient at X. (Input)

**X** — Vector of length N containing the point at which the gradient is to be checked. (Input)

**INFO** — Integer vector of length N. (Output)

INFO(I) = 0 means the user-supplied gradient is a poor estimate of the numerical gradient at the point X(I).

INFO(I) = 1 means the user-supplied gradient is a good estimate of the numerical gradient at the point X(I).

INFO(I) = 2 means the user-supplied gradient disagrees with the numerical gradient at the point X(I), but it might be impossible to calculate the numerical gradient.

INFO(I) = 3 means the user-supplied gradient and the numerical gradient are both zero at X(I), and, therefore, the gradient should be rechecked at a different point.

## Optional Arguments

N — Dimension of the problem. (Input)

Default: N = SIZE (X,1).

## FORTRAN 90 Interface

Generic: CALL CHGRD (FCN, GRAD, X, INFO [, ...])

Specific: The specific interface names are S\_CHGRD and D\_CHGRD.

## FORTRAN 77 Interface

Single:      CALL CHGRD (FCN, GRAD, N, X, INFO)

Double:     The double precision name is DCHGRD.

## Description

The routine CHGRD uses the following finite-difference formula to estimate the gradient of a function of  $n$  variables at  $x$ :

$$g_i(x) = \frac{f(x + h_i e_i) - f(x)}{h_i} \quad \text{for } i=1, \dots, n$$

where  $h_i = \varepsilon^{1/2} \max\{|x_i|, 1/s_i\} \text{ sign}(x_i)$ ,  $\varepsilon$  is the machine epsilon,  $e_i$  is the  $i$ -th unit vector, and  $s_i$  is the scaling factor of the  $i$ -th variable.

The routine CHGRD checks the user-supplied gradient  $\nabla f(x)$  by comparing it with the finite-difference gradient  $g(x)$ . If

$$\left| g_i(x) - (\nabla f(x))_i \right| < \tau \left| (\nabla f(x))_i \right|$$

where  $\tau = \varepsilon^{1/4}$ , then  $(\nabla f(x))_i$ , which is the  $i$ -th element of  $\nabla f(x)$ , is declared correct; otherwise, CHGRD computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference,  $(\nabla f(x))_i$  is reported as incorrect. In the case of a large error bound, CHGRD uses a nearly optimal stepsize to recompute  $g_i(x)$  and reports that  $(\nabla f(x))_i$  is correct if

$$\left| g_i(x) - (\nabla f(x))_i \right| < 2\tau \left| (\nabla f(x))_i \right|$$

Otherwise,  $(\nabla f(x))_i$  is considered incorrect unless the error bound for the optimal step is greater than  $\tau \left| (\nabla f(x))_i \right|$ . In this case, the numeric gradient may be impossible to compute correctly. For more details, see Schnabel (1985).

## Comments

1.      Workspace may be explicitly provided, if desired, by use of C2GRD/DC2GRD. The reference is:

```
CALL C2GRD (FCN, GRAD, N, X, INFO, FX, XSCALE, EPSFCN,
 XNEW)
```

The additional arguments are as follows:

**FX** — The functional value at  $x$ .

**XSCALE** — Real vector of length  $N$  containing the diagonal scaling matrix.

**EPSFCN** — The relative “noise” of the function FCN.

**XNEW** — Real work vector of length N.

2. Informational errors

Type Code

4 1 The user-supplied gradient is a poor estimate of the numerical gradient.

**Example**

The user-supplied gradient of

$$f(x) = x_i + x_2 e^{-(t-x_3)2/x_4}$$

at (625, 1, 3.125, 0.25) is checked where  $t = 2.125$ .

```
USE CHGRD_INT
USE WRIRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER N
PARAMETER (N=4)
!
INTEGER INFO(N)
REAL GRAD(N), X(N)
EXTERNAL DRIV, FCN
!
! Input values for point X
! X = (625.0, 1.0, 3.125, .25)
!
DATA X/625.0E0, 1.0E0, 3.125E0, 0.25E0/
!
CALL DRIV (N, X, GRAD)
!
CALL CHGRD (FCN, GRAD, X, INFO)
CALL WRIRN ('The information vector', INFO, 1, N, 1)
!
END
!
SUBROUTINE FCN (N, X, FX)
INTEGER N
REAL X(N), FX
!
REAL EXP
INTRINSIC EXP
!
FX = X(1) + X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))
RETURN
END
!
```

```

SUBROUTINE DRIV (N, X, GRAD)
INTEGER N
REAL X(N), GRAD(N)
!
REAL EXP
INTRINSIC EXP
!
GRAD(1) = 1.0E0
GRAD(2) = EXP(-1.0E0*(2.125E0-X(3))**2/X(4))
GRAD(3) = X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))*2.0E0/X(4)* &
(2.125-X(3))
GRAD(4) = X(2)*EXP(-1.0E0*(2.125E0-X(3))**2/X(4))* &
(2.125E0-X(3))**2/(X(4)*X(4))
RETURN
END

```

## Output

```

The information vector
1 2 3 4
1 1 1 1

```

---

# CHHES

Checks a user-supplied Hessian of an analytic function.

## Required Arguments

**GRAD** — User-supplied subroutine to compute the gradient at the point  $x$ . The usage is  
 CALL GRAD (N, X, G), where

$N$  — Length of  $x$  and  $G$ . (Input)

$x$  — The point at which the gradient is evaluated.  $x$  should not be changed by  
 GRAD. (Input)

$G$  — The gradient evaluated at the point  $x$ . (Output)

GRAD must be declared EXTERNAL in the calling program.

**HESS** — User-supplied subroutine to compute the Hessian at the point  $x$ . The usage is  
 CALL HESS (N, X, H, LDH), where

$N$  — Length of  $X$ . (Input)

$x$  — The point at which the Hessian is evaluated. (Input)  
 $X$  should not be changed by HESS.

$H$  — The Hessian evaluated at the point  $x$ . (Output)

$LDH$  — Leading dimension of  $H$  exactly as specified in the dimension  
 statement of the calling program. (Input)

HESS must be declared EXTERNAL in the calling program.

**X** — Vector of length  $N$  containing the point at which the Hessian is to be checked. (Input)

**INFO** — Integer matrix of dimension  $N$  by  $N$ . (Output)

$\text{INFO}(I, J) = 0$  means the Hessian is a poor estimate for function  $I$  at the point  $X(J)$ .

$\text{INFO}(I, J) = 1$  means the Hessian is a good estimate for function  $I$  at the point  $X(J)$ .

$\text{INFO}(I, J) = 2$  means the Hessian disagrees with the numerical Hessian for function  $I$  at the point  $X(J)$ , but it might be impossible to calculate the numerical Hessian.

$\text{INFO}(I, J) = 3$  means the Hessian for function  $I$  at the point  $X(J)$  and the numerical Hessian are both zero, and, therefore, the gradient should be rechecked at a different point.

### Optional Arguments

**N** — Dimension of the problem. (Input)

Default:  $N = \text{SIZE}(X, 1)$ .

**LDINFO** — Leading dimension of **INFO** exactly as specified in the dimension statement of the calling program. (Input)

Default:  $\text{LDINFO} = \text{SIZE}(\text{INFO}, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL CHHES (GRAD, HESS, X, INFO [, ...])`

Specific: The specific interface names are `S_CHHES` and `D_CHHES`.

### FORTRAN 77 Interface

Single: `CALL CHHES (GRAD, HESS, N, X, INFO, LDINFO)`

Double: The double precision name is `DCHHES`.

### Description

The routine `CHHES` uses the following finite-difference formula to estimate the Hessian of a function of  $n$  variables at  $x$ :

$$B_{ij}(x) = \left( g_i(x + h_j e_j) - g_i(x) \right) / h_j \quad \text{for } j = 1, \dots, n$$

where

$$h_j = \epsilon^{1/2} \max\{|x_j|, 1/s_j\} \text{sign}(x_j),$$

$\epsilon$  is the machine epsilon,

$e_j$ 

is the  $j$ -th unit vector,

 $s_j$ 

is the scaling factor of the  $j$ -th variable, and

 $g_i(x)$ 

is the gradient of the function with respect to the  $i$ -th variable.

Next, CHHES checks the user-supplied Hessian  $H(x)$  by comparing it with the finite difference approximation  $B(x)$ . If

$$|B_{ij}(x) - H_{ij}(x)| < \tau |H_{ij}(x)|$$

where

$$\tau = \epsilon^{1/4},$$

then

 $H_{ij}(x)$ 

is declared correct; otherwise, CHHES computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference,

 $H_{ij}(x)$ 

is reported as incorrect. In the case of a large error bound, CHHES uses a nearly optimal stepsize to recomputed

 $B_{ij}(x)$ 

and reports that

 $B_{ij}(x)$ 

is correct if

$$|B_{ij}(x) - H_{ij}(x)| < 2\tau |H_{ij}(x)|$$

Otherwise,  $H_{ij}(x)$  is considered incorrect unless the error bound for the optimal step is greater than  $\tau |H_{ij}(x)|$ . In this case, the numeric approximation may be impossible to compute correctly. For more details, see Schnabel (1985).

## Comments

Workspace may be explicitly provided, if desired, by use of C2HES/DC2HES. The reference is

```
CALL C2HES (GRAD, HESS, N, X, INFO, LDINFO, G, HX, HS,
 XSCALE, EPSFCN, INFT, NEWX)
```

The additional arguments are as follows:

**G** — Vector of length  $N$  containing the value of the gradient GRD at  $X$ .

**HX** — Real matrix of dimension  $N$  by  $N$  containing the Hessian evaluated at  $X$ .

**HS** — Real work vector of length N.

**XSCALE** — Vector of length N used to store the diagonal scaling matrix for the variables.

**EPSFCN** — Estimate of the relative noise in the function.

**INFT** — Vector of length N. For I = 1 through N, INFT contains information about the Jacobian.

**NEWX** — Real work array of length N.

## Example

The user-supplied Hessian of

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

at (-1.2, 1.0) is checked, and the error is found.

```
USE CHHES_INT

IMPLICIT NONE
INTEGER LDINFO, N
PARAMETER (N=2, LDINFO=N)
!
INTEGER INFO(LDINFO,N)
REAL X(N)
EXTERNAL GRD, HES
!
! Input values for X
! X = (-1.2, 1.0)
!
DATA X/-1.2, 1.0/
!
CALL CHHES (GRD, HES, X, INFO)
!
END
!
SUBROUTINE GRD (N, X, UG)
INTEGER N
REAL X(N), UG(N)
!
UG(1) = -400.0*X(1)*(X(2)-X(1)*X(1)) + 2.0*X(1) - 2.0
UG(2) = 200.0*X(2) - 200.0*X(1)*X(1)
RETURN
END
!
SUBROUTINE HES (N, X, HX, LDHS)
INTEGER N, LDHS
REAL X(N), HX(LDHS,N)
!
```



INFO(I, J) = 0 means the user-supplied Jacobian is a poor estimate for function I at the point X(J).

INFO(I, J) = 1 means the user-supplied Jacobian is a good estimate for function I at the point X(J).

INFO(I, J) = 2 means the user-supplied Jacobian disagrees with the numerical Jacobian for function I at the point X(J), but it might be impossible to calculate the numerical Jacobian.

INFO(I, J) = 3 means the user-supplied Jacobian for function I at the point X(J) and the numerical Jacobian are both zero. Therefore, the gradient should be rechecked at a different point.

### Optional Arguments

*M* — The number of functions in the system of equations. (Input)  
Default: `M = SIZE (INFO,1)`.

*N* — The number of unknowns in the system of equations. (Input)  
Default: `N = SIZE (X,1)`.

*LDINFO* — Leading dimension of INFO exactly as specified in the dimension statement of the calling program. (Input)  
Default: `LDINFO = SIZE (INFO,1)`.

### FORTRAN 90 Interface

Generic:     `CALL CHJAC (FCN, JAC, X, INFO [, ...])`

Specific:    The specific interface names are `S_CHJAC` and `D_CHJAC`.

### FORTRAN 77 Interface

Single:     `CALL CHJAC (FCN, JAC, M, N, X, INFO, LDINFO)`

Double:     The double precision name is `DCHJAC`.

### Description

The routine CHJAC uses the following finite-difference formula to estimate the gradient of the *i*-th function of *n* variables at *x*:

$$g_{ij}(x) = (f_i(x + h_j e_j) - f_i(x))/h_j \quad \text{for } j = 1, \dots, n$$

where  $h_j = \epsilon^{1/4} \max\{|x_j|, 1/s_j\} \text{ sign}(x_j)$ ,  $\epsilon$  is the machine epsilon,  $e_j$  is the *j*-th unit vector, and  $s_j$  is the scaling factor of the *j*-th variable.

Next, CHJAC checks the user-supplied Jacobian  $J(x)$  by comparing it with the finite difference gradient  $g_i(x)$ . If

$$|g_{ij}(x) - J_{ij}(x)| < \tau |J_{ij}(x)|$$

where  $\tau = \epsilon^{1/4}$ , then  $J_{ij}(x)$  is declared correct; otherwise, CHJAC computes the bounds of calculation error and approximation error. When both bounds are too small to account for the difference,  $J_{ij}(x)$  is reported as incorrect. In the case of a large error bound, CHJAC uses a nearly optimal stepsize to recompute  $g_{ij}(x)$  and reports that  $J_{ij}(x)$  is correct if

$$|g_{ij}(x) - J_{ij}(x)| < 2\tau |J_{ij}(x)|$$

Otherwise,  $J_{ij}(x)$  is considered incorrect unless the error bound for the optimal step is greater than  $\tau |J_{ij}(x)|$ . In this case, the numeric gradient may be impossible to compute correctly. For more details, see Schnabel (1985).

## Comments

1. Workspace may be explicitly provided, if desired, by use of C2JAC/DC2JAC. The reference is:

```
CALL C2JAC (FCN, JAC, N, X, INFO, LDINFO, FX, FJAC, GRAD,
 XSCALE, EPSFCN, INFT, NEWX)
```

The additional arguments are as follows:

**FX** — Vector of length  $M$  containing the value of each function in **FCN** at  $X$ .

**FJAC** — Real matrix of dimension  $M$  by  $N$  containing the Jacobian of **FCN** evaluated at  $X$ .

**GRAD** — Real work vector of length  $N$  used to store the gradient of each function in **FCN**.

**XSCALE** — Vector of length  $N$  used to store the diagonal scaling matrix for the variables.

**EPSFCN** — Estimate of the relative noise in the function.

**INFT** — Vector of length  $N$ . For  $I = 1$  through  $N$ , **INFT** contains information about the Jacobian.

**NEWX** — Real work array of length  $N$ .

2. Informational errors

Type Code

|   |   |                                                                          |
|---|---|--------------------------------------------------------------------------|
| 4 | 1 | The user-supplied Jacobian is a poor estimate of the numerical Jacobian. |
|---|---|--------------------------------------------------------------------------|

## Example

The user-supplied Jacobian of

$$f_1 = 1 - x_1$$
$$f_2 = 10(x_2 - x_1^2)$$

at  $(-1.2, 1.0)$  is checked.

```
USE CHJAC_INT
USE WRIRN_INT

IMPLICIT NONE
INTEGER LDINFO, N
PARAMETER (M=2, N=2, LDINFO=M)
!
INTEGER INFO(LDINFO, N)
REAL X(N)
EXTERNAL FCN, JAC
!
! Input value for X
! X = (-1.2, 1.0)
!
DATA X/-1.2, 1.0/
!
CALL CHJAC (FCN, JAC, X, INFO)
CALL WRIRN ('The information matrix', INFO)
!
END
!
SUBROUTINE FCN (M, N, X, F)
INTEGER M, N
REAL X(N), F(M)
!
F(1) = 1.0 - X(1)
F(2) = 10.0*(X(2)-X(1)*X(1))
RETURN
END
!
SUBROUTINE JAC (M, N, X, FJAC, LDFJAC)
INTEGER M, N, LDFJAC
REAL X(N), FJAC(LDFJAC, N)
!
FJAC(1,1) = -1.0
FJAC(1,2) = 0.0
FJAC(2,1) = -20.0*X(1)
FJAC(2,2) = 10.0
RETURN
END
```

## Output

```

*** WARNING ERROR 2 from C2JAC. The numerical value of the Jacobian
*** evaluation for function 1 at the point X(2) = 1.000000E+00 and
*** the user-supplied value are both zero. The Jacobian for this
*** function should probably be re-checked at another value for
*** this point.

```

The information matrix

```

 1 2
1 1 3
2 1 1

```

---

## GGUES

Generates points in an N-dimensional space.

### Required Arguments

**A** — Vector of length N. (Input)  
See B.

**B** — Real vector of length N. (Input)  
A and B define the rectangular region in which the points will be generated, i.e.,  
 $A(I) < S(I) < B(I)$  for  $I = 1, 2, \dots, N$ . Note that if  $B(I) < A(I)$ , then  $B(I) < S(I) < A(I)$ .

**K** — The number of points to be generated. (Input)

**IDO** — Initialization parameter. (Input/Output)  
IDO must be set to zero for the first call. GGUES resets IDO to 1 and returns the first generated point in S. Subsequent calls should be made with IDO = 1.

**S** — Vector of length N containing the generated point. (Output)  
Each call results in the next generated point being stored in S.

### Optional Arguments

**N** — Dimension of the space. (Input)  
Default:  $N = \text{SIZE}(B,1)$ .

### FORTRAN 90 Interface

Generic: CALL GGUES (A, B, K, IDO, S [, ...])

Specific: The specific interface names are S\_GGUES and D\_GGUES.

### FORTRAN 77 Interface

Single: CALL GGUES (N, A, B, K, IDO, S)

Double: The double precision name is DGGUES.

## Description

The routine `GGUES` generates starting points for algorithms that optimize functions of several variables—or, almost equivalently—algorithms that solve simultaneous nonlinear equations.

The routine `GGUES` is based on systematic placement of points to optimize the dispersion of the set. For more details, see Aird and Rice (1977).

## Comments

1. Workspace may be explicitly provided, if desired, by use of `G2UES/DG2UES`. The reference is:

```
CALL G2UES (N, A, B, K, IDO, S, WK, IWK)
```

The additional arguments are:

**WK** — Work vector of length  $N$ .  $WK$  must be preserved between calls to `G2UES`.

**IWK** — Work vector of length 10.  $IWK$  must be preserved between calls to `G2UES`.

2. Informational error

Type Code

4 1 Attempt to generate more than  $K$  points.

3. The routine `GGUES` may be used with any nonlinear optimization routine that requires starting points. The rectangle to be searched (defined by  $A$ ,  $B$ , and  $N$ ) must be determined; and the number of starting points,  $K$ , must be chosen. One possible use for `GGUES` would be to call `GGUES` to generate a point in the chosen rectangle. Then, call the nonlinear optimization routine using this point as an initial guess for the solution. Repeat this process  $K$  times. The number of iterations that the optimization routine is allowed to perform should be quite small (5 to 10) during this search process. The best (or best several) point(s) found during the search may be used as an initial guess to allow the optimization routine to determine the optimum more accurately. In this manner, an  $N$  dimensional rectangle may be effectively searched for a global optimum of a nonlinear function. The choice of  $K$  depends upon the nonlinearity of the function being optimized. A function with many local optima requires a larger value than a function with only a few local optima.

## Example

We want to search the rectangle with vertices at coordinates (1, 1), (3, 1), (3, 2), and (1, 2) ten times for a global optimum of a nonlinear function. To do this, we need to generate starting points. The following example illustrates the use of `GGUES` in this process:

```

 USE GGUES_INT
 USE UMACH_INT

 IMPLICIT NONE
!
! Variable Declarations
 INTEGER N
 PARAMETER (N=2)
!
 INTEGER IDO, J, K, NOUT
 REAL A(N), B(N), S(N)
!
! Initializations
!
! A = (1.0, 1.0)
! B = (3.0, 2.0)
!
 DATA A/1.0, 1.0/
 DATA B/3.0, 2.0/
!
 CALL UMACH (2, NOUT)
 WRITE (NOUT,99998)
99998 FORMAT (' Point Number', 7X, 'Generated Point')
!
 K = 10
 IDO = 0
 DO 10 J=1, K
 CALL GGUES (A, B, K, IDO, S)
!
 WRITE (NOUT,99999) J, S(1), S(2)
99999 FORMAT (1X, I7, 14X, '(', F4.1, ', ', F6.3, ')')
!
 10 CONTINUE
!
 END

```

## Output

| Point Number | Generated Point |
|--------------|-----------------|
| 1            | ( 1.5, 1.125)   |
| 2            | ( 2.0, 1.500)   |
| 3            | ( 2.5, 1.750)   |
| 4            | ( 1.5, 1.375)   |
| 5            | ( 2.0, 1.750)   |
| 6            | ( 1.5, 1.625)   |
| 7            | ( 2.5, 1.250)   |
| 8            | ( 1.5, 1.875)   |
| 9            | ( 2.0, 1.250)   |
| 10           | ( 2.5, 1.500)   |



# Chapter 9: Basic Matrix/Vector Operations

---

## Routines

|      |                                                                              |                    |
|------|------------------------------------------------------------------------------|--------------------|
| 9.1. | <b>Basic Linear Algebra Subprograms (BLAS)</b>                               | 1576               |
|      | <b>Programming Notes for Level 1 BLAS</b> .....                              | 1576               |
|      | Set a vector to a constant value, $x_i \leftarrow a$ .....                   | <b>SSET</b> 1579   |
|      | Copy a vector, $y_i \leftarrow x_i$ .....                                    | <b>SCOPY</b> 1579  |
|      | Scale a vector by a constant, $x_i \leftarrow ax_i$ .....                    | <b>SSCAL</b> 1579  |
|      | Set a vector to a constant multiple of a vector, $y_i \leftarrow ax_i$ ..... | <b>SVCAL</b> 1580  |
|      | Add a constant to a vector, $x_i \leftarrow x_i + a$ .....                   | <b>SADD</b> 1580   |
|      | Subtract a vector from a constant, $x_i \leftarrow a - x_i$ .....            | <b>SSUB</b> 1580   |
|      | Add a multiple of one vector to another, $y_i \leftarrow ax_i + y_i$ .....   | <b>SAXPY</b> 1580  |
|      | Swap two vectors, $y_i \leftrightarrow x_i$ .....                            | <b>SSWAP</b> 1581  |
|      | Compute $x^T y$ or $x^H y$ .....                                             | <b>SDOT</b> 1581   |
|      | Compute extended precision $x^T y$ or $x^H y$ .....                          | <b>DSDOT</b> 1581  |
|      | Compute extended precision $a + x^T y$ or $a + x^H y$ .....                  | <b>SDSDOT</b> 1582 |
|      | Compute $ACC + b + x^T y$<br>with extended precision accumulator .....       | <b>SDDOTI</b> 1582 |
|      | Compute $z_i \leftarrow x_i y_i$ .....                                       | <b>SHPROD</b> 1583 |
|      | Compute $\sum x_i y_i z_i$ .....                                             | <b>SXYZ</b> 1583   |
|      | Compute $\sum x_i$ .....                                                     | <b>SSUM</b> 1583   |
|      | Compute $\sum  x_i $ .....                                                   | <b>SASUM</b> 1583  |
|      | Compute $\ x\ _2$ .....                                                      | <b>SNRM2</b> 1583  |
|      | Compute $\prod x_i$ .....                                                    | <b>SPRDCT</b> 1584 |
|      | Find the index $i$ such that $x_i = \min_j x_j$ .....                        | <b>ISMIN</b> 1584  |
|      | Find the index $i$ such that $x_i = \max_j x_j$ .....                        | <b>ISMAX</b> 1584  |
|      | Find the first index $i$ such that $ x_i  = \min_j  x_j $ .....              | <b>ISAMIN</b> 1584 |
|      | Find the first index $i$ such that $ x_i  = \max_j  x_j $ .....              | <b>ISAMAX</b> 1585 |
|      | Construct a Givens rotation .....                                            | <b>SROTG</b> 1585  |
|      | Apply a Givens rotation .....                                                | <b>SROT</b> 1586   |
|      | Construct a modified Givens rotation .....                                   | <b>SROTMG</b> 1586 |
|      | <b>Programming Notes for Level 2 and Level 3 BLAS</b> .....                  | 1588               |

|                                                                            |                                         |             |
|----------------------------------------------------------------------------|-----------------------------------------|-------------|
| Apply a modified Givens rotation .....                                     | <a href="#">SROTM</a>                   | 1587        |
| Matrix-vector multiply, general .....                                      | <a href="#">SGEMV</a>                   | 1592        |
| Matrix-vector multiply, banded .....                                       | <a href="#">SGBMV</a>                   | 1592        |
| Matrix-vector multiply, Hermitian .....                                    | <a href="#">CHEMV</a>                   | 1592        |
| Matrix-vector multiply, packed Hermitian .....                             | <a href="#">CHPMV</a>                   | 1593        |
| Matrix-vector multiply, Hermitian and banded.....                          | <a href="#">CHBMV</a>                   | 1593        |
| Matrix-vector multiply, symmetric and real .....                           | <a href="#">SSYMV</a>                   | 1593        |
| Matrix-vector multiply, packed symmetric, real .....                       | <a href="#">SSPMV</a>                   | 1593        |
| Matrix-vector multiply, symmetric and banded.....                          | <a href="#">SSBMV</a>                   | 1593        |
| Matrix-vector multiply, triangular .....                                   | <a href="#">STRMV</a>                   | 1593        |
| Matrix-vector multiply, triangular and banded .....                        | <a href="#">STBMV</a>                   | 1594        |
| Matrix-vector solve, triangular .....                                      | <a href="#">STRSV</a>                   | 1594        |
| Matrix-vector solve, triangular and banded.....                            | <a href="#">STBSV</a>                   | 1595        |
| Matrix-vector multiply, packed triangular.....                             | <a href="#">STPMV</a>                   | 1594        |
| Matrix-vector solve, packed triangular .....                               | <a href="#">STPSV</a>                   | 1595        |
| Rank-one matrix update, general and real.....                              | <a href="#">SGER</a>                    | 1595        |
| Rank-one matrix update, general, complex,<br>and transpose.....            | <a href="#">CGERU</a>                   | 1595        |
| Rank-one matrix update, general, complex,<br>and conjugate transpose ..... | <a href="#">CGERC</a>                   | 1596        |
| Rank-one matrix update,<br>Hermitian and conjugate transpose .....         | <a href="#">CHER</a>                    | 1596        |
| Hermitian, packed and conjugate transpose.....                             | <a href="#">CHPR</a>                    |             |
| Rank-two matrix update,<br>Hermitian and conjugate transpose .....         | <a href="#">CHER2</a>                   | 1596        |
| Rank-two matrix update,<br>Hermitian, packed and conjugate transpose.....  | <a href="#">CHPR2</a>                   | 1596        |
| Rank-one matrix update, symmetric and real .....                           | <a href="#">SSYR</a>                    | 1597        |
| Rank-one matrix update, packed symmetric and real.....                     | <a href="#">SSPR</a>                    | 1597        |
| Rank-two matrix update, symmetric and real.....                            | <a href="#">SSYR2</a>                   | 1597        |
| Rank-two matrix update, packed symmetric and real .....                    | <a href="#">SSPR2</a>                   | 1597        |
| Matrix-matrix multiply, general .....                                      | <a href="#">SGEMM</a>                   | 1598        |
| Matrix-matrix multiply, symmetric.....                                     | <a href="#">SSYMM</a>                   | 1598        |
| Matrix-matrix multiply, Hermitian .....                                    | <a href="#">CHEMM</a>                   | 1598        |
| Rank- $k$ update, symmetric.....                                           | <a href="#">SSYRK</a>                   | 1598        |
| Rank- $k$ update, Hermitian .....                                          | <a href="#">CHERK</a>                   | 1599        |
| Rank- $2k$ update, symmetric.....                                          | <a href="#">SSYR2K</a>                  | 1599        |
| Rank- $2k$ update, Hermitian.....                                          | <a href="#">CHER2K</a>                  | 1599        |
| Matrix-matrix multiply, triangular .....                                   | <a href="#">STRMM</a>                   | 1600        |
| Matrix-matrix solve, triangular .....                                      | <a href="#">STRSM</a>                   | 1600        |
| <b>Programming Notes Using BLAS for NVIDIA .....</b>                       |                                         | <b>1601</b> |
| Gets the switchover value .....                                            | <a href="#">CUBLAS_GET</a>              | 1607        |
| Sets the switchover value.....                                             | <a href="#">CUBLAS_SET</a>              | 1609        |
| Maintains buffer sizes.....                                                | <a href="#">CHECK_BUFFER_ALLOCATION</a> | 1611        |
| Prints error messages .....                                                | <a href="#">CUDA_ERROR_PRINT</a>        | 1612        |
| <b>9.2. Other Matrix/Vector Operations</b>                                 |                                         |             |
| 9.2.1 Matrix Copy                                                          |                                         |             |
| Real general .....                                                         | <a href="#">CRGRG</a>                   | 1615        |

|       |                                                        |       |      |
|-------|--------------------------------------------------------|-------|------|
|       | Complex general.....                                   | CCGCG | 1616 |
|       | Real band.....                                         | CRBRB | 1617 |
|       | Complex band.....                                      | CCBCB | 1619 |
| 9.2.2 | Matrix Conversion                                      |       |      |
|       | Real general to real band .....                        | CRGRB | 1621 |
|       | Real band to real general .....                        | CRBRG | 1622 |
|       | Complex general to complex band .....                  | CCGCB | 1624 |
|       | Complex band to complex general .....                  | CCBCG | 1626 |
|       | Real general to complex general.....                   | CRGCG | 1627 |
|       | Real rectangular to complex rectangular.....           | CRRCR | 1629 |
|       | Real band to complex band.....                         | CRBCB | 1631 |
|       | Real symmetric to real general .....                   | CSFRG | 1632 |
|       | Complex Hermitian to complex general.....              | CHFCG | 1634 |
|       | Real symmetric band to real band.....                  | CSBRB | 1635 |
|       | Complex Hermitian band to complex band.....            | CHBCB | 1637 |
|       | Real rectangular matrix to its transpose .....         | TRNRR | 1639 |
| 9.2.3 | Matrix Multiplication                                  |       |      |
|       | Compute $X^T X$ .....                                  | MXTXF | 1641 |
|       | Compute $X^T Y$ .....                                  | MXTYF | 1643 |
|       | Compute $XY^T$ .....                                   | MXYTF | 1645 |
|       | Multiply two real rectangular matrices .....           | MRRRR | 1647 |
|       | Multiply two complex rectangular matrices.....         | MCRCR | 1649 |
|       | Compute matrix Hadamard product.....                   | HRRRR | 1651 |
|       | Compute the bilinear form $x^T A y$ .....              | BLINF | 1653 |
|       | Compute the matrix polynomial $p(A)$ .....             | POLRG | 1655 |
| 9.2.4 | Matrix-Vector Multiplication                           |       |      |
|       | Real rectangular matrix times a real vector .....      | MURRV | 1657 |
|       | Real band matrix times a real vector .....             | MURBV | 1659 |
|       | Complex rectangular matrix times a complex vector..... | MUCRV | 1661 |
|       | Complex band matrix times a complex vector .....       | MUCBV | 1663 |
| 9.2.5 | Matrix Addition                                        |       |      |
|       | Real band matrix plus a real band matrix .....         | ARBRB | 1665 |
|       | Complex band matrix plus a complex band matrix.....    | ACBCB | 1667 |
| 9.2.6 | Matrix Norm                                            |       |      |
|       | $\infty$ -norm of a real rectangular matrix .....      | NRIRR | 1670 |
|       | 1-norm of a real rectangular matrix .....              | NR1RR | 1671 |
|       | Frobenius norm of a real rectangular matrix.....       | NR2RR | 1673 |
|       | 1-norm of a real band matrix.....                      | NR1RB | 1674 |
|       | 1-norm of a complex band matrix .....                  | NR1CB | 1676 |
| 9.2.7 | Distance Between Two Points                            |       |      |
|       | Euclidean distance.....                                | DISL2 | 1677 |
|       | 1-norm distance .....                                  | DISL1 | 1679 |
|       | $\infty$ -norm distance.....                           | DISLI | 1681 |
| 9.2.8 | Vector Convolutions                                    |       |      |
|       | Convolution of real vectors .....                      | VCONR | 1683 |
|       | Convolution of complex vectors.....                    | VCONC | 1685 |

### 9.3. Extended Precision Arithmetic

|                                                                            |       |      |
|----------------------------------------------------------------------------|-------|------|
| Initialize a real accumulator, $ACC \leftarrow a$ .....                    | DQINI | 1687 |
| Store a real accumulator, $a \leftarrow ACC$ .....                         | DQSTO | 1687 |
| Add to a real accumulator, $ACC \leftarrow ACC + a$ .....                  | DQADD | 1687 |
| Add a product to a real accumulator, $ACC \leftarrow ACC + ab$ .....       | DQMUL | 1687 |
| Initialize a complex accumulator, $ACC \leftarrow a$ .....                 | ZQINI | 1687 |
| Store a complex accumulator, $a \leftarrow ACC$ .....                      | ZQSTO | 1687 |
| Add to a complex accumulator, $ACC \leftarrow ACC + a$ .....               | ZQADD | 1687 |
| Add a product to a complex accumulator,<br>$ACC \leftarrow ACC + ab$ ..... | ZQMUL | 1687 |

---

## Basic Linear Algebra Subprograms

The basic linear algebra subprograms, normally referred to as the BLAS, are routines for low-level operations such as dot products, matrix times vector, and matrix times matrix. Lawson et al. (1979) published the original set of 38 BLAS. The IMSL BLAS collection includes these 38 subprograms plus additional ones that extend their functionality. Since Dongarra et al. (1988 and 1990) published extensions to this set, it is customary to refer to the original 38 as Level 1 BLAS. The Level 1 operations are performed on one or two vectors of data. An extended set of subprograms perform operations involving a matrix and one or two vectors. These are called the Level 2 BLAS (see [Specification of the Level 1 BLAS](#)). An additional extended set of operations on matrices is called the Level 3 BLAS (see [Specification of the Level 3 BLAS](#)).

Users of the BLAS will often benefit from using versions of the BLAS supplied by hardware vendors, if available. This can provide for more efficient execution of many application programs. The BLAS provided by IMSL are written in FORTRAN. Those supplied by vendors may be written in other languages, such as assembler. The documentation given below for the BLAS is compatible with a vendor's version of the BLAS that conforms to the published specifications.

Users having an NVIDIA GPGPU or NVIDIA board can make use of a subset of the BLAS written for the NVIDIA board through the IMSL Libraries. The board is used for problems that exceed a certain size, `NSTART`. For smaller values a standard version is used. This value can be changed for any routine to any alternate value by using subprograms together with use association of the module `CUDABLAS_LIBRARY`. Documentation and further descriptions are provided in the section below [Programming Notes for BLAS Using NVIDIA](#). [Table 9.2](#) has the names of NVIDIA routines that are implemented marked with **GREEN**.

### Programming Notes for Level 1 BLAS

The Level 1 BLAS do not follow the usual IMSL naming conventions. Instead, the names consist of a prefix of one or more of the letters "I", "S", "D", "C", and "Z"; a root name; and sometimes a suffix. For subprograms involving a mixture of data types, the output type is indicated by the first prefix letter. The suffix denotes a variant algorithm. The prefix denotes the type of the operation according to the following table:

|   |         |   |                |
|---|---------|---|----------------|
| I | Integer |   |                |
| S | Real    | C | Complex        |
| D | Double  | Z | Double Complex |

|    |                      |    |                              |
|----|----------------------|----|------------------------------|
| SD | Single and Double    | CZ | Single and Double Complex    |
| DQ | Double and Quadruple | ZQ | Double and Quadruple Complex |

Vector arguments have an increment parameter that specifies the storage space or stride between elements. The correspondence between the vectors  $x$  and  $y$  and the arguments  $SX$  and  $SY$ , and  $INCX$  and  $INCY$  is

$$x_i = \begin{cases} SX((I-1)*INCX + 1) & \text{if } INCX \geq 0 \\ SX((I-N)*INCX + 1) & \text{if } INCX < 0 \end{cases}$$

$$y_i = \begin{cases} SY((I-1)*INCY + 1) & \text{if } INCY \geq 0 \\ SY((I-N)*INCY + 1) & \text{if } INCY < 0 \end{cases}$$

Function subprograms `SXYZ` and `DXYZ` refer to a third vector argument  $z$ . The storage increment  $INCZ$  for  $z$  is defined like  $INCX$  and  $INCY$ . In the Level 1 BLAS, only positive values of  $INCX$  are allowed for operations that have a single vector argument. The loops in all of the Level 1 BLAS process the vector arguments in order of increasing  $i$ . For  $INCX, INCY, INCZ < 0$ , this implies processing in reverse storage order.

The function subprograms in the Level 1 BLAS are all illustrated by means of an assignment statement. For example, see `SDOT`. Any value of a function subprogram can be used in an expression or as a parameter passed to a subprogram as long as the data types agree.

## Descriptions of the Level 1 BLAS Subprograms

The set of Level 1 BLAS are summarized in Table 9.1. This table also lists the page numbers where the subprograms are described in more detail.

### Specification of the Level 1 BLAS

With the definitions,

$$MX = \max \{1, 1 + (N - 1)|INCX|\}$$

$$MY = \max \{1, 1 + (N - 1)|INCY|\}$$

$$MZ = \max \{1, 1 + (N - 1)|INCZ|\}$$

the subprogram descriptions assume the following FORTRAN declarations:

```

IMPLICIT INTEGER (I-N)
IMPLICIT REAL S
IMPLICIT DOUBLE PRECISION D
IMPLICIT COMPLEX C
IMPLICIT DOUBLE COMPLEX Z

INTEGER IX (MX)
REAL SX (MX), SY (MY), SZ (MZ),
 SPARAM (5)
DOUBLE PRECISION DX (MX), DY (MY), DZ (MZ),
 DPARAM (5)

DOUBLE PRECISION DACC (2), DZACC (4)
COMPLEX CX (MX), CY (MY)
DOUBLE COMPLEX ZX (MX), ZY (MY)

```

Since FORTRAN 77 does not include the type `DOUBLE COMPLEX`, subprograms with `DOUBLE COMPLEX` arguments are not available for all systems. Some systems use the declaration `COMPLEX * 16` instead of `DOUBLE COMPLEX`.

In the following descriptions, the original BLAS are marked with an \* in the left column.

*Table 9.1: Level 1 Basic Linear Algebra Subprograms*

| Operation                                                | Integer | Real             | Double           | Complex          | Double Complex   | Pg.  |
|----------------------------------------------------------|---------|------------------|------------------|------------------|------------------|------|
| $x_i \leftarrow a$                                       | ISET    | SSET             | DSET             | CSET             | ZSET             | 1579 |
| $y_i \leftarrow x_i$                                     | ICOPY   | SCOPY            | DCOPY            | CCOPY            | ZCOPY            | 1579 |
| $x_i \leftarrow ax_i$<br>$a \in \mathbf{R}$              |         | SSCAL            | DSCAL            | CSCAL<br>CSSCAL  | ZSCAL<br>ZDSCAL  | 1579 |
| $y_i \leftarrow ax_i$<br>$a \in \mathbf{R}$              |         | SVCAL            | DVCAL            | CVCAL<br>CSVCAL  | ZVCAL<br>ZDVCAL  | 1580 |
| $x_i \leftarrow x_i + a$                                 | IADD    | SADD             | DADD             | CADD             | ZADD             | 1580 |
| $x_i \leftarrow a - x_i$                                 | ISUB    | SSUB             | DSUB             | CSUB             | ZSUB             | 1580 |
| $y_i \leftarrow ax_i + y_i$                              |         | SAXPY            | DAXPY            | CAXPY            | ZAXPY            | 1580 |
| $y_i \leftrightarrow x_i$                                | ISWAP   | SSWAP            | DSWAP            | CSWAP            | ZSWAP            | 1581 |
| $x \cdot y$<br>$\bar{x} \cdot y$                         |         | SDOT             | DDOT             | CDOTU<br>CDOTC   | ZDOTU<br>ZDOTC   | 1581 |
| $x \cdot y^\dagger$<br>$\bar{x} \cdot y^\dagger$         |         | DSDOT            |                  | CZDOTU<br>CZDOTC | ZQDOTU<br>ZQDOTC | 1581 |
| $a + x \cdot y^\dagger$<br>$a + \bar{x} \cdot y^\dagger$ |         | SDSDOT           | DQDDOT           | CZUDOT<br>CZCDOT | ZQUDOT<br>ZQCDOT | 1582 |
| $b + x \cdot y^\dagger$<br>ACC + $b + x \cdot y^\dagger$ |         | SDDOTI<br>SDDOTA | DQDOTI<br>DQDOTA | CZDOTI<br>CZDOTA | ZQDOTI<br>ZQDOTA | 1582 |
| $z_i \leftarrow x_i y_i$                                 |         | SHPROD           | DHPROD           |                  |                  | 1583 |
| $\sum x_i y_i z_i$                                       |         | SXYZ             | DXYZ             |                  |                  | 1583 |
| $\sum x_i$                                               | ISUM    | SSUM             | DSUM             |                  |                  | 1583 |
| $\sum  x_i $                                             |         | SASUM            | DASUM            | SCASUM           | DZASUM           | 1583 |
| $\ x\ _2$                                                |         | SNRM2            | DNRM2            | SCNRM2           | DZNRM2           | 1583 |
| $\prod x_i$                                              |         | SPRDCT           | DPRDCT           |                  |                  | 1584 |
| $i : x_i = \min_j x_j$                                   | IIMIN   | ISMIN            | IDMIN            |                  |                  | 1584 |
| $i : x_i = \max_j x_j$                                   | IIMAX   | ISMAX            | IDMAX            |                  |                  | 1584 |

| Operation                           | Integer | Real   | Double | Complex       | Double Complex | Pg.  |
|-------------------------------------|---------|--------|--------|---------------|----------------|------|
| $i :  x_i  = \min_j  x_j $          |         | ISAMIN | IDAMIN | ICAMIN        | IZAMIN         | 1584 |
| $i :  x_i  = \max_j  x_j $          |         | ISAMAX | IDAMAX | ICAMAX        | IZAMAX         | 1585 |
| Construct Givens rotation           |         | SROTG  | DROTG  | CROTG         | ZROTG          | 1585 |
| Apply Givens rotation               |         | SROT   | DROT   | CROT<br>CSROT | ZROT<br>ZDROT  | 1586 |
| Construct modified Givens transform |         | SROTMG | DROTMG |               |                | 1586 |
| Apply modified Givens transform     |         | SROTM  | DROTM  | CSROTM        | ZDROTM         | 1587 |

†Higher precision accumulation used

### Set a Vector to a Constant Value

```
CALL ISET (N, IA, IX, INCX)
CALL SSET (N, SA, SX, INCX)
CALL DSET (N, DA, DX, INCX)
CALL CSET (N, CA, CX, INCX)
CALL ZSET (N, ZA, ZX, INCX)
```

These subprograms set  $x_i \leftarrow a$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

### Copy a Vector

```
CALL ICOPY (N, IX, INCX, IY, INCY)
*CALL SCOPY (N, SX, INCX, SY, INCY)
*CALL DCOPY (N, DX, INCX, DY, INCY)
*CALL CCOPY (N, CX, INCX, CY, INCY)
CALL ZCOPY (N, ZX, INCX, ZY, INCY)
```

These subprograms set  $y_i \leftarrow x_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

### Scale a Vector

```
*CALL SSCAL (N, SA, SX, INCX)
*CALL DSCAL (N, DA, DX, INCX)
*CALL CSCAL (N, CA, CX, INCX)
CALL ZSCAL (N, ZA, ZX, INCX)
*CALL CSSCAL (N, SA, CX, INCX)
CALL ZDSCAL (N, DA, ZX, INCX)
```

These subprograms set  $x_i \leftarrow ax_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately. CAUTION: For CSSCAL and ZDSCAL, the scalar quantity  $a$  is real and the vector  $x$  is complex.

### Multiply a Vector by a Constant

```
CALL SVCAL (N, SA, SX, INCX, SY, INCY)
CALL DVCAL (N, DA, DX, INCX, DY, INCY)
CALL CVCAL (N, CA, CX, INCX, CY, INCY)
CALL ZVCAL (N, ZA, ZX, INCX, ZY, INCY)
CALL CSVCAL (N, SA, CX, INCX, CY, INCY)
CALL ZDVCAL (N, DA, ZX, INCX, ZY, INCY)
```

These subprograms set  $y_i \leftarrow ax_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately. CAUTION: For CSVCAL and ZDVCAL, the scalar quantity  $a$  is real and the vector  $x$  is complex.

### Add a Constant to a Vector

```
CALL IADD (N, IA, IX, INCX)
CALL SADD (N, SA, SX, INCX)
CALL DADD (N, DA, DX, INCX)
CALL CADD (N, CA, CX, INCX)
CALL ZADD (N, ZA, ZX, INCX)
```

These subprograms set  $x_i \leftarrow x_i + a$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

### Subtract a Vector from a Constant

```
CALL ISUB (N, IA, IX, INCX)
CALL SSUB (N, SA, SX, INCX)
CALL DSUB (N, DA, DX, INCX)
CALL CSUB (N, CA, CX, INCX)
CALL ZSUB (N, ZA, ZX, INCX)
```

These subprograms set  $x_i \leftarrow a - x_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

### Constant Times a Vector Plus a Vector

```
*CALL SAXPY (N, SA, SX, INCX, SY, INCY)
*CALL DAXPY (N, DA, DX, INCX, DY, INCY)
*CALL CAXPY (N, CA, CX, INCX, CY, INCY)
CALL ZAXPY (N, ZA, ZX, INCX, ZY, INCY)
```

These subprograms set  $y_i \leftarrow ax_i + y_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

## Swap Two Vectors

```
CALL ISWAP (N, IX, INCX, IY, INCY)
*CALL SSWAP (N, SX, INCX, SY, INCY)
*CALL DSWAP (N, DX, INCX, DY, INCY)
*CALL CSWAP (N, CX, INCX, CY, INCY)
CALL ZSWAP (N, ZX, INCX, ZY, INCY)
```

These subprograms perform the exchange  $y_i \leftrightarrow x_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

## Dot Product

```
*SW = SDOT (N, SX, INCX, SY, INCY)
*DZ = DDOT (N, DX, INCX, DY, INCY)
*CW = CDOTU (N, CX, INCX, CY, INCY)
*CW = CDOTC (N, CX, INCX, CY, INCY)
ZW = ZDOTU (N, ZX, INCX, ZY, INCY)
ZW = ZDOTC (N, ZX, INCX, ZY, INCY)
```

The function subprograms SDOT, DDOT, CDOTU, and ZDOTU compute

$$\sum_{i=1}^N x_i y_i$$

The function subprograms CDOTC and ZDOTC compute

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

The suffix C indicates that the complex conjugates of  $x_i$  are used. The suffix U indicates that the unconjugated values of  $x_i$  are used. If  $N \leq 0$ , then the subprograms return zero.

## Dot Product with Higher Precision Accumulation

```
*DW = DSDOT (N, SX, INCX, SY, INCY)
CW = CZDOTC (N, CX, INCX, CY, INCY)
CW = CZDOTU (N, CX, INCX, CY, INCY)
ZW = ZQDOTC (N, ZX, INCX, ZY, INCY)
ZW = ZQDOTU (N, ZX, INCX, ZY, INCY)
```

The function subprogram DSDOT computes

$$\sum_{i=1}^N x_i y_i$$

using double precision accumulation. The function subprograms CZDOTU and ZQDOTU compute

$$\sum_{i=1}^N x_i y_i$$

using double and quadruple complex accumulation, respectively. The function subprograms CZDOTC and ZQDOTC compute

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

using double and quadruple complex accumulation, respectively. If  $N \leq 0$ , then the subprograms return zero.

### Constant Plus Dot Product with Higher Precision Accumulation

```
*SW = SDSDOT (N, SA, SX, INCX, SY, INCY)
DW = DQDDOT (N, DA, DX, INCX, DY, INCY)
CW = CZCDOT (N, CA, CX, INCX, CY, INCY)
CW = CZUDOT (N, CA, CX, INCX, CY, INCY)
ZW = ZQCDOT (N, ZA, ZX, INCX, ZY, INCY)
ZW = ZQUDOT (N, ZA, ZX, INCX, ZY, INCY)
```

The function subprograms SDSDOT, DQDDOT, CZUDOT, and ZQUDOT compute

$$a + \sum_{i=1}^N x_i y_i$$

using higher precision accumulation where SDSDOT uses double precision accumulation, DQDDOT uses quadruple precision accumulation, CZUDOT uses double complex accumulation, and ZQUDOT uses quadruple complex accumulation. The function subprograms CZCDOT and ZQCDOT compute

$$a + \sum_{i=1}^N \bar{x}_i y_i$$

using double complex and quadruple complex accumulation, respectively. If  $N \leq 0$ , then the subprograms return zero.

### Dot Product Using the Accumulator

```
SW = SDDOTI (N, SB, DACC, SX, INCX, SY, INCY)
SW = SDDOTA (N, SB, DACC, SX, INCX, SY, INCY)
CW = CZDOTI (N, CB, DACC, CX, INCX, CY, INCY)
CW = CZDOTA (N, CB, DACC, CX, INCX, CY, INCY)
*DW = DQDOTI (N, DB, DACC, DX, INCX, DY, INCY)
*DW = DQDOTA (N, DB, DACC, DX, INCX, DY, INCY)
ZW = ZQDOTI (N, ZB, DZACC, ZX, INCX, ZY, INCY)
ZW = ZQDOTA (N, ZB, DZACC, ZX, INCX, ZY, INCY)
```

The variable DACC, a double precision array of length two, is used as a quadruple precision accumulator. DZACC, a double precision array of length four, is its complex analog. The function subprograms, with a name ending in I, initialize DACC to zero. All of the function subprograms then compute

$$DACC + b + \sum_{i=1}^N x_i y_i$$

and store the result in DACC. The result, converted to the precision of the function, is also returned as the function value. If  $N \leq 0$ , then the function subprograms return zero.

## Hadamard Product

```
CALL SHPROD (N, SX, INCX, SY, INCY, SZ, INCZ)
CALL DHPROD (N, DX, INCX, DY, INCY, DZ, INCZ)
```

These subprograms set  $z_i \leftarrow x_i y_i$  for  $i = 1, 2, \dots, N$ . If  $N \leq 0$ , then the subprograms return immediately.

## Triple Inner Product

```
SW = SXYZ (N, SX, INCX, SY, INCY, SZ, INCZ)
DW = DXYZ (N, DX, INCX, DY, INCY, DZ, INCZ)
```

These function subprograms compute

$$\sum_{i=1}^N x_i y_i z_i$$

If  $N \leq 0$  then the subprograms return zero.

## Sum of the Elements of a Vector

```
IW = ISUM (N, IX, INCX)
SW = SSUM (N, SX, INCX)
DW = DSUM (N, DX, INCX)
```

These function subprograms compute

$$\sum_{i=1}^N x_i$$

If  $N \leq 0$ , then the subprograms return zero.

## Sum of the Absolute Values of the Elements of a Vector

```
*SW = SASUM (N, SX, INCX)
*DZ = DASUM (N, DX, INCX)
*SW = SCASUM (N, CX, INCX)
DW = DZASUM (N, ZX, INCX)
```

The function subprograms SASUM and DASUM compute

$$\sum_{i=1}^N |x_i|$$

The function subprograms SCASUM and DZASUM compute

$$\sum_{i=1}^N [|\Re x_i| + |\Im x_i|]$$

If  $N \leq 0$ , then the subprograms return zero. CAUTION: For SCASUM and DZASUM, the function subprogram returns a real value.

## Euclidean or $\ell_2$ Norm of a Vector

```
*SW = SNRM2 (N, SX, INCX)
*DW = DNRM2 (N, DX, INCX)
```

\*SW = SCNRM2 (N, CX, INCX)  
DW = DZNRM2 (N, ZX, INCX)

These function subprograms compute

$$\left[ \sum_{i=1}^N |x_i|^2 \right]^{1/2}$$

If  $N \leq 0$ , then the subprograms return zero. CAUTION: For SCNRM2 and DZNRM2, the function subprogram returns a real value.

### Product of the Elements of a Vector

SW = SPRDCT (N, SX, INCX)  
DW = DPRDCT (N, DX, INCX)

These function subprograms compute

$$\prod_{i=1}^N x_i$$

If  $N \leq 0$ , then the subprograms return zero.

### Index of Element Having Minimum Value

IW = IIMIN (N, IX, INCX)  
IW = ISMIN (N, SX, INCX)  
IW = IDMIN (N, DX, INCX)

These function subprograms compute the smallest index  $i$  such that  $x_i = \min_{1 \leq j \leq N} x_j$ . If  $N \leq 0$ , then the subprograms return zero.

### Index of Element Having Maximum Value

IW = IIMAX (N, IX, INCX)  
IW = ISMAX (N, SX, INCX)  
IW = IDMAX (N, DX, INCX)

These function subprograms compute the smallest index  $i$  such that  $x_i = \max_{1 \leq j \leq N} x_j$ . If  $N \leq 0$ , then the subprograms return zero.

### Index of Element Having Minimum Absolute Value

IW = ISAMIN (N, SX, INCX)  
IW = IDAMIN (N, DX, INCX)  
IW = ICAMIN (N, CX, INCX)  
IW = IZAMIN (N, ZX, INCX)

The function subprograms ISAMIN and IDAMIN compute the smallest index  $i$  such that  $|x_i| = \min_{1 \leq j \leq N} |x_j|$ . The function subprograms ICAMIN and IZAMIN compute the smallest index  $i$  such that

$$|\Re x_i| + |\Im x_i| = \min_{1 \leq j \leq N} [|\Re x_j| + |\Im x_j|]$$

If  $N \leq 0$ , then the subprograms return zero.

### Index of Element Having Maximum Absolute Value

\*IW = ISAMAX (N, SX, INCX)

\*IW = IDAMAX (N, DX, INCX)

\*IW = ICAMAX (N, CX, INCX)

IW = IZAMAX (N, ZX, INCX)

The function subprograms ISAMAX and IDAMAX compute the smallest index  $i$  such that

$|x_i| = \max_{1 \leq j \leq N} |x_j|$ . The function subprograms ICAMAX and IZAMAX compute the smallest index  $i$  such that

$$|\Re x_i| + |\Im x_i| = \max_{1 \leq j \leq N} [|\Re x_j| + |\Im x_j|]$$

If  $N \leq 0$ , then the subprograms return zero.

### Construct a Givens Plane Rotation

\*CALL SROTG (SA, SB, SC, SS)

\*CALL DROTG (SA, SB, SC, SS)

Given the values  $a$  and  $b$ , these subprograms compute

$$c = \begin{cases} a/r & \text{if } r \neq 0 \\ 1 & \text{if } r = 0 \end{cases}$$

and

$$s = \begin{cases} b/r & \text{if } r \neq 0 \\ 1 & \text{if } r = 0 \end{cases}$$

where  $r = \sigma(a^2 + b^2)^{1/2}$  and

$$\sigma = \begin{cases} \text{sign}(a) & \text{if } |a| > |b| \\ \text{sign}(b) & \text{otherwise} \end{cases}$$

Then, the values  $c$ ,  $s$  and  $r$  satisfy the matrix equation

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

The introduction of  $\sigma$  is not essential to the computation of the Givens rotation matrix; but its use permits later stable reconstruction of  $c$  and  $s$  from just one stored number, an idea due to Stewart (1976). For this purpose, the subprogram also computes

$$z = \begin{cases} s & \text{if } |s| < c \text{ or } c = 0 \\ 1/c & \text{if } 0 < |c| \leq s \end{cases}$$

In addition to returning  $c$  and  $s$ , the subprograms return  $r$  overwriting  $a$ , and  $z$  overwriting  $b$ .

Reconstruction of  $c$  and  $s$  from  $z$  can be done as follows:

If  $z = 1$ , then set  $c = 0$  and  $s = 1$

If  $|z| < 1$ , then set

$$c = \sqrt{1 - z^2} \quad \text{and} \quad s = z$$

If  $|z| > 1$ , then set

$$c = 1/z \quad \text{and} \quad s = \sqrt{1 - c^2}$$

### Apply a Plane Rotation

```
*CALL SROT (N, SX, INCX, SY, INCY, SC, SS)
*CALL DROT (N, DX, INCX, DY, INCY, DC, DS)
 CALL CSROT (N, CX, INCX, CY, INCY, SC, SS)
 CALL ZDROT (N, ZX, INCX, ZY, INCY, DC, DS)
```

These subprograms compute

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

If  $N \leq 0$ , then the subprograms return immediately. CAUTION: For CSROT and ZDROT, the scalar quantities  $c$  and  $s$  are real, and  $x$  and  $y$  are complex.

### Construct a Modified Givens Transformation

```
*CALL SROTMG (SD1, SD2, SX1, SY1, SPARAM)
*CALL DROTMG (DD1, DD2, DX1, DY1, DPARAM)
```

The input quantities  $d_1$ ,  $d_2$ ,  $x_1$  and  $y_1$  define a 2-vector  $[w_1, z_1]^T$  by the following:

$$\begin{bmatrix} w_i \\ z_i \end{bmatrix} = \begin{bmatrix} \sqrt{d_1} & 0 \\ 0 & \sqrt{d_2} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

The subprograms determine the modified Givens rotation matrix  $H$  that transforms  $y_1$ , and thus,  $z_1$  to zero. They also replace  $d_1$ ,  $d_2$  and  $x_1$  with

$$\tilde{d}_1, \tilde{d}_2 \quad \text{and} \quad \tilde{x}_1$$

respectively. That is,

$$\begin{bmatrix} \tilde{w}_1 \\ 0 \end{bmatrix} = \begin{bmatrix} \sqrt{\tilde{d}_1} & 0 \\ 0 & \sqrt{\tilde{d}_2} \end{bmatrix} H \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \sqrt{\tilde{d}_1} & 0 \\ 0 & \sqrt{\tilde{d}_2} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_1 \\ 0 \end{bmatrix}$$

A representation of this matrix is stored in the array `SPARAM` or `DPARAM`. The form of the matrix  $H$  is flagged by `PARAM(1)`.

`PARAM(1) = 1`. In this case,

$$|d_1 x_1^2| \leq |d_2 y_1^2|$$

and

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

The elements `PARAM(3)` and `PARAM(4)` are not changed.

`PARAM(1) = 0`. In this case,

$$|d_1 x_1^2| > |d_2 y_1^2|$$

and

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

The elements `PARAM(2)` and `PARAM(5)` are not changed.

`PARAM(1) = -1`. In this case, rescaling was done and

$$H = \begin{bmatrix} \text{PARAM}(2) & \text{PARAM}(4) \\ \text{PARAM}(3) & \text{PARAM}(5) \end{bmatrix}$$

`PARAM(1) = -2`. In this case,  $H = I$  where  $I$  is the identity matrix. The elements `PARAM(2)`, `PARAM(3)`, `PARAM(4)` and `PARAM(5)` are not changed.

The values of  $d_1$ ,  $d_2$  and  $x_1$  are changed to represent the effect of the transformation. The quantity  $y_1$ , which would be zeroed by the transformation, is left unchanged.

The input value of  $d_1$  should be nonnegative, but  $d_2$  can be negative for the purpose of removing data from a least-squares problem.

See Lawson et al. (1979) for further details.

### Apply a Modified Givens Transformation

```
*CALL SROTM (N, SX, INCX, SY, INCY, SPARAM)
*CALL DROTM (N, DX, INCX, DY, INCY, DPARAM)
CALL CSROTM (N, CX, INCX, CY, INCY, SPARAM)
CALL ZDROTM (N, ZX, INCX, ZY, INCY, DPARAM)
```

If `PARAM(1) = 1.0`, then these subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \text{PARAM}(2) & 1 \\ -1 & \text{PARAM}(5) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \dots, N$$

If  $\text{PARAM}(1) = 0.0$ , then the subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} 1 & \text{PARAM}(4) \\ \text{PARAM}(3) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \dots, N$$

If  $\text{PARAM}(1) = -1.0$ , then the subprograms compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \text{PARAM}(2) & \text{PARAM}(4) \\ \text{PARAM}(3) & \text{PARAM}(5) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{ for } i = 1, \dots, N$$

If  $N \leq 0$  or if  $\text{PARAM}(1) = -2.0$ , then the subprograms return immediately. CAUTION: For CSROTM and ZDROTM, the scalar quantities  $\text{PARAM}(*)$  are real and  $x$  and  $y$  are complex.

## Programming Notes for Level 2 and Level 3 BLAS

For definitions of the matrix data structures used in the discussion below, see [Reference Material](#). The Level 2 and Level 3 BLAS, like the Level 1 BLAS, do not follow the IMSL naming conventions. Instead, the names consist of a prefix of one of the letters “S”, “D”, “C”, or “Z”. Next is a root name denoting the kind of matrix. This is followed by a suffix indicating the type of the operation.<sup>1</sup> The prefix denotes the type of operation according to the following table:

|   |        |   |         |         |
|---|--------|---|---------|---------|
| S | Real   | C | Complex |         |
| D | Double | Z | Double  | Complex |

The root names for the kind of matrix:

|    |            |    |                 |    |                   |
|----|------------|----|-----------------|----|-------------------|
| GE | General    | GB | General Band    | SP | Symmetric Packed  |
| SY | Symmetric  | SB | Symmetric Band  | TP | Triangular Packed |
| HE | Hermitian  | HB | Hermitian Band  | HP | Hermitian Packed  |
| TR | Triangular | TB | Triangular Band |    |                   |

The suffixes for the type of operation:

|    |                               |     |                             |
|----|-------------------------------|-----|-----------------------------|
| MV | Matrix-Vector Product         | SV  | Solve for Vector            |
| R  | Rank-One Update               |     |                             |
| RU | Rank-One Update, Unconjugated | RC  | Rank-One Update, Conjugated |
| R2 | Rank-Two Update               |     |                             |
| MM | Matrix-Multiply               | SM  | Symmetric Matrix Multiply   |
| RK | Rank-K Update                 | R2K | Rank 2K Update              |

<sup>1</sup>IMSL does not support any extended precision versions of the Level 2 BLAS.

The specifications of the operations are provided by subprogram arguments of CHARACTER\*1 data type. Both lower and upper case of the letter have the same meaning:

|                       |     |                                     |
|-----------------------|-----|-------------------------------------|
| TRANS, TRANSA, TRANSB | 'N' | No Transpose                        |
|                       | 'T' | Transpose                           |
|                       | 'C' | Conjugate and Transpose             |
| UPLO                  | 'L' | Lower Triangular                    |
|                       | 'U' | Upper Triangular                    |
| DIAGNL                | 'N' | Non-unit Triangular                 |
|                       | 'U' | Unit Triangular                     |
| SIDE                  | 'L' | Multiply "A" Matrix on Left side or |
|                       | 'R' | Right side of the "B" matrix        |

Note: See the "Triangular Mode" section in the [Reference Material](#) for definitions of these terms.

## Descriptions of the Level 2 and Level 3 BLAS

The subprograms for Level 2 and Level 3 BLAS that perform operations involving the expression  $\beta y$  or  $\beta C$  do not require that the contents of  $y$  or  $C$  be defined when  $\beta = 0$ . In that case, the expression  $\beta y$  or  $\beta C$  is defined to be zero. Note that for the `_GEMV` and `_GBMV` subprograms, the dimensions of the vectors  $x$  and  $y$  are implied by the specification of the operation. If `TRANS = 'N'`, the dimension of  $y$  is  $m$ ; if `TRANS = 'T'` or `'C'`, the dimension of  $y$  is  $n$ . The Level 2 and Level 3 BLAS are summarized in Table 9.2. This table also lists the page numbers where the subprograms are described in more detail.

## Specification of the Level 2 BLAS

Type and dimension for variables occurring in the subprogram specifications are

|                  |                                          |
|------------------|------------------------------------------|
| INTEGER          | INCX, INCY, NCODA, NLCA, NUCA, LDA, M, N |
| CHARACTER*1      | DIAGNL, TRANS, UPLO                      |
| REAL             | SALPHA, SBETA, SX(*), SY(*), SA(LDA,*)   |
| DOUBLE PRECISION | DALPHA, DBETA, DX(*), DY(*), DA(LDA,*)   |
| COMPLEX          | CALPHA, CBETA, CX(*), CY(*), CA(LDA,*)   |
| DOUBLE COMPLEX   | ZALPHA, ZBETA, ZX(*), ZY(*), ZA(LDA,*)   |

There is a lower bound on the leading dimension LDA. It must be  $\geq$  the number of rows in the matrix that is contained in this array. Vector arguments have an increment parameter that specifies the storage space or stride between elements. The correspondence between the vector  $x$ ,  $y$  and the arguments SX, SY and INCX, INCY is

$$x_i = \begin{cases} \text{SX}((I-1)*\text{INCX} + 1) & \text{if } \text{INCX} > 0 \\ \text{SX}((I-N)*\text{INCX} + 1) & \text{if } \text{INCX} < 0 \end{cases}$$

$$y_i = \begin{cases} \text{SY}((I-1)*\text{INCY} + 1) & \text{if } \text{INCY} > 0 \\ \text{SY}((I-N)*\text{INCY} + 1) & \text{if } \text{INCY} < 0 \end{cases}$$

In the Level 2 BLAS, only nonzero values of INCX, INCY are allowed for operations that have vector arguments. The Level 3 BLAS do not refer to INCX, INCY.

### Specification of the Level 3 BLAS

Type and dimension for variables occurring in the subprogram specifications are

|                  |                                                   |
|------------------|---------------------------------------------------|
| INTEGER          | K, LDA, LDB, LDC, M, N                            |
| CHARACTER*1      | DIAGNL, TRANS, TRANSA, TRANSB, SIDE, UPLO         |
| REAL             | SALPHA, SBETA, SA(LDA,*), SB(LDB,*),<br>SC(LDC,*) |
| DOUBLE PRECISION | DALPHA, DBETA, DA(LDA,*), DB(LDB,*),<br>DC(LDC,*) |
| COMPLEX          | CALPHA, CBETA, CA(LDA,*), CB(LDB,*),<br>CC(LDC,*) |
| DOUBLE COMPLEX   | ZALPHA, ZBETA, ZA(LDA,*), ZB(LDB,*),<br>ZC(LDC,*) |

Each of the integers K, M, N must be  $\geq 0$ . It is an error if any of them are  $< 0$ . If any of them are  $= 0$ , the subprograms return immediately. There are lower bounds on the leading dimensions LDA, LDB, LDC. Each must be  $\geq$  the number of rows in the matrix that is contained in this array. The names marked with **GREEN** indicate that versions implemented using the NVIDIA CUBLAS library and NVIDIA hardware are available. It may be advantageous to use the NVIDIA versions provided the vector and matrix sizes are large enough. See the section [Programming Notes for BLAS Using NVIDIA](#) for further details.

*Table 9.2: Level 2 and Level 3 Basic Linear Algebra Subprograms – **GREEN** Denotes NVIDIA Version Available*

| Operation                                    | Real         | Double       | Complex      | Double Complex | Pg.  |
|----------------------------------------------|--------------|--------------|--------------|----------------|------|
| Matrix-Vector Multiply, General              | <b>SGEMV</b> | <b>DGEMV</b> | <b>CGEMV</b> | <b>ZGEMV</b>   | 1592 |
| Matrix-Vector Multiply, Banded               | <b>SGBMV</b> | <b>DGBMV</b> | <b>CGBMV</b> | <b>ZGBMV</b>   | 1592 |
| Matrix-Vector Multiply, Hermitian            |              |              | CHEMV        | ZHEMV          | 1592 |
| Matrix-Vector Multiply, Hermitian and Banded |              |              | CHBMV        | ZHBMV          | 1593 |
| Matrix-Vector Multiply Symmetric and Real    | SSYMV        | DSYMV        |              |                | 1593 |
| Matrix-Vector Multiply, Symmetric and Banded | SSBMV        | DSBMV        |              |                | 1593 |

| Operation                                                         | Real   | Double | Complex | Double Complex | Pg.  |
|-------------------------------------------------------------------|--------|--------|---------|----------------|------|
| Matrix-Vector Multiply, Triangular                                | STRMV  | DTRMV  | CTRMV   | ZTRMV          | 1593 |
| Matrix-Vector Multiply, Triangular and Banded                     | STBMV  | DTBMV  | CTBMV   | ZTBMV          | 1594 |
| Matrix-Vector Multiply, Packed Triangular                         | STPMV  | DTPMV  | CTPMV   | ZTPMV          | 1594 |
| Matrix-Vector Solve, Triangular                                   | STRSV  | DTRSV  | CTRSV   | ZTRSV          | 1594 |
| Matrix-Vector Solve, Packed Triangular                            | STPSV  | DTPSV  | CTPSV   | ZTPSV          | 1595 |
| Matrix-Vector Solve, Triangular and Banded                        | STBSV  | DTBSV  | CTBSV   | ZTBSV          | 1595 |
| Rank-One Matrix Update, General and Real                          | SGER   | DGER   |         |                | 1595 |
| Rank-One Matrix Update, General, Complex and Transpose            |        |        | CGERU   | ZGERU          | 1595 |
| Rank-One Matrix Update, General, Complex, and Conjugate Transpose |        |        | CGERC   | ZGERC          | 1596 |
| Rank-One Matrix Update, Hermitian and Conjugate Transpose         |        |        | CHER    | ZHER           | 1596 |
| Rank-Two Matrix Update, Hermitian and Conjugate Transpose         |        |        | CHER2   | ZHER2          | 1596 |
| Rank-Two Matrix Update, Packed and Conjugate Transpose            |        |        | CHPR2   | ZHPR2          |      |
| Rank-One Matrix Update, Symmetric and Real                        | SSYR   | DSYR   |         |                | 1597 |
| Rank-Two Matrix Update, Symmetric and Real                        | SSYR2  | DSYR2  |         |                | 1597 |
| Rank-Two Matrix Update, Packed Symmetric and Real                 | SSPR2  | DSPR2  |         |                | 1597 |
| Packed Symmetric or Hermitian Matrix-Vector Multiply              | SSPMV  | DSPMV  | CHPMV   | ZHPMV          | 1593 |
| Packed Symmetric or Hermitian Outer Product Update                | SSPR   | DSPR   | CHPR    | ZHPR           | 1597 |
| Matrix--Matrix Multiply, General                                  | SGEMM  | DGEMM  | CGEMM   | ZGEMM          | 1598 |
| Matrix-Matrix Multiply, Symmetric                                 | SSYMM  | DSYMM  | CSYMM   | ZSYMM          | 1598 |
| Matrix-Matrix Multiply, Hermitian                                 |        |        | CHEMM   | ZHEMM          | 1598 |
| Rank - $k$ Update, Symmetric                                      | SSYRK  | DSYRK  | CSYRK   | ZSYRK          | 1598 |
| Rank - $k$ Update, Hermitian                                      |        |        | CHERK   | ZHERK          | 1599 |
| Rank - $2k$ Update, Symmetric                                     | SSYR2K | DSYR2K | CSYR2K  | ZSYR2K         | 1599 |
| Rank - $2k$ Update, Hermitian                                     |        |        | CHER2K  | ZHER2K         | 1599 |
| Matrix-Matrix Multiply, Triangular                                | STRMM  | DTRMM  | CTRMM   | ZTRMM          | 1600 |

| Operation                       | Real  | Double | Complex | Double Complex | Pg.  |
|---------------------------------|-------|--------|---------|----------------|------|
| Matrix-Matrix solve, Triangular | STRSM | DTRSM  | CTRSM   | ZTRSM          | 1600 |

### Matrix–Vector Multiply, General

```
CALL SGEMV (TRANS, M, N, SALPHA, SA, LDA, SX, INCX, SBETA, SY, INCY)
CALL DGEMV (TRANS, M, N, DALPHA, DA, LDA, DX, INCX, DBETA, DY, INCY)
CALL CGEMV (TRANS, M, N, CALPHA, CA, LDA, CX, INCX, CBETA, CY, INCY)
CALL ZGEMV (TRANS, M, N, ZALPHA, ZA, LDA, ZX, INCX, ZBETA, ZY, INCY)
```

For all data types,  $A$  is an  $M \times N$  matrix. These subprograms set  $y$  to one of the expressions:  
 $y \leftarrow \alpha Ax + \beta y$ ,  $y \leftarrow \alpha A^T x + \beta y$ , or for complex data,

$$y \leftarrow \alpha \bar{A}^T + \beta y$$

The character flag TRANS determines the operation.

### Matrix–Vector Multiply, Banded

```
CALL SGBMV (TRANS, M, N, NLCA, NUCA, SALPHA, SA, LDA, SX, INCX, SBETA, SY, INCY)
CALL DGBMV (TRANS, M, N, NLCA, NUCA, DALPHA, DA, LDA, DX, INCX, DBETA, DY, INCY)
CALL CGBMV (TRANS, M, N, NLCA, NUCA, CALPHA, CA, LDA, CX, INCX, BETA, CY, INCY)
CALL ZGBMV (TRANS, M, N, NLCA, NUCA, ZALPHA, ZA, LDA, ZX, INCX, ZBETA, ZY, INCY)
```

For all data types,  $A$  is an  $M \times N$  matrix with NLCA lower codiagonals and NUCA upper codiagonals. The matrix is stored in band storage mode. These subprograms set  $y$  to one of the expressions:  $y \leftarrow \alpha Ax + \beta y$ ,  $y \leftarrow \alpha A^T x + \beta y$ , or for complex data,

$$y \leftarrow \alpha \bar{A}^T x + \beta y$$

The character flag TRANS determines the operation.

### Matrix-Vector Multiply, Hermitian

```
CALL CHEMV (UPLO, N, CALPHA, CA, LDA, CX, INCX, CBETA, CY, INCY)
CALL ZHEMV (UPLO, N, ZALPHA, ZA, LDA, ZX, INCX, ZBETA, ZY, INCY)
```

For complex types,  $A$  is an  $N \times N$  matrix. These subprograms set  $y \leftarrow \alpha Ax + \beta y$  where  $A$  is an Hermitian matrix. The matrix  $A$  is either referenced using the upper or lower triangular part. The character flag UPLO determines the part used.

### Matrix-Vector Multiply, Packed Hermitian

```
CALL CHPMV (UPLO, N, CALPHA, CAP, CX, INCX, CBETA, CY, INCY)
CALL ZHPMV (UPLO, N, ZALPHA, ZAP, ZX, INCX, ZBETA, ZY, INCY)
```

For complex types,  $A$  is an  $N \times N$  matrix. These subprograms set  $y \leftarrow \alpha Ax + \beta y$  where  $A$  is an Hermitian matrix. The matrix  $A$  is either referenced using the packed upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Vector Multiply, Hermitian and Banded

```
CALL CHBMV (UPLO, N, NCODA, CALPHA, CA, LDA, CX, INCX, CBETA, CY, INCY)
CALL ZHBMV (UPLO, N, NCODA, ZALPHA, ZA, LDA, ZX, INCX, ZBETA, ZY, INCY)
```

For all data types,  $A$  is an  $N \times N$  matrix with `NCODA` codiagonals. The matrix is stored in band Hermitian storage mode. These subprograms set  $y \leftarrow \alpha Ax + \beta y$ . The matrix  $A$  is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Vector Multiply, Symmetric and Real

```
CALL SSMV (UPLO, N, SALPHA, SA, LDA, SX, INCX, SBETA, SY, INCY)
CALL DSMV (UPLO, N, DALPHA, DA, LDA, DX, INCX, DBETA, DY, INCY)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $y \leftarrow \alpha Ax + \beta y$  where  $A$  is a symmetric matrix. The matrix  $A$  is either referenced using the upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Vector Multiply, Packed Symmetric and Real

```
CALL SSPMV (UPLO, N, SALPHA, SAP, SX, INCX, SBETA, SY, INCY)
CALL DSPMV (UPLO, N, DALPHA, DAP, DX, INCX, DBETA, DY, INCY)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $y \leftarrow \alpha Ax + \beta y$  where  $A$  is a [packed symmetric matrix](#). The matrix  $A$  is either referenced using the packed upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Vector Multiply, Symmetric and Banded

```
CALL SSBMV (UPLO, N, NCODA, SALPHA, SA, LDA, SX, INCX, SBETA, SY, INCY)
CALL DSBMV (UPLO, N, NCODA, DALPHA, DA, LDA, DX, INCX, DBETA, DY, INCY)
```

For all data types,  $A$  is an  $N \times N$  matrix with `NCODA` codiagonals. The matrix is stored in band symmetric storage mode. These subprograms set  $y \leftarrow \alpha Ax + \beta y$ . The matrix  $A$  is either referenced using its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Vector Multiply, Triangular

```
CALL STRMV (UPLO, TRANS, DIAGNL, N, SA, LDA, SX, INCX)
CALL DTRMV (UPLO, TRANS, DIAGNL, N, DA, LDA, DX, INCX)
CALL CTRMV (UPLO, TRANS, DIAGNL, N, CA, LDA, CX, INCX)
CALL ZTRMV (UPLO, TRANS, DIAGNL, N, ZA, LDA, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  triangular matrix. These subprograms set  $x$  to one of the expressions:  $x \leftarrow Ax$ ,  $x \leftarrow A^T x$ , or for complex data,

$$x \leftarrow \bar{A}^T x$$

The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

### Matrix-Vector Multiply, Packed Triangular

```
CALL STPMV (UPLO, TRANS, DIAGNL, N, SAP, SX, INCX)
CALL DTPMV (UPLO, TRANS, DIAGNL, N, DAP, DX, INCX)
CALL CTPMV (UPLO, TRANS, DIAGNL, N, CAP, CX, INCX)
CALL ZTPMV (UPLO, TRANS, DIAGNL, N, ZAP, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  [packed triangular matrix](#). These subprograms set  $x$  to one of the expressions:  $x \leftarrow Ax$ ,  $x \leftarrow A^T x$ , or for complex data,

$$x \leftarrow \bar{A}^T x$$

The matrix  $A$  is either referenced using the packed upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

### Matrix-Vector Multiply, Triangular and Banded

```
CALL STBMV (UPLO, TRANS, DIAGNL, N, NCODA, SA, LDA, SX, INCX)
CALL DTBMV (UPLO, TRANS, DIAGNL, N, NCODA, DA, LDA, DX, INCX)
CALL CTBMV (UPLO, TRANS, DIAGNL, N, NCODA, CA, LDA, CX, INCX)
CALL ZTBMV (UPLO, TRANS, DIAGNL, N, NCODA, ZA, LDA, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  matrix with `NCODA` codiagonals. The matrix is stored in band triangular storage mode. These subprograms set  $x$  to one of the expressions:  $x \leftarrow Ax$ ,  $x \leftarrow A^T x$ , or for complex data,

$$x \leftarrow \bar{A}^T x$$

The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `UPLO`, `TRANS`, and `DIAGNL` determine the part of the matrix used and the operation performed.

### Matrix-Vector Solve, Triangular

```
CALL STRSV (UPLO, TRANS, DIAGNL, N, SA, LDA, SX, INCX)
CALL DTRSV (UPLO, TRANS, DIAGNL, N, DA, LDA, DX, INCX)
CALL CTRSV (UPLO, TRANS, DIAGNL, N, CA, LDA, CX, INCX)
CALL ZTRSV (UPLO, TRANS, DIAGNL, N, ZA, LDA, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  triangular matrix. These subprograms solve  $x$  for one of the expressions:  $x \leftarrow A^{-1}x$ ,  $x \leftarrow (A^{-1})^T x$ , or for complex data,

$$x \leftarrow (\bar{A}^T)^{-1} x \equiv (A^H)^{-1} x$$

The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags UPLO, TRANS, and DIAGNL determine the part of the matrix used and the operation performed.

### Matrix-Vector Solve, Triangular and Banded

```
CALL STBSV (UPLO, TRANS, DIAGNL, N, NCODA, SA, LDA, SX, INCX)
CALL DTBSV (UPLO, TRANS, DIAGNL, N, NCODA, DA, LDA, DX, INCX)
CALL CTBSV (UPLO, TRANS, DIAGNL, N, NCODA, CA, LDA, CX, INCX)
CALL ZTBSV (UPLO, TRANS, DIAGNL, N, NCODA, ZA, LDA, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  triangular matrix with NCODA codiagonals. The matrix is stored in band triangular storage mode. These subprograms solve  $x$  for one of the expressions:  $x \leftarrow A^{-1}x$ ,  $x \leftarrow (A^{-T})^{-1}x$ , or for complex data,

$$x \leftarrow (\bar{A}^T)^{-1} x \equiv (A^H)^{-1} x$$

The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags UPLO, TRANS, and DIAGNL determine the part of the matrix used and the operation performed.

### Matrix-Vector Solve, Packed Triangular

```
CALL STPSV (UPLO, TRANS, DIAGNL, N, SAP, SX, INCX)
CALL DTPSV (UPLO, TRANS, DIAGNL, N, DAP, DX, INCX)
CALL CTPSV (UPLO, TRANS, DIAGNL, N, CAP, CX, INCX)
CALL ZTPSV (UPLO, TRANS, DIAGNL, N, ZAP, ZX, INCX)
```

For all data types,  $A$  is an  $N \times N$  **packed triangular matrix**. These subprograms solve  $x$  for one of the expressions:  $x \leftarrow A^{-1}x$ ,  $x \leftarrow (A^{-1})^T x$ , or for complex data,

$$x \leftarrow (\bar{A}^T)^{-1} x \equiv (A^H)^{-1} x$$

The matrix  $A$  is either referenced using its packed upper or lower triangular part and is unit or nonunit triangular. The character flags UPLO, TRANS, and DIAGNL determine the part of the matrix used and the operation performed.

### Rank-One Matrix Update, General and Real

```
CALL SGER (M, N, SALPHA, SX, INCX, SY, INCY, SA, LDA)
CALL DGER (M, N, DALPHA, DX, INCX, DY, INCY, DA, LDA)
```

For all data types,  $A$  is an  $M \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha xy^T$ .

### Rank-One Matrix Update, General, Complex, and Transpose

```
CALL CGERU (M, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)
CALL ZGERU (M, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)
```

For all data types,  $A$  is an  $M \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha xy^T$ .

### Rank-One Matrix Update, General, Complex, and Conjugate Transpose

CALL CGERC (M, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)

CALL ZGERC (M, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)

For all data types,  $A$  is an  $M \times N$  matrix. These subprograms set

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

### Rank-One Matrix Update, Hermitian and Conjugate Transpose

CALL CHER (UPLO, N, SALPHA, CX, INCX, CA, LDA)

CALL ZHER (UPLO, N, DALPHA, ZX, INCX, ZA, LDA)

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set

$$A \leftarrow A + \alpha x \bar{x}^T$$

where  $A$  is Hermitian. The matrix  $A$  is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used. CAUTION: Notice the scalar parameter  $\alpha$  is real, and the data in the matrix and vector are complex.

### Rank-One Matrix Update, Packed Hermitian and Conjugate Transpose

CALL CHPR (UPLO, N, SALPHA, CX, INCX, CAP)

CALL ZHPR (UPLO, N, DALPHA, ZX, INCX, ZAP)

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set

$$A \leftarrow A + \alpha x \bar{x}^T$$

where  $A$  is [packed Hermitian](#). The matrix  $A$  is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used. CAUTION: Notice the scalar parameter  $\alpha$  is real, and the data in the matrix and vector are complex.

### Rank-Two Matrix Update, Hermitian and Conjugate Transpose

CALL CHER2 (UPLO, N, CALPHA, CX, INCX, CY, INCY, CA, LDA)

CALL ZHER2 (UPLO, N, ZALPHA, ZX, INCX, ZY, INCY, ZA, LDA)

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set

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

where  $A$  is an Hermitian matrix. The matrix  $A$  is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-Two Matrix Update, Packed Hermitian and Conjugate Transpose

CALL CHPR2 (UPLO, N, CALPHA, CX, INCX, CY, INCY, CAP)

CALL ZHPR2 (UPLO, N, ZALPHA, ZX, INCX, ZY, INCY, ZAP)

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set

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

where  $A$  is a packed Hermitian matrix. The matrix  $A$  is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-One Matrix Update, Symmetric and Real

```
CALL SSYR (UPLO, N, SALPHA, SX, INCX, SA, LDA)
CALL DSYR (UPLO, N, DALPHA, DX, INCX, DA, LDA)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha x x^T$  where  $A$  is a symmetric matrix. The matrix  $A$  is either referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-One Matrix Update, Packed Symmetric and Real

```
CALL SSPR (UPLO, N, SALPHA, SX, INCX, SAP)
CALL DSPR (UPLO, N, DALPHA, DX, INCX, DAP)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha x x^T$  where  $A$  is a packed symmetric matrix. The matrix  $A$  is either referenced using the packed upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-One Matrix Update, Packed Hermitian

```
CALL CHPR (UPLO, N, SALPHA, CX, INCX, CAP)
CALL ZHPR (UPLO, N, DALPHA, ZX, INCX, ZAP)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha x x^T$  where  $A$  is a packed Hermitian matrix. The matrix  $A$  is either referenced using the packed upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-Two Matrix Update, Symmetric and Real

```
CALL SSYR2 (UPLO, N, SALPHA, SX, INCX, SY, INCY, SA, LDA)
CALL DSYR2 (UPLO, N, DALPHA, DX, INCX, DY, INCY, DA, LDA)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha x y^T + \alpha y x^T$  where  $A$  is a symmetric matrix. The matrix  $A$  is referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Rank-Two Matrix Update, Packed Symmetric and Real

```
CALL SSPR2 (UPLO, N, SALPHA, SX, INCX, SY, INCY, SAP)
CALL DSPR2 (UPLO, N, DALPHA, DX, INCX, DY, INCY, DAP)
```

For all data types,  $A$  is an  $N \times N$  matrix. These subprograms set  $A \leftarrow A + \alpha x y^T + \alpha y x^T$  where  $A$  is a packed symmetric matrix. The matrix  $A$  is referenced by its upper or lower triangular part. The character flag `UPLO` determines the part used.

### Matrix-Matrix Multiply, General

CALL SGEMM (TRANSA, TRANSB, M, N, K, SALPHA, SA, LDA, SB, LDB, SBETA, SC, LDC)

CALL DGEMM (TRANSA, TRANSB, M, N, K, DALPHA, DA, LDA, DB, LDB, DBETA, DC, LDC)

CALL CGEMM (TRANSA, TRANSB, M, N, K, CALPHA, CA, LDA, CB, LDB, CBETA, CC, LDC)

CALL ZGEMM (TRANSA, TRANSB, M, N, K, ZALPHA, ZA, LDA, ZB, LDB, ZBETA, ZC, LDC)

For all data types, these subprograms set  $C_{M \times N}$  to one of the expressions:

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

$$\text{or for complex data, } C \leftarrow \alpha A \bar{B}^T + \beta C, C \leftarrow \alpha \bar{A}^T B + \beta C, C \leftarrow \alpha A^T \bar{B}^T + \beta C,$$

$$C \leftarrow \alpha \bar{A}^T B^T + \beta C, C \leftarrow \alpha \bar{A}^T \bar{B}^T + \beta C$$

The character flags `TRANSA` and `TRANSB` determine the operation to be performed. Each matrix product has dimensions that follow from the fact that  $C$  has dimension  $M \times N$ .

### Matrix-Matrix Multiply, Symmetric

CALL SSYMM (SIDE, UPLO, M, N, SALPHA, SA, LDA, SB, LDB, SBETA, SC, LDC)

CALL DSYMM (SIDE, UPLO, M, N, DALPHA, DA, LDA, DB, LDB, DBETA, DC, LDC)

CALL CSYMM (SIDE, UPLO, M, N, CALPHA, CA, LDA, CB, LDB, CBETA, CC, LDC)

CALL ZSYMM (SIDE, UPLO, M, N, ZALPHA, ZA, LDA, ZB, LDB, ZBETA, ZC, LDC)

For all data types, these subprograms set  $C_{M \times N}$  to one of the expressions:  $C \leftarrow \alpha AB + \beta C$  or  $C \leftarrow \alpha BA + \beta C$ , where  $A$  is a symmetric matrix. The matrix  $A$  is referenced either by its upper or lower triangular part. The character flags `SIDE` and `UPLO` determine the part of the matrix used and the operation performed.

### Matrix-Matrix Multiply, Hermitian

CALL CHEMM (SIDE, UPLO, M, N, CALPHA, CA, LDA, CB, LDB, CBETA, CC, LDC)

CALL ZHEMM (SIDE, UPLO, M, N, ZALPHA, ZA, LDA, ZB, LDB, ZBETA, ZC, LDC)

For all data types, these subprograms set  $C_{M \times N}$  to one of the expressions:  $C \leftarrow \alpha AB + \beta C$  or  $C \leftarrow \alpha BA + \beta C$ , where  $A$  is an Hermitian matrix. The matrix  $A$  is referenced either by its upper or lower triangular part. The character flags `SIDE` and `UPLO` determine the part of the matrix used and the operation performed.

### Rank- $k$ Update, Symmetric

CALL SSYRK (UPLO, TRANS, N, K, SALPHA, SA, LDA, SBETA, SC, LDC)

CALL DSYRK (UPLO, TRANS, N, K, DALPHA, DA, LDA, DBETA, DC, LDC)

CALL CSYRK (UPLO, TRANS, N, K, CALPHA, CA, LDA, CBETA, CC, LDC)

CALL ZSYRK (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZBETA, ZC, LDC)

For all data types, these subprograms set  $C_{M \times N}$  to one of the expressions:  $C \leftarrow \alpha AA^T + \beta C$  or  $C \leftarrow \alpha A^T A + \beta C$ . The matrix  $C$  is referenced either by its upper or lower triangular part. The

character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. In subprogram `CSYRK` and `ZSYRK`, only values 'N' or 'T' are allowed for `TRANS`; 'C' is not acceptable.

### Rank- $k$ Update, Hermitian

```
CALL CHERK (UPLO, TRANS, N, K, SALPHA, CA, LDA, SBETA, CC, LDC)
CALL ZHERK (UPLO, TRANS, N, K, DALPHA, ZA, LDA, DBETA, ZC, LDC)
```

For all data types, these subprograms set  $C_{N \times N}$  to one of the expressions:

$$C \leftarrow \alpha A \bar{A}^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T A + \beta C$$

The matrix  $C$  is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. CAUTION: Notice the scalar parameters  $\alpha$  and  $\beta$  are real, and the data in the matrices are complex. Only values 'N' or 'C' are allowed for `TRANS`; 'T' is not acceptable.

### Rank- $2k$ Update, Symmetric

```
CALL SSYR2K (UPLO, TRANS, N, K, SALPHA, SA, LDA, SB, LDB, SBETA, SC, LDC)
CALL DSYR2K (UPLO, TRANS, N, K, DALPHA, DA, LDA, DB, LDB, DBETA, DC, LDC)
CALL CSYR2K (UPLO, TRANS, N, K, CALPHA, CA, LDA, CB, LDB, CBETA, CC, LDC)
CALL ZSYR2K (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZB, LDB, ZBETA, ZC, LDC)
```

For all data types, these subprograms set  $C_{N \times N}$  to one of the expressions:

$$C \leftarrow \alpha A B^T + \alpha \beta A^T + \beta C \text{ or } C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

The matrix  $C$  is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. In subprogram `CSYR2K` and `ZSYR2K`, only values 'N' or 'T' are allowed for `TRANS`; 'C' is not acceptable.

### Rank- $2k$ Update, Hermitian

```
CALL CHER2K (UPLO, TRANS, N, K, CALPHA, CA, LDA, CB, LDB, SBETA, CC, LDC)
CALL ZHER2K (UPLO, TRANS, N, K, ZALPHA, ZA, LDA, ZB, LDB, DBETA, ZC, LDC)
```

For all data types, these subprograms set  $C_{N \times N}$  to one of the expressions:

$$C \leftarrow \alpha A \bar{B}^T + \bar{\alpha} B \bar{A}^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T B + \bar{\alpha} \bar{B}^T A + \beta C$$

The matrix  $C$  is referenced either by its upper or lower triangular part. The character flags `UPLO` and `TRANS` determine the part of the matrix used and the operation performed. CAUTION: Notice the scalar parameter  $\beta$  is real, and the data in the matrices are complex. In subprogram `CHER2K` and `ZHER2K`, only values 'N' or 'C' are allowed for `TRANS`; 'T' is not acceptable.

### Matrix-Matrix Multiply, Triangular

```
CALL STRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, SALPHA, SA, LDA, SB, LDB)
CALL DTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, DALPHA, DA, LDA, DB, LDB)
CALL CTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, CALPHA, CA, LDA, CB, LDB)
CALL ZTRMM (SIDE, UPLO, TRANSA, DIAGNL, M, N, ZALPHA, ZA, LDA, ZB, LDB)
```

For all data types, these subprograms set  $B_{M \times N}$  to one of the expressions:

$$B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, B \leftarrow \alpha BA, B \leftarrow \alpha BA^T,$$

or for complex data,

$$B \leftarrow \alpha \bar{A}^T B, \text{ or } B \leftarrow \alpha B \bar{A}^T$$

where  $A$  is a triangular matrix. The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `SIDE`, `UPLO`, `TRANSA`, and `DIAGNL` determine the part of the matrix used and the operation performed.

### Matrix-Matrix Solve, Triangular

```
CALL STRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, SALPHA, SA, LDA, SB, LDB)
CALL DTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, DALPHA, DA, LDA, DB, LDB)
CALL CTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, CALPHA, CA, LDA, CB, LDB)
CALL ZTRSM (SIDE, UPLO, TRANSA, DIAGNL, M, N, ZALPHA, ZA, LDA, ZB, LDB)
```

For all data types, these subprograms set  $B_{M \times N}$  to one of the expressions:

$$B \leftarrow \alpha A^{-1} B, B \leftarrow \alpha B A^{-1}, B \leftarrow \alpha (A^{-1})^T B, B \leftarrow \alpha B (A^{-1})^T,$$

or for complex data,

$$B \leftarrow \alpha (\bar{A}^T)^{-1} B, \text{ or } B \leftarrow \alpha B (\bar{A}^T)^{-1}$$

where  $A$  is a triangular matrix. The matrix  $A$  is either referenced using its upper or lower triangular part and is unit or nonunit triangular. The character flags `SIDE`, `UPLO`, `TRANSA`, and `DIAGNL` determine the part of the matrix used and the operation performed.

---

## Programming Notes for BLAS Using NVIDIA

This reference material is intended for users who want to use the computational resources of their NVIDIA GPU board for BLAS. Users who do not have the NVIDIA GPU board can ignore this section.

### Rationale, General Algorithm and an Example

NVIDIA Corp. implemented certain Level 1, 2 and 3 BLAS in their Library, *CUDA CUBLAS Library*, V3.1, July, 2010. The NVIDIA external names and argument protocols are different from the equivalent Fortran names and argument addressing. See [Table 9.2](#) for names marked in the color **GREEN**. IMSL has written these marked Fortran BLAS so that they call equivalent NVIDIA C language codes from the *CUBLAS* library. No direct use or knowledge of C is required by a Fortran programmer in order to take advantage of these codes. It is necessary that a user code or application package be compiled with a Fortran 2003 compiler that has implemented the C Interoperability Standard feature. See *The Fortran 2003 Handbook*, Adams, et al., p. 561. IMSL's use of this feature is the key to providing a portable version of these Fortran-callable IMSL/NVIDIA BLAS. The program or application is then compiled and linked using IMSL and NVIDIA libraries that contain these BLAS.

---

**Note:** An NVIDIA Graphics Processing Unit (GPU) is required to take advantage of the BLAS.

---

The strategy for using the attached NVIDIA GPU is given by the following algorithm:

- *If* the maximum of vector or matrix dimensions are *larger than* a switchover array size, **NSTART**, and NVIDIA provides a CUBLAS code, *then*
- Copy the required vector and matrix data from the CPU to the GPU
- Compute the result on the GPU
- Copy the result from the GPU to the CPU
- *Else*, use the IMSL equivalent version of the BLAS routine that does not use the GPU.

Normally a code that calls a IMSL/NVIDIA BLAS code does not have to be aware of the copy steps or the switchover size, **NSTART**. These are hidden from the user code. In the first algorithm step, a working block is allocated on the GPU for each array argument. A table within the IMSL module, **CUBLAS\_LIBRARY**, records the sizes and GPU addresses of these blocks. If the sizes are too small for the current problem size and data type the blocks are reallocated to be of adequate size. The same working block on the GPU may be used for many calls to the IMSL/NVIDIA BLAS. The IMSL versions of the BLAS also allow a user to define individual values of **NSTART** for each routine. This is important because using the GPU may be slower than using a CPU Fortran version until a switchover array size is reached. Thereafter the GPU version is typically faster and increasingly much faster as the problem size increases. The default value of **NSTART=32** is used for each vector/matrix argument of each routine but it may not be optimal. This default allows the routines to function correctly without initial attention to this value.

The user can change the default switchover value for all IMSL/NVIDIA BLAS vector/matrix arguments by setting `NSTART` to the desired value prior to calling the BLAS routine. Additionally, users can reset this value for each individual vector/matrix argument of the routines listed in [Table 9.2](#) and marked with the color `GREEN` by using the IMSL routine `CUBLAS_SET(...)`. **Note** that `CUBLAS_SET` cannot be used prior to an initial call to a BLAS code. The switchover values can be obtained using the IMSL routine `CUBLAS_GET(...)`.

The floating point results obtained using the CPU vs. the GPU will likely differ in units of the low order bits in each component. These differences come from non-equivalent strategies of floating point arithmetic and rounding modes that are implemented in the NVIDIA board. This can be an important detail when comparing results for purposes of benchmarking or code regression. Generally either result should be acceptable for numerical work.

As an added feature, the user can flag when the data values for a vector or matrix are present on the GPU and hence suppress the IMSL/NVIDIA BLAS code from first copying the data. This is often important since the data movement from the CPU to the GPU may be a significant part of the computation time. If there is no indication that the data is present, it is copied from the CPU to the GPU each time a routine is called. The necessity of copying for each use of a BLAS code depends on the application. Valid results are always copied back from the GPU to the CPU memory. *The indication that data for that positional array argument requires no initial copy step is that the switchover value for that array argument is negative. The absolute value is used as the switchover value. Caution: Be sure and reset this to a positive value when the argument requires an initial copy step.*

In [Tables 9.3-9.5](#), we list an *enumeration* that includes the routines in [Table 9.2](#) marked with the color `GREEN`. Note the prefix to each name joined with the string `'CUDABLAS_'`. There are enumerated names that currently do not use the NVIDIA hardware. They are included in anticipation of future additions that will use the `CUBLAS` library.

There are four utility routines provided in the IMSL module `CUDABLAS_LIBRARY` that can be used to help manage the use of NVIDIA BLAS. These utilities appear in [Table 9.7](#) and are described in more detail in the [routines section](#) of these notes.

For example, to set the value at **500** wherein the GPU is first used for the Level-2 routine `'SGEMV'` first positional array argument, `'A(*,*)'`, i.e. `Array_Arg = 1`, execute the code:

```
USE CUDABLAS_LIBRARY
 INTEGER ISWITCH, Array_Arg
 ...
 ISWITCH=500

 Array_Arg = 1

 ! Switch to using GPU when largest size of A(*,*) > 500.

 CALL CUBLAS_SET(CUDABLAS_SGEMV, Array_Arg, ISWITCH)
```

When the positional array argument, `'A(*,*)'` does not have to be copied for each subsequent use of `'SGEMV'`:

```
USE CUDABLAS_LIBRARY
 INTEGER ISWITCH, Array_Arg

 Array_Arg = 1
```

```
ISWITCH=CUBLAS_GET(CUDABLAS_SGEMV, Array_Arg)
```

*! Avoid copying data from CPU to GPU for subsequent calls to 'SGEMV'*

```
CALL CUBLAS_SET(CUDABLAS_SGEMV, Array_Arg, -abs(ISWITCH))
```

*! Make several calls to 'SGEMV' with A(\*,\*) maintained unchanged on the GPU.*

*! Reset flag for copying A(\*,\*) when this matrix-vector product sequence is completed.*

```
CALL CUBLAS_SET(CUDABLAS_SGEMV, Array_Arg, abs(ISWITCH))
```

Some NVIDIA hardware does not have working double precision versions of BLAS because there is no double precision arithmetic available. However, the double precision code itself is part of the *CUDA CUBLAS* library. It will appear to execute even though it will not give correct results when the device has no double precision arithmetic. When the IMSL software detects that the correct results are not returned, a warning error message will be printed. The user may instruct the application to henceforth use the Fortran code by setting the switchover value to zero. For example, if it is known that the hardware does not support `DOUBLE PRECISION`, then a code that has calls to 'DGEMM' will use an alternate version of this routine. Therefore, ignoring the error message and continuing the code will result in using the alternate version to compute the result. That code would include:

```
USE CUDABLAS_LIBRARY
```

*! Flag first array argument A(\*,\*) to avoid use of the GPU for DGEMM:*

```
CALL CUBLAS_SET(CUDABLAS_DGEMM, 1, 0)
```

If it is necessary to know if the GPU or the CPU version of 'SGEMM' was used following a call to that code, the inquiry code would include:

```
USE CUDABLAS_LIBRARY
```

*! Get the current status for the last call to SGEMM with the INTEGER function*

*! CUBLAS\_GET. The value ISWITCH=0 if an alternate was used, and ISWITCH=1 if the*

*! GPU was used.*

```
ISWITCH = CUBLAS_GET(CUDABLAS_SGEMM, 4)
```

## Enumeration of IMSL/NVIDIA BLAS

*Table 9.3. Enumeration of Level-1 BLAS*

|                 |                 |                |                |
|-----------------|-----------------|----------------|----------------|
| CUDABLAS_SROTG  | CUDABLAS_DROTG  | CUDABLAS_CROTG | CUDABLAS_ZROTG |
| CUDABLAS_SROTMG | CUDABLAS_DROTMG |                |                |
| CUDABLAS_SROT   | CUDABLAS_DROT   | CUDABLAS_CROT  | CUDABLAS_ZROT  |
| CUDABLAS_SROTM  | CUDABLAS_DROTM  | CUDABLAS_CSROT | CUDABLAS_ZSROT |
| CUDABLAS_SSWAP  | CUDABLAS_DSWAP  | CUDABLAS_CSWAP | CUDABLAS_ZSWAP |
| CUDABLAS_SCOPY  | CUDABLAS_DCOPY  | CUDABLAS_CCOPY | CUDABLAS_ZCOPY |
| CUDABLAS_SAXPY  | CUDABLAS_DAXPY  | CUDABLAS_CAXPY | CUDABLAS_ZAXPY |
| CUDABLAS_SDOT   | CUDABLAS_DDOT   | CUDABLAS_CDOTC | CUDABLAS_ZDOTC |

|                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|
| CUDABLAS_SSDOT  | CUDABLAS_DSDOT  | CUDABLAS_CDOTU  | CUDABLAS_ZDOTU  |
| CUDABLAS_SSCAL  | CUDABLAS_DSCAL  | CUDABLAS_CSCAL  | CUDABLAS_ZSCAL  |
|                 |                 | CUDABLAS_CSSCAL | CUDABLAS_ZSSCAL |
| CUDABLAS_SNRM2  | CUDABLAS_DNRM2  | CUDABLAS_SCNRM2 | CUDABLAS_DZNRM2 |
| CUDABLAS_SASUM  | CUDABLAS_DASUM  | CUDABLAS_SCASUM | CUDABLAS_DZASUM |
| CUDABLAS_ISAMIN | CUDABLAS_IDAMIN | CUDABLAS_ICAMIN | CUDABLAS_IZAMIN |
| CUDABLAS_ISAMAX | CUDABLAS_IDAMAX | CUDABLAS_ICAMAX | CUDABLAS_IZAMAX |

**Table 9.4. Enumeration of Level-2 BLAS**

|                |                |                |                |
|----------------|----------------|----------------|----------------|
| CUDABLAS_SGEMV | CUDABLAS_DGEMV | CUDABLAS_CGEMV | CUDABLAS_ZGEMV |
| CUDABLAS_SGBMV | CUDABLAS_DGBMV | CUDABLAS_CGBMV | CUDABLAS_ZGBMV |
| CUDABLAS_SSYMV | CUDABLAS_DSYMV | CUDABLAS_CHEMV | CUDABLAS_ZHEMV |
| CUDABLAS_SSBMV | CUDABLAS_DSBMV | CUDABLAS_CHBMV | CUDABLAS_ZHBMV |
| CUDABLAS_SSPMV | CUDABLAS_DSPMV | CUDABLAS_CHPMV | CUDABLAS_ZHPMV |
| CUDABLAS_STRMV | CUDABLAS_DTRMV | CUDABLAS_CTRMV | CUDABLAS_ZTRMV |
| CUDABLAS_STBMV | CUDABLAS_DTBMV | CUDABLAS_CTBMV | CUDABLAS_ZTBMV |
| CUDABLAS_STPMV | CUDABLAS_DTPMV | CUDABLAS_CTPMV | CUDABLAS_ZTPMV |
| CUDABLAS_STRSV | CUDABLAS_DTRSV | CUDABLAS_CTRSV | CUDABLAS_ZTRSV |
| CUDABLAS_STBSV | CUDABLAS_DTBSV | CUDABLAS_CTBSV | CUDABLAS_ZTBSV |
| CUDABLAS_STPSV | CUDABLAS_DTPSV | CUDABLAS_CTPSV | CUDABLAS_ZTPSV |
| CUDABLAS_SGER  | CUDABLAS_DGER  | CUDABLAS_CGERU | CUDABLAS_ZGERU |
|                |                | CUDABLAS_CGERC | CUDABLAS_ZGERC |
| CUDABLAS_SSYR  | CUDABLAS_DSYR  | CUDABLAS_CHER  | CUDABLAS_ZHER  |
| CUDABLAS_SSYR2 | CUDABLAS_DSYR2 | CUDABLAS_CHER2 | CUDABLAS_ZHER2 |
| CUDABLAS_SSPR  | CUDABLAS_DSPR  | CUDABLAS_CHPR  | CUDABLAS_ZHPR  |
| CUDABLAS_SSPR2 | CUDABLAS_DSPR2 | CUDABLAS_CHPR2 | CUDABLAS_ZHPR2 |

**Table 9.5. Enumeration of Level-3 BLAS**

|                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|
| CUDABLAS_SGEMM  | CUDABLAS_DGEMM  | CUDABLAS_CGEMM  | CUDABLAS_ZGEMM  |
| CUDABLAS_SSYMM  | CUDABLAS_DSYMM  | CUDABLAS_CSYMM  | CUDABLAS_ZSYMM  |
| CUDABLAS_SSYRK  | CUDABLAS_DSYRK  | CUDABLAS_CSYRK  | CUDABLAS_ZSYRK  |
| CUDABLAS_SSYR2K | CUDABLAS_DSYR2K | CUDABLAS_CSYR2K | CUDABLAS_ZSYR2K |
| CUDABLAS_STRMM  | CUDABLAS_DTRMM  | CUDABLAS_CTRMM  | CUDABLAS_ZTRMM  |
| CUDABLAS_STRSM  | CUDABLAS_DTRSM  | CUDABLAS_CTRSM  | CUDABLAS_ZTRSM  |
|                 |                 | CUDABLAS_CHEMM  | CUDABLAS_ZHEMM  |
|                 |                 | CUDABLAS_CHERK  | CUDABLAS_ZHERK  |
|                 |                 | CUDABLAS_CHER2K | CUDABLAS_ZHER2K |

**Table 9.6. Public Symbols and Parameters in Module CUDABLAS\_LIBRARY**

|                                                |                                                 |
|------------------------------------------------|-------------------------------------------------|
| CUBLAS_STATUS_SUCCESS=0                        | CUBLAS_STATUS_NOT_INITIALIZED=1                 |
| CUBLAS_STATUS_ALLOC_FAILED=3                   | CUBLAS_STATUS_INVALID_VALUE=7                   |
| CUBLAS_STATUS_ARCH_MISMATCH=8                  | CUBLAS_STATUS_MAPPING_ERROR=11                  |
| CUBLAS_STATUS_EXECUTION_FAILED=13              | CUBLAS_STATUS_INTERNAL_ERROR=14                 |
| FSIZE=4                                        | DSIZE=8                                         |
| CSIZE=8                                        | ZSIZE=16                                        |
| SKIND=kind(1.E0)                               | DKIND=kind(1.D0)                                |
| SZERO=0.E0                                     | DZERO=0.D0                                      |
| SONE=1.E0                                      | DONE=1.D0                                       |
| LEVEL=6 ( <i>IMSL Error or Warning Level</i> ) | NSTART(=32) ( <i>Default Switchover Value</i> ) |

**Table 9.7. Subprograms Packaged in Module CUDABLAS\_LIBRARY**

| <b>Fortran Name Implemented in Module</b> |
|-------------------------------------------|
| <a href="#">CUBLAS_GET</a>                |
| <a href="#">CUBLAS_SET</a>                |
| <a href="#">CHECK_BUFFER_ALLOCATION</a>   |
| <a href="#">CUDA_ERROR_PRINT</a>          |

Table 9.8 lists a number of NVIDIA Helper subprograms called within the CUDABLAS\_LIBRARY Modules. These are mostly for internal use only but are documented in the case that a knowledgeable NVIDIA Library user chooses to make use of them.

**Table 9.8. NVIDIA Helper Subprograms Called in Module CUDABLAS\_LIBRARY**

| <b>Fortran Usage Name in Module</b>                      | <b>NVIDIA External C Name</b>                         |
|----------------------------------------------------------|-------------------------------------------------------|
| ISW = cublasInit()                                       | cublasInit()                                          |
| ISW = cublasShutdown()                                   | cublasShutdown()                                      |
| ISW = cublasError()                                      | cublasError()                                         |
| ISW = cublasAlloc(n, datasize, c_ptr)                    | cublasAlloc(n, datasize, c_ptr)                       |
| ISW = cublasFree(c_ptr)                                  | cublasFree(c_ptr)                                     |
| ISW = cublasSetVector(n, datasize, x, incx, y, incy)     | cublasSetVector<br>(n, datasize, x, incx, y, incy)    |
| ISW = cublasGetVector(n, datasize, x, incx, y, incy)     | cublasGetVector<br>(n, datasize, x, incx, y, incy)    |
| ISW = cublasSetMatrix(m, n, datasize, A, lda, devA, ldd) | cublasSetMatrix(m, n, datasize,<br>A, lda, devA, ldd) |
| ISW = cublasGetMatrix(m, n, datasize, devA, lda, B, ldb) | cublasGetMatrix(m, n, datasize,<br>devA, lda, B, ldb) |

In Table 9.8 the arguments `c_ptr`, `x`, `y`, `A`, `devA`, and `B` are *C pointers* to arrays either on the GPU or the CPU. These are instantiated with calls to helper routine `cublasAlloc()` or by use of the Fortran 2003 intrinsic function `c_loc(...)` for array arguments residing on the CPU. This intrinsic returns a *C pointer* to a Fortran object. The helper function `cublasError()` is called from each of the double precision IMSL/NVIDIA BLAS codes to assess the availability of double precision floating point hardware on the GPU.

The NVIDIA Environmental Subprograms listed in Table 9.9 provide details about the runtime working environment.

*Table 9.9. NVIDIA Environmental Subprograms*

| Fortran Usage Name in Module                                                             | NVIDIA External Name                   |
|------------------------------------------------------------------------------------------|----------------------------------------|
| ISW = <code>cudaGetDeviceCount(ICOUNT)</code>                                            | <code>cudaGetDeviceCount()</code>      |
| ISW = <code>cudaSetDevice(IDEVICE), {0 indexed}</code>                                   | <code>cudaSetDevice()</code>           |
| ISW = <code>cudaGetDeviceProperties &amp; (&lt;TYPE&gt; cudaDeviceProp, IDEVICE )</code> | <code>cudaGetDeviceProperties()</code> |

One argument for `cudaGetDeviceProperties` is a Fortran derived type, `cudaDeviceProp`, with a C binding. This contains technical information about the device, including its name. This C character string, `NAME(*)`, is terminated with `C_NULL_CHAR`. The derived type, `cudaDeviceProp` is described below:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>TYPE, BIND(C) :: cudaDeviceProp</b></p> <p><b>CHARACTER(C_CHAR) NAME(256)</b></p> <p><b>INTEGER(C_SIZE_T) totalGlobalMem</b></p> <p><b>INTEGER(C_SIZE_T) sharedMemPerBlock</b></p> <p><b>INTEGER(C_INT) regsPerBlock</b></p> <p><b>INTEGER(C_INT) warpSize</b></p> <p><b>INTEGER(C_SIZE_T) memPitch</b></p> <p><b>INTEGER(C_INT) maxThreadsPerBlock</b></p> <p><b>INTEGER(C_INT) maxThreadsDim(3)</b></p> <p><b>INTEGER(C_INT) maxGridSize(3)</b></p> <p><b>INTEGER(C_SIZE_T) totalConstMem</b></p> <p><b>INTEGER(C_INT) major</b></p> <p><b>INTEGER(C_INT) minor</b></p> <p><b>INTEGER(C_INT) clockRate</b></p> <p><b>INTEGER(C_SIZE_T) textureAlignment</b></p> <p><b>INTEGER(C_INT) deviceOverlap</b></p> <p><b>INTEGER(C_INT) multiProcessorCount</b></p> <p><b>INTEGER(C_INT) kernelExecTimeoutEnabled</b></p> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```
INTEGER(C_INT) integrated
INTEGER(C_INT) canMapHostMemory
INTEGER(C_INT) computeMode
INTEGER(C_INT) concurrentKernels
```

```
END TYPE
```

### Required NVIDIA Copyright Notice:

© 2005–2010 by NVIDIA Corporation. All rights reserved.

Portions of the NVIDIA **SGEMM** and **DGEMM** library routines were written by Vasily Volkov and are subject to the Modified Berkeley Software Distribution License as follows:

Copyright (©) 2007-09, Regents of the University of California

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. ( See *CUDA CUBLAS Library, Version 3.1, July, 2010*, for these remaining conditions.)

---

## CUBLAS\_GET

Returns the switchover value for a positional array argument for a specified BLAS routine.

### Function Return Value

**CUBLAS\_GET** — The array size switchover value used to switch between use of the NVIDIA device or standard Fortran BLAS routine. (Output)  
When **ARRAY\_ARGUMENT** is set to 4, the return value will be 0 or 1 indicating whether the Fortran BLAS routine was used on the last use of the specified routine. ( 0 = Fortran BLAS was used , 1= NVIDIA device was used).

### Required Arguments

**ENUM** — An enumerator which specifies the BLAS routine for which the switchover value is described. (Input)  
**ENUM** must be one of the values defined in Tables 9.3, 9.4, or 9.5.

**ARRAY\_ARGUMENT** — An integer indicating the array argument of the BLAS routine for which information is to be retrieved. The array argument is specified by its position in the calling sequence, i.e. 1 = array argument 1, 2 = array argument 2, etc. (Input)  
For example, for the BLAS routine **SGEMM**, array **A** is **ARRAY\_ARGUMENT** =1, array **B** is **ARRAY\_ARGUMENT** = 2, and array **C** is **ARRAY\_ARGUMENT** =3. Setting **ARRAY\_ARGUMENT** to 4 will dictate that **CUBLAS\_GET** returns a 0, 1 value indicating which was last used – the standard Fortran BLAS routine or the NVIDIA device, respectively.

## FORTRAN 90 Interface

Generic: CUBLAS\_GET (ENUM, ARRAY\_ARGUMENT)

### Description

This routine can be used to either retrieve the array size switchover value, `NSTART`, for a specified array of a specified BLAS routine or retrieve a 0, 1 flag which indicates whether the NVIDIA device was used for the last specified BLAS routine called.

### Example

In this example the switchover value for array `A` of the BLAS routine `SGEMM` is first retrieved by making a call to `CUBLAS_GET`. Then `CUBLAS_SET` is used to inform `CUDABLAS_SGEMM` not to copy array `A` from the CPU to the GPU after the initial copy. Then, `CUBLAS_SET` is used to reset the switchover value back to its original setting. Finally, `CUBLAS_GET` is used to query whether or not the NVIDIA device was used on the last call to `SGEMM`.

```
USE CUDABLAS_LIBRARY
USE UMACH_INT
INTEGER ARRAY_ARGUMENT, IDEVICE, ISWITCH, NOUT
INTEGER, PARAMETER :: N=500
REAL ALPHA, BETA, A(N,N), B(N,N), C(N,N), D(N,N)

ALPHA = 1.0
BETA = 1.0
A = 2.0
B = 3.0
C = 4.0

! ARRAY A IS THE FIRST ARRAY IN
! THE SGEMM CALLING SEQUENCE
ARRAY_ARGUMENT = 1

! GET THE CURRENT SWITCHOVER VALUE
! FOR SGEMM

ISWITCH = CUBLAS_GET (CUDABLAS_SGEMM, ARRAY_ARGUMENT)
! PERFORM AN ARRAY MULTIPLICATION

CALL SGEMM ('N', 'N', N, N, N, ALPHA, A, N, B, N, BETA, D, N)

! AVOID COPYING A FROM THE CPU TO
! THE GPU HENCEFORTH

CALL CUBLAS_SET (CUDABLAS_SGEMM, ARRAY_ARGUMENT, -ABS(ISWITCH))

! PERFORM A SECOND ARRAY MULTIPLICATION

CALL SGEMM ('N', 'N', N, N, N, ALPHA, A, N, C, N, BETA, C, N)

! RESET THE SWITCHOVER VALUE FOR
! SGEMM BACK TO ITS ORIGINAL VALUE
```

```

CALL CUBLAS_SET (CUDABLAS_SGEMM, ARRAY_ARGUMENT, ISWITCH)

!
! DETERMINE WHETHER OR NOT THE GPU
! WAS USED FOR THE LAST SGEMM CALL
ARRAY_ARGUMENT = 4
IDEVICE = CUBLAS_GET (CUDABLAS_SGEMM, ARRAY_ARGUMENT)

!
! PRINT THE RESULT OF THE LAST QUERY
CALL UMACH (2, NOUT)
IF (IDEVICE .EQ. 0) THEN
 WRITE(NOUT, *) 'THE STANDARD FORTRAN BLAS SGEMM WAS USED.'
ELSE
 WRITE(NOUT, *) 'THE NVIDIA DEVICE SGEMM WAS USED.'
END IF
END

```

## Output

The NVIDIA DEVICE SGEMM WAS USED.

---

# CUBLAS\_SET

Sets the switchover value for an array used by a specified BLAS routine.

## Required Arguments

**ENUM** — An enumerator which specifies the BLAS routine for which the switchover value is to be set. (Input)

ENUM must be one of the values defined in Tables 9.3, 9.4, or 9.5.

**ARRAY\_ARGUMENT** — An integer indicating the array argument of the BLAS routine for which information is to be set. The array argument is specified by its position in the calling sequence, i.e. 1 = array argument 1, 2 = array argument 2, etc. (Input)

For example, for the BLAS routine SGEMM, array *A* is ARRAY\_ARGUMENT = 1, array *B* is ARRAY\_ARGUMENT = 2, and array *C* is ARRAY\_ARGUMENT = 3.

**NSTART** — Defines the array size that is used as the switchover point for the array specified by ARRAY\_ARGUMENT when the BLAS routine specified by ENUM is used. (Input)

For arrays  $\geq |NSTART|$  the NVIDIA device will be used. For arrays  $< |NSTART|$  a standard Fortran BLAS routine will be used. Setting NSTART to a negative value indicates that no array copy need be performed for the array specified by ARRAY\_ARGUMENT. Setting NSTART to 0 indicates that the NVIDIA hardware is not used for the specified BLAS routine.

## FORTRAN 90 Interface

Generic: CALL CUBLAS\_SET (ENUM, ARRAY\_ARGUMENT, NSTART)

## Description

This routine allows the user to set the array size values that will be used by a specified BLAS routine as the switchover point for using a standard Fortran Blas routine versus the CUDABLAS routine with the NVIDIA device. The routine can also be used to inform the NVIDIA device to not perform an array copy after the initial copy has been performed.

## Example

In this example the switchover for array A of the BLAS routine SGEMM is first retrieved by making a call to CUBLAS\_GET. Then CUBLAS\_SET is used to inform CUDABLAS\_SGEMM not to copy array A from the CPU to the GPU after the initial copy. Then, CUBLAS\_SET is used to reset the switchover value back to its original setting. Finally, CUBLAS\_GET is used to query whether or not the NVIDIA device was used on the last call to SGEMM.

```
USE CUDABLAS_LIBRARY
USE UMACH_INT
INTEGER ARRAY_ARGUMENT, IDEVICE, ISWITCH, NOUT
INTEGER, PARAMETER :: N=500
REAL ALPHA, BETA, A(N,N), B(N,N), C(N,N), D(N,N)

ALPHA = 1.0
BETA = 1.0
A = 2.0
B = 3.0
C = 4.0
!
! ARRAY A IS THE FIRST ARRAY IN
! THE SGEMM CALLING SEQUENCE
ARRAY_ARGUMENT = 1
!
! GET THE CURRENT SWITCHOVER VALUE
! FOR SGEMM
ISWITCH = CUBLAS_GET (CUDABLAS_SGEMM, ARRAY_ARGUMENT)
!
! PERFORM AN ARRAY MULTIPLICATION
CALL SGEMM ('N', 'N', N, N, N, ALPHA, A, N, B, N, BETA, D, N)
!
! AVOID COPYING A FROM THE CPU TO
! THE GPU HENCEFORTH
CALL CUBLAS_SET (CUDABLAS_SGEMM, ARRAY_ARGUMENT, -ABS(ISWITCH))
!
! PERFORM A SECOND ARRAY MULTIPLICATION
CALL SGEMM ('N', 'N', N, N, N, ALPHA, A, N, C, N, BETA, C, N)
!
! RESET THE SWITCHOVER VALUE FOR
! SGEMM BACK TO ITS ORIGINAL VALUE
CALL CUBLAS_SET (CUDABLAS_SGEMM, ARRAY_ARGUMENT, ISWITCH)
!
! DETERMINE WHETHER OR NOT THE GPU
! WAS USED FOR THE LAST SGEMM CALL
ARRAY_ARGUMENT = 4
```

```

IDEVICE = CUBLAS_GET (CUDABLAS_SGEMM, ARRAY_ARGUMENT)
!
PRINT THE RESULT OF THE LAST QUERY
CALL UMACH (2, NOUT)
IF (IDevice .EQ. 0) THEN
 WRITE(NOUT, *) 'THE STANDARD FORTRAN BLAS SGEMM WAS USED.'
ELSE
 WRITE(NOUT, *) 'THE NVIDIA DEVICE SGEMM WAS USED.'
END IF
END

```

## Output

The NVIDIA DEVICE SGEMM WAS USED.

---

# CHECK\_BUFFER\_ALLOCATION

Maintains buffer sizes on the NVIDIA device and performs one-time initialization.

## Required Arguments

*ISZ* — An array of length 5. (Input/Output)  
The elements of *ISZ* contain the following:

| ISZ Element | Description                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ISZ(1)      | Array size for the first array which appears in the argument list of the BLAS routine being called. If an error occurs while trying to allocate space for this array, an error flag is returned.  |
| ISZ(2)      | Array size for the second array which appears in the argument list of the BLAS routine being called. If an error occurs while trying to allocate space for this array, an error flag is returned. |
| ISZ(3)      | Array size for the third array which appears in the argument list of the BLAS routine being called. If an error occurs while trying to allocate space for this array, an error flag is returned.  |
| ISZ(4)      | Not used.                                                                                                                                                                                         |
| ISZ(5)      | The word size to be used when allocating the buffer. This element should be one of the public parameters in the CUDABLAS_LIBRARY module FSIZE = 4, DSIZE = 8, CSIZE = 8, or ZSIZE = 16.           |

If any of the first three elements of *ISZ* is set to zero on input then the GPU buffer for that argument is deallocated.

## FORTRAN 90 Interface

Generic: CALL CHECK\_BUFFER\_ALLOCATION (ISZ)

## Description

---

**Note:** Normally, this routine is not called by the user.

---

This routine is used internally to initialize the buffer sizes for the arrays on the NVIDIA device. Space is allocated for the arrays in initialization only. If an error occurs during the allocation of the first, second, or third positional array then an error flag is returned in `ISZ(1)`, `ISZ(2)`, or `ISZ(3)`, respectively. If `ISZ(1)`, `ISZ(2)`, or `ISZ(3)` is set to zero on input then the space for the array designated by the respective element is deallocated.

---

## CUDA\_ERROR\_PRINT

Prints error messages generated through the use of the CUDABLAS Library using the IMSL error handler.

### Required Arguments

**ISZ** — An array of length 5. (Input/Output)  
The elements of `ISZ` contain the following:

| ISZ Element         | Description                                                                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ISZ(1)</code> | NOT used.                                                                                                                                                                                  |
| <code>ISZ(2)</code> | NOT used.                                                                                                                                                                                  |
| <code>ISZ(3)</code> | Used to pass the argument number of the BLAS routine being called which is in error.                                                                                                       |
| <code>ISZ(4)</code> | The enumeration value which identifies the name of the routine for which an error has occurred. <code>ISZ(4)</code> can be one of the enumerated values listed in Tables 9.3, 9.4, or 9.5. |
| <code>ISZ(5)</code> | NOT used.                                                                                                                                                                                  |

**NARRAY\_ARGS** — The number of array arguments for which the error occurred. (Input)

**ERROR\_NUMBER** — An integer which identifies the error which occurred. (Input)

`ERROR_NUMBER` can be one of the following:

| ERROR_NUMBER Value  | Error Description                                                                        |
|---------------------|------------------------------------------------------------------------------------------|
| <code>ISZ(1)</code> | "GPU memory allocation error for array argument = <code>%(i1)</code> of <code>"</code> . |
| <code>ISZ(2)</code> | "CPU or GPU copy failure for array argument = <code>%(i1)</code> of <code>"</code> .     |

| ERROR_NUMBER<br>Value | Error Description                                                 |
|-----------------------|-------------------------------------------------------------------|
| ISZ(3)                | "GPU or CPU copy failure for array argument = %(i1), the result:" |
| ISZ(4)                | "Error in routine argument =. %(i1) of Fortran version. "         |
| ISZ(5)                | "Double precision hardware on GPU not available for ".            |

## FORTRAN 90 Interface

Generic:    CALL CUDA\_ERROR\_PRINT (ISZ, NARRAY\_ARGS, ERROR\_NUMBER)

## Description

---

**Note:** Normally, this routine is not called by the user.

---

This routine is used internally to process and print error messages generated through the use of the *CUDABLAS* Library.

---

## Other Matrix/Vector Operations

This section describes a set of routines for matrix/vector operations. The matrix copy and conversion routines are summarized by the following table:

| From            | To           |                 |           |              |
|-----------------|--------------|-----------------|-----------|--------------|
|                 | Real General | Complex General | Real Band | Complex Band |
| Real General    | CRGRG        | CRGCG           | CRGRB     |              |
| Complex General |              | CCGCG           |           | CCGCB        |
| Real Band       | CRBRG        |                 | CRBRB     | CRBCB        |
| Complex Band    |              | CCBCG           |           | CCBCB        |
| Symmetric Full  | CSFRG        |                 |           |              |
| Hermitian Full  |              | CHFCG           |           |              |
| Symmetric Band  |              |                 | CSBRB     |              |
| Hermitian Band  |              |                 |           | CHBCB        |

The matrix multiplication routines are summarized as follows:

| <i>AB</i>        | <i>A</i>   |               |           |              |
|------------------|------------|---------------|-----------|--------------|
| <i>B</i>         | Real Rect. | Complex Rect. | Real Band | Complex Band |
| Real Rectangular | MRRRR      |               |           |              |
| Complex Rect.    |            | MCRCR         |           |              |
| Vector           | MURRV      | MUCRV         | MURBV     | MUCBV        |

The matrix norm routines are summarized as follows:

| $\ A\ $        | Real Rectangular | Real Band | Complex Band |
|----------------|------------------|-----------|--------------|
| $\infty$ -norm | NRIRR            |           |              |
| 1-norm         | NR1RR            | NR1RB     | NR1CB        |
| Frobenius      | NR2RR            |           |              |

---

# CRGRG

Copies a real general matrix.

## Required Arguments

*A* — Matrix of order *N*. (Input)

*B* — Matrix of order *N* containing a copy of *A*. (Output)

## Optional Arguments

*N* — Order of the matrices. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDB* = SIZE (*B*,1).

## FORTRAN 90 Interface

Generic:    CALL CRGRG (*A*, *B* [, ...])

Specific:   The specific interface names are S\_CRGRG and D\_CRGRG.

## FORTRAN 77 Interface

Single:     CALL CRGRG (*N*, *A*, *LDA*, *B*, *LDB*)

Double:     The double precision name is DCRGRG.

## Description

The routine CRGRG copies the real  $N \times N$  general matrix *A* into the real  $N \times N$  general matrix *B*.

## Example

A real  $3 \times 3$  general matrix is copied into another real  $3 \times 3$  general matrix.

```
USE CRGRG_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N
PARAMETER (LDA=3, LDB=3, N=3)
```

```

!
REAL A(LDA,N) , B(LDB,N)
!
! Set values for A
! A = (0.0 1.0 1.0)
! (-1.0 0.0 1.0)
! (-1.0 -1.0 0.0)
!
DATA A/0.0, 2* - 1.0, 1.0, 0.0, -1.0, 2*1.0, 0.0/
! Copy real matrix A to real matrix B
CALL CRGRG (A, B)
!
! Print results
CALL WRRRN ('B', B)
END

```

## Output

```

 B
 1 2 3
1 0.000 1.000 1.000
2 -1.000 0.000 1.000
3 -1.000 -1.000 0.000

```

---

## CCGCG

Copies a complex general matrix.

### Required Arguments

*A* — Complex matrix of order *N*. (Input)

*B* — Complex matrix of order *N* containing a copy of *A*. (Output)

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDB* = SIZE (*B*,1).

### FORTRAN 90 Interface

Generic: CALL CCGCG (*A*, *B* [, ...])

Specific: The specific interface names are *S\_CCGCG* and *D\_CCGCG*.

## FORTRAN 77 Interface

Single:      CALL CCGCG (N, A, LDA, B, LDB)

Double:     The double precision name is DCCGCG.

## Description

The routine CCGCG copies the complex  $N \times N$  general matrix  $A$  into the complex  $N \times N$  general matrix  $B$ .

## Example

A complex  $3 \times 3$  general matrix is copied into another complex  $3 \times 3$  general matrix.

```
USE CCGCG_INT
USE WRNCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N
PARAMETER (LDA=3, LDB=3, N=3)
!
COMPLEX A(LDA,N), B(LDB,N)
! Set values for A
! A = (0.0+0.0i 1.0+1.0i 1.0+1.0i)
! (-1.0-1.0i 0.0+0.0i 1.0+1.0i)
! (-1.0-1.0i -1.0-1.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), 2*(-1.0,-1.0), (1.0,1.0), (0.0,0.0), &
 (-1.0,-1.0), 2*(1.0,1.0), (0.0,0.0)/
! Copy matrix A to matrix B
CALL CCGCG (A, B)
! Print results
CALL WRNCRN ('B', B)
END
```

## Output

```

 B
 1 2 3
1 (0.000, 0.000) (1.000, 1.000) (1.000, 1.000)
2 (-1.000,-1.000) (0.000, 0.000) (1.000, 1.000)
3 (-1.000,-1.000) (-1.000,-1.000) (0.000, 0.000)
```

---

## CRBRB

Copies a real band matrix stored in band storage mode.

## Required Arguments

$A$  — Real band matrix of order  $N$ . (Input)

*NLCA* — Number of lower codiagonals in *A*. (Input)

*NUCA* — Number of upper codiagonals in *A*. (Input)

*B* — Real band matrix of order *N* containing a copy of *A*. (Output)

*NLCB* — Number of lower codiagonals in *B*. (Input)  
NLCB must be at least as large as NLCA.

*NUCB* — Number of upper codiagonals in *B*. (Input)  
NUCB must be at least as large as NUCA.

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDB* = SIZE (*B*,1).

### FORTRAN 90 Interface

Generic:     CALL CRBRB (*A*, *NLCA*, *NUCA*, *B*, *NLCB*, *NUCB* [, ...] )

Specific:    The specific interface names are *S\_CRBRB* and *D\_CRBRB*.

### FORTRAN 77 Interface

Single:     CALL CRBRB (*N*, *A*, *LDA*, *NLCA*, *NUCA*, *B*, *LDB*, *NLCB*, *NUCB*)

Double:     The double precision name is *DCRBRB*.

### Description

The routine *CRBRB* copies the real band matrix *A* in band storage mode into the real band matrix *B* in band storage mode.

### Example

A real band matrix of order 3, in band storage mode with one upper codiagonal, and one lower codiagonal is copied into another real band matrix also in band storage mode.

```
USE CRBRB_INT
```

```

USE WRRRN_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
PARAMETER (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)

!
REAL A(LDA,N), B(LDB,N)
! Set values for A (in band mode)
! A = (0.0 1.0 1.0)
! (1.0 1.0 1.0)
! (1.0 1.0 0.0)
!
DATA A/0.0, 7*1.0, 0.0/

! Copy A to B
CALL CRBRB (A, NLCA, NUCA, B, NLCB, NUCB)

! Print results
CALL WRRRN ('B', B)
END

```

## Output

|   | B     |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 0.000 | 1.000 | 1.000 |
| 2 | 1.000 | 1.000 | 1.000 |
| 3 | 1.000 | 1.000 | 0.000 |

---

## CCBCB

Copies a complex band matrix stored in complex band storage mode.

### Required Arguments

**A** — Complex band matrix of order *N*. (Input)

**NLCA** — Number of lower codiagonals in *A*. (Input)

**NUCA** — Number of upper codiagonals in *A*. (Input)

**B** — Complex matrix of order *N* containing a copy of *A*. (Output)

**NLCB** — Number of lower codiagonals in *B*. (Input)  
NLCB must be at least as large as NLCA.

**NUCB** — Number of upper codiagonals in *B*. (Input)  
NUCB must be at least as large as NUCA.

### Optional Arguments

**N** — Order of the matrices *A* and *B*. (Input)  
Default: *N* = SIZE (*A*,2).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: LDA = SIZE (A,1).

**LDB** — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: LDB = SIZE (B,1).

### **FORTRAN 90 Interface**

Generic: CALL CCBCB (A, NLCA, NUCA, B, NLCB, NUCB [, ...])

Specific: The specific interface names are S\_CCBCB and D\_CCBCB.

### **FORTRAN 77 Interface**

Single: CALL CCBCB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB)

Double: The double precision name is DCCBCB.

### **Description**

The routine CCBCB copies the complex band matrix *A* in band storage mode into the complex band matrix *B* in band storage mode.

### **Example**

A complex band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied into another complex band matrix in band storage mode.

```
USE CCBCB_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
PARAMETER (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)
!
COMPLEX A(LDA,N), B(LDB,N)
! Set values for A (in band mode)
! A = (0.0+0.0i 1.0+1.0i 1.0+1.0i)
! (1.0+1.0i 1.0+1.0i 1.0+1.0i)
! (1.0+1.0i 1.0+1.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), 7*(1.0,1.0), (0.0,0.0)/
! Copy A to B
CALL CCBCB (A, NLCA, NUCA, B, NLCB, NUCB)
! Print results
CALL WRCRN ('B', B)
END
```

## Output

```

 B
 1 2 3
1 (0.000, 0.000) (1.000, 1.000) (1.000, 1.000)
2 (1.000, 1.000) (1.000, 1.000) (1.000, 1.000)
3 (1.000, 1.000) (1.000, 1.000) (0.000, 0.000)
```

---

## CRGRB

Converts a real general matrix to a matrix in band storage mode.

### Required Arguments

*A* — Real *N* by *N* matrix. (Input)

*NLC* — Number of lower codiagonals in *B*. (Input)

*NUC* — Number of upper codiagonals in *B*. (Input)

*B* — Real (*NUC* + 1 + *NLC*) by *N* array containing the band matrix in band storage mode.  
(Output)

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDB* = SIZE (*B*,1).

### FORTRAN 90 Interface

Generic: CALL CRGRB (*A*, *NLC*, *NUC*, *B* [, ...])

Specific: The specific interface names are S\_CRGRB and D\_CRGRB.

### FORTRAN 77 Interface

Single: CALL CRGRB (*N*, *A*, *LDA*, *NLC*, *NUC*, *B*, *LDB*)

Double: The double precision name is DCRGRB.

## Description

The routine CRGRB converts the real general  $N \times N$  matrix  $A$  with  $m_u = \text{NUC}$  upper codiagonals and  $m_l = \text{NLC}$  lower codiagonals into the real band matrix  $B$  of order  $N$ . The first  $m_u$  rows of  $B$  then contain the upper codiagonals of  $A$ , the next row contains the main diagonal of  $A$ , and the last  $m_l$  rows of  $B$  contain the lower codiagonals of  $A$ .

## Example

A real  $4 \times 4$  matrix with one upper codiagonal and three lower codiagonals is copied to a real band matrix of order 4 in band storage mode.

```
 USE CRGRB_INT
 USE WRRRN_INT

 IMPLICIT NONE

! Declare variables
 INTEGER LDA, LDB, N, NLC, NUC
 PARAMETER (LDA=4, LDB=5, N=4, NLC=3, NUC=1)

!
 REAL A(LDA,N), B(LDB,N)

! Set values for A
! A = (1.0 2.0 0.0 0.0)
! (-2.0 1.0 3.0 0.0)
! (0.0 -3.0 1.0 4.0)
! (-7.0 0.0 -4.0 1.0)
!
 DATA A/1.0, -2.0, 0.0, -7.0, 2.0, 1.0, -3.0, 0.0, 0.0, 3.0, 1.0, &
 -4.0, 0.0, 0.0, 4.0, 1.0/

! Convert A to band matrix B
 CALL CRGRB (A, NLC, NUC, B)

! Print results
 CALL WRRRN ('B', B)
 END
```

## Output

|   | B      |        |        |       |
|---|--------|--------|--------|-------|
|   | 1      | 2      | 3      | 4     |
| 1 | 0.000  | 2.000  | 3.000  | 4.000 |
| 2 | 1.000  | 1.000  | 1.000  | 1.000 |
| 3 | -2.000 | -3.000 | -4.000 | 0.000 |
| 4 | 0.000  | 0.000  | 0.000  | 0.000 |
| 5 | -7.000 | 0.000  | 0.000  | 0.000 |

---

## CRBRG

Converts a real matrix in band storage mode to a real general matrix.

## Required Arguments

*A* — Real  $(NUC + 1 + NLC)$  by  $N$  array containing the band matrix in band storage mode. (Input)

*NLC* — Number of lower codiagonals in *A*. (Input)

*NUC* — Number of upper codiagonals in *A*. (Input)

*B* — Real  $N$  by  $N$  array containing the matrix. (Output)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{SIZE}(A,1)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDB = \text{SIZE}(B,1)$ .

## FORTRAN 90 Interface

Generic:     CALL CRBRG (A, NLC, NUC, B [, ...])

Specific:    The specific interface names are S\_CRBRG and D\_CRBRG.

## FORTRAN 77 Interface

Single:     CALL CRBRG (N, A, LDA, NLC, NUC, B, LDB)

Double:     The double precision name is DCRBRG.

## Description

The routine CRBRG converts the real band matrix *A* of order  $N$  in band storage mode into the real  $N \times N$  general matrix *B* with  $m_u = NUC$  upper codiagonals and  $m_l = NLC$  lower codiagonals. The first  $m_u$  rows of *A* are copied to the upper codiagonals of *B*, the next row of *A* is copied to the diagonal of *B*, and the last  $m_l$  rows of *A* are copied to the lower codiagonals of *B*.

## Example

A real band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied to a  $3 \times 3$  real general matrix.

```
USE CRBRG_INT
USE WRRRN_INT
```

```

 IMPLICIT NONE
!
! Declare variables
 INTEGER LDA, LDB, N, NLC, NUC
 PARAMETER (LDA=3, LDB=3, N=3, NLC=1, NUC=1)
!
 REAL A(LDA,N), B(LDB,N)
!
! Set values for A (in band mode)
! A = (0.0 1.0 1.0)
! (4.0 3.0 2.0)
! (2.0 2.0 0.0)
!
 DATA A/0.0, 4.0, 2.0, 1.0, 3.0, 2.0, 1.0, 2.0, 0.0/
!
! Convert band matrix A to matrix B
 CALL CRBRG (A, NLC, NUC, B)
!
! Print results
 CALL WRRRN ('B', B)
 END

```

## Output

|   | B     |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 4.000 | 1.000 | 0.000 |
| 2 | 2.000 | 3.000 | 1.000 |
| 3 | 0.000 | 2.000 | 2.000 |

---

## CCGCB

Converts a complex general matrix to a matrix in complex band storage mode.

### Required Arguments

**A** — Complex  $N$  by  $N$  array containing the matrix. (Input)

**NLC** — Number of lower codiagonals in **B**. (Input)

**NUC** — Number of upper codiagonals in **B**. (Input)

**B** — Complex  $(NUC + 1 + NLC)$  by  $N$  array containing the band matrix in band storage mode. (Output)

### Optional Arguments

**N** — Order of the matrices **A** and **B**. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

**LDB** — Leading dimension of **B** exactly as specified in the dimension statement of the calling program. (Input)  
 Default: LDB = SIZE (B,1).

### FORTRAN 90 Interface

Generic: CALL CCGCB (A, NLC, NUC, B [, ...])

Specific: The specific interface names are S\_CCGCB and D\_CCGCB.

### FORTRAN 77 Interface

Single: CALL CCGCB (N, A, LDA, NLC, NUC, B, LDB)

Double: The double precision name is DCCGCB.

### Description

The routine CCGCB converts the complex general matrix  $A$  of order  $N$  with  $m_u = \text{NUC}$  upper codiagonals and  $m_l = \text{NLC}$  lower codiagonals into the complex band matrix  $B$  of order  $N$  in band storage mode. The first  $m_u$  rows of  $B$  then contain the upper codiagonals of  $A$ , the next row contains the main diagonal of  $A$ , and the last  $m_l$  rows of  $B$  contain the lower codiagonals of  $A$ .

### Example

A complex general matrix of order 4 with one upper codiagonal and three lower codiagonals is copied to a complex band matrix of order 4 in band storage mode.

```

USE CCGCB_INT
USE WRNCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NLC, NUC
PARAMETER (LDA=4, LDB=5, N=4, NLC=3, NUC=1)
!
COMPLEX A(LDA,N), B(LDB,N)
! Set values for A
! A = (1.0+0.0i 2.0+1.0i 0.0+0.0i 0.0+0.0i)
! (-2.0+1.0i 1.0+0.0i 3.0+2.0i 0.0+0.0i)
! (0.0+0.0i -3.0+2.0i 1.0+0.0i 4.0+3.0i)
! (-7.0+1.0i 0.0+0.0i -4.0+3.0i 1.0+0.0i)
!
DATA A/(1.0,0.0), (-2.0,1.0), (0.0,0.0), (-7.0,1.0), (2.0,1.0), &
 (1.0,0.0), (-3.0,2.0), (0.0,0.0), (0.0,0.0), (3.0,2.0), &
 (1.0,0.0), (-4.0,3.0), (0.0,0.0), (0.0,0.0), (4.0,3.0), &
 (1.0,0.0)/
!
! Convert A to band matrix B
CALL CCGCB (A, NLC, NUC, B)
!
! Print results
CALL WRNCRN ('B', B)
END

```

## Output

```

 B
 2
 1 1 2 3 4
1 (0.000, 0.000) (2.000, 1.000) (3.000, 2.000) (4.000, 3.000)
2 (1.000, 0.000) (1.000, 0.000) (1.000, 0.000) (1.000, 0.000)
3 (-2.000, 1.000) (-3.000, 2.000) (-4.000, 3.000) (0.000, 0.000)
4 (0.000, 0.000) (0.000, 0.000) (0.000, 0.000) (0.000, 0.000)
5 (-7.000, 1.000) (0.000, 0.000) (0.000, 0.000) (0.000, 0.000)
```

---

## CCBCG

Converts a complex matrix in band storage mode to a complex matrix in full storage mode.

### Required Arguments

*A* — Complex ( $NUC + 1 + NLC$ ) by  $N$  matrix containing the band matrix in band mode. (Input)

*NLC* — Number of lower codiagonals in *A*. (Input)

*NUC* — Number of upper codiagonals in *A*. (Input)

*B* — Complex  $N$  by  $N$  matrix containing the band matrix in full mode. (Output)

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

### FORTRAN 90 Interface

Generic:     CALL CCBCG (A, NLC, NUC, B [, ...])

Specific:    The specific interface names are S\_CCBCG and D\_CCBCG.

### FORTRAN 77 Interface

Single:     CALL CCBCG (N, A, LDA, NLC, NUC, B, LDB)

Double:      The double precision name is DCCBCG.

## Description

The routine CCBCG converts the complex band matrix  $A$  of order  $N$  with  $m_u = \text{NUC}$  upper codiagonals and  $m_l = \text{NLC}$  lower codiagonals into the  $N \times N$  complex general matrix  $B$ . The first  $m_u$  rows of  $A$  are copied to the upper codiagonals of  $B$ , the next row of  $A$  is copied to the diagonal of  $B$ , and the last  $m_l$  rows of  $A$  are copied to the lower codiagonals of  $B$ .

## Example

A complex band matrix of order 4 in band storage mode with one upper codiagonal and three lower codiagonals is copied into a  $4 \times 4$  complex general matrix.

```
USE CCBCG_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NLC, NUC
PARAMETER (LDA=5, LDB=4, N=4, NLC=3, NUC=1)

!
COMPLEX A(LDA,N), B(LDB,N)
! Set values for A (in band mode)
! A = (0.0+0.0i 2.0+1.0i 3.0+2.0i 4.0+3.0i)
! (1.0+0.0i 1.0+0.0i 1.0+0.0i 1.0+0.0i)
! (-2.0+1.0i -3.0+2.0i -4.0+3.0i 0.0+0.0i)
! (0.0+0.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
! (-7.0+1.0i 0.0+0.0i 0.0+0.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), (1.0,0.0), (-2.0,1.0), (0.0,0.0), (-7.0,1.0), &
 (2.0,1.0), (1.0,0.0), (-3.0,2.0), 2*(0.0,0.0), (3.0,2.0), &
 (1.0,0.0), (-4.0,3.0), 2*(0.0,0.0), (4.0,3.0), (1.0,0.0), &
 3*(0.0,0.0)/
!
! Convert band matrix A to matrix B
CALL CCBCG (A, NLC, NUC, B)
!
! Print results
CALL WRCRN ('B', B)
END
```

## Output

```

 B
 1 2 3 4
1 (1.000, 0.000) (2.000, 1.000) (0.000, 0.000) (0.000, 0.000)
2 (-2.000, 1.000) (1.000, 0.000) (3.000, 2.000) (0.000, 0.000)
3 (0.000, 0.000) (-3.000, 2.000) (1.000, 0.000) (4.000, 3.000)
4 (-7.000, 1.000) (0.000, 0.000) (-4.000, 3.000) (1.000, 0.000)
```

---

## CRGCG

Copies a real general matrix to a complex general matrix.

## Required Arguments

*A* — Real matrix of order *N*. (Input)

*B* — Complex matrix of order *N* containing a copy of *A*. (Output)

## Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDB* = SIZE (*B*,1).

## FORTRAN 90 Interface

Generic:     CALL CRGCG (*A*, *B* [, ...])

Specific:    The specific interface names are *S\_CRGCG* and *D\_CRGCG*.

## FORTRAN 77 Interface

Single:     CALL CRGCG (*N*, *A*, *LDA*, *B*, *LDB*)

Double:     The double precision name is *DCRGCG*.

## Description

The routine *CRGCG* copies a real  $N \times N$  matrix to a complex  $N \times N$  matrix.

## Example

A  $3 \times 3$  real matrix is copied to a  $3 \times 3$  complex matrix.

```
USE CRGCG_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N
PARAMETER (LDA=3, LDB=3, N=3)
!
REAL A(LDA, N)
COMPLEX B(LDB, N)
! Set values for A
```

```

! A = (2.0 1.0 3.0)
! (4.0 1.0 0.0)
! (-1.0 2.0 0.0)
!
DATA A/2.0, 4.0, -1.0, 1.0, 1.0, 2.0, 3.0, 0.0, 0.0/
! Convert real A to complex B
CALL CRGCG (A, B)
! Print results
CALL WRRCRN ('B', B)
END

```

## Output

```

 B
 1 2 3
1 (2.000, 0.000) (1.000, 0.000) (3.000, 0.000)
2 (4.000, 0.000) (1.000, 0.000) (0.000, 0.000)
3 (-1.000, 0.000) (2.000, 0.000) (0.000, 0.000)

```

---

## CRRCR

Copies a real rectangular matrix to a complex rectangular matrix.

### Required Arguments

**A** — Real *NRA* by *NCA* rectangular matrix. (Input)

**B** — Complex *NRB* by *NCB* rectangular matrix containing a copy of *A*. (Output)

### Optional Arguments

**NRA** — Number of rows in *A*. (Input)

Default: *NRA* = *SIZE* (*A*,1).

**NCA** — Number of columns in *A*. (Input)

Default: *NCA* = *SIZE* (*A*,2).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

**NRB** — Number of rows in *B*. (Input)

It must be the same as *NRA*.

Default: *NRB* = *SIZE* (*B*,1).

**NCB** — Number of columns in *B*. (Input)

It must be the same as *NCA*.

Default: *NCB* = *SIZE* (*B*,2).

**LDB** — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)  
 Default: LDB = SIZE (B,1).

### **FORTRAN 90 Interface**

Generic: CALL CRRCR (A, B [, ...])

Specific: The specific interface names are S\_CRRCR and D\_CRRCR.

### **FORTRAN 77 Interface**

Single: CALL CRRCR (NRA, NCA, A, LDA, NRB, NCB, B, LDB)

Double: The double precision name is DCRRCR.

### **Description**

The routine CRRCR copies a real rectangular matrix to a complex rectangular matrix.

### **Example**

A  $3 \times 2$  real matrix is copied to a  $3 \times 2$  complex matrix.

```

USE CRRCR_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, NCA, NCB, NRA, NRB
PARAMETER (LDA=3, LDB=3, NCA=2, NCB=2, NRA=3, NRB=3)
!
REAL A(LDA,NCA)
COMPLEX B(LDB,NCB)
! Set values for A
! A = (1.0 4.0)
! (2.0 5.0)
! (3.0 6.0)
!
DATA A/1.0, 2.0, 3.0, 4.0, 5.0, 6.0/
! Convert real A to complex B
CALL CRRCR (A, B)
! Print results
CALL WRCRN ('B', B)
END

```

### **Output**

```

 B
 1 2
1 (1.000, 0.000) (4.000, 0.000)

```

```
2 (2.000, 0.000) (5.000, 0.000)
3 (3.000, 0.000) (6.000, 0.000)
```

---

## CRBCB

Converts a real matrix in band storage mode to a complex matrix in band storage mode.

### Required Arguments

*A* — Real band matrix of order *N*. (Input)

*NLCA* — Number of lower codiagonals in *A*. (Input)

*NUCA* — Number of upper codiagonals in *A*. (Input)

*B* — Complex matrix of order *N* containing a copy of *A*. (Output)

*NLCB* — Number of lower codiagonals in *B*. (Input)

*NLCB* must be at least as large as *NLCA*.

*NUCB* — Number of upper codiagonals in *B*. (Input)

*NUCB* must be at least as large as *NUCA*.

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)

Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDB* = SIZE (*B*,1).

### FORTRAN 90 Interface

Generic: CALL CRBCB (*A*, *NLCA*, *NUCA*, *B*, *NLCB*, *NUCB* [, ...])

Specific: The specific interface names are *S\_CRBCB* and *D\_CRBCB*.

### FORTRAN 77 Interface

Single: CALL CRBCB (*N*, *A*, *LDA*, *NLCA*, *NUCA*, *B*, *LDB*, *NLCB*, *NUCB*)

Double: The double precision name is *DCRBCB*.

## Description

The routine CRBCB converts a real band matrix in band storage mode with NUCA upper codiagonals and NLCA lower codiagonals into a complex band matrix in band storage mode with NUCB upper codiagonals and NLCB lower codiagonals.

## Example

A real band matrix of order 3 in band storage mode with one upper codiagonal and one lower codiagonal is copied into another complex band matrix in band storage mode.

```
USE CRBCB_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDB, N, NLCA, NLCB, NUCA, NUCB
PARAMETER (LDA=3, LDB=3, N=3, NLCA=1, NLCB=1, NUCA=1, NUCB=1)
!
REAL A(LDA,N)
COMPLEX B(LDB,N)
!
! Set values for A (in band mode)
! A = (0.0 1.0 1.0)
! (1.0 1.0 1.0)
! (1.0 1.0 0.0)
!
DATA A/0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0/
!
! Convert real band matrix A
! to complex band matrix B
CALL CRBCB (A, NLCA, NUCA, B, NLCB, NUCB)
!
! Print results
CALL WRCRN ('B', B)
END
```

## Output

```

 B
 1 2 3
1 (0.000, 0.000) (1.000, 0.000) (1.000, 0.000)
2 (1.000, 0.000) (1.000, 0.000) (1.000, 0.000)
3 (1.000, 0.000) (1.000, 0.000) (0.000, 0.000)
```

---

## CSFRG

Extends a real symmetric matrix defined in its upper triangle to its lower triangle.

### Required Arguments

*A* — *N* by *N* symmetric matrix of order *N* to be filled out. (Input/Output)

## Optional Arguments

$N$  — Order of the matrix  $A$ . (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

## FORTRAN 90 Interface

Generic: `CALL CSFRG (A [,...])`

Specific: The specific interface names are `S_CSFRG` and `D_CSFRG`.

## FORTRAN 77 Interface

Single: `CALL CSFRG (N, A, LDA)`

Double: The double precision name is `DCSFRG`.

## Description

The routine `CSFRG` converts an  $N \times N$  matrix  $A$  in symmetric mode into a general matrix by filling in the lower triangular portion of  $A$  using the values defined in its upper triangular portion.

## Example

The lower triangular portion of a real  $3 \times 3$  symmetric matrix is filled with the values defined in its upper triangular portion.

```
USE CSFRG_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
!
REAL A(LDA,N)
!
! Set values for A
! A = (0.0 3.0 4.0)
! (1.0 5.0)
! (2.0)
!
DATA A/3*0.0, 3.0, 1.0, 0.0, 4.0, 5.0, 2.0/
!
! Fill the lower portion of A
CALL CSFRG (A)
!
! Print results
CALL WRRRN ('A', A)
END
```

## Output

```
 A
 1 2 3
1 0.000 3.000 4.000
2 3.000 1.000 5.000
3 4.000 5.000 2.000
```

---

## CHFCG

Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.

### Required Arguments

*A* — Complex Hermitian matrix of order *N*. (Input/Output)  
On input, the upper triangle of *A* defines a Hermitian matrix. On output, the lower triangle of *A* is defined so that *A* is Hermitian.

### Optional Arguments

*N* — Order of the matrix. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

### FORTRAN 90 Interface

Generic: CALL CHFCG (*A* [, ...])

Specific: The specific interface names are S\_CHFCG and D\_CHFCG.

### FORTRAN 77 Interface

Single: CALL CHFCG (*N*, *A*, *LDA*)

Double: The double precision name is DCHFCG.

### Description

The routine CHFCG converts an  $N \times N$  complex matrix *A* in Hermitian mode into a complex general matrix by filling in the lower triangular portion of *A* using the values defined in its upper triangular portion.

### Comments

Informational errors

| Type | Code |                                                                                   |
|------|------|-----------------------------------------------------------------------------------|
| 3    | 1    | The matrix is not Hermitian. It has a diagonal entry with a small imaginary part. |
| 4    | 2    | The matrix is not Hermitian. It has a diagonal entry with an imaginary part.      |

### Example

A complex  $3 \times 3$  Hermitian matrix defined in its upper triangle is extended to its lower triangle.

```

USE CHFCG_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, N
PARAMETER (LDA=3, N=3)
!
COMPLEX A(LDA,N)
! Set values for A
! A = (1.0+0.0i 1.0+1.0i 1.0+2.0i)
! (2.0+0.0i 2.0+2.0i)
! (3.0+0.0i)
!
DATA A/(1.0,0.0), 2*(0.0,0.0), (1.0,1.0), (2.0,0.0), (0.0,0.0), &
 (1.0,2.0), (2.0,2.0), (3.0,0.0)/
! Fill in lower Hermitian matrix
CALL CHFCG (A)
! Print results
CALL WRCRN ('A', A)
END

```

### Output

```

 A
 1 2 3
1 (1.000, 0.000) (1.000, 1.000) (1.000, 2.000)
2 (1.000,-1.000) (2.000, 0.000) (2.000, 2.000)
3 (1.000,-2.000) (2.000,-2.000) (3.000, 0.000)

```

---

## CSBRB

Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.

### Required Arguments

*A* — Real band symmetric matrix of order *N*. (Input)

*NUCA* — Number of codiagonals in *A*. (Input)

*B* — Real band matrix of order *N* containing a copy of *A*. (Output)

**NLCB** — Number of lower codiagonals in *B*. (Input)  
NLCB must be at least as large as NUCA.

**NUCB** — Number of upper codiagonals in *B*. (Input)  
NUCB must be at least as large as NUCA.

### Optional Arguments

**N** — Order of the matrices *A* and *B*. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

**LDB** — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

### FORTRAN 90 Interface

Generic:     CALL CSBRB (A, NUCA, B, NLCB, NUCB [, ...])

Specific:    The specific interface names are S\_CSBRB and D\_CSBRB.

### FORTRAN 77 Interface

Single:     CALL CSBRB (N, A, LDA, NUCA, B, LDB, NLCB, NUCB)

Double:     The double precision name is DCSBRB.

### Description

The routine CSBRB copies a real matrix *A* stored in symmetric band mode to a matrix *B* stored in band mode. The lower codiagonals of *B* are set using the values from the upper codiagonals of *A*.

### Example

A real matrix of order 4 in band symmetric storage mode with 2 upper codiagonals is copied to a real matrix in band storage mode with 2 upper codiagonals and 2 lower codiagonals.

```
USE CSBRB_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NLCB, NUCA, NUCB
PARAMETER (N=4, NUCA=2, LDA=NUCA+1, NLCB=NUCA, NUCB=NUCA, &
 LDB=NLCB+NUCB+1)
```

```

!
REAL A(LDA,N) , B(LDB,N)
! Set values for A, in band mode
! A = (0.0 0.0 2.0 1.0)
! (0.0 2.0 3.0 1.0)
! (1.0 2.0 3.0 4.0)
!
DATA A/2*0.0, 1.0, 0.0, 2.0, 2.0, 2.0, 3.0, 3.0, 1.0, 1.0, 4.0/
! Copy A to B
CALL CSBRB (A, NUCA, B, NLCB, NUCB)
! Print results
CALL WRRRN ('B', B)
END

```

## Output

|   | B     |       |       |       |
|---|-------|-------|-------|-------|
|   | 1     | 2     | 3     | 4     |
| 1 | 0.000 | 0.000 | 2.000 | 1.000 |
| 2 | 0.000 | 2.000 | 3.000 | 1.000 |
| 3 | 1.000 | 2.000 | 3.000 | 4.000 |
| 4 | 2.000 | 3.000 | 1.000 | 0.000 |
| 5 | 2.000 | 1.000 | 0.000 | 0.000 |

---

## CHBCB

Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.

### Required Arguments

*A* — Complex band Hermitian matrix of order *N*. (Input)

*NUCA* — Number of codiagonals in *A*. (Input)

*B* — Complex band matrix of order *N* containing a copy of *A*. (Output)

*NLCB* — Number of lower codiagonals in *B*. (Input)  
*NLCB* must be at least as large as *NUCA*.

*NUCB* — Number of upper codiagonals in *B*. (Input)  
*NUCB* must be at least as large as *NUCA*.

### Optional Arguments

*N* — Order of the matrices *A* and *B*. (Input)  
Default: *N* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

**LDB** — Leading dimension of  $B$  exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDB = SIZE(B,1)$ .

### **FORTRAN 90 Interface**

Generic:     CALL CHBCB (A, NUCA, B, NLCB, NUCB [, ...])

Specific:    The specific interface names are S\_CHBCB and D\_CHBCB.

### **FORTRAN 77 Interface**

Single:      CALL CHBCB (N, A, LDA, NUCA, B, LDB, NLCB, NUCB)

Double:     The double precision name is DCHBCB.

### **Description**

The routine CSBRB copies a complex matrix  $A$  stored in Hermitian band mode to a matrix  $B$  stored in complex band mode. The lower codiagonals of  $B$  are filled using the values in the upper codiagonals of  $A$ .

### **Comments**

Informational errors

Type   Code

- |   |   |                                                                                                   |
|---|---|---------------------------------------------------------------------------------------------------|
| 3 | 1 | An element on the diagonal has a complex part that is near zero, the complex part is set to zero. |
| 4 | 1 | An element on the diagonal has a complex part that is not zero.                                   |

### **Example**

A complex Hermitian matrix of order 3 in band Hermitian storage mode with one upper codiagonal is copied to a complex matrix in band storage mode.

```

USE CHBCB_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NLCB, NUCA, NUCB
PARAMETER (N=3, NUCA=1, LDA=NUCA+1, NLCB=NUCA, NUCB=NUCA, &
 LDB=NLCB+NUCB+1)
!
COMPLEX A(LDA,N), B(LDB,N)
! Set values for A (in band mode)
! A = (0.0+0.0i -1.0+1.0i -2.0+2.0i)
! (1.0+0.0i 1.0+0.0i 1.0+0.0i)
!

```

```

DATA A/(0.0,0.0), (1.0,0.0), (-1.0,1.0), (1.0,0.0), (-2.0,2.0), &
 (1.0,0.0)/
!
! Copy a complex Hermitian band matrix
! to a complex band matrix
CALL CHBCB (A, NUCA, B, NLCB, NUCB)
!
! Print results
CALL WRCRN ('B', B)
END

```

## Output

```

 B
 1 2 3
1 (0.000, 0.000) (-1.000, 1.000) (-2.000, 2.000)
2 (1.000, 0.000) (1.000, 0.000) (1.000, 0.000)
3 (-1.000,-1.000) (-2.000,-2.000) (0.000, 0.000)

```

---

## TRNRR

Transposes a rectangular matrix.

### Required Arguments

**A** — Real *NRA* by *NCA* matrix in full storage mode. (Input)

**B** — Real *NRB* by *NCB* matrix in full storage mode containing the transpose of *A*. (Output)

### Optional Arguments

**NRA** — Number of rows of *A*. (Input)  
Default: *NRA* = *SIZE* (*A*,1).

**NCA** — Number of columns of *A*. (Input)  
Default: *NCA* = *SIZE* (*A*,2).

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

**NRB** — Number of rows of *B*. (Input)  
*NRB* must be equal to *NCA*.  
Default: *NRB* = *SIZE* (*B*,1).

**NCB** — Number of columns of *B*. (Input)  
*NCB* must be equal to *NRA*.  
Default: *NCB* = *SIZE* (*B*,2).

**LDB** — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDB* = *SIZE* (*B*,1).

## FORTRAN 90 Interface

Generic:     CALL TRNRR (A, B [, ...])

Specific:    The specific interface names are S\_TRNRR and D\_TRNRR.

## FORTRAN 77 Interface

Single:     CALL TRNRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB)

Double:     The double precision name is DTRNRR.

## Description

The routine TRNRR computes the transpose  $B = A^T$  of a real rectangular matrix  $A$ .

## Comments

If  $LDA = LDB$  and  $NRA = NCA$ , then  $A$  and  $B$  can occupy the same storage locations; otherwise,  $A$  and  $B$  must be stored separately.

## Example

Transpose the  $5 \times 3$  real rectangular matrix  $A$  into the  $3 \times 5$  real rectangular matrix  $B$ .

```
USE TRNRR_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NCB, NRA, NRB
PARAMETER (NCA=3, NCB=5, NRA=5, NRB=3)
!
REAL A(NRA,NCA), B(NRB,NCB)
! Set values for A
! 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)
! (51.0 52.0 53.0)
!
DATA A/11.0, 21.0, 31.0, 41.0, 51.0, 12.0, 22.0, 32.0, 42.0, &
 52.0, 13.0, 23.0, 33.0, 43.0, 53.0/
! B = transpose(A)
CALL TRNRR (A, B)
! Print results
CALL WRRRN ('B = trans(A)', B)
END
```

## Output

```
 B = trans(A)
 1 2 3 4 5
1 11.00 21.00 31.00 41.00 51.00
2 12.00 22.00 32.00 42.00 52.00
3 13.00 23.00 33.00 43.00 53.00
```

---

## MXTXF



Computes the transpose product of a matrix,  $A^T A$ .

### Required Arguments

**A** — Real  $NRA$  by  $NCA$  rectangular matrix. (Input)  
The transpose product of **A** is to be computed.

**B** — Real  $NB$  by  $NB$  symmetric matrix containing the transpose product  $A^T A$ . (Output)

### Optional Arguments

**NRA** — Number of rows in **A**. (Input)  
Default:  $NRA = \text{SIZE}(A,1)$ .

**NCA** — Number of columns in **A**. (Input)  
Default:  $NCA = \text{SIZE}(A,2)$ .

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

**NB** — Order of the matrix **B**. (Input)  
**NB** must be equal to **NCA**.  
Default:  $NB = \text{SIZE}(B,1)$ .

**LDB** — Leading dimension of **B** exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

### FORTRAN 90 Interface

Generic:    `CALL MXTXF (A, B [, ...])`

Specific:    The specific interface names are `S_MXTXF` and `D_MXTXF`.

## FORTRAN 77 Interface

Single:      CALL MXTXF (NRA, NCA, A, LDA, NB, B, LDB)

Double:     The double precision name is DMXTXF.

## Description

The routine MXTXF computes the real general matrix  $B = A^T A$  given the real rectangular matrix  $A$ .

## Example

Multiply the transpose of a  $3 \times 4$  real matrix by itself. The output matrix will be a  $4 \times 4$  real symmetric matrix.

```
 USE MXTXF_INT
 USE WRRRN_INT

 IMPLICIT NONE
!
! Declare variables
 INTEGER NB, NCA, NRA
 PARAMETER (NB=4, NCA=4, NRA=3)
!
 REAL A(NRA,NCA), B(NB,NB)
!
! Set values for A
! A = (3.0 1.0 4.0 2.0)
! (0.0 2.0 1.0 -1.0)
! (6.0 1.0 3.0 2.0)
!
 DATA A/3.0, 0.0, 6.0, 1.0, 2.0, 1.0, 4.0, 1.0, 3.0, 2.0, -1.0, &
 2.0/
!
! Compute B = trans(A)*A
 CALL MXTXF (A, B)
!
! Print results
 CALL WRRRN ('B = trans(A)*A', B)
 END
```

## Output

```
 B = trans(A)*A
 1 2 3 4
1 45.00 9.00 30.00 18.00
2 9.00 6.00 9.00 2.00
3 30.00 9.00 26.00 13.00
4 18.00 2.00 13.00 9.00
```

---

# MXTYF



Multiplies the transpose of matrix  $A$  by matrix  $B$ ,  $A^T B$ .

## Required Arguments

$A$  — Real  $NRA$  by  $NCA$  matrix. (Input)

$B$  — Real  $NRB$  by  $NCB$  matrix. (Input)

$C$  — Real  $NCA$  by  $NCB$  matrix containing the transpose product  $A^T B$ . (Output)

## Optional Arguments

$NRA$  — Number of rows in  $A$ . (Input)  
Default:  $NRA = \text{SIZE}(A,1)$ .

$NCA$  — Number of columns in  $A$ . (Input)  
Default:  $NCA = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

$NRB$  — Number of rows in  $B$ . (Input)  
 $NRB$  must be the same as  $NRA$ .  
Default:  $NRB = \text{SIZE}(B,1)$ .

$NCB$  — Number of columns in  $B$ . (Input)  
Default:  $NCB = \text{SIZE}(B,2)$ .

$LDB$  — Leading dimension of  $B$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDB = \text{SIZE}(B,1)$ .

$NRC$  — Number of rows of  $C$ . (Input)  
 $NRC$  must be equal to  $NCA$ .  
Default:  $NRC = \text{SIZE}(C,1)$ .

$NCC$  — Number of columns of  $C$ . (Input)  
 $NCC$  must be equal to  $NCB$ .  
Default:  $NCC = \text{SIZE}(C,2)$ .

**LDC** — Leading dimension of  $C$  exactly as specified in the dimension statement of the calling program. (Input)  
 Default:  $LDC = \text{SIZE}(C,1)$ .

### FORTRAN 90 Interface

Generic:     CALL MXTYF (A, B, C [, ...])

Specific:    The specific interface names are S\_MXTYF and D\_MXTYF.

### FORTRAN 77 Interface

Single:      CALL MXTYF (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)

Double:     The double precision name is DMXTYF.

### Description

The routine MXTYF computes the real general matrix  $C = A^T B$  given the real rectangular matrices  $A$  and  $B$ .

### Example

Multiply the transpose of a  $3 \times 4$  real matrix by a  $3 \times 3$  real matrix. The output matrix will be a  $4 \times 3$  real matrix.

```

USE MXTYF_INT
USE WRRRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER NCA, NCB, NCC, NRA, NRB, NRC
PARAMETER (NCA=4, NCB=3, NCC=3, NRA=3, NRB=3, NRC=4)
!
REAL A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
!
! Set values for B
! B = (-1.0 2.0 0.0)
! (3.0 0.0 -1.0)
! (0.0 5.0 2.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0/
!
! Compute C = trans(A)*B
CALL MXTYF (A, B, C)
!
! Print results

```

```
CALL WRRRN ('C = trans(A)*B', C)
END
```

## Output

```
 C = trans(A)*B
 1 2 3
1 8.00 12.00 1.00
2 12.00 5.00 -2.00
3 -5.00 14.00 5.00
4 0.00 5.00 2.00
```

---

## MXYTF

Multiplies a matrix  $A$  by the transpose of a matrix  $B$ ,  $AB^T$ .

### Required Arguments

$A$  — Real  $NRA$  by  $NCA$  rectangular matrix. (Input)

$B$  — Real  $NRB$  by  $NCB$  rectangular matrix. (Input)

$C$  — Real  $NRC$  by  $NCC$  rectangular matrix containing the transpose product  $AB^T$ . (Output)

### Optional Arguments

$NRA$  — Number of rows in  $A$ . (Input)

Default:  $NRA = \text{SIZE}(A,1)$ .

$NCA$  — Number of columns in  $A$ . (Input)

Default:  $NCA = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{SIZE}(A,1)$ .

$NRB$  — Number of rows in  $B$ . (Input)

Default:  $NRB = \text{SIZE}(B,1)$ .

$NCB$  — Number of columns in  $B$ . (Input)

$NCB$  must be the same as  $NCA$ .

Default:  $NCB = \text{SIZE}(B,2)$ .

$LDB$  — Leading dimension of  $B$  exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDB = \text{SIZE}(B,1)$ .

$NRC$  — Number of rows of  $C$ . (Input)

$NRC$  must be equal to  $NRA$ .

Default:  $NRC = \text{SIZE}(C,1)$ .

**NCC** — Number of columns of *C*. (Input)

NCC must be equal to NRB.

Default: NCC = SIZE (C,2).

**LDC** — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)

Default: LDC = SIZE (C,1).

## FORTRAN 90 Interface

Generic: CALL MXYTF (A, B, C [, ...])

Specific: The specific interface names are S\_MXYTF and D\_MXYTF.

## FORTRAN 77 Interface

Single: CALL MXYTF (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)

Double: The double precision name is DMXYTF.

## Description

The routine MXYTF computes the real general matrix  $C = AB^T$  given the real rectangular matrices *A* and *B*.

## Example

Multiply a  $3 \times 4$  real matrix by the transpose of a  $3 \times 4$  real matrix. The output matrix will be a  $3 \times 3$  real matrix.

```
USE MXYTF_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NCB, NCC, NRA, NRB, NRC
PARAMETER (NCA=4, NCB=4, NCC=3, NRA=3, NRB=3, NRC=3)
!
REAL A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
!
! Set values for B
! B = (-1.0 2.0 0.0 2.0)
! (3.0 0.0 -1.0 -1.0)
! (0.0 5.0 2.0 5.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
```

```

DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0, 2.0, -1.0, &
 5.0/
!
! Compute C = A*trans(B)
CALL MXYTF (A, B, C)
! Print results
CALL WRRRN ('C = A*trans(B)', C)
END

```

## Output

```

 C = A*trans(B)
 1 2 3
1 -1.00 1.00 4.00
2 5.00 10.00 18.00
3 2.00 3.00 14.00

```

---

## MRRRR



Multiplies two real rectangular matrices,  $AB$ .

### Required Arguments

$A$  — Real  $NRA$  by  $NCA$  matrix in full storage mode. (Input)

$B$  — Real  $NRB$  by  $NCB$  matrix in full storage mode. (Input)

$C$  — Real  $NRC$  by  $NCC$  matrix containing the product  $AB$  in full storage mode. (Output)

### Optional Arguments

$NRA$  — Number of rows of  $A$ . (Input)  
Default:  $NRA = \text{SIZE}(A,1)$ .

$NCA$  — Number of columns of  $A$ . (Input)  
Default:  $NCA = \text{SIZE}(A,2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

$NRB$  — Number of rows of  $B$ . (Input)  
 $NRB$  must be equal to  $NCA$ .  
Default:  $NRB = \text{SIZE}(B,1)$ .

$NCB$  — Number of columns of  $B$ . (Input)  
Default:  $NCB = \text{SIZE}(B,2)$ .

**LDB** — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDB = SIZE (B,1)`.

**NRC** — Number of rows of *C*. (Input)

NRC must be equal to NRA.

Default: `NRC = SIZE (C,1)`.

**NCC** — Number of columns of *C*. (Input)

NCC must be equal to NCB.

Default: `NCC = SIZE (C,2)`.

**LDC** — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDC = SIZE (C,1)`.

## **FORTRAN 90 Interface**

Generic: `CALL MRRRR (A, B, C [, ...])`

Specific: The specific interface names are `S_MRRRR` and `D_MRRRR`.

## **FORTRAN 77 Interface**

Single: `CALL MRRRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)`

Double: The double precision name is `DMRRRR`.

## **Description**

Given the real rectangular matrices *A* and *B*, `MRRRR` computes the real rectangular matrix  $C = AB$ .

## **Example**

Multiply a  $3 \times 4$  real matrix by a  $4 \times 3$  real matrix. The output matrix will be a  $3 \times 3$  real matrix.

```
USE MRRRR_INT

USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NCB, NCC, NRA, NRB, NRC
PARAMETER (NCA=4, NCB=3, NCC=3, NRA=3, NRB=4, NRC=3)
!
REAL A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
```

```

!
!
! Set values for B
! B = (-1.0 0.0 2.0)
! (3.0 5.0 2.0)
! (0.0 0.0 -1.0)
! (2.0 -1.0 5.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
DATA B/-1.0, 3.0, 0.0, 2.0, 0.0, 5.0, 0.0, -1.0, 2.0, 2.0, -1.0, &
5.0/
!
! Compute C = A*B
CALL MRRRR (A, B, C)
!
! Print results
CALL WRRRN ('C = A*B', C)
END

```

## Output

```

 C = A*B
 1 2 3
1 -1.00 0.00 0.00
2 9.00 20.00 15.00
3 3.00 4.00 9.00

```

---

## MCRCR

Multiplies two complex rectangular matrices,  $AB$ .

### Required Arguments

- $A$  — Complex  $NRA$  by  $NCA$  rectangular matrix. (Input)
- $B$  — Complex  $NRB$  by  $NCB$  rectangular matrix. (Input)
- $C$  — Complex  $NRC$  by  $NCC$  rectangular matrix containing the product  $A * B$ . (Output)

### Optional Arguments

- $NRA$  — Number of rows of  $A$ . (Input)  
Default:  $NRA = SIZE(A,1)$ .
- $NCA$  — Number of columns of  $A$ . (Input)  
Default:  $NCA = SIZE(A,2)$ .
- $LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = SIZE(A,1)$ .
- $NRB$  — Number of rows of  $B$ . (Input)  
 $NRB$  must be equal to  $NCA$ .  
Default:  $NRB = SIZE(B,1)$ .

**NCB** — Number of columns of *B*. (Input)

Default: `NCB = SIZE (B,2)`.

**LDB** — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDB = SIZE (B,1)`.

**NRC** — Number of rows of *C*. (Input)

NRC must be equal to NRA.

Default: `NRC = SIZE (C,1)`.

**NCC** — Number of columns of *C*. (Input)

NCC must be equal to NCB.

Default: `NCC = SIZE (C,2)`.

**LDC** — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDC = SIZE (C,1)`.

## **FORTRAN 90 Interface**

Generic: `CALL MCRCR (A, B, C [, ...])`

Specific: The specific interface names are `S_MCRCR` and `D_MCRCR`.

## **FORTRAN 77 Interface**

Single: `CALL MCRCR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)`

Double: The double precision name is `DMCRCR`.

## **Description**

Given the complex rectangular matrices *A* and *B*, `MCRCR` computes the complex rectangular matrix  $C = AB$ .

## **Example**

Multiply a  $3 \times 4$  complex matrix by a  $4 \times 3$  complex matrix. The output matrix will be a  $3 \times 3$  complex matrix.

```
USE MCRCR_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NCB, NCC, NRA, NRB, NRC
PARAMETER (NCA=4, NCB=3, NCC=3, NRA=3, NRB=4, NRC=3)
!
```

```

COMPLEX A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
!
! Set values for A
! A = (1.0 + 1.0i -1.0+ 2.0i 0.0 + 1.0i 0.0 - 2.0i)
! (3.0 + 7.0i 6.0 - 4.0i 2.0 - 1.0i 0.0 + 1.0i)
! (1.0 + 0.0i 1.0 - 2.0i -2.0+ 0.0i 0.0 + 0.0i)
!
! Set values for B
! B = (2.0 + 1.0i 3.0 + 2.0i 3.0 + 1.0i)
! (2.0 - 1.0i 4.0 - 2.0i 5.0 - 3.0i)
! (1.0 + 0.0i 0.0 - 1.0i 0.0 + 1.0i)
! (2.0 + 1.0i 1.0 + 2.0i 0.0 - 1.0i)
!
DATA A/(1.0,1.0), (3.0,7.0), (1.0,0.0), (-1.0,2.0), (6.0,-4.0), &
 (1.0,-2.0), (0.0,1.0), (2.0,-1.0), (-2.0,0.0), (0.0,-2.0), &
 (0.0,1.0), (0.0,0.0)/
DATA B/(2.0,1.0), (2.0,-1.0), (1.0,0.0), (2.0,1.0), (3.0,2.0), &
 (4.0,-2.0), (0.0,-1.0), (1.0,2.0), (3.0,1.0), (5.0,-3.0), &
 (0.0,1.0), (0.0,-1.0)/
!
! Compute C = A*B
CALL MCRCR (A, B, C)
!
! Print results
CALL WRCRN ('C = A*B', C)
END

```

## Output

```

 C = A*B
 1 2 3
1 (3.00, 5.00) (6.00, 13.00) (0.00, 17.00)
2 (8.00, 4.00) (8.00, -2.00) (22.00,-12.00)
3 (0.00, -4.00) (3.00, -6.00) (2.00,-14.00)

```

---

# HRRRR

Computes the Hadamard product of two real rectangular matrices.

## Required Arguments

**A** — Real *NRA* by *NCA* rectangular matrix. (Input)

**B** — Real *NRB* by *NCB* rectangular matrix. (Input)

**C** — Real *NRC* by *NCC* rectangular matrix containing the Hadamard product of **A** and **B**. (Output)

If **A** is not needed, then **C** can share the same storage locations as **A**. Similarly, if **B** is not needed, then **C** can share the same storage locations as **B**.

## Optional Arguments

**NRA** — Number of rows of **A**. (Input)

Default: *NRA* = *SIZE* (**A**,1).

**NCA** — Number of columns of A. (Input)

Default:  $NCA = SIZE(A,2)$ .

**LDA** — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = SIZE(A,1)$ .

**NRB** — Number of rows of B. (Input)

NRB must be equal to NRA.

Default:  $NRB = SIZE(B,1)$ .

**NCB** — Number of columns of B. (Input)

NCB must be equal to NCA.

Default:  $NCB = SIZE(B,2)$ .

**LDB** — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDB = SIZE(B,1)$ .

**NRC** — Number of rows of C. (Input)

NRC must be equal to NRA.

Default:  $NRC = SIZE(C,1)$ .

**NCC** — Number of columns of C. (Input)

NCC must be equal to NCA.

Default:  $NCC = SIZE(C,2)$ .

**LDC** — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDC = SIZE(C,1)$ .

## **FORTRAN 90 Interface**

Generic: `CALL HRRRR (A, B, C [, ...])`

Specific: The specific interface names are `S_HRRRR` and `D_HRRRR`.

## **FORTRAN 77 Interface**

Single: `CALL HRRRR (NRA, NCA, A, LDA, NRB, NCB, B, LDB, NRC, NCC, C, LDC)`

Double: The double precision name is `DHRRRR`.

## **Description**

The routine `HRRRR` computes the Hadamard product of two real matrices  $A$  and  $B$  and returns a real matrix  $C$ , where  $C_{ij} = A_{ij}B_{ij}$ .

## Example

Compute the Hadamard product of two  $4 \times 4$  real matrices. The output matrix will be a  $4 \times 4$  real matrix.

```
USE HRRRR_INT
USE WRRRN_INT

IMPLICIT NONE

! Declare variables
INTEGER NCA, NCB, NCC, NRA, NRB, NRC
PARAMETER (NCA=4, NCB=4, NCC=4, NRA=4, NRB=4, NRC=4)

!
REAL A(NRA,NCA), B(NRB,NCB), C(NRC,NCC)
! Set values for A
! A = (-1.0 0.0 -3.0 8.0)
! (2.0 1.0 7.0 2.0)
! (3.0 -2.0 2.0 -6.0)
! (4.0 1.0 -5.0 -8.0)
!
! Set values for B
! B = (2.0 3.0 0.0 -10.0)
! (1.0 -1.0 4.0 2.0)
! (-1.0 -2.0 7.0 1.0)
! (2.0 1.0 9.0 0.0)
!
DATA A/-1.0, 2.0, 3.0, 4.0, 0.0, 1.0, -2.0, 1.0, -3.0, 7.0, 2.0, &
 -5.0, 8.0, 2.0, -6.0, -8.0/
DATA B/2.0, 1.0, -1.0, 2.0, 3.0, -1.0, -2.0, 1.0, 0.0, 4.0, 7.0, &
 9.0, -10.0, 2.0, 1.0, 0.0/
! Compute Hadamard product of A and B
CALL HRRRR (A, B, C)
! Print results
CALL WRRRN ('C = A (*) B', C)
END
```

## Output

```
 C = A (*) B
 1 2 3 4
1 -2.00 0.00 0.00 -80.00
2 2.00 -1.00 28.00 4.00
3 -3.00 4.00 14.00 -6.00
4 8.00 1.00 -45.00 0.00
```

---

## BLINF

This function computes the bilinear form  $x^T Ay$ .

### Function Return Value

*BLINF* — The value of  $x^T Ay$  is returned in *BLINF*. (Output)

## Required Arguments

*A* — Real *NRA* by *NCA* matrix. (Input)

*X* — Real vector of length *NRA*. (Input)

*Y* — Real vector of length *NCA*. (Input)

## Optional Arguments

*NRA* — Number of rows of *A*. (Input)

Default: *NRA* = *SIZE* (*A*, 1).

*NCA* — Number of columns of *A*. (Input)

Default: *NCA* = *SIZE* (*A*, 2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = *SIZE* (*A*, 1).

## FORTRAN 90 Interface

Generic: `BLINF (A, X, Y [, ...])`

Specific: The specific interface names are `S_BLINF` and `D_BLINF`.

## FORTRAN 77 Interface

Single: `BLINF (NRA, NCA, A, LDA, X, Y)`

Double: The double precision name is `DBLINF`.

## Description

Given the real rectangular matrix *A* and two vectors *x* and *y*, `BLINF` computes the bilinear form  $x^T A y$ .

## Comments

The quadratic form can be computed by calling `BLINF` with the vector *X* in place of the vector *Y*.

## Example

Compute the bilinear form  $x^T A y$ , where *x* is a vector of length 5, *A* is a  $5 \times 2$  matrix and *y* is a vector of length 2.

```
USE BLINF_INT
```

```

USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER NCA, NRA
PARAMETER (NCA=2, NRA=5)
!
INTEGER NOUT
REAL A(NRA,NCA), VALUE, X(NRA), Y(NCA)
! Set values for A
! A = (-2.0 2.0)
! (3.0 -6.0)
! (-4.0 7.0)
! (1.0 -8.0)
! (0.0 10.0)
! Set values for X
! X = (1.0 -2.0 3.0 -4.0 -5.0)
! Set values for Y
! Y = (-6.0 3.0)
!
DATA A/-2.0, 3.0, -4.0, 1.0, 0.0, 2.0, -6.0, 7.0, -8.0, 10.0/
DATA X/1.0, -2.0, 3.0, -4.0, -5.0/
DATA Y/-6.0, 3.0/
!
! Compute bilinear form
VALUE = BLINF(A,X,Y)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The bilinear form trans(x)*A*y = ', VALUE
END

```

## Output

The bilinear form  $\text{trans}(x) \cdot A \cdot y = 195.000$

# POLRG



Evaluates a real general matrix polynomial.

## Required Arguments

**A** —  $N$  by  $N$  matrix for which the polynomial is to be computed. (Input)

**COEF** — Vector of length  $N_{\text{COEF}}$  containing the coefficients of the polynomial in order of increasing power. (Input)

**B** —  $N$  by  $N$  matrix containing the value of the polynomial evaluated at **A**. (Output)

## Optional Arguments

*N* — Order of the matrix *A*. (Input)

Default: *N* = SIZE (*A*,1).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = SIZE (*A*,1).

*NCOEF* — Number of coefficients. (Input)

Default: *NCOEF* = SIZE (*COEF*,1).

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDB* = SIZE (*B*,1).

## FORTRAN 90 Interface

Generic:     CALL POLRG (*A*, *COEF*, *B* [, ...])

Specific:    The specific interface names are *S\_POLRG* and *D\_POLRG*.

## FORTRAN 77 Interface

Single:     CALL POLRG (*N*, *A*, *LDA*, *NCOEF*, *COEF*, *B*, *LDB*)

Double:     The double precision name is *DPOLRG*.

## Description

Let  $m = \text{NCOEF}$  and  $c = \text{COEF}$ .

The routine *POLRG* computes the matrix polynomial

$$B = \sum_{k=1}^m c_k A^{k-1}$$

using Horner's scheme

$$B = \left( \dots \left( (c_m A + c_{m-1} I) A + c_{m-2} I \right) A + \dots + c_1 I \right)$$

where *I* is the  $N \times N$  identity matrix.

## Comments

Workspace may be explicitly provided, if desired, by use of *P2LRG/DP2LRG*. The reference is

```
CALL P2LRG (N, A, LDA, NCOEF, COEF, B, LDB, WORK)
```

The additional argument is

**WORK** — Work vector of length  $N * N$ .

## Example

This example evaluates the matrix polynomial  $3I + A + 2A^2$ , where  $A$  is a  $3 \times 3$  matrix.

```
USE POLRG_INT
USE WRRRN_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, LDB, N, NCOEF
PARAMETER (N=3, NCOEF=3, LDA=N, LDB=N)
!
REAL A(LDA,N), B(LDB,N), COEF(NCOEF)
! Set values of A and COEF
!
! A = (1.0 3.0 2.0)
! (-5.0 1.0 7.0)
! (1.0 5.0 -4.0)
!
! COEF = (3.0, 1.0, 2.0)
!
DATA A/1.0, -5.0, 1.0, 3.0, 1.0, 5.0, 2.0, 7.0, -4.0/
DATA COEF/3.0, 1.0, 2.0/
!
! Evaluate B = 3I + A + 2*A**2
CALL POLRG (A, COEF, B)
!
! Print B
CALL WRRRN ('B = 3I + A + 2*A**2', B)
END
```

## Output

```
B = 3I + A + 2*A**2
 1 2 3
1 -20.0 35.0 32.0
2 -11.0 46.0 -55.0
3 -55.0 -19.0 105.0
```

---

# MURRV

Multiplies a real rectangular matrix by a vector.

## Required Arguments

**A** — Real  $NRA$  by  $NCA$  rectangular matrix. (Input)

**X** — Real vector of length  $NX$ . (Input)

*Y* — Real vector of length *NY* containing the product  $A * X$  if *IPATH* is equal to 1 and the product  $\text{trans}(A) * X$  if *IPATH* is equal to 2. (Output)

### Optional Arguments

*NRA* — Number of rows of *A*. (Input)  
Default: *NRA* = *SIZE* (*A*,1).

*NCA* — Number of columns of *A*. (Input)  
Default: *NCA* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*,1).

*NX* — Length of the vector *X*. (Input)  
*NX* must be equal to *NCA* if *IPATH* is equal to 1. *NX* must be equal to *NRA* if *IPATH* is equal to 2.  
Default: *NX* = *SIZE* (*X*,1).

*IPATH* — Integer flag. (Input)  
*IPATH* = 1 means the product  $Y = A * X$  is computed. *IPATH* = 2 means the product  $Y = \text{trans}(A) * X$  is computed, where  $\text{trans}(A)$  is the transpose of *A*.  
Default: *IPATH* = 1.

*NY* — Length of the vector *Y*. (Input)  
*NY* must be equal to *NRA* if *IPATH* is equal to 1. *NY* must be equal to *NCA* if *IPATH* is equal to 2.  
Default: *NY* = *SIZE* (*Y*,1).

### FORTRAN 90 Interface

Generic:     CALL MURRV (*A*, *X*, *Y* [, ...])

Specific:    The specific interface names are *S\_MURRV* and *D\_MURRV*.

### FORTRAN 77 Interface

Single:     CALL MURRV (*NRA*, *NCA*, *A*, *LDA*, *NX*, *X*, *IPATH*, *NY*, *Y*)

Double:     The double precision name is *DMURRV*.

### Description

If *IPATH* = 1, *MURRV* computes  $y = Ax$ , where *A* is a real general matrix and *x* and *y* are real vectors. If *IPATH* = 2, *MURRV* computes  $y = A^T x$ .

## Example

Multiply a  $3 \times 3$  real matrix by a real vector of length 3. The output vector will be a real vector of length 3.

```
 USE MURRV_INT
 USE WRRRN_INT

 IMPLICIT NONE

! Declare variables
 INTEGER LDA, NCA, NRA, NX, NY
 PARAMETER (NCA=3, NRA=3, NX=3, NY=3)

!
 INTEGER IPATH
 REAL A(NRA,NCA), X(NX), Y(NY)

! Set values for A and X
! A = (1.0 0.0 2.0)
! (0.0 3.0 0.0)
! (4.0 1.0 2.0)
!
! X = (1.0 2.0 1.0)
!
!
 DATA A/1.0, 0.0, 4.0, 0.0, 3.0, 1.0, 2.0, 0.0, 2.0/
 DATA X/1.0, 2.0, 1.0/

! Compute y = Ax
 IPATH = 1
 CALL MURRV (A, X, Y)

! Print results
 CALL WRRRN ('y = Ax', Y, 1, NY, 1)
 END
```

## Output

```
 y = Ax
 1 2 3
3.000 6.000 8.000
```

---

# MURBV

Multiplies a real band matrix in band storage mode by a real vector.

## Required Arguments

**A** — Real  $NLCA + NUCA + 1$  by  $N$  band matrix stored in band mode. (Input)

**NLCA** — Number of lower codiagonals in **A**. (Input)

**NUCA** — Number of upper codiagonals in **A**. (Input)

**X** — Real vector of length  $NX$ . (Input)

$Y$  — Real vector of length  $NY$  containing the product  $A * X$  if  $IPATH$  is equal to 1 and the product  $trans(A) * X$  if  $IPATH$  is equal to 2. (Output)

### Optional Arguments

$N$  — Order of the matrix. (Input)  
Default:  $N = SIZE(A, 2)$ .

$LDA$  — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = SIZE(A, 1)$ .

$NX$  — Length of the vector  $X$ . (Input)  
 $NX$  must be equal to  $N$ .  
Default:  $NX = SIZE(X, 1)$ .

$IPATH$  — Integer flag. (Input)  
 $IPATH = 1$  means the product  $Y = A * X$  is computed.  $IPATH = 2$  means the product  $Y = trans(A) * X$  is computed, where  $trans(A)$  is the transpose of  $A$ .  
Default:  $IPATH = 1$ .

$NY$  — Length of vector  $Y$ . (Input)  
 $NY$  must be equal to  $N$ .  
Default:  $NY = SIZE(Y, 1)$ .

### FORTRAN 90 Interface

Generic: `CALL MURBV (A, NLCA, NUCA, X, Y [, ...])`

Specific: The specific interface names are `S_MURBV` and `D_MURBV`.

### FORTRAN 77 Interface

Single: `CALL MURBV (N, A, LDA, NLCA, NUCA, NX, X, IPATH, NY, Y)`

Double: The double precision name is `DMURBV`.

### Description

If  $IPATH = 1$ , `MURBV` computes  $y = Ax$ , where  $A$  is a real band matrix and  $x$  and  $y$  are real vectors.  
If  $IPATH = 2$ , `MURBV` computes  $y = A^T x$ .

### Example

Multiply a real band matrix of order 6, with two upper codiagonals and two lower codiagonals stored in band mode, by a real vector of length 6. The output vector will be a real vector of length 6.

```

USE MURBV_INT
USE WRRRN_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, N, NLCA, NUCA, NX, NY
PARAMETER (LDA=5, N=6, NLCA=2, NUCA=2, NX=6, NY=6)

!
INTEGER IPATH
REAL A(LDA,N), X(NX), Y(NY)

! Set values for A (in band mode)
! A = (0.0 0.0 1.0 2.0 3.0 4.0)
! (0.0 1.0 2.0 3.0 4.0 5.0)
! (1.0 2.0 3.0 4.0 5.0 6.0)
! (-1.0 -2.0 -3.0 -4.0 -5.0 0.0)
! (-5.0 -6.0 -7.0 -8.0 0.0 0.0)
!
! Set values for X
! X = (-1.0 2.0 -3.0 4.0 -5.0 6.0)
!
DATA A/0.0, 0.0, 1.0, -1.0, -5.0, 0.0, 1.0, 2.0, -2.0, -6.0, &
 1.0, 2.0, 3.0, -3.0, -7.0, 2.0, 3.0, 4.0, -4.0, -8.0, 3.0, &
 4.0, 5.0, -5.0, 0.0, 4.0, 5.0, 6.0, 0.0, 0.0/
DATA X/-1.0, 2.0, -3.0, 4.0, -5.0, 6.0/

! Compute y = Ax
IPATH = 1
CALL MURBV (A, NLCA, NUCA, X, Y)

! Print results
CALL WRRRN ('y = Ax', Y, 1, NY, 1)
END

```

## Output

```

 y = Ax
 1 2 3 4 5 6
-2.00 7.00 -11.00 17.00 10.00 29.00

```

---

# MUCRV

Multiplies a complex rectangular matrix by a complex vector.

## Required Arguments

**A** — Complex NRA by NCA rectangular matrix. (Input)

**X** — Complex vector of length NX. (Input)

**Y** — Complex vector of length NY containing the product  $A * X$  if IPATH is equal to 1 and the product  $\text{trans}(A) * X$  if IPATH is equal to 2. (Output)

## Optional Arguments

**NRA** — Number of rows of *A*. (Input)

Default: `NRA = SIZE (A,1)`.

**NCA** — Number of columns of *A*. (Input)

Default: `NCA = SIZE (A,2)`.

**LDA** — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: `LDA = SIZE (A,1)`.

**NX** — Length of the vector *x*. (Input)

*NX* must be equal to *NCA* if *IPATH* is equal to 1. *NX* must be equal to *NRA* if *IPATH* is equal to 2.

Default: `NX = SIZE (X,1)`.

**IPATH** — Integer flag. (Input)

*IPATH* = 1 means the product  $Y = A * X$  is computed. *IPATH* = 2 means the product  $Y = \text{trans}(A) * X$  is computed, where  $\text{trans}(A)$  is the transpose of *A*.

Default: `IPATH = 1`.

**NY** — Length of the vector *y*. (Input)

*NY* must be equal to *NRA* if *IPATH* is equal to 1. *NY* must be equal to *NCA* if *IPATH* is equal to 2.

Default: `NY = SIZE (Y,1)`.

## FORTRAN 90 Interface

Generic: `CALL MUCRV (A, X, Y [, ...])`

Specific: The specific interface names are `S_MUCRV` and `D_MUCRV`.

## FORTRAN 77 Interface

Single: `CALL MUCRV (NRA, NCA, A, LDA, NX, X, IPATH, NY, Y)`

Double: The double precision name is `DMUCRV`.

## Description

If *IPATH* = 1, `MUCRV` computes  $y = Ax$ , where *A* is a complex general matrix and *x* and *y* are complex vectors. If *IPATH* = 2, `MUCRV` computes  $y = A^T x$ .

## Example

Multiply a  $3 \times 3$  complex matrix by a complex vector of length 3. The output vector will be a complex vector of length 3.

```

USE MUCRV_INT
USE WRCRN_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NRA, NX, NY
PARAMETER (NCA=3, NRA=3, NX=3, NY=3)
!
INTEGER IPATH
COMPLEX A(NRA,NCA), X(NX), Y(NY)
!
! Set values for A and X
! A = (1.0 + 2.0i 3.0 + 4.0i 1.0 + 0.0i)
! (2.0 + 1.0i 3.0 + 2.0i 0.0 - 1.0i)
! (2.0 - 1.0i 1.0 + 0.0i 0.0 + 1.0i)
!
! X = (1.0 - 1.0i 2.0 - 2.0i 0.0 - 1.0i)
!
DATA A/(1.0,2.0), (2.0,1.0), (2.0,-1.0), (3.0,4.0), (3.0,2.0), &
 (1.0,0.0), (1.0,0.0), (0.0,-1.0), (0.0,1.0)/
DATA X/(1.0,-1.0), (2.0,-2.0), (0.0,-1.0)/
! Compute y = Ax
IPATH = 1
CALL MUCRV (A, X, Y)
! Print results
CALL WRCRN ('y = Ax', Y, 1, NY, 1)
END

```

## Output

```

 y = Ax
 1 2 3
(17.00, 2.00) (12.00, -3.00) (4.00, -5.00)

```

---

## MUCBV

Multiplies a complex band matrix in band storage mode by a complex vector.

### Required Arguments

*A* — Complex  $NLCA + NUCA + 1$  by  $N$  band matrix stored in band mode. (Input)

*NLCA* — Number of lower codiagonals in *A*. (Input)

*NUCA* — Number of upper codiagonals in *A*. (Input)

*X* — Complex vector of length  $NX$ . (Input)

*Y* — Complex vector of length  $NY$  containing the product  $A * X$  if *IPATH* is equal to 1 and the product  $\text{trans}(A) * X$  if *IPATH* is equal to 2. (Output)

## Optional Arguments

*N* — Order of the matrix. (Input)

Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default:  $LDA = \text{SIZE}(A,1)$ .

*NX* — Length of the vector *X*. (Input)

*NX* must be equal to *N*.

Default:  $NX = \text{SIZE}(X,1)$ .

*IPATH* — Integer flag. (Input)

*IPATH* = 1 means the product  $Y = A * X$  is computed. *IPATH* = 2 means the product  $Y = \text{trans}(A) * X$  is computed, where  $\text{trans}(A)$  is the transpose of *A*.

Default: *IPATH* = 1.

*NY* — Length of vector *Y*. (Input)

*NY* must be equal to *N*.

Default:  $NY = \text{SIZE}(Y,1)$ .

## FORTRAN 90 Interface

Generic:     CALL MUCBV (A, NLCA, NUCA, X, Y [, ...])

Specific:    The specific interface names are S\_MUCBV and D\_MUCBV.

## FORTRAN 77 Interface

Single:     CALL MUCBV (N, A, LDA, NLCA, NUCA, NX, X, IPATH, NY, Y)

Double:     The double precision name is DMUCBV.

## Description

If *IPATH* = 1, MUCBV computes  $y = Ax$ , where *A* is a complex band matrix and *x* and *y* are complex vectors. If *IPATH* = 2, MUCBV computes  $y = A^T x$ .

## Example

Multiply the transpose of a complex band matrix of order 4, with one upper codiagonal and two lower codiagonals stored in band mode, by a complex vector of length 3. The output vector will be a complex vector of length 3.

```
USE MUCBV_INT
USE WRNCRN_INT

IMPLICIT NONE
```

```

! Declare variables
INTEGER LDA, N, NLCA, NUCA, NX, NY
PARAMETER (LDA=4, N=4, NLCA=2, NUCA=1, NX=4, NY=4)
!
INTEGER IPATH
COMPLEX A(LDA,N), X(NX), Y(NY)
!
! Set values for A (in band mode)
! A = (0.0+ 0.0i 1.0+ 2.0i 3.0+ 4.0i 5.0+ 6.0i)
! (-1.0- 1.0i -1.0- 1.0i -1.0- 1.0i -1.0- 1.0i)
! (-1.0+ 2.0i -1.0+ 3.0i -2.0+ 1.0i 0.0+ 0.0i)
! (2.0+ 0.0i 0.0+ 2.0i 0.0+ 0.0i 0.0+ 0.0i)
!
! Set values for X
! X = (3.0 + 4.0i 0.0 + 0.0i 1.0 + 2.0i -2.0 - 1.0i)
!
DATA A/(0.0,0.0), (-1.0,-1.0), (-1.0,2.0), (2.0,0.0), (1.0,2.0), &
 (-1.0,-1.0), (-1.0,3.0), (0.0,2.0), (3.0,4.0), (-1.0,-1.0), &
 (-2.0,1.0), (0.0,0.0), (5.0,6.0), (-1.0,-1.0), (0.0,0.0), &
 (0.0,0.0)/
DATA X/(3.0,4.0), (0.0,0.0), (1.0,2.0), (-2.0,-1.0)/
! Compute y = Ax
IPATH = 2
CALL MUCBV (A, NLCA, NUCA, X, Y, IPATH=IPATH)
! Print results
CALL WRCRN ('y = Ax', Y, 1, NY, 1)
END

```

## Output

```

 y = Ax
 1 2 3 4
(3.00, -3.00) (-10.00, 7.00) (6.00, -3.00) (-6.00, 19.00)

```

---

## ARBRB

Adds two band matrices, both in band storage mode.

### Required Arguments

**A** —  $N$  by  $N$  band matrix with *NLCA* lower codiagonals and *NUCA* upper codiagonals stored in band mode with dimension  $(NLCA + NUCA + 1)$  by  $N$ . (Input)

**NLCA** — Number of lower codiagonals of **A**. (Input)

**NUCA** — Number of upper codiagonals of **A**. (Input)

**B** —  $N$  by  $N$  band matrix with *NLCB* lower codiagonals and *NUCB* upper codiagonals stored in band mode with dimension  $(NLCB + NUCB + 1)$  by  $N$ . (Input)

**NLCB** — Number of lower codiagonals of **B**. (Input)

**NUCB** — Number of upper codiagonals of **B**. (Input)

*C* — *N* by *N* band matrix with *NLCC* lower codiagonals and *NUCC* upper codiagonals containing the sum *A* + *B* in band mode with dimension (*NLCC* + *NUCC* + 1) by *N*.  
(Output)

*NLCC* — Number of lower codiagonals of *C*. (Input)  
*NLCC* must be at least as large as  $\max(\text{NLCA}, \text{NLCA})$ .

*NUCC* — Number of upper codiagonals of *C*. (Input)  
*NUCC* must be at least as large as  $\max(\text{NUCA}, \text{NUCA})$ .

### Optional Arguments

*N* — Order of the matrices *A*, *B* and *C*. (Input)  
Default: *N* = `SIZE (A,2)`.

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = `SIZE (A,1)`.

*LDB* — Leading dimension of *B* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDB* = `SIZE (B,1)`.

*LDC* — Leading dimension of *C* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDC* = `SIZE (C,1)`.

### FORTRAN 90 Interface

Generic:     `CALL ARBRB (A, NLCA, NUCA, B, NLCA, NUCA, C, NLCC, NUCC [, ...])`

Specific:    The specific interface names are `S_ARBRB` and `D_ARBRB`.

### FORTRAN 77 Interface

Single:     `CALL ARBRB (N, A, LDA, NLCA, NUCA, B, LDB, NLCA, NUCA, C, LDC, NLCC, NUCC)`

Double:     The double precision name is `DARBRB`.

### Description

The routine `ARBRB` adds two real matrices stored in band mode, returning a real matrix stored in band mode.

## Example

Add two real matrices of order 4 stored in band mode. Matrix *A* has one upper codiagonal and one lower codiagonal. Matrix *B* has no upper codiagonals and two lower codiagonals. The output matrix *C*, has one upper codiagonal and two lower codiagonals.

```
USE ARBRB_INT
USE WRRRN_INT

IMPLICIT NONE

!
! Declare variables
INTEGER LDA, LDB, LDC, N, NLCA, NLCB, NLCC, NUCA, NUCB, NUCC
PARAMETER (LDA=3, LDB=3, LDC=4, N=4, NLCA=1, NLCB=2, NLCC=2, &
 NUCA=1, NUCB=0, NUCC=1)
!
REAL A(LDA,N), B(LDB,N), C(LDC,N)
!
! Set values for A (in band mode)
! A = (0.0 2.0 3.0 -1.0)
! (1.0 1.0 1.0 1.0)
! (0.0 3.0 4.0 0.0)
!
! Set values for B (in band mode)
! B = (3.0 3.0 3.0 3.0)
! (1.0 -2.0 1.0 0.0)
! (-1.0 2.0 0.0 0.0)
!
DATA A/0.0, 1.0, 0.0, 2.0, 1.0, 3.0, 3.0, 1.0, 4.0, -1.0, 1.0, &
 0.0/
DATA B/3.0, 1.0, -1.0, 3.0, -2.0, 2.0, 3.0, 1.0, 0.0, 3.0, 0.0, &
 0.0/
!
! Add A and B to obtain C (in band
! mode)
CALL ARBRB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC)
!
! Print results
CALL WRRRN ('C = A+B', C)
END
```

## Output

```
 C = A+B
 1 2 3 4
1 0.000 2.000 3.000 -1.000
2 4.000 4.000 4.000 4.000
3 1.000 1.000 5.000 0.000
4 -1.000 2.000 0.000 0.000
```

---

## ACBCB

Adds two complex band matrices, both in band storage mode.

### Required Arguments

*A* — *N* by *N* complex band matrix with *NLCA* lower codiagonals and *NUCA* upper codiagonals stored in band mode with dimension (*NLCA* + *NUCA* + 1) by *N*. (Input)

**NLCA** — Number of lower codiagonals of A. (Input)

**NUCA** — Number of upper codiagonals of A. (Input)

**B** —  $N$  by  $N$  complex band matrix with **NLCB** lower codiagonals and **NUCB** upper codiagonals stored in band mode with dimension  $(\text{NLCB} + \text{NUCB} + 1)$  by  $N$ . (Input)

**NLCB** — Number of lower codiagonals of B. (Input)

**NUCB** — Number of upper codiagonals of B. (Input)

**C** —  $N$  by  $N$  complex band matrix with **NLCC** lower codiagonals and **NUCC** upper codiagonals containing the sum  $A + B$  in band mode with dimension  $(\text{NLCC} + \text{NUCC} + 1)$  by  $N$ . (Output)

**NLCC** — Number of lower codiagonals of C. (Input)  
NLCC must be at least as large as  $\max(\text{NLCA}, \text{NLCB})$ .

**NUCC** — Number of upper codiagonals of C. (Input)  
NUCC must be at least as large as  $\max(\text{NUCA}, \text{NUCB})$ .

### Optional Arguments

**N** — Order of the matrices A, B and C. (Input)  
Default:  $N = \text{SIZE}(A, 2)$ .

**LDA** — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDA} = \text{SIZE}(A, 1)$ .

**LDB** — Leading dimension of B exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDB} = \text{SIZE}(B, 1)$ .

**LDC** — Leading dimension of C exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $\text{LDC} = \text{SIZE}(C, 1)$ .

### FORTRAN 90 Interface

Generic:     CALL ACBCB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC [, ...])

Specific:    The specific interface names are S\_ACBCB and D\_ACBCB.

## FORTRAN 77 Interface

Single:      CALL ACBCB (N, A, LDA, NLCA, NUCA, B, LDB, NLCB, NUCB, C, LDC, NLCC,  
                  NUCC)

Double:      The double precision name is DACBCB.

## Description

The routine ACBCB adds two complex matrices stored in band mode, returning a complex matrix stored in band mode.

## Example

Add two complex matrices of order 4 stored in band mode. Matrix *A* has two upper codiagonals and no lower codiagonals. Matrix *B* has no upper codiagonals and two lower codiagonals. The output matrix *C* has two upper codiagonals and two lower codiagonals.

```
USE ACBCB_INT
USE WRCRN_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, LDB, LDC, N, NLCA, NLCB, NLCC, NUCA, NUCB, NUCC
PARAMETER (LDA=3, LDB=3, LDC=5, N=3, NLCA=0, NLCB=2, NLCC=2, &
 NUCA=2, NUCB=0, NUCC=2)
!
COMPLEX A(LDA,N), B(LDB,N), C(LDC,N)
!
! Set values for A (in band mode)
!
! A = (0.0 + 0.0i 0.0 + 0.0i 3.0 - 2.0i)
! (0.0 + 0.0i -1.0+ 3.0i 6.0 + 0.0i)
! (1.0 + 4.0i 5.0 - 2.0i 3.0 + 1.0i)
!
!
! Set values for B (in band mode)
!
! B = (3.0 + 1.0i 4.0 + 1.0i 7.0 - 1.0i)
! (-1.0- 4.0i 9.0 + 3.0i 0.0 + 0.0i)
! (2.0 - 1.0i 0.0 + 0.0i 0.0 + 0.0i)
!
!
DATA A/(0.0,0.0), (0.0,0.0), (1.0,4.0), (0.0,0.0), (-1.0,3.0), &
 (5.0,-2.0), (3.0,-2.0), (6.0,0.0), (3.0,1.0)/
DATA B/(3.0,1.0), (-1.0,-4.0), (2.0,-1.0), (4.0,1.0), (9.0,3.0), &
 (0.0,0.0), (7.0,-1.0), (0.0,0.0), (0.0,0.0)/
!
! Compute C = A+B
CALL ACBCB (A, NLCA, NUCA, B, NLCB, NUCB, C, NLCC, NUCC)
!
! Print results
CALL WRCRN ('C = A+B', C)
END
```

## Output

```

 C = A+B
 1 2 3
1 (0.00, 0.00) (0.00, 0.00) (3.00, -2.00)
2 (0.00, 0.00) (-1.00, 3.00) (6.00, 0.00)
3 (4.00, 5.00) (9.00, -1.00) (10.00, 0.00)
```

```
4 (-1.00, -4.00) (9.00, 3.00) (0.00, 0.00)
5 (2.00, -1.00) (0.00, 0.00) (0.00, 0.00)
```

---

## NRIRR

Computes the infinity norm of a real matrix.

### Required Arguments

*A* — Real *NRA* by *NCA* matrix whose infinity norm is to be computed. (Input)

*ANORM* — Real scalar containing the infinity norm of *A*. (Output)

### Optional Arguments

*NRA* — Number of rows of *A*. (Input)

Default: *NRA* = *SIZE* (*A*,1).

*NCA* — Number of columns of *A*. (Input)

Default: *NCA* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

### FORTRAN 90 Interface

Generic:     CALL NRIRR (*A*, *ANORM* [, ...])

Specific:    The specific interface names are *S\_NRIRR* and *D\_NRIRR*.

### FORTRAN 77 Interface

Single:     CALL NRIRR (*NRA*, *NCA*, *A*, *LDA*, *ANORM*)

Double:     The double precision name is *DNRIRR*.

### Description

The routine *NRIRR* computes the infinity norm of a real rectangular matrix *A*. If *m* = *NRA* and *n* = *NCA*, then the  $\infty$ -norm of *A* is

$$\|A\|_{\infty} = \max_{1 \leq i \leq m} \sum_{j=1}^n |A_{ij}|$$

This is the maximum of the sums of the absolute values of the row elements.

## Example

Compute the infinity norm of a  $3 \times 4$  real rectangular matrix.

```
USE NRIRR_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NRA
PARAMETER (NCA=4, NRA=3)
!
INTEGER NOUT
REAL A(NRA,NCA), ANORM
!
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
! Compute the infinity norm of A
CALL NRIRR (A, ANORM)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The infinity norm of A is ', ANORM
END
```

## Output

The infinity norm of A is 8.00000

---

# NR1RR

Computes the 1-norm of a real matrix.

## Required Arguments

*A* — Real *NRA* by *NCA* matrix whose 1-norm is to be computed. (Input)

*ANORM* — Real scalar containing the 1-norm of *A*. (Output)

## Optional Arguments

*NRA* — Number of rows of *A*. (Input)  
Default: *NRA* = *SIZE* (*A*,1).

*NCA* — Number of columns of *A*. (Input)  
Default: *NCA* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
 Default: *LDA* = *SIZE* (*A*,1).

### **FORTRAN 90 Interface**

Generic:    CALL NR1RR (A, ANORM [, ...])

Specific:   The specific interface names are S\_NR1RR and D\_NR1RR.

### **FORTRAN 77 Interface**

Single:     CALL NR1RR (NRA, NCA, A, LDA, ANORM)

Double:     The double precision name is DNR1RR.

### **Description**

The routine NR1RR computes the 1-norm of a real rectangular matrix *A*. If *m* = *NRA* and *n* = *NCA*, then the 1-norm of *A* is

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |A_{ij}|$$

This is the maximum of the sums of the absolute values of the column elements.

### **Example**

Compute the 1-norm of a 3 × 4 real rectangular matrix.

```

USE NR1RR_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER NCA, NRA
PARAMETER (NCA=4, NRA=3)
!
INTEGER NOUT
REAL A(NRA,NCA), ANORM
!
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
! Compute the L1 norm of A
CALL NR1RR (A, ANORM)
! Print results

```

```
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The 1-norm of A is ', ANORM
END
```

## Output

The 1-norm of A is        6.00000

---

# NR2RR

Computes the Frobenius norm of a real rectangular matrix.

## Required Arguments

*A* — Real *NRA* by *NCA* rectangular matrix. (Input)

*ANORM* — Frobenius norm of *A*. (Output)

## Optional Arguments

*NRA* — Number of rows of *A*. (Input)

Default: *NRA* = *SIZE* (*A*,1).

*NCA* — Number of columns of *A*. (Input)

Default: *NCA* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)

Default: *LDA* = *SIZE* (*A*,1).

## FORTRAN 90 Interface

Generic:    `CALL NR2RR (A, ANORM [, ...])`

Specific:    The specific interface names are `S_NR2RR` and `D_NR2RR`.

## FORTRAN 77 Interface

Single:    `CALL NR2RR (NRA, NCA, A, LDA, ANORM)`

Double:    The double precision name is `DNR2RR`.

## Description

The routine `NR2RR` computes the Frobenius norm of a real rectangular matrix *A*. If *m* = *NRA* and *n* = *NCA*, then the Frobenius norm of *A* is

$$\|A\|_2 = \left[ \sum_{i=1}^m \sum_{j=1}^n A_{ij}^2 \right]^{1/2}$$

### Example

Compute the Frobenius norm of a  $3 \times 4$  real rectangular matrix.

```

USE NR2RR_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER LDA, NCA, NRA
PARAMETER (LDA=3, NCA=4, NRA=3)
!
INTEGER NOUT
REAL A(LDA,NCA), ANORM
!
! Set values for A
! A = (1.0 0.0 2.0 0.0)
! (3.0 4.0 -1.0 0.0)
! (2.0 1.0 2.0 1.0)
!
DATA A/1.0, 3.0, 2.0, 0.0, 4.0, 1.0, 2.0, -1.0, 2.0, 0.0, 0.0, &
1.0/
!
! Compute Frobenius norm of A
CALL NR2RR (A, ANORM)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The Frobenius norm of A is ', ANORM
END

```

### Output

The Frobenius norm of A is        6.40312

---

## NR1RB

Computes the 1-norm of a real band matrix in band storage mode.

### Required Arguments

*A* — Real (*NUCA* + *NLCA* + 1) by *N* array containing the *N* by *N* band matrix in band storage mode. (Input)

*NLCA* — Number of lower codiagonals of *A*. (Input)

*NUCA* — Number of upper codiagonals of *A*. (Input)

*ANORM* — Real scalar containing the 1-norm of *A*. (Output)

## Optional Arguments

*N* — Order of the matrix. (Input)  
Default: *N* = SIZE (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = SIZE (*A*,1).

## FORTRAN 90 Interface

Generic: CALL NR1RB (*A*, *NLCA*, *NUCA*, *ANORM* [, ...])

Specific: The specific interface names are *S\_NR1RB* and *D\_NR1RB*.

## FORTRAN 77 Interface

Single: CALL NR1RB (*N*, *A*, *LDA*, *NLCA*, *NUCA*, *ANORM*)

Double: The double precision name is *DNR1RB*.

## Description

The routine *NR1RB* computes the 1-norm of a real band matrix *A*. The 1-norm of a matrix *A* is

$$\|A\|_1 = \max_{1 \leq j \leq N} \sum_{i=1}^N |A_{ij}|$$

This is the maximum of the sums of the absolute values of the column elements.

## Example

Compute the 1-norm of a  $4 \times 4$  real band matrix stored in band mode.

```
USE NR1RB_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER LDA, N, NLCA, NUCA
PARAMETER (LDA=4, N=4, NLCA=2, NUCA=1)
!
INTEGER NOUT
REAL A(LDA,N), ANORM
!
! Set values for A (in band mode)
! A = (0.0 2.0 2.0 3.0)
! (-2.0 -3.0 -4.0 -1.0)
! (2.0 1.0 0.0 0.0)
```

```

! (0.0 1.0 0.0 0.0)
!
DATA A/0.0, -2.0, 2.0, 0.0, 2.0, -3.0, 1.0, 1.0, 2.0, -4.0, 0.0, &
 0.0, 3.0, -1.0, 2*0.0/
! Compute the L1 norm of A
CALL NR1RB (A, NLCA, NUCA, ANORM)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The 1-norm of A is ', ANORM
END

```

## Output

The 1-norm of A is        7.00000

---

## NR1CB

Computes the 1-norm of a complex band matrix in band storage mode.

### Required Arguments

*A* — Complex ( $NUCA + NLCA + 1$ ) by  $N$  array containing the  $N$  by  $N$  band matrix in band storage mode. (Input)

*NLCA* — Number of lower codiagonals of  $A$ . (Input)

*NUCA* — Number of upper codiagonals of  $A$ . (Input)

*ANORM* — Real scalar containing the 1-norm of  $A$ . (Output)

### Optional Arguments

*N* — Order of the matrix. (Input)  
Default:  $N = \text{SIZE}(A,2)$ .

*LDA* — Leading dimension of  $A$  exactly as specified in the dimension statement of the calling program. (Input)  
Default:  $LDA = \text{SIZE}(A,1)$ .

### FORTRAN 90 Interface

Generic:     CALL NR1CB (A, NLCA, NUCA, ANORM [, ...])

Specific:    The specific interface names are S\_NR1CB and D\_NR1CB.

### FORTRAN 77 Interface

Single:      CALL NR1CB (N, A, LDA, NLCA, NUCA, ANORM)

Double: The double precision name is DNR1CB.

## Description

The routine NR1CB computes the 1-norm of a complex band matrix  $A$ . The 1-norm of a complex matrix  $A$  is

$$\|A\|_1 = \max_{1 \leq j \leq N} \sum_{i=1}^N [|\Re A_{ij}| + |\Im A_{ij}|]$$

## Example

Compute the 1-norm of a complex matrix of order 4 in band storage mode.

```
USE NR1CB_INT
USE UMACH_INT

IMPLICIT NONE

! Declare variables
INTEGER LDA, N, NLCA, NUCA
PARAMETER (LDA=4, N=4, NLCA=2, NUCA=1)

!
INTEGER NOUT
REAL ANORM
COMPLEX A(LDA,N)

!
! Set values for A (in band mode)
! A = (0.0+0.0i 2.0+3.0i -1.0+1.0i -2.0-1.0i)
! (-2.0+3.0i 1.0+0.0i -4.0-1.0i 0.0-4.0i)
! (2.0+2.0i 4.0+6.0i 3.0+2.0i 0.0+0.0i)
! (0.0-1.0i 2.0+1.0i 0.0+0.0i 0.0+0.0i)
!
DATA A/(0.0,0.0), (-2.0,3.0), (2.0,2.0), (0.0,-1.0), (2.0,3.0), &
 (1.0,0.0), (4.0,6.0), (2.0,1.0), (-1.0,1.0), (-4.0,-1.0), &
 (3.0,2.0), (0.0,0.0), (-2.0,-1.0), (0.0,-4.0), (0.0,0.0), &
 (0.0,0.0)/

! Compute the L1 norm of A
CALL NR1CB (A, NLCA, NUCA, ANORM)

! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The 1-norm of A is ', ANORM
END
```

## Output

The 1-norm of A is 19.0000

---

## DISL2

This function computes the Euclidean (2-norm) distance between two points.

## Function Return Value

*DISL2* — Euclidean (2-norm) distance between the points  $x$  and  $y$ . (Output)

## Required Arguments

$X$  — Vector of length  $\max(N * |INCX|, 1)$ . (Input)

$Y$  — Vector of length  $\max(N * |INCY|, 1)$ . (Input)

## Optional Arguments

$N$  — Length of the vectors  $x$  and  $y$ . (Input)  
Default:  $N = \text{SIZE}(X,1)$ .

*INCX* — Displacement between elements of  $x$ . (Input)  
The  $I$ -th element of  $x$  is  $x(1 + (I - 1) * INCX)$  if *INCX* is greater than or equal to zero or  $x(1 + (I - N) * INCX)$  if *INCX* is less than zero.  
Default: *INCX* = 1.

*INCY* — Displacement between elements of  $y$ . (Input)  
The  $I$ -th element of  $y$  is  $y(1 + (I - 1) * INCY)$  if *INCY* is greater than or equal to zero or  $y(1 + (I - N) * INCY)$  if *INCY* is less than zero.  
Default: *INCY* = 1.

## FORTRAN 90 Interface

Generic: `DISL2 (X, Y [, ...])`

Specific: The specific interface names are `S_DISL2` and `D_DISL2`.

## FORTRAN 77 Interface

Single: `DISL2 (N, X, INCX, Y, INCY)`

Double: The double precision function name is `DDISL2`.

## Description

The function `DISL2` computes the Euclidean (2-norm) distance between two points  $x$  and  $y$ . The Euclidean distance is defined to be

$$\left[ \sum_{i=1}^N (x_i - y_i)^2 \right]^{1/2}$$

## Example

Compute the Euclidean (2-norm) distance between two vectors of length 4.

```
 USE DISL2_INT
 USE UMACH_INT

 IMPLICIT NONE
! Declare variables
 INTEGER INCX, INCY, N
 PARAMETER (N=4)
!
 INTEGER NOUT
 REAL VAL, X(N), Y(N)
!
! Set values for X and Y
! X = (1.0 -1.0 0.0 2.0)
!
! Y = (4.0 2.0 1.0 -3.0)
!
 DATA X/1.0, -1.0, 0.0, 2.0/
 DATA Y/4.0, 2.0, 1.0, -3.0/
! Compute L2 distance
 VAL = DISL2(X,Y)
! Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,*) ' The 2-norm distance is ', VAL
 END
```

## Output

```
The 2-norm distance is 6.63325
```

---

# DISL1

This function computes the 1-norm distance between two points.

## Function Return Value

*DISL1* — 1-norm distance between the points *X* and *Y*. (Output)

## Required Arguments

*X* — Vector of length  $\max(N * |INCX|, 1)$ . (Input)

*Y* — Vector of length  $\max(N * |INCY|, 1)$ . (Input)

## Optional Arguments

*N* — Length of the vectors *X* and *Y*. (Input)  
Default:  $N = \text{SIZE}(X,1)$ .

**INCX** — Displacement between elements of  $X$ . (Input)

The  $I$ -th element of  $X$  is  $X(1 + (I - 1) * INCX)$  if  $INCX$  is greater than or equal to zero or  $X(1 + (I - N) * INCX)$  if  $INCX$  is less than zero.

Default:  $INCX = 1$ .

**INCY** — Displacement between elements of  $Y$ . (Input)

The  $I$ -th element of  $Y$  is  $Y(1 + (I - 1) * INCY)$  if  $INCY$  is greater than or equal to zero or  $Y(1 + (I - N) * INCY)$  if  $INCY$  is less than zero.

Default:  $INCY = 1$ .

## FORTRAN 90 Interface

Generic: `DISL1 (X, Y [, ...])`

Specific: The specific interface names are `S_DISL1` and `D_DISL1`.

## FORTRAN 77 Interface

Single: `DISL1 (N, X, INCX, Y, INCY)`

Double: The double precision function name is `DDISL1`.

## Description

The function `DISL1` computes the 1-norm distance between two points  $x$  and  $y$ . The 1-norm distance is defined to be

$$\sum_{i=1}^N |x_i - y_i|$$

## Example

Compute the 1-norm distance between two vectors of length 4.

```
USE DISL1_INT
USE UMACH_INT

IMPLICIT NONE
!
! DECLARE VARIABLES
INTEGER INCX, INCY, N
PARAMETER (N=4)
!
INTEGER NOUT
REAL VAL, X(N), Y(N)
!
! SET VALUES FOR X AND Y
! X = (1.0 -1.0 0.0 2.0)
!
! Y = (4.0 2.0 1.0 -3.0)
```

```

!
 DATA X/1.0, -1.0, 0.0, 2.0/
 DATA Y/4.0, 2.0, 1.0, -3.0/
!
 VAL = DISL1(X,Y) Compute L1 distance
!
 Print results
 CALL UMACH (2, NOUT)
 WRITE (NOUT,*) ' The 1-norm distance is ', VAL
 END

```

## Output

```
The 1-norm distance is 12.0000
```

---

## DISLI

This function computes the infinity norm distance between two points.

### Function Return Value

*DISLI* — Infinity norm distance between the points *X* and *Y*. (Output)

### Required Arguments

*X* — Vector of length  $\max(N * |INCX|, 1)$ . (Input)

*Y* — Vector of length  $\max(N * |INCY|, 1)$ . (Input)

### Optional Arguments

*N* — Length of the vectors *X* and *Y*. (Input)

Default:  $N = \text{SIZE}(X,1)$ .

*INCX* — Displacement between elements of *X*. (Input)

The *I*-th element of *X* is  $X(1 + (I - 1) * INCX)$  if *INCX* is greater than or equal to zero or  $X(1 + (I - N) * INCX)$  if *INCX* is less than zero.

Default:  $INCX = 1$ .

*INCY* — Displacement between elements of *Y*. (Input)

The *I*-th element of *Y* is  $Y(1 + (I - 1) * INCY)$  if *INCY* is greater than or equal to zero or  $Y(1 + (I - N) * INCY)$  if *INCY* is less than zero.

Default:  $INCY = 1$ .

### FORTRAN 90 Interface

Generic: `DISLI (X, Y [, ...])`

Specific: The specific interface names are `S_DISLI` and `D_DISLI`.

## FORTRAN 77 Interface

Single: DISLI (N, X, INCX, Y, INCY)

Double: The double precision function name is DDISLI.

## Description

The function DISLI computes the  $\infty$ -norm distance between two points  $x$  and  $y$ . The  $\infty$ -norm distance is defined to be

$$\max_{1 \leq i \leq N} |x_i - y_i|$$

## Example

Compute the  $\infty$ -norm distance between two vectors of length 4.

```
USE DISLI_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER INCX, INCY, N
PARAMETER (N=4)
!
INTEGER NOUT
REAL VAL, X(N), Y(N)
!
! Set values for X and Y
! X = (1.0 -1.0 0.0 2.0)
! Y = (4.0 2.0 1.0 -3.0)
!
DATA X/1.0, -1.0, 0.0, 2.0/
DATA Y/4.0, 2.0, 1.0, -3.0/
!
! Compute L-infinity distance
VAL = DISLI(X,Y)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,*) ' The infinity-norm distance is ', VAL
END
```

## Output

The infinity-norm distance is 5.00000

---

# VCONR



Computes the convolution of two real vectors.

## Required Arguments

*X* — Vector of length *NX*. (Input)

*Y* — Vector of length *NY*. (Input)

*Z* — Vector of length *NZ* containing the convolution  $z = x * y$ . (Output)

## Optional Arguments

*NX* — Length of the vector *x*. (Input)  
Default:  $NX = \text{SIZE}(X,1)$ .

*NY* — Length of the vector *y*. (Input)  
Default:  $NY = \text{SIZE}(Y,1)$ .

*NZ* — Length of the vector *z*. (Input)  
*NZ* must be at least  $NX + NY - 1$ .  
Default:  $NZ = \text{SIZE}(Z,1)$ .

## FORTRAN 90 Interface

Generic:    CALL VCONR (X, Y, Z [, ...])

Specific:   The specific interface names are S\_VCONR and D\_VCONR.

## FORTRAN 77 Interface

Single:     CALL VCONR (NX, X, NY, Y, NZ, Z)

Double:    The double precision name is DVCONR.

## Description

The routine VCONR computes the convolution *z* of two real vectors *x* and *y*. Let  $n_x = NX$ ,  $n_y = NY$  and  $n_z = NZ$ . The vector *z* is defined to be

$$z_j = \sum_{k=1}^{n_x} x_{j-k+1} y_k \quad \text{for } j = 1, 2, \dots, n_z$$

where  $n_z = n_x + n_y - 1$ . If the index  $j - k + 1$  is outside the range  $1, 2, \dots, n_x$ , then  $x_{j - k + 1}$  is taken to be zero.

The fast Fourier transform is used to compute the convolution. Define the complex vector  $u$  of length  $n_z = n_x + n_y - 1$  to be

$$u = (x_1, x_2, \dots, x_{n_x}, 0, \dots, 0)$$

The complex vector  $v$ , also of length  $n_z$ , is defined similarly using  $y$ . Then, by the Fourier convolution theorem,

$$\hat{w}_i = \hat{u}_i \hat{v}_i \quad \text{for } i = 1, 2, \dots, n_z$$

where the  $\hat{u}$  indicates the Fourier transform of  $u$  computed via IMSL routines [FFTCF](#) and [FFTCB](#) (see [Chapter 6, Transforms](#)) is used to compute the complex vector  $w$  from  $\hat{w}$ . The vector  $z$  is then found by taking the real part of the vector  $w$ .

## Comments

Workspace may be explicitly provided, if desired, by use of `V2ONR/DV2ONR`. The reference is

```
CALL V2ONR (NX, X, NY, Y, NZ, Z, XWK, YWK, ZWK, WK)
```

The additional arguments are as follows:

**XWK** — Complex work array of length  $NX + NY - 1$ .

**YWK** — Complex work array of length  $NX + NY - 1$ .

**ZWK** — Complex work array of length  $NX + NY - 1$ .

**WK** — Real work array of length  $6 * (NX + NY - 1) + 15$ .

## Example

In this example, the convolution of a vector  $x$  of length 8 and a vector  $y$  of length 3 is computed. The resulting vector  $z$  is of length  $8 + 3 - 1 = 10$ . (The vector  $y$  is sometimes called a *filter*.)

```
USE VCONR_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER NX, NY, NZ
PARAMETER (NX=8, NY=3, NZ=NX+NY-1)
!
REAL X(NX), Y(NY), Z(NZ)
! Set values for X
! X = (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0)
! Set values for Y
! Y = (0.0 0.0 1.0)
```

```

!
 DATA X/1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0/
 DATA Y/0.0, 0.0, 1.0/
!
! Compute vector convolution
! Z = X * Y
 CALL VCONR (X,Y,Z)
!
! Print results
 CALL WRRRN ('Z = X (*) Y', Z, 1, NZ, 1)
END

```

## Output

```

 Z = X (*) Y
 1 2 3 4 5 6 7 8 9 10
0.000 0.000 1.000 2.000 3.000 4.000 5.000 6.000 7.000 8.000

```

---

## VCONC



Computes the convolution of two complex vectors.

### Required Arguments

*X* — Complex vector of length *NX*. (Input)

*Y* — Complex vector of length *NY*. (Input)

*Z* — Complex vector of length *NZ* containing the convolution  $Z = X * Y$ . (Output)

### Optional Arguments

*NX* — Length of the vector *X*. (Input)  
Default:  $NX = \text{SIZE}(X,1)$ .

*NY* — Length of the vector *Y*. (Input)  
Default:  $NY = \text{SIZE}(Y,1)$ .

*NZ* — Length of the vector *Z*. (Input)  
*NZ* must be at least  $NX + NY - 1$ .  
Default:  $NZ = \text{SIZE}(Z,1)$ .

### FORTRAN 90 Interface

Generic:    CALL VCONC (X, Y, Z [, ...])

Specific:   The specific interface names are *S\_VCONC* and *D\_VCONC*.

## FORTRAN 77 Interface

Single:      CALL VCONC (NX, X, NY, Y, NZ, Z)

Double:     The double precision name is DVCONC.

## Description

The routine VCONC computes the convolution  $z$  of two complex vectors  $x$  and  $y$ . Let  $n_x = NX$ , then  $n_y = NY$  and  $n_z = NZ$ . The vector  $z$  is defined to be

$$z_j = \sum_{k=1}^{n_x} x_{j-k+1} y_k \quad \text{for } j = 1, 2, \dots, n_z$$

where  $n_z = n_x + n_y - 1$ . If the index  $j - k + 1$  is outside the range  $1, 2, \dots, n_x$ , then  $x_{j-k+1}$  is taken to be zero.

The fast Fourier transform is used to compute the convolution. Define the complex vector  $u$  of length  $n_z = n_x + n_y - 1$  to be

$$u = (x_1, x_2, \dots, x_{n_z}, 0, \dots, 0)$$

The complex vector  $v$ , also of length  $n_z$ , is defined similarly using  $y$ . Then, by the Fourier convolution theorem,

$$\hat{z}_i = \hat{u}_i \hat{v}_i \quad \text{for } i = 1, 2, \dots, n_z$$

where the  $\hat{u}$  indicates the Fourier transform of  $u$  computed using IMSL routine FFTCF (see [Chapter 6, Transforms](#)). The complex vector  $z$  is computed from  $\hat{w}$  via IMSL routine FFTCB (see [Chapter 6, Transforms](#)).

## Comments

Workspace may be explicitly provided, if desired, by use of V2ONC/DV2ONC. The reference is

```
CALL V2ONC (NX, X, NY, Y, NZ, Z, XWK, YWK, WK)
```

The additional arguments are as follows:

**XWK** — Complex work array of length  $NX + NY - 1$ .

**YWK** — Complex work array of length  $NX + NY - 1$ .

**WK** — Real work array of length  $6 * (NX + NY - 1) + 15$ .

## Example

In this example, the convolution of a vector  $x$  of length 4 and a vector  $y$  of length 3 is computed. The resulting vector  $z$  is of length  $4 + 3 - 1$  is sometimes called a *filter*.)

```
USE VCONC_INT
USE WRRCRN_INT

IMPLICIT NONE
INTEGER NX, NY, NZ
PARAMETER (NX=4, NY=3, NZ=NX+NY-1)
!
COMPLEX X(NX), Y(NY), Z(NZ)
! Set values for X
! X = (1.0+2.0i 3.0+4.0i 5.0+6.0i 7.0+8.0i)
! Set values for Y
! Y = (0.0+0i 0.0+0i 1.0+0i)
!
DATA X/(1.0,2.0), (3.0,4.0), (5.0,6.0), (7.0,8.0)/
DATA Y/(0.0,0.0), (0.0,0.0), (1.0,1.0)/
! Compute vector convolution
! Z = X * Y
CALL VCONC (X,Y,Z)
!
! Print results
CALL WRRCRN ('Z = X (*) Y', Z, 1, NZ, 1)
END
```

## Output

```

 Z = X (*) Y
 1 2 3 4
(0.00, 0.00) (0.00, 0.00) (-1.00, 3.00) (-1.00, 7.00)
 5 6
(-1.00, 11.00) (-1.00, 15.00)
```

---

## Extended Precision Arithmetic

This section describes a set of routines for mixed precision arithmetic. The routines are designed to allow the computation and use of the full quadruple precision result from the multiplication of two double precision numbers. An array called the accumulator stores the result of this multiplication. The result of the multiplication is added to the current contents of the accumulator. It is also possible to add a double precision number to the accumulator or to store a double precision approximation in the accumulator.

The mixed double precision arithmetic routines are described below. The accumulator array, `QACC`, is a double precision array of length 2. Double precision variables are denoted by `DA` and `DB`. Available operations are:

Initialize a real accumulator,  $QACC \leftarrow DA$ .

```
CALL DQINI (DA, QACC)
```

Store a real accumulator,  $DA \leftarrow QACC$ .

```
CALL DQSTO (QACC, DA)
```

Add to a real accumulator,  $QACC \leftarrow QACC + DA$ .

```
CALL DQADD (DA, QACC)
```

Add a product to a real accumulator,  $QACC \leftarrow QACC + DA*DB$ .

```
CALL DQMUL (DA, DB, QACC)
```

There are also mixed double complex arithmetic versions of the above routines. The accumulator, ZACC, is a double precision array of length 4. Double complex variables are denoted by ZA and ZB. Available operations are:

Initialize a complex accumulator,  $ZACC \leftarrow ZA$ .

```
CALL ZQINI (ZA, ZACC)
```

Store a complex accumulator,  $ZA \leftarrow ZACC$ .

```
CALL ZQSTO (ZACC, ZA)
```

Add to a complex accumulator,  $ZACC \leftarrow ZACC + ZA$ .

```
CALL ZQADD (ZA, ZACC)
```

Add a product to a complex accumulator,  $ZACC \leftarrow ZACC + ZA * ZB$ .

```
CALL ZQMUL (ZA, ZB, ZACC)
```

## Example

In this example, the value of  $1.0D0/3.0D0$  is computed in quadruple precision using Newton's method. Four iterations of

$$x_{k+1} = x_k + \left( x_k - ax_k^2 \right)$$

with  $a = 3$  are taken. The error  $ax - 1$  is then computed. The results are accurate to approximately twice the usual double precision accuracy, as given by the IMSL routine `DMACH(4)`, in the Reference Material section of this manual. Since `DMACH` is machine dependent, the actual accuracy obtained is also machine dependent.

```
USE IMSL_LIBRARIES

IMPLICIT NONE
INTEGER I, NOUT
DOUBLE PRECISION A, DACC(2), DMACH, ERROR, SACC(2), X(2), X1, X2, EPSQ
!
CALL UMACH (2, NOUT)
A = 3.0D0
CALL DQINI (1.0001D0/A, X)
!
! Compute X(K+1) = X(K) - A*X(K)*X(K)
! + X(K)
DO 10 I=1, 4
 X1 = X(1)
 X2 = X(2)
!
! Compute X + X
```

```

 CALL DQADD (X1, X)
 CALL DQADD (X2, X)
!
 CALL DQINI (0.0D0, DACC)
 CALL DQMUL (X1, X1, DACC)
 CALL DQMUL (X1, X2, DACC)
 CALL DQMUL (X1, X2, DACC)
 CALL DQMUL (X2, X2, DACC)
!
 CALL DQINI (0.0D0, SACC)
 CALL DQMUL (-A, DACC(1), SACC)
 CALL DQMUL (-A, DACC(2), SACC)
!
 CALL DQADD (SACC(1), X)
 CALL DQADD (SACC(2), X)
10 CONTINUE
!
 CALL DQINI (0.0D0, SACC)
 CALL DQMUL (A, X(1), SACC)
 CALL DQMUL (A, X(2), SACC)
 CALL DQADD (-1.0D0, SACC)
 CALL DQSTO (SACC, ERROR)
!
 ERROR should be less than MACHEPS**2
 EPSQ = AMACH(4)
 EPSQ = EPSQ * EPSQ
 WRITE (NOUT,99999) ERROR, ERROR/EPSQ
!
99999 FORMAT (' A*X - 1 = ', D15.7, ' = ', F10.5, '*MACHEPS**2')
 END

```

## Output

```

A*X - 1 = 0.6162976D-32 = 0.12500*MACHEPS**2

```



# Chapter 10: Linear Algebra Operators and Generic Functions

---

## Routines

|                                                                                                             |                           |      |
|-------------------------------------------------------------------------------------------------------------|---------------------------|------|
| <b>1.1. Operators</b>                                                                                       |                           |      |
| Computes matrix-matrix or matrix-vector product .....                                                       | <a href="#">.x.</a>       | 1707 |
| Computes transpose matrix-matrix product.....                                                               | <a href="#">.tx.</a>      | 1711 |
| Computes matrix- transpose matrix product.....                                                              | <a href="#">.xt.</a>      | 1714 |
| Computes conjugate transpose matrix-matrix product.....                                                     | <a href="#">.hx.</a>      | 1717 |
| Computes matrix-conjugate transpose matrix product.....                                                     | <a href="#">.xh.</a>      | 1720 |
| Computes the transpose of a matrix.....                                                                     | <a href="#">.t.</a>       | 1723 |
| Computes conjugate transpose of a matrix.....                                                               | <a href="#">.h.</a>       | 1726 |
| Computes the inverse matrix.....                                                                            | <a href="#">.i.</a>       | 1728 |
| Computes inverse matrix-matrix product.....                                                                 | <a href="#">.ix.</a>      | 1730 |
| Computes matrix-inverse matrix product.....                                                                 | <a href="#">.xi.</a>      | 1740 |
| <b>10.2 Functions</b>                                                                                       |                           |      |
| Computes the Cholesky factorization of a positive-definite,<br>symmetric or self-adjoint matrix .....       | <a href="#">CHOL</a>      | 1743 |
| Computes the condition number of a matrix.....                                                              | <a href="#">COND</a>      | 1746 |
| Computes the determinant of a rectangular matrix .....                                                      | <a href="#">DET</a>       | 1750 |
| Constructs a square diagonal matrix .....                                                                   | <a href="#">DIAG</a>      | 1753 |
| Extracts the diagonal terms of a matrix .....                                                               | <a href="#">DIAGONALS</a> | 1754 |
| Computes the eigenvalue-eigenvector decomposition of an<br>ordinary or generalized eigenvalue problem ..... | <a href="#">EIG</a>       | 1755 |
| Creates the identity matrix.....                                                                            | <a href="#">EYE</a>       | 1759 |
| Computes the Discrete Fourier Transform of one<br>complex sequence.....                                     | <a href="#">FFT</a>       | 1761 |
| Discrete Fourier Transform of<br>several complex or real sequences .....                                    | <a href="#">FFT_BOX</a>   | 1763 |
| Computes the inverse of the Discrete Fourier<br>Transform of one complex sequence .....                     | <a href="#">IFFT</a>      | 1765 |
| Computes the inverse Discrete Fourier Transform of<br>several complex or real sequences .....               | <a href="#">IFFT_BOX</a>  | 1767 |
| Tests for NaN.....                                                                                          | <a href="#">isNaN</a>     | 1769 |
| Returns the value for NaN .....                                                                             | <a href="#">NaN</a>       | 1770 |
| Computes the norm of an array.....                                                                          | <a href="#">NORM</a>      | 1771 |
| Orthogonalizes the columns of a matrix .....                                                                | <a href="#">ORTH</a>      | 1774 |

|                                                             |                      |      |
|-------------------------------------------------------------|----------------------|------|
| Generates random numbers .....                              | <a href="#">RAND</a> | 1777 |
| Computes the mathematical rank of a matrix.....             | <a href="#">RANK</a> | 1779 |
| Computes the singular value decomposition of a matrix ..... | <a href="#">SVD</a>  | 1780 |
| Normalizes the columns of a matrix. ....                    | <a href="#">UNIT</a> | 1783 |

---

## Usage Notes

This chapter describes numerical linear algebra, Fourier transforms, random number generation, and other utility software packaged as *defined operations* that are executed with a function notation similar to standard mathematics. The resulting interface alters the way libraries are presented to the user. Many computations of numerical linear algebra are documented here as operators and generic functions. A notation is developed reminiscent of matrix algebra. This allows the Fortran user to express mathematical formulas in terms of operators. The operators can be used with both dense and sparse matrices.

A comprehensive Fortran module, *linear\_operators*, defines the operators and functions. Its use provides this simplification. Subroutine calls and the use of type-dependent procedure names are largely avoided. This makes a rapid development cycle possible, at least for the purposes of experiments and proof-of-concept. The goal is to provide the Fortran programmer with an interface, operators, and functions that are useful and succinct. The modules can be used with or added to existing Fortran programs, but the operators provide a more readable program whenever they apply. This approach may require more hidden working storage. The size of the executable program may be larger than alternatives using subroutines. There are applications wherein the operator and function interface does not have the functionality that is available using subroutine libraries. To retain greater flexibility, some users will continue to require the techniques of calling subroutines.

A parallel computation for many of the defined operators and functions has been implemented. The type of problem solved is a simple one: several independent problems of the same data type and size. Most of the detailed communication for parallel computation is hidden from the user. Those functions having this data type computed in parallel are marked in **bold type**. The section “[Dense Matrix Parallelism Using MPI](#)” gives an introduction on how users should write their codes to use machines on a network.

A number of examples, in addition to those shown in this document, are supplied in the product examples directory. The name of the example code is shown in parentheses in the example heading, for those examples that are included with the product.

---

## Matrix Optional Data Changes

To reset tolerances for determining singularity and to allow for other data changes, non-allocated “hidden” variables are defined within the modules. These variables can be allocated first, then assigned values which result in the use of different tolerances or greater efficiency in the executable program. The non-allocated variables, whose scope is limited to the module, are hidden from the casual user. Default values or rules are applied if these arrays are not allocated. In more detail, the inverse matrix operator “.i.” applied to a square matrix first uses the *LU* factorization code `LIN_SOL_GEN` and row pivoting. The default value for a small diagonal term is defined to be:

$$\text{sqrt}(\text{epsilon}(\text{A}) * \text{sum}(\text{abs}(\text{A})) / (\text{n} * \text{n} + 1))$$

If the system is singular, a generalized matrix inverse is computed with the *QR* factorization code `LIN_SOL_LSQ` using this same tolerance. Both row and column pivoting are used. If the system is singular, an error message will be printed and a Fortran 90 `STOP` is executed. Users may want to change this rule. This is illustrated by continuing and not printing the error message. The following is an additional source to accomplish this, for all following invocations of the operator “.i.”:

```
allocate(s_inv_options(1))
s_inv_options(1) = skip_error_processing
B = .i. A
```

There are additional self-documenting integer parameters, packaged in the module *linear\_operators*, that allow users other choices, such as changing the value of the tolerance, as noted above. Included is the ability to have the option apply for just the next invocation of the operator. Options are available that allow optional data to be passed to supporting Fortran 90 subroutines. This is illustrated in the following example:

### Operator\_ex36.f90

```
use linear_operators

implicit none

! This is the equivalent of Example 4 for LIN_GEIG_GEN (using operators).

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0, zero=0d0
real(kind(1d0)) a(n,n), b(n,n), bta(n), err
complex(kind(1d0)) alpha(n), v(n,n)

! Generate random matrices for both A and B.
A = rand(A); B = rand(B)

! Set the option, a larger tolerance than default for lin_sol_lsq.
allocate(d_eig_options(6))
d_eig_options(1) = options_for_lin_geig_gen
d_eig_options(2) = 4
d_eig_options(3) = d_lin_geig_gen_for_lin_sol_lsq
d_eig_options(4) = 2
d_eig_options(5) = d_options(d_lin_sol_lsq_set_small, &
 sqrt(epsilon(one))*norm(B,1))
d_eig_options(6) = d_lin_sol_lsq_no_sing_mess

! Compute the generalized eigenvalues.
alpha = EIG(A, B=B, D=bta, W=V)

! Check the residuals.
err = norm((A .x. V .x. diag(bta)) - (B .x. V .x. diag(alpha)),1)/&
 (norm(A,1)*norm(bta,1)+norm(B,1)*norm(alpha,1))

if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_GEIG_GEN (operators) is correct.'
end if

! Clean up the allocated array. This is good housekeeping.
```

```

deallocate(d_eig_options)
end

```

Note that in this example one first allocates the array by which the user will pass the new options for EIG to use. This array is named `d_eig_options` in accordance with the name of the unallocated option array specified in the documentation for EIG. A size of 6 is specified because a total of six options must be passed to EIG to accomplish the resetting of the singular value tolerance and to turn off the printing of the error message when the matrix is singular. The first entry of `d_eig_options` specifies which of the options for EIG will be set. The next entry designates the number of entries which follows that apply to "options\_for\_lin\_geig\_gen". The third entry specifies the option value of LIN\_GEIG\_GEN to be set, `d_lin_geig_gen_for_lin_sol_lsq`. The fourth entry specifies the number of entries that follow which apply to LIN\_SOL\_LSQ. Finally, the fifth and sixth entries set the two LIN\_SOL\_LSQ options that we desire.

---

## Dense Matrix Computations




---

For a detailed description of MPI Capability see [“Dense Matrix Parallelism Using MPI.”](#)

---

This section is concerned with methods for computing with dense matrices. Consider a Fortran 90 code fragment that solves a linear system of algebraic equations,  $Ay = b$ , then computes the residual  $r = b - Ay$ . A standard mathematical notation is often used to write the solution,

$$y = A^{-1}b$$

A user thinks: “matrix and right-hand side yields solution.” The code shows the computation of this mathematical solution using a defined Fortran operator “.ix.”, and random data obtained with the function, *rand*. This operator is read “inverse matrix times.” The residuals are computed with another defined Fortran operator “.x.”, read “matrix times vector.” Once a user understands the equivalence of a mathematical formula with the corresponding Fortran operator, it is possible to write this program with little effort. The last line of the example before `end` is discussed below.

```

USE linear_operators

integer,parameter :: n=3; real A(n,n), y(n), b(n), r(n)
A=rand(A); b=rand(b); y = A .ix. b
r = b - (A .x. y) ! Parentheses are needed
end

```

The IMSL Fortran Numerical Library provides additional lower-level software that implements the operation “.ix.”, the function *rand*, matrix multiply “.x.”, and others not used in this example. Standard matrix products and inverse operations of matrix algebra are shown in the following table:

| Defined Array Operation                                  | Matrix Operation | Alternative in Fortran 90                                 |
|----------------------------------------------------------|------------------|-----------------------------------------------------------|
| A .x. B                                                  | $AB$             | matmul(A, B)                                              |
| .i. A                                                    | $A^{-1}$         | lin_sol_gen<br>lin_sol_lsq                                |
| .t. A, .h. A                                             | $A^T, A^H$       | transpose(A)<br>conjg(transpose(A))                       |
| A .ix. B                                                 | $A^{-1}B$        | lin_sol_gen<br>lin_sol_lsq                                |
| B .xi. A                                                 | $BA^{-1}$        | lin_sol_gen<br>lin_sol_lsq                                |
| A .tx. B, or (.t. A) .x. B<br>A .hx. B, or (.h. A) .x. B | $A^T B, A^H B$   | matmul(transpose(A), B)<br>matmul(conjg(transpose(A)), B) |
| B .xt. A, or B .x. (.t. A)<br>B .xh. A, or B .x. (.h. A) | $BA^T, BA^H$     | matmul(B, transpose(A))<br>matmul(B, conjg(transpose(A))) |

The IMSL operators apply generically to all standard precisions and floating-point data types – real and complex – and to objects that are broader in scope than arrays with a fixed number of dimensions. For example, the matrix product “.x.” applies to matrix times vector and matrix times matrix represented as Fortran 90 arrays. It also applies to “independent matrix products.” For this, use the notion: *a box of problems* to refer to independent linear algebra computations, of the same kind and dimension, but different data. The *racks* of the box are the distinct problems. In terms of Fortran 90 arrays, a rank-3, assumed-shape array is the data structure used for a box. The first two dimensions are the data for a matrix; the third dimension is the rack number. Each problem is independent of other problems in consecutive racks of the box. We use parallelism of an underlying network of processors, and MPI, when computing these disjoint problems.

In addition to the operators .ix., .xi., .i., and .x., additional operators .t., .h., .tx., .hx., .xt., and .xh. are provided for complex matrices. Since the transpose matrix is defined for complex matrices, this meaning is kept for the defined operations. In order to write one defined operation for both real and complex matrices, use the conjugate-transpose in all cases. This will result in only real operations when the data arrays are real.

For sums and differences of vectors and matrices, the intrinsic array operations “+” and “-” are available. It is not necessary to have separate defined operations. A parsing rule in Fortran 90 states that the result of a defined operation involving two quantities has a *lower precedence* than any intrinsic operation. This explains the parentheses around the next-to-last line containing the sub-expression “A .x. y” found in the example. Users are advised to always include parentheses around array expressions that are mixed with defined operations, or whenever there is possible confusion without them. The next-to-last line of the example results in computing the residual associated with the solution, namely  $r = b - Ay$ . Ideally, this residual is zero when the system has a unique solution. It will be computed as a non-zero vector due to rounding errors and conditioning of the problem.

# Dense Matrix Functions



For a detailed description of MPI Capability see “[Dense Matrix Parallelism Using MPI.](#)”

Several decompositions and functions required for numerical linear algebra follow. The convention of enclosing optional quantities in brackets, “[ ]” is used. The functions that use MPI for parallel execution of the box data type are marked in **bold**.

| Defined Array Functions                   | Matrix Operation                                                                                                                                                                                                        |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S=SVD(A [,U=U, V=V])                      | $A = USV^T$                                                                                                                                                                                                             |
| E=EIG(A [[,B=B, D=D],<br>V=V, W=W])       | $(AV = VE), AVD = BVE$<br>$(AW = WE), AWD = BWE$                                                                                                                                                                        |
| R=CHOL(A)                                 | $A = R^T R$                                                                                                                                                                                                             |
| Q=ORTH(A [,R=R])                          | $(A = QR), Q^T Q = I$                                                                                                                                                                                                   |
| U=UNIT(A)                                 | $[u_1, \dots] = [a_1 / \ a_1\ , \dots]$                                                                                                                                                                                 |
| F=DET(A)                                  | Det(A) = determinant                                                                                                                                                                                                    |
| K=RANK(A)                                 | rank(A) = rank                                                                                                                                                                                                          |
| P=NORM(A[,type= i])                       | $p = \ A\ _1 = \max_j \left( \sum_{i=1}^m  a_{ij}  \right)$<br>$p = \ A\ _2 = s_1 = \text{largest singular value}$<br>$p = \ A\ _{\infty \leftrightarrow \text{huge}(1)} = \max_i \left( \sum_{j=1}^n  a_{ij}  \right)$ |
| C=COND(A)                                 | $s_1 / s_{\text{rank}(A)}$                                                                                                                                                                                              |
| Z=EYE(N)                                  | $Z = I_N$                                                                                                                                                                                                               |
| A=DIAG(X)                                 | $A = \text{diag}(x_1, \dots)$                                                                                                                                                                                           |
| X=DIAGONALS(A)                            | $x = (a_{11}, \dots)$                                                                                                                                                                                                   |
| Y=FFT (X,[WORK=W]);<br>X=IFFT(Y,[WORK=W]) | Discrete Fourier Transform, Inverse                                                                                                                                                                                     |

| Defined Array Functions                           | Matrix Operation                              |
|---------------------------------------------------|-----------------------------------------------|
| Y=FFT_BOX (X,[WORK=W]);<br>X=IFFT_BOX(Y,[WORK=W]) | Discrete Fourier Transform for Boxes, Inverse |
| A=RAND(A)                                         | Random numbers, $0 < A < 1$                   |
| L=isNaN(A)                                        | Test for NaN, <i>if (l) then...</i>           |

In certain functions, the optional arguments are inputs while other optional arguments are outputs. To illustrate the example of the box **SVD** function, a code is given that computes the singular value decomposition and the reconstruction of the random matrix box,  $A$ , using the computed factors,  $R = USV^T$ . Mathematically  $R = A$ , but this will be true, only approximately, due to rounding errors. The value  $units\_of\_error = \|A - R\|/( \|A\| \epsilon)$ , shows the merit of this approximation.

---

## Dense Matrix Parallelism Using MPI



### General Remarks

The central theme we use for the computing functions of the box data type is that of delivering results to a distinguished node of the machine. One of the design goals was to shield much of the complexity of distributed computing from the user.

The nodes are numbered by their “ranks.” Each node has *rank value* `MP_RANK`. There are `MP_NPROCS` nodes, so `MP_RANK = 0, 1, ..., MP_NPROCS-1`. The root node has `MP_RANK = 0`. Most of the elementary MPI material is found in Gropp, Lusk, and Skjellum (1994) and Snir, Otto, Huss-Lederman, Walker, and Dongarra (1996). Although IMSL Fortran Numerical Library users are for the most part shielded from the complexity of MPI, it is desirable for some users to learn this important topic. Users should become familiar with any referenced MPI routines and the documentation of their usage. MPI routines are not discussed here, because that is best found in the above references.

The IMSL Fortran Numerical Library algorithm for allocating the racks of the box to the processors consists of creating a schedule for the processors, followed by communication and execution of this schedule. The efficiency may be improved by using the nodes according to a specific *priority order*. This order can reflect information such as a powerful machine on the network other than the user’s work station, or even transient network behavior. The IMSL Fortran Numerical Library allows users to define this order, but a default order is provided. A setup function establishes an order based on timing matrix products of a size given by the user. See [Parallel Example 4](#) for an illustration of this usage.

### Getting Started with Modules `MPI_setup_int` and `MPI_node_int`

The `MPI_setup_int` and `MPI_node_int` modules are part of the IMSL Fortran Numerical Library and not part of MPI itself. Following a call to the function `MP_SETUP()`, the module `MPI_node_int` will contain information about the number of processors, the rank of a processor, the communicator for IMSL Fortran Numerical Library, and the usage priority order of the node

machines. Since `MPI_node_int` is used by `MPI_setup_int`, it is not necessary to explicitly use this module. If neither `MP_SETUP()` nor `MPI_Init()` is called, then the box data type will compute entirely on one node. No routine from MPI will be called.

```

MODULE MPI_NODE_INT
 INTEGER, ALLOCATABLE :: MPI_NODE_PRIORITY(:)
 INTEGER, SAVE :: MP_LIBRARY_WORLD = huge(1)
 LOGICAL, SAVE :: MPI_ROOT_WORKS = .TRUE.
 INTEGER, SAVE :: MP_RANK = 0, MP_NPROCS = 1
END MODULE

```

When the function `MP_SETUP()` is called with no arguments, the following events occur:

- If MPI has not been initialized, it is first initialized. This step uses the routines `MPI_Initialized()` and possibly `MPI_Init()`. Users who choose not to call `MP_SETUP()` must make the required initialization call before using any IMSL Fortran Numerical Library code that relies on MPI for its execution. If the user's code calls an IMSL Fortran Numerical Library function utilizing the box data type and MPI has not been initialized, then the computations are performed on the root node. The only MPI routine always called in this context is `MPI_Initialized()`. The name `MP_SETUP` is pushed onto the subprogram or call stack.
- If `MP_LIBRARY_WORLD` equals its initial value (`=huge(1)`) then `MPI_COMM_WORLD`, the default MPI communicator, is duplicated and becomes its handle. This uses the routine `MPI_Comm_dup()`. Users can change the handle of `MP_LIBRARY_WORLD` as required by their application code. Often this issue can be ignored.
- The integers `MP_RANK` and `MP_NPROCS` are respectively the node's rank and the number of nodes in the communicator, `MP_LIBRARY_WORLD`. Their values require the routines `MPI_Comm_size()` and `MPI_Comm_rank()`. The default values are important when MPI is not initialized and a box data type is computed. In this case the root node is the only node and it will do all the work. No calls to MPI communication routines are made when `MP_NPROCS = 1` when computing the box data type functions. A program can temporarily assign this value to force box data type computation entirely at the root node. This is desirable for problems where using many nodes would be less efficient than using the root node exclusively.
- The array `MPI_NODE_PRIORITY(:)` is unallocated unless the user allocates it. The IMSL Fortran Numerical Library codes use this array for assigning tasks to processors, if it is allocated. If it is not allocated, the default priority of the nodes is `(0, 1, ..., MP_NPROCS-1)`. Use of the function call `MP_SETUP(N)` allocates the array, as explained below. Once the array is allocated its size is `MP_NPROCS`. The contents of the array is a permutation of the integers `0, ..., MP_NPROCS-1`. Nodes appearing at the start of the list are used first for parallel computing. A node other than the root can avoid any computing, except receiving the schedule, by setting the value `MPI_NODE_PRIORITY(I) < 0`. This means that node `|MPI_NODE_PRIORITY(I)|` will be sent the task schedule but will not perform any significant work as part of box data type function evaluations.

- The LOGICAL flag `MPI_ROOT_WORKS` designates whether or not the root node participates in the major computation of the tasks. The root node communicates with the other nodes to complete the tasks but can be designated to do no other work. Since there may be only one processor, this flag has the default value `.TRUE.`, assuring that one node exists to do work. When more than one processor is available users can consider assigning `MPI_ROOT_WORKS=.FALSE.` This is desirable when the alternate nodes have equal or greater computational resources compared with the root node. [Parallel Example 4](#) illustrates this usage. A single problem is given a box data type, with one rack. The computing is done at the node, other than the root, with highest priority. This example requires more than one processor since the root does no work.

When the generic function `MP_SETUP(N)` is called, where `N` is a positive integer, a call to `MP_SETUP()` is first made, using no argument. Use just one of these calls to `MP_SETUP()`. This initializes the MPI system and the other parameters described above. The array `MPI_NODE_PRIORITY(:)` is allocated with size `MP_NPROCS`. Then DOUBLE PRECISION matrix products  $C = AB$ , where  $A$  and  $B$  are  $N$  by  $N$  matrices, are computed at each node and the elapsed time is recorded. These elapsed times are sorted and the contents of `MPI_NODE_PRIORITY(:)` are permuted in accordance with the shortest times yielding the highest priority. All the nodes in the communicator `MP_LIBRARY_WORLD` are timed. The array `MPI_NODE_PRIORITY(:)` is then broadcast from the root to the remaining nodes of `MP_LIBRARY_WORLD` using the routine `MPI_Bcast()`. Timing matrix products to define the node priority is relevant because the effort to compute  $C$  is comparable to that of many linear algebra computations of similar size. Users are free to define their own node priority and broadcast the array `MPI_NODE_PRIORITY(:)` to the alternate nodes in the communicator.

To print any IMSL Fortran Numerical Library error messages that have occurred at any node, and to finalize MPI, use the function call `MP_SETUP('Final')`. Case of the string 'Final' is not important. Any error messages pending will be discarded after printing on the root node. This is triggered by popping the name 'MP\_SETUP' from the subprogram stack or returning to Level 1 in the stack. Users can obtain error messages by popping the stack to Level 1 and still continuing with MPI calls. This requires executing call `elpop('MP_SETUP')`. To continue on after summarizing errors execute call `elsh('MP_SETUP')`. More details about the error processor are found in Reference Material chapter of this manual.

Messages are printed by nodes from largest rank to smallest, which is the root node. Use of the routine `MPI_Finalize()` is made within `MP_SETUP('Final')`, which shuts down MPI. After `MPI_Finalize()` is called, the value of `MP_NPROCS = 0`. This flags that MPI has been initialized and terminated. It cannot be initialized again in the same program unit execution. No MPI routine is defined when `MP_NPROCS` has this value.

## Using Processors

There are certain pitfalls to avoid when using IMSL Fortran Numerical Library and box data types as implemented with MPI. A fundamental requirement is to allow all processors to participate in parts of the program where their presence is needed for correctness. It is incorrect to have a program unit that restricts nodes from executing a block of code required when computing with the box data type. On the other hand it is appropriate to restrict computations with rank-2 arrays to the root node. This is not required, but the results for the alternate nodes are normally discarded. This will avoid gratuitous error messages that may appear at alternate nodes.

Observe that only the root has a correct result for a box data type function. Alternate nodes have the constant value one as the result. The reason for this is that during the computation of the

functions, sub-problems are allocated to the alternate nodes by the root, but for only the root to utilize the result. If a user needs a value at the other nodes, then the root must send it to the nodes. See Parallel Example 3 for an illustration of this usage. Convergence information is computed at the root node and broadcast to the others. Without this step some nodes would not terminate the loop even when corrections at the root become small. This would cause the program to be incorrect.

---

## Sparse Matrix Computations

### Introduction

This section is concerned with methods for computing with sparse matrices. Our primary goal is to give the appearance of simplicity and allow ease-of-use in dealing with these calculations. The underlying principle in our design is to use Fortran 2003 standard support for derived types with initialized and allocatable components. To perform data storage and conversions we use overloaded assignment to hide complexity. The operations currently supported are:

- defining entries of the matrices,
- adding sparse matrices,
- forming products of sparse matrices and dense vectors or matrices,
- solving linear systems of algebraic equations
- condition number computation
- conversion of sparse matrices or dense arrays to the converse
- storage management operations

The definition of the sparse matrices starts with a *triplet* consisting of the row and column indices and a value at that entry. By setting a flag in the derived type `SLU_Options`, repeated values may be accumulated to yield a value that is the sum of all triplets for that matrix entry. A diagram for constructing a single precision sparse  $10000 \times 10000$  matrix, `H`, is illustrated with the pseudocode fragment:

```
Use linear_operators
```

```
Integer I, J; Real(Kind(1.e0)) value, x(10000)
```

```
Type(s_sparse) A
```

```
Type(s_hbc_sparse) H
```

```
Define non-zero values of A with repeated overloaded assignments
```

```
A = s_entry(I, J, value).
```

```
Convert to computational Harwell-Boeing form with the overloaded assignment H = A.
```

```
Compute with sparse matrix H, e.g. x = H .ix. x.
```

A basic feature is that there are four sparse matrix derived types, *Types* (`s_hbc_sparse`), (`d_hbc_sparse`), (`c_hbc_sparse`), and (`z_hbc_sparse`). These respectively handle single, double, complex and double-complex data. The defined operators work with a sparse matrix and a

corresponding dense array of the same precision and data type. There is no mixing of data types such as a sparse double precision matrix multiplied by a single precision vector. To accommodate that case an intermediate double precision quantity will be created that ascends the single precision vector to a double precision vector. The table below shows the operations that are valid with sparse matrix types.

| Mathematical Operation                            | Operation Notation      | Input Terms                                     | Output Terms            |
|---------------------------------------------------|-------------------------|-------------------------------------------------|-------------------------|
| $y = H^{-1}x$                                     | $y = H \text{ .ix. } x$ | $H_{n \times n}$ sparse, $x(1:k), k \geq n$     | $y(1:n)$                |
| $y = x^T H^{-1} \equiv H^{-T} x$                  | $y = x \text{ .xi. } H$ | $H_{n \times n}$ sparse, $x(1:k), k \geq n$     | $y(1:n)$                |
| $Y = H^{-1} X_{n \times r}$                       | $Y = H \text{ .ix. } X$ | $H_{n \times n}$ sparse, $X(1:k,1:r), k \geq n$ | $Y(1:n,1:r)$            |
| $Y = X_{r \times n} H^{-1} \equiv (H^{-T} X^T)^T$ | $Y = X \text{ .xi. } H$ | $H_{n \times n}$ sparse, $X(1:r,1:n), k \geq n$ | $Y(1:r,1:n)$            |
| $y = Hx$                                          | $y = H \text{ .x. } x$  | $H_{m \times n}$ sparse, $x(1:k), k \geq n$     | $y(1:m)$                |
| $y = x^T H \equiv H^T x$                          | $y = x \text{ .x. } H$  | $H_{m \times n}$ sparse, $x(1:k), k \geq m$     | $y(1:n)$                |
| $Y = HX_{n \times r}$                             | $Y = H \text{ .x. } X$  | $H_{m \times n}$ sparse, $X(1:k,1:r), k \geq n$ | $Y(1:m,1:r)$            |
| $Y = X_{r \times m} H$                            | $Y = X \text{ .x. } H$  | $H_{m \times n}$ sparse, $X(1:r,1:k), k \geq m$ | $Y(1:r,1:n)$            |
| $K = H^T$                                         | $K = \text{.t. } H$     | $H_{m \times n}$ sparse                         | $K_{n \times m}$ sparse |
| $K = H^H = \overline{H}^T$                        | $K = \text{.h. } H$     | $H_{m \times n}$ sparse, complex                | $K_{n \times m}$ sparse |
| $y = H^T x$                                       | $y = H \text{ .tx. } x$ | $H_{m \times n}$ sparse, $x(1:k), k \geq m$     | $y(1:n)$                |
| $Y = H^T X_{m \times r}$                          | $Y = H \text{ .tx. } X$ | $H_{m \times n}$ sparse, $X(1:k,1:r), k \geq m$ | $Y(1:n,1:r)$            |
| $y = x^T H$                                       | $Y = x \text{ .tx. } H$ | $H_{m \times n}$ sparse, $x(1:k), k \geq m$     | $y(1:n)$                |
| $Y = X_{r \times m}^T H$                          | $Y = X \text{ .tx. } H$ | $H_{m \times n}$ sparse, $X(1:k,1:r), k \geq m$ | $Y(1:r,1:n)$            |
| $y = Hx^T$                                        | $y = H \text{ .xt. } x$ | $H_{m \times n}$ sparse, $x(1:k), k \geq n$     | $y(1:m)$                |
| $Y = HX_{n \times r}^T$                           | $Y = H \text{ .xt. } X$ | $H_{m \times n}$ sparse, $x(1:k,1:r), k \geq n$ | $Y(1:m,1:r)$            |
| $y = xH^T$                                        | $y = x \text{ .xt. } H$ | $H_{m \times n}$ sparse, $x(1:k), k \geq n$     | $y(1:m)$                |

| Mathematical Operation                              | Operation Notation | Input Terms                                               | Output Terms |
|-----------------------------------------------------|--------------------|-----------------------------------------------------------|--------------|
| $Y = X_{r \times n} H^T$                            | $Y = X .xt. H$     | $H_{m \times n}$ sparse, $x(1:r,1:k), k \geq n$           | $Y(1:r,1:m)$ |
| $y = H^H x = \bar{H}^T x$                           | $y = H .hx. x$     | $H_{m \times n}$ sparse <sup>4</sup> , $x(1:k), k \geq m$ | $y(1:n)$     |
| $Y = H^H X_{m \times r} = \bar{H}^T X_{m \times r}$ | $Y = H .hx. X$     | $H_{m \times n}$ sparse, $X(1:k,1:r), k \geq m$           | $Y(1:n,1:r)$ |
| $y = x^H H = \bar{x}^T H$                           | $Y = x .hx. H$     | $H_{m \times n}$ sparse, $x(1:k), k \geq m$               | $y(1:n)$     |
| $Y = X_{r \times m}^H H = \bar{X}_{r \times m}^T H$ | $Y = X .hx. H$     | $H_{m \times n}$ sparse, $X(1:k,1:r), k \geq m$           | $Y(1:r,1:n)$ |
| $y = Hx^H = H\bar{x}^T$                             | $y = H .xh. x$     | $H_{m \times n}$ sparse, $x(1:k), k \geq n$               | $y(1:m)$     |
| $Y = HX_{n \times r}^H = H\bar{X}_{n \times r}^T$   | $Y = H .xh. X$     | $H_{m \times n}$ sparse, $x(1:k,1:r), k \geq n$           | $Y(1:m,1:r)$ |
| $y = xH^H = x\bar{H}^T$                             | $y = x .xh. H$     | $H_{m \times n}$ sparse, $x(1:k), k \geq n$               | $y(1:m)$     |
| $Y = X_{r \times n} H^H = X_{r \times n} \bar{H}^T$ | $Y = X .xh. H$     | $H_{m \times n}$ sparse, $x(1:r,1:k), k \geq n$           | $Y(1:r,1:m)$ |

## Derived Type Definitions

A derived type is used for the entries (triplets or coordinate format) of a sparse matrix, which consists of row and column coordinates and a corresponding value:

```

type s_entry
 integer irow
 integer jcol
 real(kind(1.e0)) value
end type

```

Additionally, type (`d_entry`), type (`c_entry`), and type (`z_entry`) are defined similarly. These support double precision, complex and complex-double precision accuracy and types.

Thus for a sparse matrix  $A$ , the entry at the intersection of row `irow` and column `jcol` is the scalar `value`. We define a sparse matrix representation in terms of a collection of triplets. This is a convenient way for a user to define a sparse matrix. This representation is used to define the matrix entries in a user's program using overloaded assignment. There is no implied order on the collection of triplets that define this sparse matrix. Our experience shows that for writing application code the technique of using triplets to define the matrix entries is convenient and provides a workable transition from mathematical definitions of the entries to computer code. Also note that there is generally no need for the programmer to allocate the components of a matrix of type `s_sparse` when using the overloaded assignment: `s_sparse = s_entry`. The software handles this detail by reallocating and expanding those components of the `s_sparse`

<sup>4</sup> The operators `.hx.` and `.xh.` apply to sparse complex matrices only. For real matrices use the `.tx.` and `.xt.` operators.

matrix as required. (For this task we use the Fortran 2003 intrinsic subroutine `move_alloc()`, when it is available. This routine provides an efficient way to perform a reallocation.) The amount reallocated is controlled by an expansion factor that is a component of the derived type `SLU_options`.

```

type s_sparse
 integer :: mrows = 0
 integer :: ncols = 0
 integer :: numnz = 0
 integer, allocatable, dimension(:) :: irow
 integer, allocatable, dimension(:) :: jcol
 real(kind(1.e0)), allocatable, dimension(:) :: value
 type (SLU_options) options
end type

```

When performing matrix computations we use the [Harwell-Boeing](#) column-oriented derived type. The row indices, for each column, are unique and increasing. The values in the `colptr(1:ncols)` component mark the start of the row indices and corresponding matrix entries for that column. The value `colptr(ncols+1)-1` will equal the value `numnz` after the matrix is defined with non-zero entries. The row indices for each column are in array `irow(:)`. They are unique and sorted into increasing order.

```

type s_hbc_sparse
 integer :: mrows = 0
 integer :: ncols = 0
 integer :: numnz = 0
 integer, allocatable, dimension(:) :: irow
 integer, allocatable, dimension(:) :: colptr
 real(kind(1.e0)), allocatable, dimension(:) :: value
 type (SLU_options) options
end type

```

Additionally we support types (`d_hbc_sparse`), type (`c_hbc_sparse`), and type (`z_hbc_sparse`). These will have analogous support for the operations defined with type (`s_hbc_sparse`) and others that follow. From now on we only mention type (`s_hbc_sparse`).

All components of the type (`s_sparse`) object are self-explanatory except for the one named `type(SLU_options)`. This component contains various parameters for managing the data structure, and for computing matrix products and linear system solutions. Normally these components do not need to be changed from their default values.

The derived type `SLU_options` carries extra required information. That data needed for [SuperLU<sup>5</sup>](#) is labeled with a comment. The remaining data is needed by IMSL codes that call on SuperLU. Of particular importance is the `Sequence` attribute statement. This prevents the Fortran compiler from rearranging the order of the components. Maintaining this order is required since the derived type `SLU_options` is passed to a IMSL C code that uses the information as a C

---

<sup>5</sup> SuperLU is used to support the defined operations `.ix.` and `.xi.`, and the condition number function, `cond()`. SuperLU is well-tested. Distributed and threaded versions are available but these are not used here in our software at present. SuperLU was developed by James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. **Note** that the authors do not support the package in the context used in the IMSL Libraries.

*structure.* The Sequence statement orders the Fortran-defined data so that it matches the C code structure.

```

Type SLU_options
Sequence
Integer :: unique = 1 ! Each new entry is unique -IMSL
Integer :: Accumulate = 0
 ! Accumulate or assemble duplicated entries in
 ! a ?_sparse matrix. This flag is checked
 ! when executing an overloaded assignment
 ! with a Harwell-Boeing = ?_sparse matrix.
 ! The default is not to accumulate (0)
 ! Assign the value 1 to accumulate.
Integer :: handle(2) = 0
 ! Each HBC matrix requiring an LU
 ! decomposition will have allocated
 ! arrays whose start is pointed to by
 ! this value. In cases where the OS
 ! uses 64 bit addressing 8 bytes are used.
Integer :: Info = - 1
 ! Flag returned after LU factorization (SuperLU)
Integer :: Fact = 0 !DOFACT - SuperLU
Integer :: Equil = 1 !YES
Integer :: ColPerm = 3 !COLAMD
Integer :: Trans = 0 !NOTRANS
Integer :: IterRefine = 1 !REFINE
Integer :: PrintStat = 0 !NO
Integer :: SymmetricMode = 0 !NO
Integer :: PivotGrowth = 0 !NO
Integer :: ConditionNumber = 0 !NO
Integer :: RowPerm = 0 !NO
Integer :: ReplaceTinyPivot = 0 !NO
Integer :: SolveInitialized = 0 !NO
Integer :: RefineInitialized = 0 !NO
Real (Kind(1.d0)) :: DiagPivotThresh = 1.d0 ! SuperLU
Real (Kind(1.d0)) :: expansion_factor = 1.2 ! VNI -
! The factor to use when expanding storage. Any value > 1.
! can be used such that the integer part of this factor times
! any integer > 9 provides at least a value of 1 increase.
Integer :: Cond_Iteration_Max = 30
! Maximum number of Lanczos and inverse iterations with sparse COND().
Integer Alignment_Dummy
End Type

```

## Overloaded Assignments

A natural way to define a sparse matrix is in terms of its triplets. The basic tool used here to define all the non-zero entries is *overloaded assignment*. Fortran 90, and further updates to the standard, supports a hidden subroutine call, packaged in a module, when an assignment is executed between differing derived types. Thus if a Fortran program has a declaration `type(s_sparse) A`, then the overloaded assignment statement

```
A = s_entry(I, J, value)
```

has the effect of calling subroutines that result in joining the matrix entry *value* at the intersection of row *I* and column *J*. The components of *A* are managed to hold any number of values. The

number of rows, columns and non-zero values are updated as new triplets are assigned. Also the arrays that hold the triplets are re-allocated and expanded, as required, to hold newly assigned triplets.

The code snippet for this operation, and others that follow, will require use of the module `linear_operators`. If new space is required in the assignment, a reallocation of the components of the matrix `A` will occur. The user does not have to manage the details.

```
Use linear_operators
Type(s_sparse) A
...

{For all entries in A, A = s_entry(I, J, Value)}
```

### **Sparse = Collection of Triplets**

The Harwell-Boeing sparse matrix data types are used for computations. An assignment, `H = A`, implies deallocating any allocated components of `H`, allocating new storage, and sorting the collection of triplets provided as input in the sparse matrix `A`. If the accumulation flag is set in `H%options%accumulate`, the duplicate row indices in a column are reduced to a single entry and the corresponding values are added to yield a final value. The assignment `H = 0` deallocates the allocated components and returns `H` to its initialized state, except for any changes to the component `SLU_options`. A similar comment holds for the assignment, `A = 0`.

```
Use linear_operators
Type(s_sparse) A
Type(s_hbc_sparse) H
...

{For all nonzero matrix entries, A = s_entry(I, J, Value)}
```

```
H = A
A = 0 ! Clear and deallocate components of A
...
H = 0 ! Clear and deallocate components of H
```

### **Sparse = Dense**

The non-zero entries of the dense array are converted to a Harwell-Boeing sparse matrix. As a first step any allocated components are cleared and then allocated as needed to hold the non-zero values of the dense array. The specific dimensions of array `D` are arbitrary.

```
Use linear_operators
Type(s_hbc_sparse) H
Integer, parameter :: M=1000, N=1000
Real (kind(1.e0)) D(M,N)
{Define entries of D}
H = D
```

## Dense = Sparse

For some applications it is convenient to expand a sparse matrix into a dense matrix. The specific dimensions of array `D` are arbitrary.

```
Use linear_operators
Type(s_hbc_sparse) H
Integer, parameter :: M=1000, N=1000
Real (kind(1.e0)) D(M,N)
{Define entries of H}
D = H
```

## Scalar = s\_hbc\_entry(Sparse, I, J)

This assignment gets the value at the intersection of row `I` and column `J` of the Harwell-Boeing sparse matrix. There must be type agreement with the function and sparse matrix type. Use a prefix of `d_`, `c_`, or `z_` for double, complex, or double complex values.

```
Use linear_operators
Type(s_hbc_sparse) H
Real (kind(1.e0)) value
{Define entries of H, I and J}
value = s_hbc_entry(H, I, J)
```

---

**.X.**



Computes matrix-matrix or matrix-vector product.

### Operator Return Value

Matrix containing the product of A and B. (Output)

### Required Operands

**A** — Left operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
Note that A and B cannot both be `?_hbc_sparse`.

**B** — Right operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
Note that A and B cannot both be `?_hbc_sparse`.

If A has rank one, B must have rank two.  
If B has rank one, A must have rank two.  
If A has rank three, B must have rank three.  
If B has rank three, A must have rank three.

### FORTRAN 90 Interface

A .x. B

### Description

Computes the product of matrix or vector A and matrix or vector B. The results are in a precision and data type that ascends to the most accurate or complex operand.

Rank three operation is defined as follows:

```
do i = 1, min(size(A,3), size(B,3))
 X(:, :, i) = A(:, :, i) .x. B(:, :, i)
end do
```

`.x.` can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

## Examples

### Dense Matrix Example (operator\_ex03.f90)

```
use linear_operators
implicit none

! This is the equivalent of Example 3 for LIN_SOL_GEN using operators.
integer, parameter :: n=32
real(kind(1e0)) :: one=1e0, zero=0e0, A(n,n), b(n), x(n)
real(kind(1e0)) change_new, change_old
real(kind(1d0)) :: d_zero=0d0, c(n), d(n,n), y(n)

! Generate a random matrix and right-hand side.
A = rand(A); b= rand(b)

! Save double precision copies of the matrix and right-hand side.
D = A
c = b

! Compute single precision inverse to compute the iterative refinement.
A = .i. A

! Start solution at zero. Update it to an accurate solution
! with each iteration.
y = d_zero
change_old = huge(one)

iterative_refinement: do
! Compute the residual with higher accuracy than the data.
b = c - (D .x. y)

! Compute the update in single precision.
x = A .x. b
y = x + y
change_new = norm(x)

! Exit when changes are no longer decreasing.
if (change_new >= change_old) exit iterative_refinement
change_old = change_new
end do iterative_refinement

write (*,*) 'Example 3 for LIN_SOL_GEN (operators) is correct.'
end
```

### Sparse Matrix Example

Consider the one-dimensional Dirichlet problem

$$\frac{d^2u}{dx^2} = f(x), \quad a < x < b, \quad u(a) = u_a = u_1, \quad u(b) = u_b = u_N$$

Using a standard approach to solving this involves approximating the second derivative operator with central divided differences

$$\frac{d^2u}{dx^2} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}, \quad h = (b-a)/(N-1), \quad i = 2, \dots, N-1, \quad N > 2$$

This leads to the sparse linear algebraic system  $Mu = w$ . The definitions for these terms are implied in the following Fortran program.

```

Subroutine document_ex1
! Illustrate a 1D Poisson equation with Dirichlet boundary conditions.
! This module defines the structures and overloaded assignment code.
 Use linear_operators
 Implicit None
!
 Integer :: I
 Integer, Parameter :: N = 1000
 Real (Kind(1.d0)) :: f, h, r, w (N), a = 0.d0, b = 1.d0, &
 u_a = 0.d0, u_b = 1.d0, u (N)
 Type (d_sparse) M
 Type (d_hbc_sparse) K
 External f
! Define the difference used.
 h = (b-a) / (N-1)
 r = 1.d0 / h ** 2
! Fill in the matrix entries.
! Isolated equation for the left boundary condition.
 M = d_entry (1, 1, r)
 Do I = 2, N - 1
 M = d_entry (I, I-1, r)
 M = d_entry (I, I, -2*r)
 M = d_entry (I, I+1, r)
 End Do
! Isolated equation for the right boundary condition.
 M = d_entry (N, N, r)
! Fill in the right-hand side (a dense vector).
 Do I = 2, N - 1
 w (I) = f (a+(I-1)*h)
 End Do
! Insert the known end conditions. These should be satisfied
! almost exactly, up to rounding errors.
 w (1) = u_a * r
 w (N) = u_b * r
! Ready to solve ...
! Conversion to Harwell-Boeing format using overloaded assignment
 K = M
! Solve the system using an IMSL defined operator.
 u = K .ix. w
! The parentheses are needed because of precedence rules.
! Compute residuals and overwrite w(:) with these values.
 w = w - (K .x. u)
End Subroutine
!
Function f (x)

```

```

 Real (Kind(1.d0)) :: f, x
! Define a hat function, peaked at x=0.5.
 If (x <= 0.5d0) Then
 f = x
 Else
 f = 1.d0 - x
 End If
End Function

```

### Parallel Example (parallel\_ex03.f90)

This example shows the box data type used while obtaining an accurate solution of several systems. Important in this example is the fact that only the root will achieve convergence, which controls program flow out of the loop. Therefore the nodes must share the root's view of convergence, and that is the reason for the broadcast of the update from root to the nodes. Note that when writing an explicit call to an MPI routine there must be the line `INCLUDE 'mpif.h'`, placed just after the `IMPLICIT NONE` statement. Any number of nodes can be used.

```

 use linear_operators
 use mpi_setup_int

 implicit none
 INCLUDE 'mpif.h'

! This is the equivalent of Parallel Example 3 for .i. and iterative
! refinement with box data types, operators and functions.
 integer, parameter :: n=32, nr=4
 integer IERROR
 real(kind(1e0)) :: one=1e0, zero=0e0
 real(kind(1e0)) :: A(n,n,nr), b(n,1,nr), x(n,1,nr)
 real(kind(1e0)) change_old(nr), change_new(nr)
 real(kind(1d0)) :: d_zero=0d0, c(n,1,nr), D(n,n,nr), y(n,1,nr)

! Setup for MPI.
 MP_NPROCS=MP_SETUP()

! Generate a random matrix and right-hand side.
 A = rand(A); b= rand(b)

! Save double precision copies of the matrix and right-hand side.
 D = A
 c = b

! Get single precision inverse to compute the iterative refinement.
 A = .i. A

! Start solution at zero. Update it to a more accurate solution
! with each iteration.
 y = d_zero
 change_old = huge(one)

 ITERATIVE_REFINEMENT: DO

! Compute the residual with higher accuracy than the data.
 b = c - (D .x. y)

```

```

! Compute the update in single precision.
 x = A .x. b
 y = x + y
 change_new = norm(x)

! All processors must share the root's test of convergence.
 CALL MPI_BCAST(change_new, nr, MPI_REAL, 0, &
 MP_LIBRARY_WORLD, IERROR)

! Exit when changes are no longer decreasing.
 if (ALL(change_new >= change_old)) exit iterative_refinement
 change_old = change_new
end DO ITERATIVE_REFINEMENT

 IF(MP_RANK == 0) write (*,*) 'Parallel Example 3 is correct.'

! See to any error messages and quit MPI.
MP_NPROCS=MP_SETUP('Final')
end

```

---

**.tx.**



Computes transpose matrix-matrix or transpose matrix-vector product.

### Operator Return Value

Matrix containing the product of  $A^T$  and B. (Output)

### Required Operands

**A** — Left operand matrix. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
 Note that A and B cannot both be `?_hbc_sparse`.

**B** — Right operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
 Note that A and B cannot both be `?_hbc_sparse`.

If A has rank three, B must have rank three.  
 If B has rank three, A must have rank three.

## FORTRAN 90 Interface

A .tx. B

### Description

Computes the product of the transpose of matrix A and matrix or vector B. The results are in a precision and data type that ascends to the most accurate or complex operand.

Rank three operation is defined as follows:

```
do i = 1, min(size(A,3), size(B,3))
 X(:, :, i) = A(:, :, i) .tx. B(:, :, i)
end do
```

.tx. can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

### Examples

#### Dense Matrix Example (operator\_ex05.f90)

```
use linear_operators
implicit none

! This is the equivalent of Example 1 for LIN_SOL_SELF using operators
! and functions.
integer, parameter :: m=64, n=32
real(kind(1e0)) :: one=1.0e0, err
real(kind(1e0)) A(n,n), b(n,n), C(m,n), d(m,n), x(n,n)

! Generate two rectangular random matrices.
C = rand(C); d=rand(d)

! Form the normal equations for the rectangular system.
A = C .tx. C; b = C .tx. d

! Compute the solution for Ax = b, A is symmetric.
x = A .ix. b

! Check the results.
err = norm(b - (A .x. x))/(norm(A)+norm(b))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SOL_SELF (operators) is correct.'
end if

end
```

#### Sparse Matrix Example

```
use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H
integer, parameter :: N=3
```

```

real (kind(1.e0)) x(N,N), y(N,N), B(N,N)
real (kind(1.e0)) err

S = s_entry (1, 1, 2.0)
S = s_entry (1, 3, 1.0)
S = s_entry (2, 2, 4.0)
S = s_entry (3, 3, 6.0)
H = S ! sparse
X = H ! dense equivalent of H
B = rand(B)
Y = H .tx. B
call wrrrn ('H', X)
call wrrrn ('B', b)
call wrrrn ('H.tx. B ', y)

! Check the results.
err = norm(y - (X .tx. B))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .tx. operator is correct.'
end if

end

```

### Output

|   | H     |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 2.000 | 0.000 | 1.000 |
| 2 | 0.000 | 4.000 | 0.000 |
| 3 | 0.000 | 0.000 | 6.000 |

|   | B      |        |        |
|---|--------|--------|--------|
|   | 1      | 2      | 3      |
| 1 | 0.8711 | 0.4467 | 0.4743 |
| 2 | 0.8315 | 0.7257 | 0.4518 |
| 3 | 0.6839 | 0.0561 | 0.6972 |

|   | H .tx. B |       |       |
|---|----------|-------|-------|
|   | 1        | 2     | 3     |
| 1 | 1.742    | 0.893 | 0.949 |
| 2 | 3.326    | 2.903 | 1.807 |
| 3 | 4.975    | 0.784 | 4.657 |

Sparse example for .tx. operator is correct.

### Parallel Example (parallel\_ex05.f90)

```

use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 5 using box data types,
! operators and functions.

integer, parameter :: m=64, n=32, nr=4
real(kind(1e0)) :: one=1e0, err(nr)

```

```

 real(kind(1e0)), dimension(n,n,nr) :: A, b, x
 real(kind(1e0)), dimension(m,n,nr) :: C, d

! Setup for MPI.
 mp_nprocs = mp_setup()

! Generate two rectangular random matrices, only
! at the root node.
 if (mp_rank == 0) then
 C = rand(C); d=rand(d)
 endif

! Form the normal equations for the rectangular system.
 A = C .tx. C; b = C .tx. d

! Compute the solution for Ax = b.
 x = A .ix. b

! Check the results.
 err = norm(b - (A .x. x))/(norm(A)+norm(b))
 if (ALL(err <= sqrt(epsilon(one))) .AND. MP_RANK == 0) &
 write (*,*) 'Parallel Example 5 is correct.'

! See to any error messages and quit MPI.
 mp_nprocs = mp_setup('Final')

 end

```

---

## .xt.



Computes matrix- transpose matrix product.

### Operator Return Value

Matrix containing the product of A and  $B^T$ . (Output)

### Required Operands

A — Left operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
 Note that A and B cannot both be `?_hbc_sparse`.

**B** — Right operand matrix. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)  
 Note that A and B cannot both be `?_hbc_sparse`.

If A has rank three, B must have rank three.  
 If B has rank three, A must have rank three.

## FORTRAN 90 Interface

A .xt. B

### Description

Computes the product of matrix or vector A and the transpose of matrix B. The results are in a precision and data type that ascends to the most accurate or complex operand.

Rank three operation is defined as follows:

```
do i = 1, min(size(A,3), size(B,3))
 X(:, :, i) = A(:, :, i) .xt. B(:, :, i)
end do
```

`.xt.` can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

### Examples

#### Dense Matrix Example (operator\_ex14.f90)

```
use linear_operators
implicit none

!
integer, parameter :: n=32
real(kind(1d0)) :: one=1d0, zero=0d0
real(kind(1d0)) A(n,n), P(n,n), Q(n,n), &
 S_D(n), U_D(n,n), V_D(n,n)

! Generate a random matrix.
A = rand(A)

! Compute the singular value decomposition.
S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
Q = V_D .x. diag(S_D) .xt. V_D

! Check the results.
if (norm(EYE(n) - (P .xt. P)) &
 <= sqrt(epsilon(one))) then
 if (norm(A - (P .x. Q))/norm(A) &
 <= sqrt(epsilon(one))) then
```

```

 write (*,*) 'Example 2 for LIN_SOL_SVD (operators) is correct.'
 end if
end if
end

```

### Sparse Matrix Example

```

use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H
integer, parameter :: N=3
real (kind(1.e0)) x(N,N), y(N,N), a(N,N)
real (kind(1.e0)) err
S = s_entry (1, 1, 2.0)
S = s_entry (1, 3, 1.0)
S = s_entry (2, 2, 4.0)
S = s_entry (3, 3, 6.0)
H = S ! sparse
X = H ! dense equivalent of H
A = rand(A)
Y = A .xt. H
call wrrrn ('A', A)
call wrrrn ('H', X)
call wrrrn ('A .xt. H', y)

! Check the results.
err = norm(y - (A .xt. X))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .xt. operator is correct.'
end if

end

```

### Output

|   | A      |        |        |
|---|--------|--------|--------|
|   | 1      | 2      | 3      |
| 1 | 0.5423 | 0.2380 | 0.9250 |
| 2 | 0.0844 | 0.1323 | 0.1937 |
| 3 | 0.4146 | 0.3135 | 0.7757 |

|   | H     |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 2.000 | 0.000 | 1.000 |
| 2 | 0.000 | 4.000 | 0.000 |
| 3 | 0.000 | 0.000 | 6.000 |

|   | A .xt. H |       |       |
|---|----------|-------|-------|
|   | 1        | 2     | 3     |
| 1 | 2.010    | 0.952 | 5.550 |
| 2 | 0.363    | 0.529 | 1.162 |
| 3 | 1.605    | 1.254 | 4.654 |

Sparse example for .xt. operator is correct.

## Parallel Example (parallel\_ex15.f90)

A “Polar Decomposition” of several matrices are computed. The box data type and the `SVD()` function are used. Orthogonality and small residuals are checked to verify that the results are correct.

```
use linear_operators
use mpi_setup_int
implicit none

! This is the equivalent of Parallel Example 15 using operators and,
! functions for a polar decomposition.
integer, parameter :: n=33, nr=3
real(kind(ld0)) :: one=1d0, zero=0d0
real(kind(ld0)),dimension(n,n,nr) :: A, P, Q, &
 S_D(n,nr), U_D, V_D
real(kind(ld0)) TEMP1(nr), TEMP2(nr)

! Setup for MPI:
mp_nprocs = mp_setup()

! Generate a random matrix.
if(mp_rank == 0) A = rand(A)

! Compute the singular value decomposition.
S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
Q = V_D .x. diag(S_D) .xt. V_D
! Check the results for orthogonality and
! small residuals.
TEMP1 = NORM(spread(EYE(n),3,nr) - (p .xt. p))
TEMP2 = NORM(A - (P .X. Q)) / NORM(A)
if (ALL(TEMP1 <= sqrt(epsilon(one))) .and. &
 ALL(TEMP2 <= sqrt(epsilon(one)))) then
 if(mp_rank == 0) &
 write (*,*) 'Parallel Example 15 is correct.'
end if

! See to any error messages and exit MPI.
mp_nprocs = mp_setup('Final')

end
```

---

**.hx.**



Computes conjugate transpose matrix-matrix product.

## Operator Return Value

Matrix containing the product of  $A^H$  and B. (Output)

## Required Operands

**A** — Left operand matrix. This is an array of rank 2 or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types,

?\_hbc\_sparse. (Input)

Note that A and B cannot both be ?\_hbc\_sparse.

**B** — Right operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types,

?\_hbc\_sparse. (Input)

Note that A and B cannot both be ?\_hbc\_sparse.

If A has rank three, B must have rank three.

If B has rank three, A must have rank three.

## FORTRAN 90 Interface

A .hx. B

## Description

Computes the product of the conjugate transpose of matrix A and matrix or vector B. The results are in a precision and data type that ascends to the most accurate or complex operand.

Rank three operation is defined as follows:

```
do i = 1, min(size(A,3), size(B,3))
 X(:, :, i) = A(:, :, i) .hx. B(:, :, i)
end do
```

.hx. can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

## Examples

### Dense Matrix Example (operator\_ex32.f90)

```
use linear_operators
implicit none
! This is the equivalent of Example 4 (using operators) for LIN_EIG_GEN.

integer, parameter :: n=17
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)), dimension(n,n) :: A, C
real(kind(1d0)) variation(n), eta
complex(kind(1d0)), dimension(n,n) :: U, V, e(n), d(n)
```

```

! Generate a random matrix.
 A = rand(A)

! Compute the eigenvalues, left- and right- eigenvectors.
 D = EIG(A, W=V); E = EIG(.t.A, W=U)

! Compute condition numbers and variations of eigenvalues.
 variation = norm(A)/abs(diagonals(U .hx. V))

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again. Check the
! differences compared to the estimates. They should not exceed
! the bounds.
 eta = sqrt(epsilon(one))
 C = A + eta*(2*rand(A)-1)*A
 D = EIG(C)

! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
 if (count(abs(d)-abs(e) > eta*variation) == 0) then
 write (*,*) 'Example 4 for LIN_EIG_GEN (operators) is correct.'
 end if
end

```

### Sparse Matrix Example

```

use wrcrn_int
use linear_operators

type (c_sparse) S
type (c_hbc_sparse) H
integer, parameter :: N=3
complex (kind(1.e0)) x(N,N), y(N,N), A(N,N)
real (kind(1.e0)) err
S = c_entry (1, 1, (2.0, 1.0))
S = c_entry (1, 3, (1.0, 3.0))
S = c_entry (2, 2, (4.0, -1.0))
S = c_entry (3, 3, (6.0, 2.0))
H = S ! sparse
X = H ! dense equivalent of H
A= rand(A)
Y = H .hx. A
call wrcrn ('H', X)
call wrcrn ('A', a)
call wrcrn ('H .hx. A ', y)

! Check the results.
err = norm(y - (X .hx. A))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .hx. operator is correct.'
end if

end

```

### Output

```

 H
 1 2 3
1 (2.000, 1.000) (0.000, 0.000) (1.000, 3.000)
2 (0.000, 0.000) (4.000,-1.000) (0.000, 0.000)
3 (0.000, 0.000) (0.000, 0.000) (6.000, 2.000)

 A
 1 2 3
1 (0.6278, 0.8475) (0.8007, 0.4179) (0.4512, 0.2601)
2 (0.1249, 0.4675) (0.7957, 0.1609) (0.4228, 0.0507)
3 (0.4608, 0.0891) (0.3181, 0.9180) (0.9961, 0.1939)

 H .hx. A
 1 2 3
1 (2.103, 1.067) (2.019, 0.035) (1.163, 0.069)
2 (0.032, 1.995) (3.022, 1.439) (1.640, 0.626)
3 (6.113,-1.423) (5.799, 2.888) (7.596,-1.922)
Sparse example for .hx. operator is correct.

```

### Parallel Example

```

use linear_operators
use mpi_setup_int

integer, parameter :: N=32, nr=4
complex (kind(1.e0)) A(N,N,nr), B(N,N,nr), Y(N,N,nr)
! Setup for MPI
 mp_nprocs = mp_setup()

if (mp_rank == 0) then
 A = rand(A)
 B = rand(B)
end if

Y = A .hx. B

mp_nprocs = mp_setup ('Final')

end

```

---

## .xh.



Computes a matrix-conjugate transpose matrix product.

### Operator Return Value

Matrix containing the product of  $A$  and  $B^H$ . (Output)

## Required Operands

**A** — Left operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)

Note that A and B cannot both be `?_hbc_sparse`.

**B** — Right operand matrix. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)

Note that A and B cannot both be `?_hbc_sparse`.

If A has rank three, B must have rank three.

If B has rank three, A must have rank three.

## FORTRAN 90 Interface

A .xh. B

## Description

Computes the product of matrix or vector A and the conjugate transpose of matrix B. The results are in a precision and data type that ascends to the most accurate or complex operand.

Rank three operation is defined as follows:

```
do i = 1, min(size(A,3), size(B,3))
 X(:, :, i) = A(:, :, i) .xh. B(:, :, i)
end do
```

`.xh.` can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

## Examples

### Dense Matrix Example

```
use wrarn_int
use linear_operators
integer, parameter :: N=3
complex (kind(1.e0)) A(N,N), B(N,N), Y(N,N)

A = rand(A)
B = rand(B)
Y = A .xh. B
call wrarn ('A', a)
call wrarn ('H', b)
call wrarn ('A .xh. B ', y)
end
```

## Output

```
1 A 2 3
```

```

1 (0.8071, 0.0054) (0.5617, 0.2508) (0.0223, 0.5555)
2 (0.9380, 0.5181) (0.8895, 0.9512) (0.7951, 0.6010)
3 (0.8349, 0.7291) (0.4162, 0.5255) (0.7388, 0.0309)

```

```

 B
 1 2 3
1 (0.5342, 0.2246) (0.9045, 0.0550) (0.4576, 0.3173)
2 (0.5531, 0.3362) (0.0757, 0.3970) (0.6807, 0.8625)
3 (0.3553, 0.9157) (0.0951, 0.7807) (0.4853, 0.0617)

```

```

 A .xh. B
 1 2 3
1 (1.141, 0.265) (1.085,-0.113) (0.586,-0.884)
2 (2.029, 0.900) (2.198,-0.587) (2.058,-1.036)
3 (1.363, 0.434) (1.477,-0.619) (1.775,-0.811)

```

### Sparse Matrix Example

```

use wrcrn_int
use linear_operators

type (c_sparse) S
type (c_hbc_sparse) H
integer, parameter :: N=3
complex (kind(1.e0)) x(N,N), y(N,N), A(N,N)
real (kind(1.e0)) err
S = c_entry (1, 1, (2.0, 1.0))
S = c_entry (1, 3, (1.0, 3.0))
S = c_entry (2, 2, (4.0, -1.0))
S = c_entry (3, 3, (6.0, 2.0))
H = S ! sparse
X = H ! dense equivalent of H
A= rand(A)
Y = A .xh. H
call wrcrn ('A', a)
call wrcrn ('H', X)
call wrcrn ('A .xh. H ', y)

! Check the results.
err = norm(y - (A .xh. X))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .xh. operator is correct.'
end if

end

```

### Output

```

 A
 1 2 3
1 (0.8526, 0.3532) (0.1822, 0.3938) (0.8008, 0.1308)
2 (0.5599, 0.8914) (0.7541, 0.5163) (0.8713, 0.9580)
3 (0.9947, 0.2735) (0.6237, 0.2137) (0.3802, 0.8903)

```

H

```

 1 2 3
1 (2.000, 1.000) (0.000, 0.000) (1.000, 3.000)
2 (0.000, 0.000) (4.000,-1.000) (0.000, 0.000)
3 (0.000, 0.000) (0.000, 0.000) (6.000, 2.000)

```

```

 A .xh. H
 1 2 3
1 (3.252,-2.418) (0.335, 1.757) (5.066,-0.817)
2 (5.757,-0.433) (2.500, 2.819) (7.144, 4.005)
3 (5.314,-0.698) (2.281, 1.478) (4.062, 4.581)
Sparse example for .xh. operator is correct.

```

### Parallel Example

```

use linear_operators
use mpi_setup_int

integer, parameter :: N=32, nr=4
complex (kind(1.e0)) A(N,N,nr), B(N,N,nr), Y(N,N,nr)
! Setup for MPI
 mp_nprocs = mp_setup()

if (mp_rank == 0) then
 A = rand(A)
 B = rand(B)
end if

Y = A .xh. B

mp_nprocs = mp_setup ('Final')

end

```

---

## .t.

Computes the transpose of a matrix.

### Operator Return Value

Matrix containing the transpose of A. (Output)

### Required Operand

A — Matrix for which the transpose is to be computed. This is a real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input).

### FORTRAN 90 Interface

.t. A

## Description

Computes the transpose of matrix A. The operation may be read *transpose*, and the results are the mathematical objects in a precision and data type that matches the operand. Since this is a unary operation, it has *higher* Fortran 90 precedence than any other intrinsic unary array operation.

.t. can be used with either dense or sparse matrices.

## Examples

### Dense Matrix Example (operator\_ex07.f90)

```
use linear_operators

implicit none

! This is the equivalent of Example 3 (using operators) for LIN_SOL_SELF.

integer tries
integer, parameter :: m=8, n=4, k=2
integer ipivots(n+1)
real(kind(ld0)) :: one=1.0d0, err
real(kind(ld0)) a(n,n), b(n,1), c(m,n), x(n,1), &
 e(n), ATEMP(n,n)
type(d_options) :: iopti(4)

! Generate a random rectangular matrix.
C = rand(C)

! Generate a random right hand side for use in the inverse
! iteration.
b = rand(b)

! Compute the positive definite matrix.
A = C .tx. C; A = (A+.t.A)/2

! Obtain just the eigenvalues.
E = EIG(A)

! Use packaged option to reset the value of a small diagonal.
iopti(4) = 0
iopti(1) = d_options(d_lin_sol_self_set_small,&
 epsilon(one)*abs(E(1)))

! Use packaged option to save the factorization.
iopti(2) = d_lin_sol_self_save_factors

! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
iopti(3) = d_lin_sol_self_no_sing_mess

ATEMP = A

! Compute A-eigenvalue*I as the coefficient matrix.
! Use eigenvalue number k.
```

```

 A = A - e(k)*EYE(n)

 do tries=1,2
 call lin_sol_self(A, b, x, &
 pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
 iopti(4) = d_lin_sol_self_solve_A

! Reset right-hand side in the direction of the eigenvector.
 B = UNIT(x)
 end do

! Normalize the eigenvector.
 x = UNIT(x)

! Check the results.
 b=ATEMP .x. x
 err = dot_product(x(1:n,1), b(1:n,1)) - e(k)

! If any result is not accurate, quit with no printing.
 if (abs(err) <= sqrt(epsilon(one))*E(1)) then
 write (*,*) 'Example 3 for LIN_SOL_SELF (operators) is correct.'
 end if

end

```

### Sparse Matrix Example

```

use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H, HT
integer, parameter :: N=3
real (kind(1.e0)) X(3,3), XT(3,3)
real (kind(1.e0)) err
S = s_entry (1, 1, 2.0)
S = s_entry (1, 3, 1.0)
S = s_entry (2, 2, 4.0)
S = s_entry (3, 3, 6.0)
H = S ! sparse
X = H ! dense equivalent of H
HT = .t. H
XT = HT ! dense equivalent of HT
call wrrrn ('H', X)
call wrrrn ('H Transpose', XT)

! Check the results.
err = norm(XT - (.t. X))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .t. operator is correct.'
end if

end

```

## Output

```
 H
 1 2 3
1 2.000 0.000 1.000
2 0.000 4.000 0.000
3 0.000 0.000 6.000
```

```
 H Transpose
 1 2 3
1 2.000 0.000 0.000
2 0.000 4.000 0.000
3 1.000 0.000 6.000
```

Sparse example for `.t.` operator is correct.

---

## .h.

Computes the conjugate transpose of a matrix.

### Operator Return Value

Matrix containing the conjugate transpose of A. (Output)

### Required Operand

A — Matrix for which the conjugate transpose is to be computed. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)

### FORTRAN 90 Interface

```
.h. A
```

### Description

Computes the conjugate transpose of matrix A. The operation may be read *adjoint*, and the results are the mathematical objects in a precision and data type that matches the operand. Since this is a unary operation, it has *higher* Fortran 90 precedence than any other intrinsic unary array operation.

`.h.` can be used with either dense or sparse matrices.

### Examples

#### Dense Matrix Example (operator\_ex34.f90)

```
use linear_operators
implicit none
```

! This is the equivalent of Example 2 (using operators) for LIN\_GEIG\_GEN.

```

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0, zero=0d0
real(kind(1d0)) err, alpha(n)
complex(kind(1d0)), dimension(n,n) :: A, B, C, D, V

! Generate random matrices for both A and B.
C = rand(C); D = rand(D)
A = C + .h.C; B = D .hx. D; B = (B + .h.B)/2

ALPHA = EIG(A, B=B, W=V)

! Check that residuals are small. Use a real array for alpha
! since the eigenvalues are known to be real.
err= norm((A .x. V) - (B .x. V .x. diag(alpha)),1)/&
 (norm(A,1)+norm(B,1)*norm(alpha,1))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_GEIG_GEN (operators) is correct.'
end if

end

```

### Sparse Matrix Example

```

use wrcrn_int
use linear_operators

type (c_sparse) S
type (c_hbc_sparse) H, HT
integer, parameter :: N=3
complex (kind(1.e0)) X(3,3), XT(3,3)
real (kind(1.e0)) err
S = c_entry (1, 1, (2.0, 1.0))
S = c_entry (1, 3, (1.0, 3.0))
S = c_entry (2, 2, (4.0, -1.0))
S = c_entry (3, 3, (6.0, 2.0))
H = S ! sparse
X = H ! dense equivalent of H
HT = .h. H
XT = HT ! dense equivalent of HT
call wrcrn ('H', X)
call wrcrn ('H Conjugate Transpose', XT)

! Check the results.
err = norm(XT - (.h. X))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .h. operator is correct.'
end if

end

```

### Output

```

H
1 2 3

```

```

1 (2.000, 1.000) (0.000, 0.000) (1.000, 3.000)
2 (0.000, 0.000) (4.000,-1.000) (0.000, 0.000)
3 (0.000, 0.000) (0.000, 0.000) (6.000, 2.000)

```

H Conjugate Transpose

```

1 (2.000,-1.000) (0.000, 0.000) (0.000, 0.000)
2 (0.000, 0.000) (4.000, 1.000) (0.000, 0.000)
3 (1.000,-3.000) (0.000, 0.000) (6.000,-2.000)
Sparse example for .h. operator is correct.

```

**.i.**



Computes the inverse matrix.

### Operator Return Value

Matrix containing the inverse of A. (Output)

### Required Operand

A — Matrix for which the inverse is to be computed. This is an array of rank 2 or 3. It may be real, double, complex, double complex. (Input)

### Optional Variables, Reserved Names

This operator uses the routines [LIN\\_SOL\\_GEN](#) or [LIN\\_SOL\\_LSQ](#) (See [Chapter 1, “Linear Systems”](#)).

The option and derived type names are given in the following tables:

| Option Names for .i.      | Option Value |
|---------------------------|--------------|
| Use_lin_sol_gen_only      | 1            |
| Use_lin_sol_lsq_only      | 2            |
| I_options_for_lin_sol_gen | 3            |
| I_options_for_lin_sol_lsq | 4            |
| Skip_error_processing     | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_inv_options(:)                                            | Use when setting options for calls hereafter. | ?_options    |
| ?_inv_options_once(:)                                       | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See [LIN\\_SOL\\_GEN](#) and [LIN\\_SOL\\_LSQ](#) located in [Chapter 1, “Linear Systems”](#) for the specific options for these routines.

## FORTRAN 90 Interface

.i. A

### Description

Computes the inverse matrix for square non-singular matrices using `LIN_SOL_GEN`, or the Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices using `LIN_SOL_LSQ`. The operation may be read *inverse or generalized inverse*, and the results are in a precision and data type that matches the operand.

This operator requires a single operand. Since this is a unary operation, it has *higher* Fortran 90 precedence than any other intrinsic array operation.

### Examples

#### Dense Matrix Example (operator\_ex02.f90)

```

use linear_operators
implicit none

! This is the equivalent of Example 2 for LIN_SOL_GEN using operators
! and functions.

integer, parameter :: n=32
real(kind(1e0)) :: one=1e0, err, det_A, det_i
real(kind(1e0)), dimension(n,n) :: A, inv

! Generate a random matrix.
A = rand(A)
! Compute the matrix inverse and its determinant.
inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
det_i = det(inv)
! Check the quality of both left and right inverses.
err = (norm(EYE(n)-(A .x. inv))+norm(EYE(n)-(inv.x.A)))/cond(A)
if (err <= sqrt(epsilon(one)) .and. abs(det_A*det_i - one) <= &
sqrt(epsilon(one))) &
write (*,*) 'Example 2 for LIN_SOL_GEN (operators) is correct.'
end

```

### Parallel Example (parallel\_ex02.f90)

```
use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 2 for .i. and det() with box
! data types, operators and functions.

integer, parameter :: n=32, nr=4
integer J
real(kind(1e0)) :: one=1e0
real(kind(1e0)), dimension(nr) :: err, det_A, det_i
real(kind(1e0)), dimension(n,n,nr) :: A, inv, R, S

! Setup for MPI.
MP_NPROCS=MP_SETUP()
! Generate a random matrix.
A = rand(A)
! Compute the matrix inverse and its determinant.
inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
det_i = det(inv)
! Check the quality of both left and right inverses.
DO J=1,nr; R(:, :, J)=EYE(N); END DO

S=R; R=R-(A .x. inv); S=S-(inv .x. A)
err = (norm(R)+norm(S))/cond(A)
if (ALL(err <= sqrt(epsilon(one)) .and. &
 abs(det_A*det_i - one) <= sqrt(epsilon(one))) &
 .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 2 is correct.'

! See to any error messages and quit MPI.
MP_NPROCS=MP_SETUP('Final')

end
```

---

## .ix.



Computes the product of the inverse of a matrix and a vector or matrix.

### Operator Return Value

Matrix containing the product of  $A^{-1}$  and B. (Output)

## Required Operands

**A** — Left operand matrix. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)

**B** — Right operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, or double complex. (Input)

## Optional Variables, Reserved Names

This operator uses the routines `LIN_SOL_GEN` or `LIN_SOL_LSQ` (See [Chapter 1, “Linear Systems”](#)).

The option and derived type names are given in the following tables:

| Option Names for <code>.ix.</code>      | Option Value |
|-----------------------------------------|--------------|
| <code>use_lin_sol_gen_only</code>       | 1            |
| <code>use_lin_sol_lsq_only</code>       | 2            |
| <code>ix_options_for_lin_sol_gen</code> | 3            |
| <code>ix_options_for_lin_sol_lsq</code> | 4            |
| <code>Skip_error_processing</code>      | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_invx_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_invx_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_GEN` and `LIN_SOL_LSQ` located in [Chapter 1, “Linear Systems”](#) for the specific options for these routines.

## FORTRAN 90 Interface

`A .ix. B`

## Description

Computes the product of the inverse of matrix **A** and vector or matrix **B**, for square non-singular matrices or the corresponding Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *generalized inverse times*. The results are in a precision and data type that matches the most accurate or complex operand.

`.ix.` can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

## Examples

### Dense Matrix Example (operator\_ex01.f90)

```
use linear_operators
implicit none

! This is the equivalent of Example 1 for LIN_SOL_GEN, with operators
! and functions.

integer, parameter :: n=32
real(kind(1e0)) :: one=1.0e0, err
real(kind(1e0)), dimension(n,n) :: A, b, x

! Generate random matrices for A and b:
A = rand(A); b=rand(b)

! Compute the solution matrix of Ax = b.
x = A .ix. b

! Check the results.
err = norm(b - (A .x. x))/(norm(A)*norm(x)+norm(b))
if (err <= sqrt(epsilon(one))) &
write (*,*) 'Example 1 for LIN_SOL_GEN (operators) is correct.'
end
```

### Sparse Matrix Example 1

```
use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H
integer, parameter :: N=3
real (kind(1.e0)) x(N,N), y(N,N), B(N,N)
real (kind(1.e0)) err
S = s_entry (1, 1, 2.0)
S = s_entry (1, 3, 1.0)
S = s_entry (2, 2, 4.0)
S = s_entry (3, 3, 6.0)
H = S ! sparse
X = H ! dense equivalent of H
B= rand(B)
Y = H .ix. B
call wrrrn ('H', X)
call wrrrn ('B', b)
call wrrrn ('H .ix. B ', y)

! Check the results.
err = norm(y - (X .ix. B))
if (err <= sqrt(epsilon(one))) then
write (*,*) 'Sparse example for .ix. operator is correct.'
end if

end
```

## Output

| H |       |       |       |
|---|-------|-------|-------|
|   | 1     | 2     | 3     |
| 1 | 2.000 | 0.000 | 1.000 |
| 2 | 0.000 | 4.000 | 0.000 |
| 3 | 0.000 | 0.000 | 6.000 |

| B |        |        |        |
|---|--------|--------|--------|
|   | 1      | 2      | 3      |
| 1 | 0.8292 | 0.5697 | 0.1687 |
| 2 | 0.9670 | 0.7296 | 0.0603 |
| 3 | 0.1458 | 0.2726 | 0.8809 |

| H . i x . B |        |        |        |
|-------------|--------|--------|--------|
|             | 1      | 2      | 3      |
| 1           | 0.4025 | 0.2621 | 0.0109 |
| 2           | 0.2417 | 0.1824 | 0.0151 |
| 3           | 0.0243 | 0.0454 | 0.1468 |

### Sparse Matrix Example 2: Plane Poisson Problem with Dirichlet Boundary Conditions

We want to calculate a numerical solution, which approximates the true solution of the Poisson (boundary value) problem in the solution domain  $\Omega$ , a rectangle in  $\mathbf{R}^2$ . The equation is

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \quad \text{in } \Omega$$

There are Dirichlet boundary conditions  $u = g$  on  $\partial_1\Omega$

There are further Neumann boundary conditions  $\frac{\partial u}{\partial n} = h$  on  $\partial_2\Omega$

The boundary arcs comprising  $\partial_1\Omega \cup \partial_2\Omega = \partial\Omega$  are mutually exclusive of each other. The functions  $f, g, h$  are defined on their respective domains.

We will solve an instance of this problem by using finite differences to approximate the derivatives. This will lead to a sparse system of linear algebraic equations. Note that particular cases of this problem can be solved with methods that are likely to be more efficient or more appropriate than the one illustrated here. We use this method to illustrate our matrix data handling routines and defined operators.

The area of the rectangle  $\Omega$  is  $a \times b$  with the origin fixed at the lower left or SW corner. The dimension along the  $x$  axis is  $a$  and along the  $y$  axis is  $b$ . A rectangular  $n \times m$  uniform grid is defined on  $\Omega$  where each sub-rectangle in the grid has sides

$\Delta x = a/(n-1)$  and  $\Delta y = b/(m-1)$ . What is perhaps novel in our development is that the boundary values are written into the  $(m \times n)^2$  linear system as trivial equations. This leads to more unknowns than standard approaches to this problem but the complexity of describing the

equations into computer code is reduced. The boundary conditions are naturally in place when the solution is obtained. No reshaping is required.

We number the approximate values of  $u$  at the grid points and collapse them into a single vector. Given a coordinate of the grid  $(i, j)$ ,  $((i = 1, \dots, n), j = 1, \dots, m)$ , we use the mapping  $J = i + (j - 1)n$  to define coordinate  $J$  of this vector. This mapping enables us to define the matrix that is used to solve for the values of  $u$  at the grid points.

For the Neumann boundary conditions we take  $\partial_2\Omega$  to be the East face of the rectangle. Along that edge we have  $\frac{\partial u}{\partial n} = \frac{\partial u}{\partial x}$ , and we impose the smooth interface  $h = 0$ .

Our use of finite differences is standard. For the differential equation we approximate

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \approx \left( \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} \right) + \left( \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \right) = f(x_i, y_j)$$

at the inner grid points  $(i, j)$ ,  $((i = 2, \dots, n-1), j = 2, \dots, m-1)$ . For the Neumann condition we approximate

$$\frac{\partial u}{\partial x} \approx \left( \frac{u_{n,j} - u_{n-1,j}}{\Delta x} \right) = 0, \quad j = 1, \dots, m$$

The remaining equations come from the Dirichlet conditions given on  $\partial_1\Omega$ .

To illustrate three examples of solutions to this problem we consider

1. A Laplace Equation with the boundary conditions

$$\begin{aligned} u &= 0, \text{ on the South Edge} \\ u &= 0.7, \text{ on the East Edge} \\ u &= 1, \text{ on the North Edge} \\ u &= 0.3, \text{ on the West Edge} \end{aligned}$$

The function  $f = 0$  for all  $(x, y)$ . Graphical results are shown below with the title “[Problem Case 1](#)”

2. A Poisson equation with the boundary conditions  $u = 0$  on all of the edges and  $f(x, y) = -\sin(\pi x)\sin(\pi y)$ . This problem has the solution  $u(x, y) = -f(x, y)/(2\pi^2)$ , and this identity provides a way of verifying that the accuracy is within the truncation error implied by the difference equations. Graphical results are shown with the title “[Problem Case 2](#)” The residual function verifies the expected accuracy.

3. The Laplace Equation with the boundary conditions of Problem Case 1 except that the boundary condition on the East Edge is replaced by the Neumann condition  $\frac{\partial u}{\partial x} = 0$ .

Graphical results are shown as “[Problem Case 3](#).”

```

Subroutine document_ex2
! Illustrate a 2D Poisson equation with Dirichlet and
! Neumann boundary conditions.
! These modules defines the structures and overloaded assignment code.
 Use linear_operators
 Implicit None
 Integer :: I, J, JJ, MY_CASE, IFILE
 Integer, Parameter :: N = 300, M = 300
 Real (Kind(1.d0)) :: a = 1.d0, b = 1.d0
 Real (Kind(1.d0)) :: delx, dely, r, s, pi, scale
 Real (Kind(1.d0)) :: u(N*M), w(N*M), P(N, M)
 Real (Kind(1.e0)) :: TS, TE
 CHARACTER (LEN=12) :: PR_LABEL(3) =&
 ('Laplace', 'Poisson', 'Neumann')
! Mapping function (in-line) for grid coordinate to
! matrix-vector indexing.
 JJ (I, J) = I + (J-1) * N

! Define sparse matrices to hold problem data.
 Type (d_sparse) C
 Type (d_hbc_sparse) D
! Define differences and related parameters.
 delx = a / (N-1)
 dely = b / (M-1)
 r = 1.d0 / delx ** 2
 s = 1.d0 / dely ** 2
 Do MY_CASE = 1, 3
! For MY_CASE =
! 1. Solve boundary value problem with f=0 and Dirichlet
! boundary conditions.
! 2. Solve Poisson equation with f such that a solution is known.
! Use zero boundary condtions.
! 3. Solve boundary value problem with Dirichlet condtions as in 1.
! except on the East edge. There the partial WRT x is zero.
! Set timer for building the matrix.
 Call cpu_time (TS)
 Do I = 2, N - 1
 Do J = 2, M - 1
! Write entries for second partials WRT x and y.
 C = d_entry (JJ(I, J), JJ(I-1, J), r)
 C = d_entry (JJ(I, J), JJ(I+1, J), r)
 C = d_entry (JJ(I, J), JJ(I, J), -2*(r+s))
 C = d_entry (JJ(I, J), JJ(I, J-1), s)
 C = d_entry (JJ(I, J), JJ(I, J+1), s)
 !
! Define components of the right-hand side.
 w (JJ(I, J)) = f((I-1)*delx, (J-1)*dely, MY_CASE)
 End Do
 End Do
! Write entries for Dirichlet boundary conditions.

```

```

! First do the South edge, then the West, then the North.
 Select Case (MY_CASE)
 Case (1:2)
 Do I = 1, N
 C = d_entry (JJ(I, 1), JJ(I, 1), r+s)
 w (JJ(I, 1)) = g ((I-1)*delx, 0.d0, MY_CASE) * (r+s)
 End Do
 Do J = 2, M - 1
 C = d_entry (JJ(1, J), JJ(1, J), r+s)
 w (JJ(1, J)) = g (0.d0, (J-1)*dely, MY_CASE) * (r+s)
 End Do
 Do I = 1, N
 C = d_entry (JJ(I, M), JJ(I, M), r+s)
 w (JJ(I, M)) = g ((I-1)*delx, b, MY_CASE) * (r+s)
 End Do
 Do J = 2, M - 1
 C = d_entry (JJ(N, J), JJ(N, J), (r+s))
 w (JJ(N, J)) = g (a, (J-1)*dely, MY_CASE) * (r+s)
 End Do
 Case (3)
! Write entries for the boundary values but avoid the East edge.
 Do I = 1, N - 1
 C = d_entry (JJ(I, 1), JJ(I, 1), r+s)
 w (JJ(I, 1)) = g ((I-1)*delx, 0.d0, MY_CASE) * (r+s)
 End Do
 Do J = 2, M - 1
 C = d_entry (JJ(1, J), JJ(1, J), r+s)
 w (JJ(1, J)) = g (0.d0, (J-1)*dely, MY_CASE) * (r+s)
 End Do
 Do I = 1, N - 1
 C = d_entry (JJ(I, M), JJ(I, M), r+s)
 w (JJ(I, M)) = g ((I-1)*delx, b, MY_CASE) * (r+s)
 End Do
! Write entries for the Neumann condition on the East edge.
 Do J = 1, M
 C = d_entry (JJ(N, J), JJ(N, J), 1.d0/delx)
 C = d_entry (JJ(N, J), JJ(N-2, J), -1.d0/delx)
 w (JJ(N, J)) = 0.d0
 End Do
 End Select
!
! Convert to Harwell-Boeing format for solving.
 D = C
!
 Call cpu_time (TE)
 Write (*,'(A,F6.2," S. - ",A)') "Time to build matrix = ", &
 TE - TS, PR_LABEL(MY_CASE)
! Clear sparse triplets.
 C = 0
!
! Turn off iterative refinement for maximal performance.
! This is generally not recommended unless
! the problem is known not to require it.
 If (MY_CASE == 2) D%options%iterRefine = 0
! This is the solve step.
 Call cpu_time (TS)

```

```

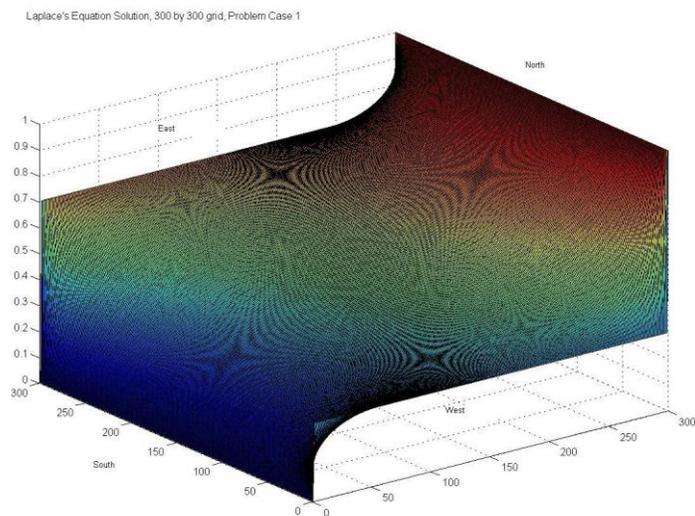
 u = D .ix. w
 Call cpu_time (TE)
 Write (*,'(A,I6," is",F6.2," S")') &
 "Time to solve system of size = ", N * M, TE - TS
! This is a second solve step using the factorization
! from the first step.
 Call cpu_time (TS)
 u = D .ix. w
 Call cpu_time (TE)
!
 If(MY_CASE == 1) then
 Write (*,'(A,I6," is",F6.2," S")') &
 "Time for a 2nd system of size (iterative refinement) =", &
 N * M, TE - TS
 Else
 Write (*,'(A,I6," is",F6.2," S")') &
 "Time for a 2nd system of size (without refinement) =", &
 N * M, TE - TS
 End if
! Convert solution vector to a 2D array of values.
 P = reshape (u , (/ N, M /))
 If (MY_CASE == 2) Then
 pi = dconst ('pi')
!
 scale = - 0.5 / pi ** 2
 Do I = 1, N
 Do J = 1, M
! This uses the known form of the solution to compute residuals.
 P (I, J) = P (I, J) - scale * f ((I-1)*delx, &
 (J-1)*dely, MY_CASE)
 End Do
 End Do
!
 write (*,*) minval (P), " = min solution error "
 write (*,*) maxval (P), " = max solution error "
 End If
 Write (*,'(A,1pE12.4/)') "Condition number of matrix", cond (D)
! Clear all matrix data for next problem case.
 D = 0
!
 End Do ! MY_CASE
Contains
 Function f (x, y, MY_CASE)
 implicit none
! Define the right-hand side function associated with the
! "del" operator.
 Real (Kind(1.d0)) x, y, f, pi
 Integer MY_CASE
 if(MY_CASE == 2) THEN
 pi = dconst ('pi')
 f = - Sin (pi*x) * Sin (pi*y)
 Else
 f = 0.d0
 End If
 End Function
!
 Function g (x, y, MY_CASE)

```

```

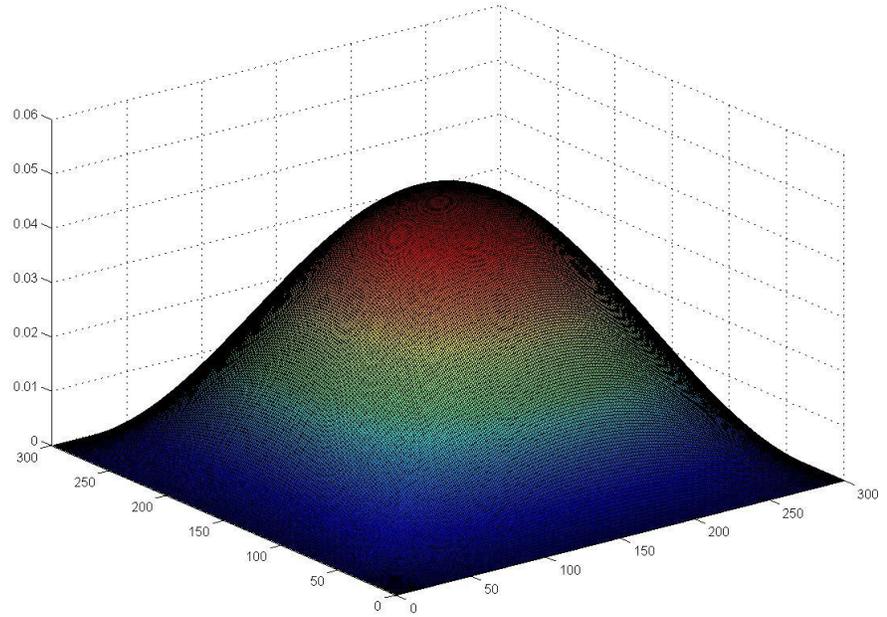
 implicit none
! Define the edge values, except along East edge, x = a.
 Real (Kind(1.d0)) x, y, g
 Integer MY_CASE
! Fill in a constant value along each edge.
 If (MY_CASE == 1 .Or. MY_CASE == 3) Then
 If (y == 0.d0) Then
 g = 0.d0
 Return
 End If
 If (y == b) Then
 g = 1.d0
 Return
 End If
 If (x == 0.d0) Then
 g = 0.3d0
 Return
 End If
 If (x == a) Then
 g = 0.7d0
 End If
 Else
 g = 0.d0
!
!
 End If
 End Function
End Subroutine

```



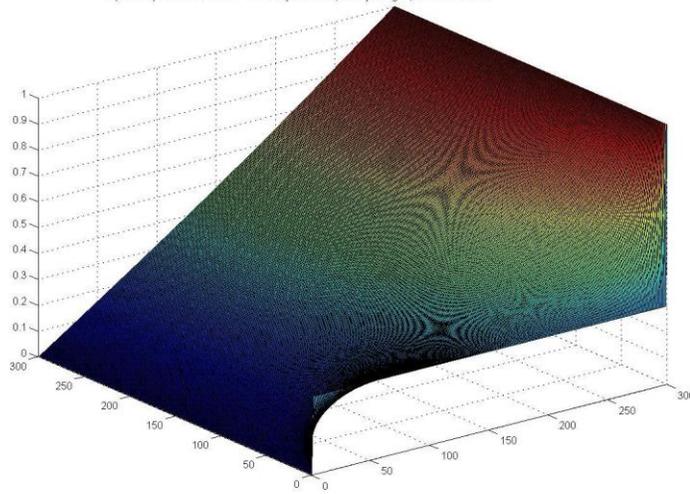
*Problem Case 1*

Poisson Equation Solution, 300 by 300 grid, Problem Case 2



*Problem Case 2*

Laplace Equation with Neumann Boundary Condition, 300 by 300 grid, Problem Case 3



*Problem Case 3*

### Parallel Example (parallel\_ex01.f90)

```

use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 1 for .ix., with box data types
! and functions.

integer, parameter :: n=32, nr=4
real(kind(1e0)) :: one=1e0
real(kind(1e0)), dimension(n,n,nr) :: A, b, x, err(nr)

! Setup for MPI.
MP_NPROCS=MP_SETUP()

! Generate random matrices for A and b:
A = rand(A); b=rand(b)

! Compute the box solution matrix of Ax = b.
x = A .ix. b

! Check the results.
err = norm(b - (A .x. x)) / (norm(A)*norm(x)+norm(b))
if (ALL(err <= sqrt(epsilon(one))) .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 1 is correct.'

! See to any error messages and quit MPI.
MP_NPROCS=MP_SETUP('Final')

end

```

---

## .xi.



Computes the product of a matrix or vector and the inverse of a matrix.

### Operator Return Value

Matrix containing the product of  $A$  and  $B^{-1}$ . (Output)

### Required Operands

$A$  — Right operand matrix or vector. This is an array of rank 1, 2, or 3. It may be real, double, complex, or double complex. (Input)

**B** — Left operand matrix. This is an array of rank 2, or 3. It may be real, double, complex, double complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. (Input)

## Optional Variables, Reserved Names

This operator uses the routines `LIN_SOL_GEN` or `LIN_SOL_LSQ` (See [Chapter 1, “Linear Systems”](#)).

The option and derived type names are given in the following tables:

| Option Names for <code>.xi.</code>      | Option Value |
|-----------------------------------------|--------------|
| <code>use_lin_sol_gen_only</code>       | 1            |
| <code>use_lin_sol_lsq_only</code>       | 2            |
| <code>xi_options_for_lin_sol_gen</code> | 3            |
| <code>xi_options_for_lin_sol_lsq</code> | 4            |
| <code>Skip_error_processing</code>      | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_xinv_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_xinv_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_GEN` and `LIN_SOL_LSQ` located in [Chapter 1, “Linear Systems”](#) for the specific options for these routines.

## FORTRAN 90 Interface

A `.xi.` B

### Description

Computes the product of matrix A and the inverse of matrix B, for square non-singular matrices or the corresponding Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *times generalized inverse*. The results are in a precision and data type that matches the most accurate or complex operand.

`.xi.` can be used with either dense or sparse matrices. It is MPI capable for dense matrices only.

### Examples

#### Dense Matrix Example

```
use linear_operators
implicit none
```

```

integer, parameter :: n=32
real(kind(1e0)) :: one=1.0e0, err
real(kind(1e0)), dimension(n,n) :: A, b, x

! Generate random matrices for A and b:
A = rand(A); b=rand(b)

! Compute the solution matrix of xA = b.
x = b .xi. A

! Check the results.
err = norm(b - (x .x. A))/(norm(A)*norm(x)+norm(b))
if (err <= sqrt(epsilon(one))) &
 write (*,*) 'Example for .xi. operator is correct.'
end

```

### Sparse Matrix Example

```

use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H
integer, parameter :: N=3
real (kind(1.e0)) x(N,N), y(N,N), a(N,N)
real (kind(1.e0)) err
S = s_entry (1, 1, 2.0)
S = s_entry (1, 3, 1.0)
S = s_entry (2, 2, 4.0)
S = s_entry (3, 3, 6.0)
H = S ! sparse
X = H ! dense equivalent of H
A = rand(A)
Y = A .xi. H
call wrrrn ('A', A)
call wrrrn ('H', X)
call wrrrn ('A .xi. H', y)

! Check the results.
err = norm(y - (A .xi. X))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Sparse example for .xi. operator is correct.'
end if

end

```

### Output

|   | A      |        |        |
|---|--------|--------|--------|
|   | 1      | 2      | 3      |
| 1 | 0.5926 | 0.5015 | 0.5368 |
| 2 | 0.4001 | 0.9529 | 0.6988 |
| 3 | 0.0412 | 0.0633 | 0.3821 |

```

 H
 1 2 3
1 2.000 0.000 1.000
2 0.000 4.000 0.000
3 0.000 0.000 6.000

```

```

 A .xi. H
 1 2 3
1 0.2963 0.1254 0.0401
2 0.2001 0.2382 0.0831
3 0.0206 0.0158 0.0602

```

Sparse example for .xi. operator is correct.

### Parallel Example

```

use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 1 for .xi., with box data types
! and functions.

integer, parameter :: n=32, nr=4
real(kind(1e0)) :: one=1e0
real(kind(1e0)), dimension(n,n,nr) :: A, b, x, err(nr)

! Setup for MPI.
MP_NPROCS=MP_SETUP()

! Generate random matrices for A and b:
A = rand(A); b=rand(b)

! Compute the box solution matrix of xA = b.
x = b .xi. A

! Check the results.
err = norm(b - (x .x. A)) / (norm(A)*norm(x)+norm(b))
if (ALL(err <= sqrt(epsilon(one))) .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 1 is correct.'

! See to any error messages and quit MPI.
MP_NPROCS=MP_SETUP('Final')

end

```

---

## CHOL



Computes the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix.

## Function Return Value

Matrix containing the Cholesky factorization of  $A$ . The factor is upper triangular,  $R^T R = A$ .  
(Output)

## Required Argument

$A$  — Matrix to be factored. This argument must be a rank-2 or rank-3 array that contains a positive-definite, symmetric or self-adjoint matrix. It may be real, double, complex, double complex. (Input)  
For rank-3 arrays each rank-2 array, (for fixed third subscript), is a positive-definite, symmetric or self-adjoint matrix. In this case, the output is a rank-3 array of Cholesky factors for the individual problems.

## Optional Arguments, Packaged Options

This function uses `LIN_SOL_SELF` (See [Chapter 1, “Linear Systems”](#)), using the appropriate options to obtain the Cholesky factorization.

The option and derived type names are given in the following tables:

| Option Names for <code>CHOL</code> | Option Value |
|------------------------------------|--------------|
| <code>Use_lin_sol_gen_only</code>  | 4            |
| <code>Use_lin_sol_lsq_only</code>  | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_chol_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_chol_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_SELF` located in [Chapter 1, “Linear Systems”](#) for the specific options for this routine.

## FORTRAN 90 Interface

`CHOL(A)`

## Description

Computes the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix,  $A$ . The factor is upper triangular,  $R^T R = A$ .

## Examples

### Dense Matrix Example (operator\_ex06.f90)

```
use linear_operators

implicit none

! This is the equivalent of Example 2 for LIN_SOL_SELF using operators
! and functions.

integer, parameter :: m=64, n=32
real(kind(1e0)) :: one=1e0, zero=0e0, err
real(kind(1e0)) A(n,n), b(n), C(m,n), d(m), cov(n,n), x(n)

! Generate a random rectangular matrix and right-hand side.
C = rand(C); d=rand(d)

! Form the normal equations for the rectangular system.
A = C .tx. C; b = C .tx. d
COV = .i. CHOL(A); COV = COV .xt. COV

! Compute the least-squares solution.
x = C .ix. d

! Compare with solution obtained using the inverse matrix.
err = norm(x - (COV .x. b))/norm(cov)

! Scale the inverse to obtain the sample covariance matrix.
COV = sum((d - (C .x. x))**2)/(m-n) * COV
! Check the results.
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_SOL_SELF (operators) is correct.'
end if

end
```

### Parallel Example (parallel\_ex06.f90)

```
use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 6 for box data types, operators
! and functions.

integer, parameter :: m=64, n=32, nr=4
real(kind(1e0)) :: one=1e0, zero=0e0, err(nr)
real(kind(1e0)), dimension(m,n,nr) :: C, d(m,1,nr)
real(kind(1e0)), dimension(n,n,nr) :: A, cov
real(kind(1e0)), dimension(n,1,nr) :: b, x

! Setup for MPI:
mp_nprocs=mp_setup()
```

```

! Generate a random rectangular matrix and right-hand side.
 if(mp_rank == 0) then
 C = rand(C); d=rand(d)
 endif

! Form the normal equations for the rectangular system.
 A = C .tx. C; b = C .tx. d
 COV = .i. CHOL(A); COV = COV .xt. COV

! Compute the least-squares solution.
 x = C .ix. d

! Compare with solution obtained using the inverse matrix.
 err = norm(x - (COV .x. b))/norm(cov)

! Check the results.
 if (ALL(err <= sqrt(epsilon(one))) .and. mp_rank == 0) &
 write (*,*) 'Parallel Example 6 is correct.'

! See to any error messages and quit MPI
 mp_nprocs=mp_setup('Final')

end

```

---

## COND



Computes the condition number of a matrix.

### Function Return Value

Computes condition number of matrix *A*. This is a scalar for the case where *A* is rank-2 or a sparse matrix. It is a rank-1 array when *A* is a dense rank-3 array. (Output)

### Required Argument

*A* — Matrix for which the condition number is to be computed. The matrix may be real, double, complex, double-complex, or one of the computational sparse matrix derived types, `?_hbc_sparse`. For an array of type real, double, complex, or double-complex the array may be of rank-2 or rank-3.

For a dense rank-3 array, each rank-2 array section, (for fixed third subscript), is a separate problem. In this case, the output is a rank-1 array of condition numbers for each problem. (Input)

### Optional Arguments, Packaged Options

*NORM\_CHOICE* — Integer indicating the type of norm to be used in computing the condition number.

| NORM_CHOICE | CONDITION Number | Square Matrix |        | Rectangular Matrix |        |
|-------------|------------------|---------------|--------|--------------------|--------|
|             |                  | Dense         | Sparse | Dense              | Sparse |
| 1           | $L_1$            | Yes           | Yes    | No                 | No     |
| 2 (Default) | $L_2$            | Yes           | Yes    | Yes                | No     |
| huge(1)     | $L_\infty$       | Yes           | Yes    | No                 | No     |

This function uses `LIN_SOL_SVD` (see [Chapter 1, “Linear Systems”](#)).

The option and derived type names are given in the following tables:

| Option Names for COND  | Option Value |
|------------------------|--------------|
| ?_cond_set_small       | 1            |
| ?_cond_for_lin_sol_svd | 2            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_cond_options(:)                                           | Use when setting options for calls hereafter. | ?_options    |
| ?_cond_options_once(:)                                      | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_SVD` located in [Chapter 1, “Linear Systems”](#) for the specific options for this routine.

## FORTRAN 90 Interface

`COND (A [,...])`

### Description

The mathematical definitions of the condition numbers which this routine estimates are:

$$l_1 \text{ condition number } \kappa_1(A) = \|A\|_1 \cdot \|A^{-1}\|_1$$

$$l_2 \text{ condition number } \kappa_2(A) = \|A\|_2 \cdot \|A^{-1}\|_2$$

$$l_\infty \text{ condition number } \kappa_\infty(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty$$

`COND` can be used with either dense or sparse matrices as follows:

|       | Square Matrix |        | Rectangular Matrix |        |
|-------|---------------|--------|--------------------|--------|
|       | Dense         | Sparse | Dense              | Sparse |
| $L_1$ | Yes           | Yes    | No                 | No     |
| $L_2$ | Yes           | Yes    | Yes                | No     |

|            |     |     |    |    |
|------------|-----|-----|----|----|
| $L_\infty$ | Yes | Yes | No | No |
|------------|-----|-----|----|----|

The generic function `COND` can be used with either dense or sparse square matrices. This function uses `LIN_SOL_SVD` for dense square and rectangular matrices in computing  $\kappa_2(A) = s_1 / s_n$ .

The function uses `LIN_SOL_GEN` for dense square matrices in computing  $\kappa_1(A)$  and  $\kappa_\infty(A)$ .

For sparse square matrices, the values returned for  $\kappa_1(A)$  and  $\kappa_\infty(A)$  are provided by the

[SuperLU](#) linear equation solver. The condition number  $\kappa_2(A) = s_1 / s_n$  is computed by an algorithm that first approximates  $s_1$  by computing the singular values of the  $k \times k$  bidiagonal matrix obtained using the Lanczos method found in Golub and Van Loan, Ed. 3, p. 495. Here  $k$  is set using the value `A%Options%Cond_Iteration_Max`, which has the default value of 30. The value  $s_n^{-2}$  is obtained using the power method, Golub and Van Loan, p. 330, iterating with the

inverse matrix  $(A^T A)^{-1} = A^{-1} A^{-T}$ . For complex matrices  $A^T$  is replaced by  $A^H = \bar{A}^T$ . The

dominant eigenvalue of this inverse matrix is  $s_n^{-2}$ . The number of iterations is limited by the parameter value  $k$  or relative accuracy equal to the cube root of machine epsilon. Some timing tests indicate that computing  $\kappa_2(A)$  for sparse matrices by this algorithm typically requires about twice the time as for a single linear solve using the defined operator `A .ix. b`.

For computation of  $\kappa_2(A)$  with rectangular sparse matrices one can use a dense matrix representation for the matrix. This is not recommended except for small problem sizes. For overdetermined systems of sparse least-squares equations  $Ax \cong b$  a related square system is given by

$$C \begin{bmatrix} x \\ r \end{bmatrix} \equiv \begin{bmatrix} A & I_{m \times m} \\ 0_{n \times n} & A^T \end{bmatrix} \begin{bmatrix} x \\ r \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

One can form  $C$ , which has more than twice the number of non-zeros as  $A$ . But  $C$  is still sparse. One can use the condition number of  $C$  as an estimate of the accuracy for the solution vector  $x$  and the residual vector  $r$ . Note that this version of the condition number is not the same as the  $l_2$  condition number of  $A$  but is relevant to determining the accuracy of the least-squares system.

## Examples

### Dense Matrix Example (operator\_ex02.f90)

```
use wrrrn_int
use linear_operators

integer, parameter :: N=3
real (kind(1.e0)) A(N,N)
```

```

real (kind(1.e0)) C1, C2, CINF
DATA A/2.0, 2.0, -4.0, 0.0, -1.0, 2.0, 0.0, 0.0, 5.0/
CINF = COND (A, norm_choice=huge(1))
C1 = COND (A, norm_choice=1)
C2 = COND (A)
call wrrrn ('A', A)
write (*,*) 'L1 condition number= ', C1
write (*,*) 'L2 condition number= ', C2
write (*,*) 'L infinity condition number= ', CINF

end

```

### Output

```

 A
 1 2 3
1 2.000 0.000 0.000
2 2.000 -1.000 0.000
3 -4.000 2.000 5.000

L1 condition number= 12.0
L2 condition number= 10.405088
L infinity condition number= 22.0

```

### Sparse Matrix Example

```

use wrrrn_int
use linear_operators

type (s_sparse) S
type (s_hbc_sparse) H
integer, parameter :: N=3
real (kind(1.e0)) X(N,N)
real (kind(1.e0)) C1, C2, CINF
S = s_entry (1, 1, 2.0)
S = s_entry (2, 1, 2.0)
S = s_entry (3, 1, -4.0)
S = s_entry (3, 2, 2.0)
S = s_entry (2, 2, -1.0)
S = s_entry (3, 3, 5.0)
H = S ! sparse
X = H ! dense equivalent of H

CINF = COND (H, norm_choice=huge(1))
C1 = COND (H, norm_choice=1)
C2 = COND (H)
call wrrrn ('H', X)
write (*,*) 'L1 condition number= ', C1
write (*,*) 'L2 condition number= ', C2
write (*,*) 'L infinity condition number= ', CINF

end

```

### Output

```

H

```

```

 1 2 3
1 2.000 0.000 0.000
2 2.000 -1.000 0.000
3 -4.000 2.000 5.000

```

```

L1 condition number= 12.0
L2 condition number= 10.405088
L infinity condition number= 22.0

```

### Parallel Example (parallel\_ex02.f90)

```

 use linear_operators
 use mpi_setup_int

 implicit none

! This is the equivalent of Parallel Example 2 for .i. and det() with box
! data types, operators and functions.

 integer, parameter :: n=32, nr=4
 integer J
 real(kind(1e0)) :: one=1e0
 real(kind(1e0)), dimension(nr) :: err, det_A, det_i
 real(kind(1e0)), dimension(n,n,nr) :: A, inv, R, S

! Setup for MPI.
 MP_NPROCS=MP_SETUP()
! Generate a random matrix.
 A = rand(A)
! Compute the matrix inverse and its determinant.
 inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
 det_i = det(inv)
! Check the quality of both left and right inverses.
 DO J=1,nr; R(:, :, J)=EYE(N); END DO

 S=R; R=R-(A .x. inv); S=S-(inv .x. A)
 err = (norm(R)+norm(S))/cond(A)
 if (ALL(err <= sqrt(epsilon(one)) .and. &
 abs(det_A*det_i - one) <= sqrt(epsilon(one))) &
 .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 2 is correct.'

! See to any error messages and quit MPI.
 MP_NPROCS=MP_SETUP('Final')

 end

```

---

## DET



Computes the determinant of a rectangular matrix.

## Function Return Value

Determinant of matrix *A*. This is a scalar for the case where *A* is rank 2. It is a rank-1 array of determinant values for the case where *A* is rank 3. (Output)

## Required Argument

*A* — Matrix for which the determinant is to be computed. This argument must be a rank-2 or rank-3 array that contains a rectangular matrix. It may be real, double, complex, double complex. (Input)

For rank-3 arrays, each rank-2 array (for fixed third subscript), is a separate matrix. In this case, the output is a rank-1 array of determinant values for each problem.

## Optional Arguments, Packaged Options

This function uses `LIN_SOL_LSQ` (see [Chapter 1, “Linear Systems”](#)) to compute the *QR* decomposition of *A*, and the logarithmic value of  $\det(A)$ , which is exponentiated for the result.

The option and derived type names are given in the following tables:

| Option Names for <code>DET</code>  | Option Value |
|------------------------------------|--------------|
| <code>?_det_for_lin_sol_lsq</code> | 1            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_det_options(:)</code>                               | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_det_options_once(:)</code>                          | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_LSQ` located in [Chapter 1, “Linear Systems”](#) for the specific options for this routine.

## FORTRAN 90 Interface

`DET (A)`

## Description

Computes the determinant of a rectangular matrix, *A*. The evaluation is based on the *QR* decomposition,

$$QAP = \begin{bmatrix} R_{k \times k} & 0 \\ 0 & 0 \end{bmatrix}$$

and  $k = \text{rank}(A)$ . Thus  $\det(A) = s \times \det(R)$  where  $s = \det(Q) \times \det(P) = \pm 1$ .

Even well-conditioned matrices can have determinants with values that have very large or very tiny magnitudes. The values may overflow or underflow. For this class of problems, the use of the logarithmic representation of the determinant found in `LIN_SOL_GEN` or `LIN_SOL_LSQ` is required.

## Examples

### Dense Matrix Example (operator\_ex02.f90)

```
use linear_operators
implicit none

! This is Example 2 for LIN_SOL_GEN using operators and functions.

integer, parameter :: n=32
real(kind(1e0)) :: one=1e0, err, det_A, det_i
real(kind(1e0)), dimension(n,n) :: A, inv

! Generate a random matrix.
A = rand(A)
! Compute the matrix inverse and its determinant.
inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
det_i = det(inv)
! Check the quality of both left and right inverses.
err = (norm(EYE(n)-(A .x. inv))+norm(EYE(n)-(inv.x.A)))/cond(A)
if (err <= sqrt(epsilon(one)) .and. abs(det_A*det_i - one) <= &
sqrt(epsilon(one))) &
write (*,*) 'Example 2 for LIN_SOL_GEN (operators) is correct.'
end
```

### Parallel Example (parallel\_ex02.f90)

```
use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 2 for .i. and det() with box
! data types, operators and functions.

integer, parameter :: n=32, nr=4
integer J
real(kind(1e0)) :: one=1e0
real(kind(1e0)), dimension(nr) :: err, det_A, det_i
real(kind(1e0)), dimension(n,n,nr) :: A, inv, R, S

! Setup for MPI.
MP_NPROCS=MP_SETUP()
! Generate a random matrix.
A = rand(A)
! Compute the matrix inverse and its determinant.
```

```

 inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
 det_i = det(inv)
! Check the quality of both left and right inverses.
 DO J=1,nr; R(:, :,J)=EYE(N); END DO

 S=R; R=R-(A .x. inv); S=S-(inv .x. A)
 err = (norm(R)+norm(S))/cond(A)
 if (ALL(err <= sqrt(epsilon(one)) .and. &
 abs(det_A*det_i - one) <= sqrt(epsilon(one))) &
 .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 2 is correct.'

! See to any error messages and quit MPI.
 MP_NPROCS=MP_SETUP('Final')

 end

```

---

## DIAG

Constructs a square diagonal matrix.

### Function Return Value

Square diagonal matrix of rank-2 if A is rank-1 or rank-3 array if A is rank-2. (Output)

### Required Argument

**A** — This is a rank-1 or rank-2 array of type real, double, complex, or double complex, containing the diagonal elements. The output is a rank-2 or rank-3 array, respectively. (Input)

### FORTRAN 90 Interface

DIAG (A)

### Description

Constructs a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array. The dimension of the matrix is the value of the size of the rank-1 array.

The use of `DIAG` may be obviated by observing that the defined operations  $C = \text{diag}(x) \cdot x \cdot A$  or  $D = B \cdot x \cdot \text{diag}(x)$  are respectively the array operations  $C = \text{spread}(x, \text{DIM}=1, \text{NCOPIES}=\text{size}(A, 1)) * A$ , and  $D = B * \text{spread}(x, \text{DIM}=2, \text{NCOPIES}=\text{size}(B, 2))$ . These array products are not as easy to read as the defined operations using `DIAG` and matrix multiply, but their use results in a more efficient code.

### Examples

#### Dense Matrix Example (operator\_ex13.f90)

```

 use linear_operators

```

```

implicit none

! This is the equivalent of Example 1 for LIN_SOL_SVD using operators
! and functions.
integer, parameter :: m=128, n=32
real(kind(1d0)) :: one=1d0, err
real(kind(1d0)) A(m,n), b(m), x(n), U(m,m), V(n,n), S(n), g(m)

! Generate a random matrix and right-hand side.
A = rand(A); b = rand(b)

! Compute the least-squares solution matrix of Ax=b.
S = SVD(A, U = U, V = V)
g = U .tx. b; x = V .x. diag(one/S) .x. g(1:n)

! Check the results.
err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_SOL_SVD (operators) is correct.'
end if

end

```

---

## DIAGONALS

Extracts the diagonal terms of a matrix.

### Function Return Value

Array containing the diagonal terms of matrix A. It is rank-1 or rank-2 depending on the rank of A. When A is a rank-3 array, the result is a rank-2 array consisting of each separate set of diagonals. (Output)

### Required Argument

A — Matrix from which to extract the diagonal. This is a rank-2 or rank-3 array of type real, double, complex, or double complex. The output is a rank-1 or rank-2 array, respectively. (Input)

### FORTRAN 90 Interface

DIAGONALS (A)

### Description

Extracts a rank-1 array whose values are the diagonal terms of the rank-2 array A. The size of the array is the smaller of the two dimensions of the rank-2 array.

## Examples

### Dense Matrix Example (operator\_ex32.f90)

```
use linear_operators
implicit none
! This is the equivalent of Example 4 (using operators) for LIN_EIG_GEN.

integer, parameter :: n=17
real(kind(ld0)), parameter :: one=1d0
real(kind(ld0)), dimension(n,n) :: A, C
real(kind(ld0)) variation(n), eta
complex(kind(ld0)), dimension(n,n) :: U, V, e(n), d(n)

! Generate a random matrix.
A = rand(A)

! Compute the eigenvalues, left- and right- eigenvectors.
D = EIG(A, W=V); E = EIG(.t.A, W=U)

! Compute condition numbers and variations of eigenvalues.
variation = norm(A)/abs(diagonals(U .hx. V))

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again. Check the
! differences compared to the estimates. They should not exceed
! the bounds.
eta = sqrt(epsilon(one))
C = A + eta*(2*rand(A)-1)*A
D = EIG(C)

! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
if (count(abs(d)-abs(e) > eta*variation) == 0) then
 write (*,*) 'Example 4 for LIN_EIG_GEN (operators) is correct.'
end if
end
```

---

## EIG



Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.

### Function Return Value

Rank-1 or rank-2 complex array of eigenvalues. (Output)

## Required Argument

*A* — Matrix for which the eigenexpansion is to be computed. This is a square rank-2 array or a rank-3 array with square first rank-2 sections of type single, double, complex, or double complex. (Input)

## Optional Arguments, Packaged Options

*B* — Matrix *B* for the generalized problem,  $Ax = eBx$ . *B* must be the same type as *A*. (Input)

*D* — Array containing the real diagonal matrix factors of the generalized eigenvalues. (Output)

*V* — Array of real eigenvectors for both the ordinary and generalized problem. Used only for the generalized problem when matrix *B* is singular. (Output)

*W* — Array of complex eigenvectors for both the ordinary and generalized problem. Do not use optional argument *v* when *w* is present. (Output)

This function uses [LIN\\_EIG\\_SELF](#), [LIN\\_EIG\\_GEN](#), and [LIN\\_GEIG\\_GEN](#) to compute the decompositions. See [Chapter 2, “Eigensystem Analysis”](#).

The option and derived type names are given in the following tables:

| Option Names for EIG     | Option Value |
|--------------------------|--------------|
| Options_for_lin_eig_self | 1            |
| Options_for_lin_eig_gen  | 2            |
| Options_for_lin_geig_gen | 3            |
| Skip_error_processing    | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_eig_options(:)                                            | Use when setting options for calls hereafter. | ?_options    |
| ?_eig_options_once(:)                                       | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See [LIN\\_EIG\\_SELF](#), [LIN\\_EIG\\_GEN](#), and [LIN\\_GEIG\\_GEN](#) located in [Chapter 2, “Eigensystems Analysis”](#) for the specific options for these routines.

## FORTRAN 90 Interface

EIG (A [,...])

## Description

Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.

For the ordinary eigenvalue problem,  $Ax = ex$ , the optional input "B=" is not used. With the generalized problem,  $Ax = eBx$ , the matrix  $B$  is passed as the array in the right-side of "B=". The optional output "D=" is an array required only for the generalized problem and then only when the matrix  $B$  is singular.

The array of real eigenvectors is an optional output for both the ordinary and the generalized problem. It is used as "V=" where the right-side array will contain the eigenvectors. If any eigenvectors are complex, the optional output "W=" must be present. In that case "V=" should not be used.

## Examples

### Dense Matrix Example 1 (operator\_ex26.f90)

```
use linear_operators

implicit none

! This is the equivalent of Example 2 (using operators) for LIN_EIG_SELF.

integer, parameter :: n=8
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)), dimension(n,n) :: A, d(n), v_s

! Generate a random self-adjoint matrix.
A = rand(A); A = A + .t.A

! Compute the eigenvalues and eigenvectors.
D = EIG(A,V=v_s)

! Check the results for small residuals.
if (norm((A .x. v_s) - (v_s .x. diag(D)))/abs(d(1)) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_EIG_SELF (operators) is correct.'
end if

end
```

### Dense Matrix Example 2 (operator\_ex33.f90)

```
use linear_operators

implicit none

! This is the equivalent of Example 1 (using operators) for LIN_GEIG_GEN.

integer, parameter :: n=32
real(kind(1d0)), parameter :: one=1d0
real(kind(1d0)) A(n,n), B(n,n), bta(n), beta_t(n), err
complex(kind(1d0)) alpha(n), alpha_t(n), V(n,n)
```

```

! Generate random matrices for both A and B.
 A = rand(A); B = rand(B)

! Compute the generalized eigenvalues.
 alpha = EIG(A, B=B, D=bta)

! Compute the full decomposition once again, A*V = B*V*values,
! and check for any error messages.
 alpha_t = EIG(A, B=B, D=beta_t, W = V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
 err = norm((A .x. V .x. diag(bta)) - (B .x. V .x. diag(alpha)),1)/&
 (norm(A,1)*norm(bta,1) + norm(B,1)*norm(alpha,1))
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for LIN_GEIG_GEN (operators) is correct.'
 end if

end

```

### Parallel Example (parallel\_ex04.f90)

Here an alternate node is used to compute the majority of a single application, and the user does not need to make any explicit calls to MPI routines. The time-consuming parts are the evaluation of the eigenvalue-eigenvector expansion, the solving step, and the residuals. To do this, the rank-2 arrays are changed to a box data type with a unit third dimension. This uses parallel computing. The node priority order is established by the initial function call, `MP_SETUP(n)`. The root is restricted from working on the box data type by assigning `MPI_ROOT_WORKS=.false.` This example anticipates that the most efficient node, other than the root, will perform the heavy computing. Two nodes are required to execute.

```

use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 4 for matrix exponential.
! The box dimension has a single rack.
integer, parameter :: n=32, k=128, nr=1
integer i
real(kind(1e0)), parameter :: one=1e0, t_max=one, delta_t=t_max/(k-1)
real(kind(1e0)) err(nr), sizes(nr), A(n,n,nr)
real(kind(1e0)) t(k), y(n,k,nr), y_prime(n,k,nr)
complex(kind(1e0)), dimension(n,nr) :: x(n,n,nr), z_0, &
 z_1(n,nr,nr), y_0, d

! Setup for MPI. Establish a node priority order.
! Restrict the root from significant computing.
! Illustrates using the 'best' performing node that
! is not the root for a single task.
 MP_NPROCS=MP_SETUP(n)

 MPI_ROOT_WORKS=.false.

```

```

! Generate a random coefficient matrix.
 A = rand(A)

! Compute the eigenvalue-eigenvector decomposition
! of the system coefficient matrix on an alternate node.
 D = EIG(A, W=X)

! Generate a random initial value for the ODE system.
 y_0 = rand(y_0)

! Solve complex data system that transforms the initial
! values, X z_0=y_0.

 z_1= X .ix. y_0 ; z_0(:,nr) = z_1(:,nr,nr)

! The grid of points where a solution is computed:
 t = (/ (i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
! With the eigenvalue-eigenvector decomposition AX = XD, this
! is an evaluation of EXP(A t)y_0 = y(t).
 y = X .x.exp(spread(d(:,nr),2,k)*spread(t,1,n))*spread(z_0(:,nr),2,k)

! This is y', derived by differentiating y(t).
 y_prime = X .x. &
spread(d(:,nr),2,k)*exp(spread(d(:,nr),2,k)*spread(t,1,n))* &
 spread(z_0(:,nr),2,k)

! Check results. Is y' - Ay = 0?
 err = norm(y_prime-(A .x. y))
 sizes=norm(y_prime)+norm(A)*norm(y)
 if (ALL(err <= sqrt(epsilon(one))*sizes) .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 4 is correct.'

! See to any error messages and quit MPI.
 MP_NPROCS=MP_SETUP('Final')

end

```

---

## EYE

Creates the identity matrix.

### Function Return Value

Identity matrix of size  $N \times N$  and type real . (Output)

### Required Argument

$N$  — Size of output identity matrix. (Input)

## FORTRAN 90 Interface

EYE(N)

### Description

Creates a rank-2 square array whose diagonals are all the value one. The off-diagonals all have value zero.

### Examples

#### Dense Matrix Example (operator\_ex07.f90)

```
use linear_operators

implicit none

! This is the equivalent of Example 3 (using operators) for LIN_SOL_SELF.

integer tries
integer, parameter :: m=8, n=4, k=2
integer ipivots(n+1)
real(kind(1d0)) :: one=1.0d0, err
real(kind(1d0)) a(n,n), b(n,1), c(m,n), x(n,1), &
 e(n), ATEMP(n,n)
type(d_options) :: iopti(4)

! Generate a random rectangular matrix.
C = rand(C)

! Generate a random right hand side for use in the inverse
! iteration.
b = rand(b)

! Compute the positive definite matrix.
A = C .tx. C; A = (A+.t.A)/2

! Obtain just the eigenvalues.
E = EIG(A)

! Use packaged option to reset the value of a small diagonal.
iopti(4) = 0
iopti(1) = d_options(d_lin_sol_self_set_small,&
 epsilon(one)*abs(E(1)))

! Use packaged option to save the factorization.
iopti(2) = d_lin_sol_self_save_factors

! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
iopti(3) = d_lin_sol_self_no_sing_mess

ATEMP = A
```

```

! Compute A-eigenvalue*I as the coefficient matrix.
! Use eigenvalue number k.
 A = A - e(k)*EYE(n)

 do tries=1,2
 call lin_sol_self(A, b, x, &
 pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
 iopti(4) = d_lin_sol_self_solve_A

! Reset right-hand side in the direction of the eigenvector.
 B = UNIT(x)
 end do

! Normalize the eigenvector.
 x = UNIT(x)

! Check the results.
 b=ATEMP .x. x
 err = dot_product(x(1:n,1), b(1:n,1)) - e(k)

! If any result is not accurate, quit with no printing.
 if (abs(err) <= sqrt(epsilon(one))*E(1)) then
 write (*,*) 'Example 3 for LIN_SOL_SELF (operators) is correct.'
 end if

end

```

---

## FFT

Computes the Discrete Fourier Transform of one complex sequence.

### Function Return Value

Complex array containing the Discrete Fourier Transform of  $x$ . The result is the complex array of the same shape and rank as  $x$ . (Output)

### Required Argument

$X$  — Array containing the sequence for which the transform is to be computed.  $x$  is an assumed shape complex array of rank 1, 2 or 3. If  $x$  is real or double, it is converted to complex internally prior to the computation. (Input)

### Optional Arguments, Packaged Options

**WORK** — A `COMPLEX` array of the same precision as the data. For rank-1 transforms the size of `WORK` is  $n+15$ . To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases the efficiency of the transform.

The option and derived type names are given in the following tables:

| Option Names for FFT | Option Value |
|----------------------|--------------|
| Options_for_fast_dft | 1            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_fft_options(:)                                            | Use when setting options for calls hereafter. | ?_options    |
| ?_fft_options_once(:)                                       | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `FAST_DFT` located in [Chapter 6, “Transforms”](#) for the specific options for this routine.

## FORTRAN 90 Interface

FFT (X [,...])

## Description

Computes the Discrete Fourier Transform of a complex sequence. This function uses `FAST_DFT`, `FAST_2DFT`, and `FAST_3DFT` from Chapter 6.

## Examples (operator\_ex37.f90)

```

use rand_gen_int
use fft_int
use ifft_int
use linear_operators

implicit none

! This is Example 4 for FAST_DFT (using operators).

integer j
integer, parameter :: n=40
real(kind(1e0)) :: err, one=1e0
real(kind(1e0)), dimension(n) :: a, b, c, yy(n,n)
complex(kind(1e0)), dimension(n) :: f, fa, fb

! Generate two random periodic sequences 'a' and 'b'.
a=rand(a); b=rand(b)

! Compute the convolution 'c' of 'a' and 'b'.
yy(1:,1)=b
do j=2,n
 yy(2:,j)=yy(1:n-1,j-1)
 yy(1,j)=yy(n,j-1)
end do

c=yy .x. a

```

```

! Compute f=inverse(transform(a)*transform(b)).
 fa = fft(a)
 fb = fft(b)
 f=ifft(fa*fb)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
 err = norm(c-f)/norm(c)
 if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for FAST_DFT (operators) is correct.'
 end if

end

```

---

## FFT\_BOX



Computes the Discrete Fourier Transform of several complex or real sequences.

### Function Return Value

Complex array containing the Discrete Fourier Transform of the sequences in  $x$ . If  $x$  is an assumed shape complex array of rank 2, 3 or 4, the result is a complex array of the same shape and rank consisting of the DFT for each of the last rank's indices. (Output)

### Required Argument

$X$  — Box containing the sequences for which the transform is to be computed.  $x$  is an assumed shape complex array of rank 2, 3 or 4. If  $x$  is real or double, it is converted to complex internally prior to the computation. (Input)

### Optional Arguments, Packaged Options

**WORK** — A `COMPLEX` array of the same precision as the data. For rank-1 transforms the size of `WORK` is  $n+15$ . To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases the efficiency of the transform

The option and derived type names are given in the following tables:

| Option Names for FFT | Option Value |
|----------------------|--------------|
| Options_for_fast_dft | 1            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_fft_box_options(:)                                        | Use when setting options for calls hereafter. | ?_options    |
| ?_fft_box_options_once(:)                                   | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `FAST_DFT` located in [Chapter 6, “Transforms”](#) for the specific options for this routine.

## FORTRAN 90 Interface

```
FFT_BOX (X [...])
```

### Description

Computes the Discrete Fourier Transform of a box of complex sequences. This function uses [FAST\\_DFT](#), [FAST\\_2DFT](#), and [FAST\\_3DFT](#) from Chapter 6.

### Examples

#### Parallel Example

```

use rand_gen_int
use fft_box_int
use ifft_box_int
use linear_operators
use mpi_setup_int

implicit none

! This is FFT_BOX example.

integer i,j
integer, parameter :: n=40, nr=4
real(kind(1e0)) :: err(nr), one=1e0
real(kind(1e0)) :: a(n,1,nr), b(n,nr), c(n,1,nr), yy(n,n,nr)
complex(kind(1e0)), dimension(n,nr) :: f, fa, fb, cc, aa

real(kind(1e0)), parameter :: zero_par=0.e0
real(kind(1e0)) :: dummy_par(0)
integer iseed_par
type(s_options) :: iopti_par(2)

! setup for MPI
MP_NPROCS = MP_SETUP()

! Set Random Number generator seed

iseed_par = 53976279
iopti_par(1)=s_options(s_rand_gen_generator_seed, zero_par)

```

```

 iopti_par(2)=s_options(iseed_par,zero_par)

 call rand_gen(dummy_par,iopt=iopti_par)

! Generate two random periodic sequences 'a' and 'b'.
 a=rand(a); b=rand(b)

! Compute the convolution 'c' of 'a' and 'b'.
 do i=1,nr
 aa(1:,i) = a(1:,1,i)
 yy(1:,1,i)=b(1:,i)
 do j=2,n
 yy(2:,j,i)=yy(1:n-1,j-1,i)
 yy(1,j,i)=yy(n,j-1,i)
 end do
 end do

 c=yy .x. a

! Compute f=inverse(transform(a)*transform(b)).
 fa = fft_box(aa)
 fb = fft_box(b)
 f=ifft_box(fa*fb)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
 do i=1,nr
 cc(1:,i) = c(1:,1,i)
 end do
 err = norm(cc-f)/norm(cc)
 if (ALL(err <= sqrt(epsilon(one))) .AND. MP_RANK == 0) then
 write (*,*) 'FFT_BOX is correct.'
 end if

 MP_NPROCS = MP_SETUP('Final')
 end

```

---

## IFFT

Computes the inverse of the Discrete Fourier Transform of one complex sequence.

### Function Return Value

Complex array containing the inverse of the Discrete Fourier Transform of  $x$ . The result is the complex array of the same shape and rank as  $x$ . (Output)

### Required Argument

$X$  — Array containing the sequence for which the inverse transform is to be computed.  $x$  is an assumed shape complex array of rank 1, 2 or 3. If  $x$  is real or double, it is converted to complex internally prior to the computation. (Input)

## Optional Arguments, Packaged Options

**WORK** — a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is  $n+15$ . To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform.

The option and derived type names are given in the following tables:

| Option Name for IFFT | Option Value |
|----------------------|--------------|
| options_for_fast_dft | 1            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_ifft_options(:)                                           | Use when setting options for calls hereafter. | ?_options    |
| ?_ifft_options_once(:)                                      | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See [FAST\\_DFT](#) located in Chapter 6, “Transforms” for the specific options for this routine.

## FORTRAN 90 Interface

IFFT (X [,...])

## Description

Computes the inverse of the Discrete Fourier Transform of a complex sequence. This function uses [FAST\\_DFT](#), [FAST\\_2DFT](#), and [FAST\\_3DFT](#) from Chapter 6.

## Example (operator\_ex37.f90)

```
use rand_gen_int
use fft_int
use ifft_int
use linear_operators

implicit none

! This is the equivalent of Example 4 for FAST_DFT (using operators).

integer j
integer, parameter :: n=40
real(kind(1e0)) :: err, one=1e0
real(kind(1e0)), dimension(n) :: a, b, c, yy(n,n)
complex(kind(1e0)), dimension(n) :: f, fa, fb

! Generate two random periodic sequences 'a' and 'b'.
a=rand(a); b=rand(b)
```

```

! Compute the convolution 'c' of 'a' and 'b'.
yy(1:,1)=b
do j=2,n
 yy(2:,j)=yy(1:n-1,j-1)
 yy(1,j)=yy(n,j-1)
end do

c=yy .x. a

! Compute f=inverse(transform(a)*transform(b)).
fa = fft(a)
fb = fft(b)
f=ifft(fa*fb)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
err = norm(c-f)/norm(c)
if (err <= sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for FAST_DFT (operators) is correct.'
end if

end

```

---

## IFFT\_BOX



Computes the inverse Discrete Fourier Transform of several complex or real sequences.

### Function Return Value

Complex array containing the inverse of the Discrete Fourier Transform of the sequences in  $x$ . If  $x$  is an assumed shape complex array of rank 2, 3 or 4, the result is a complex array of the same shape and rank consisting of the inverse DFT for each of the last rank's indices.  
(Output)

### Required Argument

$X$  — Box containing the sequences for which the inverse transform is to be computed.  $x$  is an assumed shape complex array of rank 2, 3 or 4. If  $x$  is real or double, it is converted to complex internally prior to the computation. (Input)

### Optional Arguments, Packaged Options

**WORK** — A `COMPLEX` array of the same precision as the data. For rank-1 transforms the size of `WORK` is  $n+15$ . To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases the efficiency of the transform.

The option and derived type names are given in the following tables:

| Option Names for IFFT | Option Value |
|-----------------------|--------------|
| Options_for_fast_dft  | 1            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_ifft_box_options(:)                                       | Use when setting options for calls hereafter. | ?_options    |
| ?_ifft_box_options_once(:)                                  | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `FAST_DFT` located in [Chapter 6, “Transforms”](#) for the specific options for this routine.

## FORTRAN 90 Interface

IFFT\_BOX (X [,...])

## Description

Computes the inverse of the Discrete Fourier Transform of a box of complex sequences. This function uses `FAST_DFT`, `FAST_2DFT`, and `FAST_3DFT` from Chapter 6.

## Parallel Example

```

use rand_gen_int
use fft_box_int
use ifft_box_int
use linear_operators
use mpi_setup_int

implicit none

! This is FFT_BOX example.

integer i,j
integer, parameter :: n=40, nr=4
real(kind(1e0)) :: err(nr), one=1e0
real(kind(1e0)) :: a(n,1,nr), b(n,nr), c(n,1,nr), yy(n,n,nr)
complex(kind(1e0)), dimension(n,nr) :: f, fa, fb, cc, aa

real(kind(1e0)), parameter :: zero_par=0.e0
real(kind(1e0)) :: dummy_par(0)
integer iseed_par
type(s_options) :: iopti_par(2)

! setup for MPI
MP_NPROCS = MP_SETUP()

```

```

! Set Random Number generator seed

iseed_par = 53976279
iopti_par(1)=s_options(s_rand_gen_generator_seed,zero_par)
iopti_par(2)=s_options(iseed_par,zero_par)

call rand_gen(dummy_par,iopt=iopti_par)

! Generate two random periodic sequences 'a' and 'b'.
a=rand(a); b=rand(b)

! Compute the convolution 'c' of 'a' and 'b'.
do i=1,nr
aa(1:,i) = a(1:,1,i)
yy(1:,1,i)=b(1:,i)
do j=2,n
yy(2:,j,i)=yy(1:n-1,j-1,i)
yy(1,j,i)=yy(n,j-1,i)
end do
end do

c=yy .x. a

! Compute f=inverse(transform(a)*transform(b)).
fa = fft_box(aa)
fb = fft_box(b)
f=ifft_box(fa*fb)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
do i=1,nr
cc(1:,i) = c(1:,1,i)
end do
err = norm(cc-f)/norm(cc)
if (ALL(err <= sqrt(epsilon(one))) .AND. MP_RANK == 0) then
write (*,*) 'FFT_BOX is correct.'
end if

MP_NPROCS = MP_SETUP('Final')
end

```

---

## isNaN

Tests for NaN.

### Function Return Value

Logical indicating whether or not A contains NaN. The output value tests `.true.` only if there is at least one NaN in the scalar or array. (Output)

### Required Argument

**A** — The argument can be a scalar or array of rank-1, rank-2 or rank-3. The values can be any of the four intrinsic floating-point types. (Input)

## FORTRAN 90 Interface

isNaN( A)

### Description

This is a generic logical function used to test scalars or arrays for occurrence of an IEEE 754 Standard format of floating point (ANSI/IEEE 1985) NaN, or not-a-number. Either *quiet* or *signaling* NaNs are detected without an exception occurring in the test itself. The individual array entries are each examined, with bit manipulation, until the first NaN is located. For non-IEEE formats, the bit pattern tested for single precision is `transfer(not(0),1)`. For double precision numbers `x`, the bit pattern tested is equivalent to assigning the integer array `i(1:2) = not(0)`, then testing this array with the bit pattern of the integer array `transfer(x,i)`. This function is likely to be required whenever there is the possibility that a subroutine blocked the output with NaNs in the presence of an error condition.

### Example

```
use isnan_int
implicit none

! This is the equivalent of Example 1 for NaN.
integer, parameter :: n=3
real(kind(1e0)) A(n,n); real(kind(1d0)) B(n,n)
real(kind(1e0)), external :: s_NaN
real(kind(1d0)), external :: d_NaN

! Assign NaNs to both A and B:
A = s_NaN(1e0); B = d_NaN(1d0)

! Check that NaNs are noted in both A and B:
if (isnan(A) .and. isnan(B)) then
 write (*,*) 'Example 1 for NaN is correct.'
end if

end
```

---

## NaN

Returns the value for NaN.

### Function Return Value

Returns, as a scalar, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN. For other floating point formats a special pattern is returned that tests `.true.` using the function `isnan`. (Output)

### Required Argument

*X* — Scalar value of the same type and precision as the desired result, NaN. This input value is used only to match the type of output. (Input)

## FORTRAN 90 Interface

NaN (A)

### Description

NaN returns, as a scalar, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN.

The bit pattern used for single precision is `transfer(not(0),1)`. For double precision, the bit pattern for single precision is replicated by assigning the temporary integer array `i(1:2) = not(0)`, and then using the double-precision bit pattern `transfer(i,x)` for the output value.

### Example

Arrays are assigned all NaN values, using single and double-precision formats. These are tested using the logical function routine, `isNaN`.

```
use isnan_int

implicit none

! This is the equivalent of Example 1 for NaN.
integer, parameter :: n=3
real(kind(1e0)) A(n,n); real(kind(1d0)) B(n,n)
real(kind(1e0)), external :: s_NaN
real(kind(1d0)), external :: d_NaN

! Assign NaNs to both A and B:
A = s_NaN(1e0); B = d_NaN(1d0)

! Check that NaNs are noted in both A and B:
if (isNaN(A) .and. isNaN(B)) then

 write (*,*) 'Example 1 for NaN is correct.'
end if

end
```

---

## NORM



Computes the norm of an array.

### Function Return Value

Norm of A. This is a scalar for the case where A is rank 1 or rank 2. For rank-3 arrays, the norms of each rank-2 array, in dimension 3, are computed. (Output)

## Required Argument

*A* — An array of rank-1, rank-2, or rank-3, containing the values for which the norm is to be computed. It may be real, double, complex, or double complex. (Input)

## Optional Arguments, Packaged Options

*TYPE* —Integer indicating the type of norm to be computed.

1 =  $l_1$

2 =  $l_2$  (default)

huge(1) =  $l_\infty$

Use of the option number `?_reset_default_norm` will switch the default from the  $l_2$  to the  $l_1$  or  $l_\infty$  norms. (Input)

The option and derived type names are given in the following tables:

| Option Names for <code>NORM</code>  | Option Value |
|-------------------------------------|--------------|
| <code>?_norm_for_lin_sol_svd</code> | 1            |
| <code>?_reset_default_norm</code>   | 2            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_norm_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_norm_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See `LIN_SOL_SVD` located in [Chapter 1, “Linear Systems”](#) for the specific options for this routine.

## FORTRAN 90 Interface

`NORM (A [,...])`

## Description

Computes the  $l_2$ ,  $l_1$  or  $l_\infty$  norm. The  $l_1$  and  $l_\infty$  norms are likely to be less expensive to compute than the  $l_2$  norm.

$$\|A\|_1 = \max_j \left( \sum_{i=1}^m |a_{ij}| \right)$$

$$\|A\|_2 = s_1 = \text{largest singular value}$$

$$\|A\|_{\infty \leftrightarrow \text{huge}(1)} = \max_i \left( \sum_{j=1}^n |a_{ij}| \right)$$

If the  $l_2$  norm is required, this function uses `LIN_SOL_SVD` (see [Chapter 1, “Linear Systems”](#)), to compute the largest singular value of  $A$ . For the other norms, Fortran 90 intrinsics are used.

## Examples

Compute three norms of an array

```
use norm_int
real (kind(1e0)) A(5), n_1, n_2, n_inf
A = rand (A)
! I1
n_1 = norm(A, TYPE=1)
write (*,*) n_1
! I2
n_2 = norm(A)
write (*,*) n_2
! I infinity
n_inf = norm(A, TYPE=huge(1))
write (*,*) n_inf
end
```

## Parallel Example (parallel\_ex14.f90)

A “Polar Decomposition” of several matrices are computed. The `box` data type and the `SVD()` function are used. Orthogonality and small residuals are checked to verify that the results are correct.

```
use linear_operators
use mpi_setup_int
implicit none

! This is Parallel Example 15 using operators and
! functions for a polar decomposition.
integer, parameter :: n=33, nr=3
real(kind(1d0)) :: one=1d0, zero=0d0
real(kind(1d0)), dimension(n,n,nr) :: A, P, Q, &
 S_D(n,nr), U_D, V_D
real(kind(1d0)) TEMP1(nr), TEMP2(nr)

! Setup for MPI:
mp_nprocs = mp_setup()

! Generate a random matrix.
if(mp_rank == 0) A = rand(A)
```

```

! Compute the singular value decomposition.
 S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
 P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
 Q = V_D .x. diag(S_D) .xt. V_D
! Check the results for orthogonality and
! small residuals.
 TEMP1 = NORM(spread(EYE(n),3,nr) - (p .xt. p))
 TEMP2 = NORM(A - (P .X. Q)) / NORM(A)
 if (ALL(TEMP1 <= sqrt(epsilon(one))) .and. &
 ALL(TEMP2 <= sqrt(epsilon(one)))) then
 if(mp_rank == 0) &
 write (*,*) 'Parallel Example 15 is correct.'
 end if

! See to any error messages and exit MPI.
 mp_nprocs = mp_setup('Final')

end

```

---

## ORTH



Orthogonalizes the columns of a matrix.

### Function Return Value

Orthogonal matrix  $Q$  from the decomposition  $A=QR$ . If  $A$  is rank-3,  $Q$  is rank-3. (Output)

### Required Argument

$A$  — Matrix  $A$  to be decomposed. Must be an array of rank-2 or rank-3 (box data) of type real, double, complex, or double complex. (Input)

### Optional Arguments, Packaged Options

$R$  — Upper-triangular or upper trapezoidal matrix  $R$ , from the QR decomposition. If this optional argument is present, the decomposition is complete. If  $A$  is rank-3,  $R$  is rank-3. (Output)

The option and derived type names are given in the following tables:

| Option Name for ORTH  | Option Value |
|-----------------------|--------------|
| Skip_error_processing | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_orth_options(:)                                           | Use when setting options for calls hereafter. | ?_options    |
| ?_orth_options_once(:)                                      | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”.

## FORTRAN 90 Interface

ORTH (A [,...])

### Description

Orthogonalizes the columns of a matrix. The decomposition  $A = QR$  is computed using a forward and backward sweep of the Modified Gram-Schmidt algorithm.

### Examples

#### (Operator\_ex19.f90)

```

use linear_operators
 use lin_sol_tri_int
 use rand_int
 use Numerical_Libraries

implicit none

! This is the equivalent of Example 3 (using operators) for LIN_SOL_TRI.

integer i, nopt
integer, parameter :: n=128, k=n/4, ncoda=1, lda=2
real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
real(kind(1e0)) A(lda,n), EVAL(k)
type(s_options) :: iopt(2)
real(kind(1e0)) d(n), b(n), d_t(2*n,k), c_t(2*n,k), perf_ratio, &
 b_t(2*n,k), y_t(2*n,k), eval_t(k), res(n,k)
logical small

! This flag is used to get the k largest eigenvalues.
small = .false.

! Generate the main diagonal and the co-diagonal of the
! tridiagonal matrix.
b=rand(b); d=rand(d)
A(1,1:)=b; A(2,1:)=d

! Use Numerical Libraries routine for the calculation of k
! largest eigenvalues.
CALL EVASB (N, K, A, LDA, NCODA, SMALL, EVAL)
EVAL_T = EVAL

```

```

! Use IMSL Fortran Numerical Library tridiagonal solver for inverse iteration
! calculation of eigenvectors.
 factorization_choice: do nopt=0,1

! Create k tridiagonal problems, one for each inverse
! iteration system.
 b_t(1:n,1:k) = spread(b,DIM=2,NCOPIES=k)
 c_t(1:n,1:k) = EOSHIFT(b_t(1:n,1:k),SHIFT=1,DIM=1)
 d_t(1:n,1:k) = spread(d,DIM=2,NCOPIES=k) - &
 spread(EVAL_T,DIM=1,NCOPIES=n)

! Start the right-hand side at random values, scaled downward
! to account for the expected 'blowup' in the solution.
 y_t=rand(y_t)

! Do two iterations for the eigenvectors.
 do i=1, 2
 y_t(1:n,1:k) = y_t(1:n,1:k)*epsilon(s_one)
 call lin_sol_tri(c_t, d_t, b_t, y_t, &
 iopt=iopt)
 iopt(nopt+1) = s_lin_sol_tri_solve_only
 end do

! Orthogonalize the eigenvectors. (This is the most
! intensive part of the computing.)
 y_t(1:n,1:k) = ORTH(y_t(1:n,1:k))

! See if the performance ratio is smaller than the value one.
! If it is not the code will re-solve the systems using Gaussian
! Elimination. This is an exceptional event. It is a necessary
! complication for achieving reliable results.

 res(1:n,1:k) = spread(d,DIM=2,NCOPIES=k)*y_t(1:n,1:k) + &
 spread(b,DIM=2,NCOPIES=k)* &
 EOSHIFT(y_t(1:n,1:k),SHIFT=-1,DIM=1) + &
 EOSHIFT(spread(b,DIM=2,NCOPIES=k)*y_t(1:n,1:k),SHIFT=1) &
 - y_t(1:n,1:k)*spread(EVAL_T(1:k),DIM=1,NCOPIES=n)

! If the factorization method is Cyclic Reduction and perf_ratio is
! larger than one, re-solve using Gaussian Elimination. If the
! method is already Gaussian Elimination, the loop exits
! and perf_ratio is checked at the end.
 perf_ratio = norm(res(1:n,1:k),1) / &
 norm(EVAL_T(1:k),1) / &
 epsilon(s_one) / (5*n)
 if (perf_ratio <= s_one) exit factorization_choice
 iopt(nopt+1) = s_lin_sol_tri_use_Gauss_elim

end do factorization_choice

if (perf_ratio <= s_one) then
 write (*,*) 'Example 3 for LIN_SOL_TRI (operators) is correct.'
end if

end

```

## Parallel Example

```
use linear_operators
use mpi_setup_int

integer, parameter :: N=32, nr=4
real (kind(1.e0)) A(N,N,nr), Q(N,N,nr)
! Setup for MPI
mp_nprocs = mp_setup()

if (mp_rank == 0) then
 A = rand(A)
end if

Q = orth(A)

mp_nprocs = mp_setup ('Final')

end
```

---

# RAND

Generates a scalar, rank-1, rank-2 or rank-3 array of random numbers.

## Function Return Value

Scalar, rank-1, rank-2 or rank-3 array of random numbers. The output function value matches the input argument *A* in type, kind and rank. For complex arguments, the output values will be real and imaginary parts with random values of the same type, kind, and rank. (Output)

## Required Argument

*A* — The argument must be a scalar, rank-1, rank-2, or rank-3 array of type single, double, complex, or double complex. Used only to determine the type and rank of the output. (Input)

## Optional Arguments, Packaged Options

Note: If any of the arrays `s_rand_options(:)`, `s_rand_options_once(:)`, `d_rand_options(:)`, or `d_rand_options_once(:)` are allocated, they are passed as arguments to `rand_gen` using the keyword “`iopt=`”.

The option and derived type names are given in the following table:

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_rand_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_rand_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

## FORTRAN 90 Interface

RAND(A)

### Description

Generates a scalar, rank-1, rank-2 or rank-3 array of random numbers. Each component number is positive and strictly less than one in value.

This function uses `rand_gen` to obtain the number of values required by the argument. The values are then copied using the `RESHAPE` intrinsic

### Example

```
use show_int
use rand_int

implicit none

! This is the equivalent of Example 1 for SHOW.

integer, parameter :: n=7, m=3
real(kind(1e0)) s_x(-1:n), s_m(m,n)
real(kind(1d0)) d_x(n), d_m(m,n)
complex(kind(1e0)) c_x(n), c_m(m,n)
complex(kind(1d0)) z_x(n), z_m(m,n)
integer i_x(n), i_m(m,n)
type (s_options) options(3)

! The data types printed are real(kind(1e0)), real(kind(1d0)),
! complex(kind(1e0)), complex(kind(1d0)), and INTEGER. Fill with random
! numbers and then print the contents, in each case with a label.
s_x=rand(s_x); s_m=rand(s_m)
d_x=rand(d_x); d_m=rand(d_m)
c_x=rand(c_x); c_m=rand(c_m)
z_x=rand(z_x); z_m=rand(z_m)
i_x=100*rand(s_x(1:n)); i_m=100*rand(s_m)

call show (s_x, 'Rank-1, REAL')
call show (s_m, 'Rank-2, REAL')
call show (d_x, 'Rank-1, DOUBLE')
call show (d_m, 'Rank-2, DOUBLE')
call show (c_x, 'Rank-1, COMPLEX')
call show (c_m, 'Rank-2, COMPLEX')
call show (z_x, 'Rank-1, DOUBLE COMPLEX')
call show (z_m, 'Rank-2, DOUBLE COMPLEX')
call show (i_x, 'Rank-1, INTEGER')
call show (i_m, 'Rank-2, INTEGER')

! Show 7 digits per number and -1 according to the
! natural or declared size of the array.
options(1)=show_significant_digits_is_7
options(2)=show_starting_index_is
options(3)= -1 ! The starting -1 value.
call show (s_x, &
```

```
'Rank-1, REAL with 7 digits, natural indexing', IOPT=options)
end
```

---

## RANK



Computes the mathematical rank of a matrix.

### Function Return Value

Integer rank of  $A$ . The output function value is an integer with a value equal to the number of singular values that are greater than a tolerance. (Output)

### Required Argument

$A$  — Matrix for which the rank is to be computed. The argument must be rank-2 or rank-3 (box) array of type single, double, complex, or double complex. (Input)

### Optional Arguments, Packaged Options

This function uses [LIN\\_SOL\\_SVD](#) to compute the singular values of the argument. The singular values are then compared with the value of the tolerance to compute the rank.

The option and derived type names are given in the following tables:

| Option Names for <code>RANK</code>  | Option Value |
|-------------------------------------|--------------|
| <code>?_rank_set_small</code>       | 1            |
| <code>?_rank_for_lin_sol_svd</code> | 2            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type           |
|-------------------------------------------------------------|-----------------------------------------------|------------------------|
| <code>?_rank_options(:)</code>                              | Use when setting options for calls hereafter. | <code>?_options</code> |
| <code>?_rank_options_once(:)</code>                         | Use when setting options for next call only.  | <code>?_options</code> |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See [LIN\\_SOL\\_SVD](#) located in [Chapter 1, “Linear Systems”](#) for the specific options for this routine.

### FORTRAN 90 Interface

RANK (A)

## Description

Computes the mathematical rank of a rank-2 or rank-3 array. The output function value is an integer with a value equal to the number of singular values that are greater than a tolerance. The default value for this tolerance is  $\varepsilon^{1/2} s_1$ , where  $\varepsilon$  is machine precision and  $s_1$  is the largest singular value of the matrix.

## Examples

```
use linear_operators
real (kind(1e0)) A(5,5)
A = rand (A)
write (*,*) rank(A)
A=1.0
write (*,*) rank(A)
end
```

## Output

```
5
1
```

## Parallel Example

```
use linear_operators
use mpi_setup_int

integer, parameter :: N=3, nr=4
integer r(nr)
real (kind(1.e0)) s_mat(N,N), s_box(N,N,nr)
! Setup for MPI
mp_nprocs = mp_setup()

if (mp_rank == 0) then
 s_mat = reshape((/1.,0.,0.,epsilon(1.0e0)/), (/n,n/))
 s_box = spread(s_mat,dim=3,ncopies=nr)
end if

r = rank(s_box)

mp_nprocs = mp_setup ('Final')

end
```

---

## SVD



Computes the singular value decomposition of a matrix,  $A = USV^T$ .

## Function Return Value

$m \times n$  diagonal matrix of singular values, S, from the  $A = USV^T$  decomposition. (Output)

## Required Argument

$A$  — Array of size  $m \times n$  to be decomposed. Must be rank-2 or rank-3 array of type single, double, complex, or double complex. (Input)

## Optional Arguments, Packaged Options

$U$  — Array of size  $m \times m$  containing the singular vectors U. (Output)

$V$  — Array of size  $n \times n$  containing the singular vectors V. (Output)

The option and derived type names are given in the following tables:

| Option Names for SVD    | Option Value |
|-------------------------|--------------|
| Options_for_lin_svd     | 1            |
| Options_for_lin_sol_svd | 2            |
| skip_error_processing   | 5            |

| Name of Unallocated Option Array to Use for Setting Options | Use                                           | Derived Type |
|-------------------------------------------------------------|-----------------------------------------------|--------------|
| ?_svd_options(:)                                            | Use when setting options for calls hereafter. | ?_options    |
| ?_svd_options_once(:)                                       | Use when setting options for next call only.  | ?_options    |

For a description on how to use these options, see “[Matrix Optional Data Changes](#)”. See [LIN\\_SVD](#) and [LIN\\_SOL\\_SVD](#) located in Chapter 1, “Linear Systems” for the specific options for these routines.

## FORTRAN 90 Interface

SVD (A [,...])

## Description

Computes the singular value decomposition of a rank-2 or rank-3 array,  $A = USV^T$ .

This function uses one of the routines [LIN\\_SVD](#) and [LIN\\_SOL\\_SVD](#). If a complete decomposition is required, [LIN\\_SVD](#) is used. If singular values only, or singular values and one of the right and left singular vectors are required, then [LIN\\_SOL\\_SVD](#) is called.

## Examples

### operator\_ex14.f90

```
use linear_operators
implicit none

! This is the equivalent of Example 2 for LIN_SOL_SVD using operators
! and functions.
integer, parameter :: n=32
real(kind(ld0)) :: one=1d0, zero=0d0
real(kind(ld0)) A(n,n), P(n,n), Q(n,n), &
 S_D(n), U_D(n,n), V_D(n,n)

! Generate a random matrix.
A = rand(A)

! Compute the singular value decomposition.
S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
Q = V_D .x. diag(S_D) .xt. V_D

! Check the results.
if (norm(EYE(n) - (P .xt. P)) &
 <= sqrt(epsilon(one))) then
 if (norm(A - (P .x. Q))/norm(A) &
 <= sqrt(epsilon(one))) then
 write (*,*) 'Example 2 for LIN_SOL_SVD (operators) is correct.'
 end if
end if
end
```

### Parallel Example (parallel\_ex14.f90)

Systems of least-squares problems are solved, but now using the `SVD()` function. A box data type is used. This is an example which uses optional arguments and a generic function overloaded for parallel execution of a box data type. Any number of nodes can be used.

```
use linear_operators
use mpi_setup_int
implicit none

! This is the equivalent of Parallel Example 14
! for SVD, .tx. , .x. and NORM.
integer, parameter :: m=128, n=32, nr=4
real(kind(ld0)) :: one=1d0, err(nr)
real(kind(ld0)) A(m,n,nr), b(m,1,nr), x(n,1,nr), U(m,m,nr), &
 V(n,n,nr), S(n,nr), g(m,1,nr)

! Setup for MPI:
mp_nprocs=mp_setup()
```

```

 if(mp_rank == 0) then
! Generate a random matrix and right-hand side.
 A = rand(A); b = rand(b)
 endif

! Compute the least-squares solution matrix of Ax=b.
 S = SVD(A, U = U, V = V)
 g = U .tx. b
 x = V .x. (diag(one/S) .x. g(1:n, :, :))

! Check the results.
 err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
 if (ALL(err <= sqrt(epsilon(one)))) then
 if(mp_rank == 0) &
 write (*,*) 'Parallel Example 14 is correct.'
 end if

! See to any error messages and quit MPI
 mp_nprocs = mp_setup('Final')

 end

```

---

## UNIT

Normalizes the columns of a matrix so each has Euclidean length of value one.

### Function Return Value

Matrix containing the normalized values of  $A$ . The output function value is an array of the same type and kind as  $A$ , where each column of each rank-2 principal section has Euclidean length of value one (Output)

### Required Argument

$A$  — Matrix to be normalized. The argument must be a rank-2 or rank-3 array of type single, double, complex, or double complex. (Input)

### FORTRAN 90 Interface

UNIT (A)

### Description

Normalizes the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.

This function uses a rank-2 Euclidean length subroutine to compute the lengths of the nonzero columns, which are then normalized to have lengths of value one. The subroutine carefully avoids overflow or damaging underflow by rescaling the sums of squares as required.

### Example (operator\_ex28.f90)

```
use linear_operators
```

```

implicit none

! This is the equivalent of Example 4 (using operators) for LIN_EIG_SELF.

integer, parameter :: n=64
real(kind(1e0)), parameter :: one=1d0
real(kind(1e0)), dimension(n,n) :: A, B, C, D(n), lambda(n), &
 S(n), vb_d, X, res

! Generate random self-adjoint matrices.
A = rand(A); A = A + .t.A
B = rand(B); B = B + .t.B

! Add a scalar matrix so B is positive definite.
B = B + norm(B)*EYE(n)

! Get the eigenvalues and eigenvectors for B.
S = EIG(B,V=vb_d)

! For full rank problems, convert to an ordinary self-adjoint
! problem. (All of these examples are full rank.)
if (S(n) > epsilon(one)) then
 D = one/sqrt(S)
 C = diag(D) .x. (vb_d .tx. A .x. vb_d) .x. diag(D)
 C = (C + .t.C)/2

! Get the eigenvalues and eigenvectors for C.
lambda = EIG(C,v=X)

! Compute and normalize the generalized eigenvectors.
X = UNIT(vb_d .x. diag(D) .x. X)
res = (A .x. X) - (B .x. X .x. diag(lambda))

! Check the results.
if(norm(res)/(norm(A)+norm(B)) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 4 for LIN_EIG_SELF (operators) is correct.'
end if

end if

end

```





# Chapter 11: Utilities

---

## Routines

|                                                                                                                                     |                                  |      |
|-------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|------|
| <b>11.1. ScaLAPACK Utilities</b>                                                                                                    |                                  |      |
| Sets up a processor grid.....                                                                                                       | <a href="#">ScaLAPACK_SETUP</a>  | 1792 |
| Calculates array dimensions for local arrays.....                                                                                   | <a href="#">ScaLAPACK_GETDIM</a> | 1794 |
| Reads matrix data from a file .....                                                                                                 | <a href="#">ScaLAPACK_READ</a>   | 1795 |
| Writes the matrix data to a file .....                                                                                              | <a href="#">ScaLAPACK_WRITE</a>  | 1797 |
| Reads matrix data from an array .....                                                                                               | <a href="#">ScaLAPACK_MAP</a>    | 1805 |
| Writes the matrix data to a global array .....                                                                                      | <a href="#">ScaLAPACK_UNMAP</a>  | 1807 |
| Exits ScaLAPACK usage.....                                                                                                          | <a href="#">ScaLAPACK_EXIT</a>   | 1809 |
| <b>11.2. Print</b>                                                                                                                  |                                  |      |
| Prints error messages .....                                                                                                         | <a href="#">ERROR_POST</a>       | 1810 |
| Prints rank-1 or rank-2 arrays of numbers.....                                                                                      | <a href="#">SHOW</a>             | 1813 |
| Real rectangular matrix<br>with integer row and column labels.....                                                                  | <a href="#">WRRRN</a>            | 1817 |
| Real rectangular matrix with given format and labels.....                                                                           | <a href="#">WRRRL</a>            | 1819 |
| Integer rectangular matrix<br>with integer row and column labels.....                                                               | <a href="#">WRIRN</a>            | 1822 |
| Integer rectangular matrix with given format and labels.....                                                                        | <a href="#">WRIRL</a>            | 1825 |
| Complex rectangular matrix<br>with row and column labels.....                                                                       | <a href="#">WRCRN</a>            | 1827 |
| Complex rectangular matrix<br>with given format and labels .....                                                                    | <a href="#">WRCRL</a>            | 1830 |
| Sets or retrieves options for printing a matrix .....                                                                               | <a href="#">WROPT</a>            | 1833 |
| Sets or retrieves page width and length .....                                                                                       | <a href="#">PGOPT</a>            | 1840 |
| <b>11.3. Permute</b>                                                                                                                |                                  |      |
| Elements of a vector .....                                                                                                          | <a href="#">PERMU</a>            | 1842 |
| Rows/columns of a matrix.....                                                                                                       | <a href="#">PERMA</a>            | 1844 |
| <b>11.4. Sort</b>                                                                                                                   |                                  |      |
| Sorts a rank-1 array of real numbers $x$ so the $y$ results<br>are algebraically nondecreasing, $y_1 \leq y_2 \leq \dots y_n$ ..... | <a href="#">SORT_REAL</a>        | 1846 |
| Real vector by algebraic value.....                                                                                                 | <a href="#">SVRGN</a>            | 1849 |
| Real vector by algebraic value<br>and permutations returned.....                                                                    | <a href="#">SVRGP</a>            | 1850 |
| Integer vector by algebraic value.....                                                                                              | <a href="#">SVIGN</a>            | 1852 |

|              |                                                                                    |                          |      |
|--------------|------------------------------------------------------------------------------------|--------------------------|------|
|              | Integer vector by algebraic value<br>and permutations returned .....               | <a href="#">SVIGP</a>    | 1853 |
|              | Real vector by absolute value .....                                                | <a href="#">SVRBN</a>    | 1855 |
|              | Real vector by absolute value<br>and permutations returned .....                   | <a href="#">SVRBP</a>    | 1856 |
|              | Integer vector by absolute value .....                                             | <a href="#">SVIBN</a>    | 1857 |
|              | Integer vector by absolute value<br>and permutations returned .....                | <a href="#">SVIBP</a>    | 1859 |
| <b>11.5.</b> | <b>Search</b>                                                                      |                          |      |
|              | Sorted real vector for a number .....                                              | <a href="#">SRCH</a>     | 1860 |
|              | Sorted integer vector for a number .....                                           | <a href="#">ISRCH</a>    | 1862 |
|              | Sorted character vector for a string .....                                         | <a href="#">SSRCH</a>    | 1864 |
| <b>11.6.</b> | <b>Character String Manipulation</b>                                               |                          |      |
|              | Gets the character corresponding to a<br>given ASCII value .....                   | <a href="#">ACHAR</a>    | 1867 |
|              | Get the integer ASCII value for a given character .....                            | <a href="#">IACHAR</a>   | 1868 |
|              | Gets upper case integer ASCII value for a character .....                          | <a href="#">ICASE</a>    | 1869 |
|              | Case-insensitive version comparing two strings .....                               | <a href="#">IICSR</a>    | 1870 |
|              | Case-insensitive version of intrinsic function INDEX .....                         | <a href="#">IIDEX</a>    | 1872 |
|              | Converts a character string with digits to an integer .....                        | <a href="#">CVTSI</a>    | 1873 |
| <b>11.7.</b> | <b>Time, Date, and Version</b>                                                     |                          |      |
|              | CPU time .....                                                                     | <a href="#">CPSEC</a>    | 1874 |
|              | Time of day .....                                                                  | <a href="#">TIMDY</a>    | 1875 |
|              | Today's date .....                                                                 | <a href="#">TDATE</a>    | 1876 |
|              | Number of days from January 1, 1900, to the given date ...                         | <a href="#">NDAYS</a>    | 1877 |
|              | Date for the number of days from January 1, 1900 .....                             | <a href="#">NDYIN</a>    | 1878 |
|              | Day of week for given date .....                                                   | <a href="#">IDYWK</a>    | 1880 |
|              | Version, system, and serial numbers .....                                          | <a href="#">VERML</a>    | 1881 |
| <b>11.8.</b> | <b>Random Number Generation</b>                                                    |                          |      |
|              | Generates a rank-1 array of random numbers .....                                   | <a href="#">RAND_GEN</a> | 1882 |
|              | Retrieves the current value of the seed .....                                      | <a href="#">RNGET</a>    | 1891 |
|              | Initializes a random seed .....                                                    | <a href="#">RNSET</a>    | 1892 |
|              | Selects the uniform (0,1) generator .....                                          | <a href="#">RNOPT</a>    | 1893 |
|              | Initializes the 32-bit Merseene Twister generator<br>using an array .....          | <a href="#">RNIN32</a>   | 1894 |
|              | Retrieves the current table used in the 32-bit<br>Mersenne Twister generator ..... | <a href="#">RNGE32</a>   | 1895 |
|              | Sets the current table used in the 32-bit<br>Mersenne Twister generator .....      | <a href="#">RNSE32</a>   | 1897 |
|              | Initializes the 32-bit Merseene Twister generator<br>using an array .....          | <a href="#">RNIN64</a>   | 1897 |
|              | Retrieves the current table used in the 64-bit<br>Mersenne Twister generator ..... | <a href="#">RNGE64</a>   | 1898 |
|              | Sets the current table used in the 64-bit<br>Mersenne Twister generator .....      | <a href="#">RNSE64</a>   | 1900 |
|              | Generates pseudorandom numbers (function form) .....                               | <a href="#">RNUNF</a>    | 1900 |

|               |                                                                              |                            |      |
|---------------|------------------------------------------------------------------------------|----------------------------|------|
|               | Generates pseudorandom numbers .....                                         | <a href="#">RNUN</a>       | 1902 |
| <b>11.9</b>   | <b>Low Discrepancy Sequences</b>                                             |                            |      |
|               | Shuffled Faure sequence initialization .....                                 | <a href="#">FAURE_INIT</a> | 1904 |
|               | Frees the structure containing information<br>about the Faure sequence ..... | <a href="#">FAURE_FREE</a> | 1905 |
|               | Computes a shuffled Faure sequence .....                                     | <a href="#">FAURE_NEXT</a> | 1905 |
| <b>11.10.</b> | <b>Options Manager</b>                                                       |                            |      |
|               | Gets and puts type <code>INTEGER</code> options .....                        | <a href="#">IUMAG</a>      | 1908 |
|               | Gets and puts type <code>REAL</code> options .....                           | <a href="#">UMAG</a>       | 1911 |
|               | Gets and puts type <code>DOUBLE PRECISION</code> options.....                | <a href="#">DUMAG</a>      | 1914 |
| <b>11.11.</b> | <b>Line Printer Graphics</b>                                                 |                            |      |
|               | Prints plot of up to 10 sets of points .....                                 | <a href="#">PLOT</a>       | 1914 |
| <b>11.12.</b> | <b>Miscellaneous</b>                                                         |                            |      |
|               | Decomposes an integer into its prime factors .....                           | <a href="#">PRIME</a>      | 1917 |
|               | Returns mathematical and physical constants .....                            | <a href="#">CONST</a>      | 1919 |
|               | Converts a quantity to different units .....                                 | <a href="#">CUNIT</a>      | 1921 |
|               | Computes $\sqrt{a^2 + b^2}$ without underflow or overflow .....              | <a href="#">HYPOT</a>      | 1925 |
|               | Initializes or finalizes MPI. ....                                           | <a href="#">MP_SETUP</a>   | 1926 |

---

## Usage Notes for ScaLAPACK Utilities




---

For a detailed description of MPI Requirements see “[Dense Matrix Parallelism Using MPI](#)” in Chapter 10 of this manual.

---

This section describes the use of *ScaLAPACK*, a suite of dense linear algebra solvers, applicable when a single problem size is large. We have integrated usage of IMSL Fortran Library with *ScaLAPACK*. However, the *ScaLAPACK* library, including libraries for *BLACS* and *PBLAS*, are not part of this Library. To use *ScaLAPACK* software, the required libraries must be installed on the user’s computer system. We adhered to the specification of Blackford, et al. (1997), but use only MPI for communication. The *ScaLAPACK* library includes certain *LAPACK* routines, Anderson, et al. (1995), redesigned for distributed memory parallel computers. It is written in a Single Program, Multiple Data (SPMD) style using explicit message passing for communication. Matrices are laid out in a two-dimensional block-cyclic decomposition. Using High Performance Fortran (HPF) directives, Koelbel, et al. (1994), and a *static*  $p \times q$  processor array, and following declaration of the array, `A(*,*)`, this is illustrated by:

```
INTEGER, PARAMETER :: N=500, P= 2, Q=3, MB=32, NB=32
!HPF$ PROCESSORS PROC(P,Q)
!HPF$ DISTRIBUTE A(cyclic(MB), cyclic(NB)) ONTO PROC
```

Our integration work provides modules that describe the interface to the *ScaLAPACK* library. We recommend that users include these modules when using *ScaLAPACK* or ancillary packages,

including *BLACS* and *PBLAS*. For the job of distributing data within a user's application to the block-cyclic decomposition required by *ScaLAPACK* solvers, we provide a utility that reads data from an external file and arranges the data within the distributed machines for a computational step. Another utility writes the results into an external file. We also provide similar utilities that map/unmap global arrays to/from local arrays. These utilities are used in our *ScaLAPACK* examples for brevity.

The data types supported for these utilities are **integer; single precision, real; double precision, real; single precision, complex; and double precision, complex**.

A *ScaLAPACK* library normally includes routines for:

- the solution of full-rank linear systems of equations,
- general and symmetric, positive-definite, banded linear systems of equations,
- general and symmetric, positive-definite, tri-diagonal, linear systems of equations,
- condition number estimation and iterative refinement for LU and Cholesky factorization,
- matrix inversion,
- full-rank linear least-squares problems,
- orthogonal and generalized orthogonal factorizations,
- orthogonal transformation routines,
- reductions to upper Hessenberg, bidiagonal and tridiagonal form,
- reduction of a symmetric-definite, generalized eigenproblem to standard form,
- the self-adjoint or Hermitian eigenproblem,
- the generalized self-adjoint or Hermitian eigenproblem, and
- the non-symmetric eigenproblem

*ScaLAPACK* routines are available in four data types: **single precision, real; double precision; real, single precision, complex**, and **double precision, complex**. At present, the non-symmetric eigenproblem is only available in single and double precision. More background information and user documentation is available on the World Wide Web at location [www.netlib.org/scalapack/slug/scalapack\\_slug.html](http://www.netlib.org/scalapack/slug/scalapack_slug.html).

For users with rank deficiency or simple constraints in their linear systems or least-squares problem, we have routines for:

- full or deficient rank least-squares problems with non-negativity constraints
- full or deficient rank least-squares problems with simple upper and lower bound constraints

These are available in two data types: **single precision, real**, and **double precision, real**, and they are not part of *ScaLAPACK*. The matrices are distributed in a general block-column layout.

We also provide generic interfaces to a number of *ScaLAPACK* routines through the standard IMSL Library routines. These are listed in [Table D](#) in the Introduction of this manual.

The global arrays which are to be distributed across the processor grid for use by the *ScaLAPACK* routines require that an *array descriptor* be defined for each of them. We use the *ScaLAPACK*

*TOOLS* routine *DESCINIT* to set up array descriptors in our examples. A typical call to *DESCINIT*:

```
CALL DESCINIT (DESCA, M, N, MB, NB, IRSRC, ICSRC, ICTXT, LLD, INFO)
```

Where the arguments in the above call are defined as follows for the matrix being described:

**DESCA** — An input integer vector of length 9 which is to contain the array descriptor information.

**M** — An input integer which indicates the row size of the global array which is being described.

**N** — An input integer which indicates the column size of the global array which is being described.

**MB** — An input integer which indicates the blocking factor used to distribute the rows of the matrix being described.

**NB** — An input integer which indicates the blocking factor used to distribute the columns of the matrix being described.

**IRSRC** — An input integer which indicates the processor grid row over which the first row of the array being described is distributed.

**ICSRC** — An input integer which indicates the processor grid column over which the first column of the array being described is distributed.

**ICTXT** — An input integer which indicates the BLACS context handle.

**LLD** — An input integer indicating the leading dimension of the local array which is to be used for storing the local blocks of the array being described

**INFO** — An output integer indicating whether or not the call was successful. *INFO* = 0 indicates a successful exit. *INFO* = -i indicates the i-th argument had an illegal value.

This call is equivalent to the following assignment statements:

```
DESCA (1) = 1 ! This is the descriptor
 ! type. In this case, 1.
DESCA (2) = ICTXT
DESCA (3) = M
DESCA (4) = N
DESCA (5) = MB
DESCA (6) = NB
DESCA (7) = IRSRC
DESCA (8) = ICSRC
DESCA (9) = LLD
```

The IMSL Library routines which interface with *ScaLAPACK* routines use *IRSRC* = 0 and *ICSRC* = 0 for the internal calls to *DESCINIT*.

## ScaLAPACK Supporting Modules

We recommend that users needing routines from *ScaLAPACK*, *PBLAS* or *BLACS*, Version 1.4, use modules that describe the interface to individual codes. This practice, including use of the declaration directive, `IMPLICIT NONE`, is a reliable way of writing *ScaLAPACK* application code, since the routines may have lengthy lists of arguments. Using the modules is helpful to avoid the mistakes such as missing arguments or mismatches involving Type, Kind or Rank (TKR). The modules are part of the Fortran Library product. There is a comprehensive module, `ScaLAPACK_Support`, that includes use of all the modules in the table below. This module decreases the number of lines of code for checking the interface, but at the cost of increasing source compilation time compared with using individual modules.

| Module Name                      | Contents of the Module                                                                                                                                                                                       |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ScaLAPACK_Support</code>   | All of the following modules                                                                                                                                                                                 |
| <code>ScaLAPACK_Int</code>       | All interfaces to <i>ScaLAPACK</i> routines                                                                                                                                                                  |
| <code>PBLAS_Int</code>           | All interfaces to parallel <i>BLAS</i> , or <i>PBLAS</i>                                                                                                                                                     |
| <code>BLACS_Int</code>           | All interfaces to basic linear algebra communication routines, or <i>BLACS</i>                                                                                                                               |
| <code>TOOLS_Int</code>           | Interfaces to ancillary routines used by <i>ScaLAPACK</i> , but not in other packages                                                                                                                        |
| <code>LAPACK_Int</code>          | All interfaces to <i>LAPACK</i> routines required by <i>ScaLAPACK</i>                                                                                                                                        |
| <code>ScaLAPACK_IO_Int</code>    | All interfaces to <code>ScaLAPACK_READ</code> , <code>ScaLAPACK_WRITE</code> utility routines. See this Chapter.                                                                                             |
| <code>MPI_Node_Int</code>        | The module holding data describing the MPI communicator, <code>MP_LIBRARY_WORLD</code> . See <a href="#">Dense Matrix Parallelism Using MPI</a> .                                                            |
| <code>GRIDINFO_Int</code>        | The module holding data describing the processor grid and information required to map the target array to the processors. See the <a href="#">Description</a> section of <code>ScaLAPACK_SETUP</code> below. |
| <code>ScaLAPACK_MAP_Int</code>   | The interface to the <code>ScaLAPACK_MAP</code> utility routines.                                                                                                                                            |
| <code>ScaLAPACK_UNMAP_Int</code> | The interface to the <code>ScaLAPACK_UNMAP</code> utility routines.                                                                                                                                          |

---

## ScaLAPACK\_SETUP



---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine sets up a processor grid and calculates default values for various entities to be used in mapping a global array to the processor grid. All processors in the *BLACS* context call the routine.

## Required Arguments

- M*** — The row dimension of the global array for which the local array dimensions are to be calculated. (Input)
- N*** — The column dimension of the global array for which the local array dimensions are to be calculated. (Input)
- NSQUARE*** — Input logical which indicates whether the block used for mapping the global array to the processor grid must be square. If the block must be square, set *NSQUARE* to *.TRUE.*, otherwise, set it to *.FALSE.* (Input)
- GRID1D*** — Input logical which indicates whether the processor grid is to be one dimensional or two dimensional. Set *GRID1D* to *.TRUE.* if the grid is to be one dimensional. Otherwise, set *GRID1D* to *.FALSE.* (Input)

## FORTRAN 90 Interface

Generic:    CALL ScaLAPACK\_SETUP (M, N, NSQUARE, GRID1D)

## Description

Subroutine *ScaLAPACK\_SETUP* creates a processor grid based on the number of processors being used and the *GRID1D* logical supplied by the user. The argument, *NSQUARE*, is supplied because some *ScaLAPACK* routines require that the row and column blocking factors be equal. *GRID1D* is supplied for those routines which require that the processor grid be one dimensional.

*ScaLAPACK\_SETUP* also establishes values for *MP\_M*, *MP\_N*, *MP\_NPROW*, *MP\_NPCOL*, *MP\_MB*, *MP\_NB*, *MP\_PIGRID*, *MP\_ICTXT*, *MP\_NSQUARE*, and *MP\_GRID1D* in the IMSL Fortran Library module *GRIDINFO\_INT*. The above entities are defined as follows:

*MP\_M* — The row dimension of the primary array which is to be distributed among the processors.

*MP\_N* — The column dimension of the primary array which is to be distributed among the processors.

*MP\_NPROW* — The number of rows in the processor grid.

*MP\_NPCOL* — The number of columns in the processor grid.

*MP\_MB* — The row blocking factor to be used in distributing the array.

*MP\_NB* — The column blocking factor to be used in distributing the array.

*MP\_PIGRID* — The pointer to the processor grid, *MP\_IGRID*.

*MP\_ICTXT* — The *BLACS* context ID associated with the processor grid.

*MP\_NSQUARE* — Logical indicating whether or not the block used for mapping the global array to the processor grid must be square.

*MP\_GRID1D* — Logical indicating whether or not the processor grid must be one dimensional.

*GRIDINFO\_INT* is used by *MPI\_SETUP\_INT* so users do not need to explicitly use *GRIDINFO\_INT* since they will be using *MPI\_SETUP\_INT* when they use *MPI*.

## Example

See [ScaLAPACK\\_WRITE](#).

---

# ScaLAPACK\_GETDIM



---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine calculates the row and column dimensions of a local distributed array based on the size of the array to be distributed and the row and column blocking factors to be used. All processors in the *BLACS* context call the routine.

## Required Arguments

*M* — The row dimension of the global array for which the local array dimensions are to be calculated. (Input)

*N* — The column dimension of the global array for which the local array dimensions are to be calculated. (Input)

*MB* — The row blocking factor to be used in distributing the array. (Input)

*NB* — The column blocking factor to be used in distributing the array. (Input)

*MXLDA* — The row dimension of the local array. (Output)

*MXCOL* — The column dimension of the local array. (Output)

## FORTRAN 90 Interface

Generic:     CALL ScaLAPACK\_GETDIM (M, N, MB, NB, MXLDA, MXCOL)

## Description

Subroutine `ScaLAPACK_GETDIM` calculates the row and column dimensions of a local array by using the ScaLAPACK utility `NUMROC`.

---

**Note** that [ScaLAPACK\\_SETUP](#) must be called prior to calling this routine because `ScaLAPACK_GETDIM` will use some of the global entities defined by `ScaLAPACK_SETUP`.

---

## Example

See [ScaLAPACK\\_WRITE](#).

---

# ScaLAPACK\_READ



---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by *ScaLAPACK* routines. This routine contains a call to a barrier routine so that if one process is writing the file and an alternate process is to read it, the results will be synchronized.

All processors in the *BLACS* context call the routine.

## Required Arguments

**File\_Name** — A character variable naming the file containing the matrix data. (Input)  
This file is opened with `STATUS="OLD."` If the name is misspelled or the file does not exist, or any access violation occurs, a `type = terminal` error message will occur. After the contents are read, the file is closed. This file is read with a loop logically equivalent to groups of reads:

```
READ() ((BUFFER(I,J), I=1,M), J=1, NB)
 or (optionally):
READ() ((BUFFER(I,J), J=1,N), I=1, MB)
```

**DESC\_A(\*)** — The nine integer parameters associated with the *ScaLAPACK* matrix descriptor. Values for `NB,MB,LDA` are contained in this array. (Input)

**A(LDA,\*)** — This is an assumed-size array, with leading dimension `LDA`, that will contain this processor's piece of the block-cyclic matrix. The data type for `A(*,*)` is any of five Fortran intrinsic types: **integer; single precision, real; double precision, real; single precision, complex; and double precision, complex.** (Output)

## Optional Arguments

**Format** — A character variable containing a format to be used for reading the file containing matrix data. If this argument is not present, an unformatted or list-directed read is used. (Input)

**iopt** — Derived type array with the same precision as the array `A(*,*)`, used for passing optional data to `ScaLAPACK_READ`. (Input)  
The options are as follows:

|                                                         |
|---------------------------------------------------------|
| <b>Packaged Options for <code>ScaLAPACK_READ</code></b> |
|---------------------------------------------------------|

| Option Prefix = ? | Option Name                 | Option Value |
|-------------------|-----------------------------|--------------|
| S_, d_            | ScaLAPACK_READ_UNIT         | 1            |
| S_, d_            | ScaLAPACK_READ_FROM_PROCESS | 2            |
| S_, d_            | ScaLAPACK_READ_BY_ROWS      | 3            |

`iopt(IO) = ScaLAPACK_READ_UNIT`

Sets the unit number to the value in `iopt(IO + 1)%idummy`. The default unit number is the value 11.

`iopt(IO) = ScaLAPACK_READ_FROM_PROCESS`

Sets the process number that reads the named file to the value in `iopt(IO + 1)%idummy`. The default process number is the value 0.

`iopt(IO) = ScaLAPACK_READ_BY_ROWS`

Read the matrix by rows from the named file. By default the matrix is read by columns.

## FORTRAN 90 Interface

Generic: `CALL ScaLAPACK_READ (File_Name, DESC_A, A [, ...])`

Specific: The specific interface names are `S_ScaLAPACK_READ` and `D_ScaLAPACK_READ`.

## Description

Subroutine `ScaLAPACK_READ` reads columns or rows of a problem matrix so that it is usable by a *ScaLAPACK* routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to place the data in the desired assumed-size arrays on the processors. The blocks of data are read, then transmitted and received. The block sizes, contained in the array descriptor, determines the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to  $\lceil M \times N / (MB \times NB) \rceil$ . A temporary local buffer is allocated for staging the matrix data. It is of size  $M$  by  $NB$ , when reading by columns, or  $N$  by  $MB$ , when reading by rows.

## Example

See [ScaLAPACK\\_WRITE](#).

---

# ScaLAPACK\_WRITE



---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine writes the matrix data to a file. The data is transmitted from the two-dimensional block-cyclic form used by *ScaLAPACK*. This routine contains a call to a barrier routine so that if one process is writing the file and an alternate process is to read it, the results will be synchronized. All processors in the *BLACS* context call the routine.

## Required Arguments

**File\_Name** — A character variable naming the file to receive the matrix data. (Input)  
This file is opened with “STATUS=“UNKNOWN.” If any access violation happens, a type = terminal error message will occur. If the file already exists it will be overwritten. After the contents are written, the file is closed. This file is written with a loop logically equivalent to groups of writes:

```
WRITE() ((BUFFER(I,J), I=1,M), J=1, NB)
or (optionally):
WRITE() ((BUFFER(I,J), J=1,N), I=1, MB)
```

**DESC\_A(\*)** — The nine integer parameters associated with the *ScaLAPACK* matrix descriptor. Values for NB, MB, LDA are contained in this array. (Input)

**A(LDA, \*)** — This is an assumed-size array, with leading dimension LDA, containing this processor’s piece of the block-cyclic matrix. The data type for A(\*, \*) is any of five Fortran intrinsic types: **integer**; **single precision, real**; **double precision, real**; **single precision, complex**; or **double precision, complex**. (Input)

## Optional Arguments

**Format** — A character variable containing a format to be used for writing the file that receives matrix data. If this argument is not present, an unformatted or list-directed write is used. (Input)

**iopt** — Derived type array with the same precision as the array A(\*, \*), used for passing optional data to *ScaLAPACK\_WRITE*. Use single precision when A(\*, \*) is type INTEGER. (Input)  
The options are as follows:

| Packaged Options for <i>ScaLAPACK_WRITE</i> |                              |              |
|---------------------------------------------|------------------------------|--------------|
| Option Prefix = ?                           | Option Name                  | Option Value |
| S_, d_                                      | ScaLAPACK_WRITE_UNIT         | 1            |
| S_, d_                                      | ScaLAPACK_WRITE_FROM_PROCESS | 2            |

| Packaged Options for ScaLAPACK_WRITE |                         |              |
|--------------------------------------|-------------------------|--------------|
| Option Prefix = ?                    | Option Name             | Option Value |
| S_, d_                               | ScaLAPACK_WRITE_BY_ROWS | 3            |

```
iopt(IO) = ScaLAPACK_WRITE_UNIT
```

Sets the unit number to the integer component of

`iopt(IO + 1)%idummy`. The default unit number is the value 11.

```
iopt(IO) = ScaLAPACK_WRITE_FROM_PROCESS
```

Sets the process number that writes the named file to the integer component of

`iopt(IO + 1)%idummy`. The default process number is the value 0.

```
iopt(IO) = ScaLAPACK_WRITE_BY_ROWS
```

Write the matrix by rows to the named file. By default the matrix is written by columns.

## FORTRAN 90 Interface

Generic:     CALL ScaLAPACK\_WRITE (File\_Name, DESC\_A, A [, ...])

Specific:    The specific interface names are S\_ScaLAPACK\_WRITE and  
              D\_ScaLAPACK\_WRITE.

## Description

Subroutine `ScaLAPACK_WRITE` writes columns or rows of a problem matrix output by a `ScaLAPACK` routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to extract the data from the assumed-size arrays on the processors. The blocks of data are transmitted and received, then written. The block sizes, contained in the array descriptor, determines the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to  $\lceil M \times N / (MB \times NB) \rceil$ . A temporary local buffer is allocated for staging the matrix data. It is of size  $M$  by  $NB$ , when writing by columns, or  $N$  by  $MB$ , when writing by rows.

## Example 1: Distributed Transpose of a Matrix, In Place

The program `SCPK_EX1` illustrates an *in-situ* transposition of a matrix. An  $m \times n$  matrix,  $A$ , is written to a file, by rows. The  $n \times m$  matrix,  $B = A^T$ , overwrites storage for  $A$ . Two temporary files are created and deleted. This algorithm for transposing a matrix is not efficient. It is used to illustrate the read and write routines and optional arguments for writing of data by matrix rows.

```
program scpk_ex1
! This is Example 1 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! It shows in-situ or in-place transposition of a
! block-cyclic matrix.
USE ScaLAPACK_SUPPORT
```

```

USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: M=6, N=6, NIN=10
INTEGER DESC_A(9), IERROR, INFO, I, J, K, L, MXLDA, MXCOL
LOGICAL :: GRID1D = .TRUE., NSQUARE = .TRUE.
real(kind(1d0)), allocatable :: A(:, :), A0(:, :)
real(kind(1d0)) ERROR
TYPE(d_OPTIONS) IOPT(1)

 MP_NPROCS=MP_SETUP()

! Set up a 1D processor grid and define its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(M, N, NSQUARE, GRID1D)
! Get the array descriptor entities MXLDA, and MXCOL
CALL SCALAPACK_GETDIM(M, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptor
CALL DESCINIT(DESC_A, M, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDA, INFO)
! Allocate space for local arrays
ALLOCATE(A0(MXLDA, MXCOL))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
 ALLOCATE(A(M, N))
! Fill array with a pattern that is easy to recognize.
 K=0
 DO
 K=K+1; IF(10**K > N) EXIT
 END DO
 DO J=1, N
 DO I=1, M
! The values will appear, as decimals I.J, where I is
! the row and J is the column.
 A(I, J)=REAL(I)+REAL(J)*10d0**(-K)
 END DO
 END DO

 OPEN(UNIT=NIN, FILE='test.dat', STATUS='UNKNOWN')
! Write the data by columns.
 DO J=1, N, MP_NB
 WRITE(NIN, *) ((A(I, L), I=1, M), L=J, min(N, J+MP_NB-1))
 END DO
 CLOSE(NIN)
 DEALLOCATE(A)
 ALLOCATE(A(N, M))
END IF

! Read the matrix into the local arrays.
CALL ScaLAPACK_READ('test.dat', DESC_A, A0)

! To transpose, write the matrix by rows as the first step.
! This requires an option since the default is to write
! by columns.

```

```

IOPT(1)=ScaLAPACK_WRITE_BY_ROWS
CALL ScaLAPACK_WRITE("TEST.DAT", DESC_A, A0, IOPT=IOPT)

! Resize the local storage
DEALLOCATE(A0)
CALL SCALAPACK_GETDIM(N, M, MP_NB, MP_MB, MXLDA, MXCOL)
! Set up the array descriptor
! Reshape the descriptor for the transpose of the matrix.
! The number of rows and columns are swapped.
CALL DESCINIT(DESC_A, N, M, MP_NB, MP_MB, 0, 0, MP_ICTXT, &
MXLDA, INFO)

ALLOCATE(A0(MXLDA, MXCOL))

! Read the transpose matrix

CALL ScaLAPACK_READ("TEST.DAT", DESC_A, A0)

IF(MP_RANK == 0) THEN

! Open the used files and delete when closed.
OPEN(UNIT=NIN, FILE='test.dat', STATUS='OLD')
CLOSE(NIN, STATUS='DELETE')
OPEN(UNIT=NIN, FILE='TEST.DAT', STATUS='OLD')
DO J=1, M, MP_MB
READ(NIN, *) ((A(I, L), I=1, N), L=J, min(M, J+MP_MB-1))
END DO
CLOSE(NIN, STATUS='DELETE')
DO I=1, N
DO J=1, M
! The values will appear, as decimals I.J, where I is the row
! and J is the column.
A(I, J)=REAL(J)+REAL(I)*10d0**(-K) - A(I, J)
END DO
END DO
ERROR=SUM(ABS(A))
END IF

! See to any error messages.
call elpop("Mp_setup")

! Check results on just one process.
IF(ERROR <= SQRT(EPSILON(ERROR)) .and. &
MP_RANK == 0) THEN
write(*, *) " Example 1 for BLACS is correct."
END IF

! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(A0)) DEALLOCATE(A0)

! Exit from using this process grid.
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

Example 1 for BLACS is correct.

## Additional Examples

### Example 2: Distributed Matrix Product with PBLAS

The program `SCPK_EX2` illustrates computation of the matrix product  $C_{m \times n} = A_{m \times k} B_{k \times n}$ . The matrices on the right-hand side are random. Three temporary files are created and deleted. *BLACS* and *PBLAS* are used. The problem size is such that the results are checked on one process.

```
program scpk_ex2
! This is Example 2 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! The product of two matrices is computed with PBLAS
! and checked for correctness.

USE ScaLAPACK_SUPPORT
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: K=32, M=33, N=34, NIN=10
INTEGER INFO, IA, JA, IB, JB, IC, JC, MXLDA, MXCOL, MXLDB, &
 MXCOLB, MXLDC, MXCOLC, IERROR, I, J, L, &
 DESC_A(9), DESC_B(9), DESC_C(9)
LOGICAL :: GRID1D = .TRUE., NSQUARE = .TRUE.

real(kind(1d0)) :: ALPHA, BETA, ERROR=1d0, SIZE_C
real(kind(1d0)), allocatable, dimension(:, :) :: A, B, C, X(:,), &
 A0, B0, C0

MP_NPROCS=MP_SETUP()

! Set up a 1D processor grid and define its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(M, N, NSQUARE, GRID1D)
! Get the array descriptor entities
CALL SCALAPACK_GETDIM(M, K, MP_MB, MP_NB, MXLDA, MXCOL)
CALL SCALAPACK_GETDIM(K, N, MP_NB, MP_MB, MXLDB, MXCOLB)
CALL SCALAPACK_GETDIM(M, N, MP_MB, MP_NB, MXLDC, MXCOLC)
! Set up the array descriptors
CALL DESCINIT(DESC_A, M, K, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDA, INFO)
CALL DESCINIT(DESC_B, K, N, MP_NB, MP_MB, 0, 0, MP_ICTXT, &
 MXLDB, INFO)
CALL DESCINIT(DESC_C, M, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDC, INFO)

ALLOCATE(A0(MXLDA, MXCOL), B0(MXLDB, MXCOLB), C0(MXLDC, MXCOLC))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
 ALLOCATE(A(M, K), B(K, N), C(M, N), X(M))
 CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(B)
```

```

OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')
! Write the data by columns.
DO J=1,K,MP_NB
 WRITE(NIN,*) ((A(I,L),I=1,M),L=J,min(K,J+MP_NB-1))
END DO
CLOSE(NIN)

OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
! Write the data by columns.
DO J=1,N,MP_NB
 WRITE(NIN,*) ((B(I,L),I=1,K),L=J,min(N,J+MP_NB-1))
END DO
CLOSE(NIN)
END IF

! Read the factors into the local arrays.
CALL ScaLAPACK_READ('Atest.dat', DESC_A, A0)
CALL ScaLAPACK_READ('Btest.dat', DESC_B, B0)

! Compute the distributed product C = A x B.
ALPHA=1d0; BETA=0d0
IA=1; JA=1; IB=1; JB=1; IC=1; JC=1
C0=0
CALL pdGEMM &
 ("No", "No", M, N, K, ALPHA, A0, IA, JA, &
 DESC_A, B0, IB, JB, DESC_B, BETA, &
 C0, IC, JC, DESC_C)

! Put the product back on the root node.
Call ScaLAPACK_WRITE('Ctest.dat', DESC_C, C0)

IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
OPEN(UNIT=NIN, FILE='Ctest.dat', STATUS='OLD')

! Read the data by columns.
DO J=1,N,MP_NB
 READ(NIN,*) ((C(I,L),I=1,M),L=J,min(N,J+MP_NB-1))
END DO

CLOSE(NIN,STATUS='DELETE')
SIZE_C=SUM(ABS(C)); C=C-matmul(A,B)
ERROR=SUM(ABS(C))/SIZE_C

! Open other temporary files and delete them.
OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
CLOSE(NIN,STATUS='DELETE')
OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
CLOSE(NIN,STATUS='DELETE')

END IF

! See to any error messages.
call elpop("Mp_Setup")

```

```

! Deallocate storage arrays and exit from BLACS.
IF (ALLOCATED(A)) DEALLOCATE(A)
IF (ALLOCATED(B)) DEALLOCATE(B)
IF (ALLOCATED(C)) DEALLOCATE(C)
IF (ALLOCATED(X)) DEALLOCATE(X)
IF (ALLOCATED(A0)) DEALLOCATE(A0)
IF (ALLOCATED(B0)) DEALLOCATE(B0)
IF (ALLOCATED(C0)) DEALLOCATE(C0)

! Check the results.
IF (ERROR <= SQRT(EPSILON(ALPHA)) .and. &
 MP_RANK == 0) THEN
 write(*,*) " Example 2 for BLACS and PBLAS is correct."
END IF

! Exit from using this process grid.
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

Example 2 for BLACS and PBLAS is correct.

### Example 3: Distributed Linear Solver with ScaLAPACK

The program `SCPK_EX3` illustrates solving a system of linear-algebraic equations,  $Ax = b$  by calling a *ScaLAPACK* routine directly. The right-hand side is produced by defining  $A$  and  $y$  to have random values. Then the matrix-vector product  $b = Ay$  is computed. The problem size is such that the residuals,  $x - y \approx 0$  are checked on one process. Three temporary files are created and deleted. *BLACS* are used to define the process grid and provide further information identifying each process. Then a *ScaLAPACK* routine is called directly to compute the approximate solution,  $x$ .

```

program scp_k_ex3
! This is Example 3 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! A linear system is solved with ScaLAPACK and checked.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE

INCLUDE "mpif.h"
INTEGER, PARAMETER :: N=9, NIN=10
INTEGER INFO, IA, JA, IB, JB, MXLDA, MXCOL, &
 IERROR, I, J, L, DESC_A(9), &
 DESC_B(9), BUFF(3), RBUF(3)

LOGICAL :: COMMUTE = .TRUE., NSQUARE = .TRUE., GRID1D = .TRUE.
INTEGER, ALLOCATABLE :: IPIV0(:)
real(kind(1d0)) :: ERROR=0d0, SIZE_Y
real(kind(1d0)), allocatable, dimension(:, :) :: A, B(:,) , &

```

```

X(:), Y(:), A0, B0

MP_NPROCS=MP_SETUP()

! Set up a 1D processor grid and define its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, NSQUARE, GRID1D)
! Get the array descriptor entities
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESC_A, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDA, INFO)
CALL DESCINIT(DESC_B, N, 1, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
MXLDA, INFO)

! Allocate local space for each array.
ALLOCATE(A0(MXLDA,MXCOL), B0(MXLDA,1), IPIV0(MXLDA+MP_MB))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
 ALLOCATE(A(N,N), B(N), X(N), Y(N))
 CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(Y)

! Compute the correct result.
 B=MATMUL(A,Y); SIZE_Y=SUM(ABS(Y))
 OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')

! Write the data by columns.
 DO J=1,N,MP_NB
 WRITE(NIN,*) ((A(I,L),I=1,N),L=J,min(N,J+MP_NB-1))
 END DO
 CLOSE(NIN)

 OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
! Write the data by columns.
 WRITE(NIN,*) (B(I),I=1,N)
 CLOSE(NIN)
END IF

! Read the factors into the local arrays.
CALL ScaLAPACK_READ('Atest.dat', DESC_A, A0)
CALL ScaLAPACK_READ('Btest.dat', DESC_B, B0)

! Compute the distributed product solution to $A x = b$.
IA=1; JA=1; IB=1; JB=1

CALL pdGESV (N, 1, A0, IA, JA, DESC_A, IPIV0, &
B0, IB, JB, DESC_B, INFO)

! Put the result on the root node.
Call ScaLAPACK_WRITE('Xtest.dat', DESC_B, B0)

IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
 OPEN(UNIT=NIN, FILE='Xtest.dat', STATUS='OLD')

```

```

! Read the approximate solution data.
 READ(NIN,*) X
 B=X-Y

 CLOSE(NIN,STATUS='DELETE')
 ERROR=SUM(ABS(B))/SIZE_Y

! Delete temporary files.
 OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
 CLOSE(NIN,STATUS='DELETE')
 OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
 CLOSE(NIN,STATUS='DELETE')

END IF

! See to any error messages.
call elpop("Mp_Setup")

! Deallocate storage arrays
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(B)) DEALLOCATE(B)
IF(ALLOCATED(X)) DEALLOCATE(X)
IF(ALLOCATED(Y)) DEALLOCATE(Y)
IF(ALLOCATED(A0)) DEALLOCATE(A0)
IF(ALLOCATED(B0)) DEALLOCATE(B0)
IF(ALLOCATED(IPIV0)) DEALLOCATE(IPIV0)

IF(ERROR <= SQRT(EPSILON(ERROR)) .and. MP_RANK == 0) THEN
 write(*,*) &
 " Example 3 for BLACS and ScaLAPACK solver is correct."
END IF

! Exit from using this process grid.
CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

Example 3 for BLACS and ScaLAPACK is correct.

---

## ScaLAPACK\_MAP




---

For a detailed description of MPI Requirements see [“Using ScaLAPACK Enhanced Routines”](#) in the Introduction of this manual.

---

This routine maps array data from a global array to local arrays in the two-dimensional block-cyclic form required by *ScaLAPACK* routines.

All processors in the *BLACS* context call the routine.

## Required Arguments

- A* — Global rank-1 or rank-2 array which is to be mapped to the processor grid. The data type for *A* is any of five Fortran intrinsic types: **integer; single precision, real; double precision, real; single precision, complex; double precision, complex**. Normally, the user defines *A* to be valid only on the *MP\_RANK* = 0 processor. (Input)
- DESC\_A* — An integer vector containing the nine parameters associated with the *ScaLAPACK* matrix descriptor for array *A*. See “[Usage Notes for ScaLAPACK Utilities](#)” for a description of the nine parameters. (Input)
- A0* — This is a local rank-1 or rank-2 array that will contain this processor’s piece of the block-cyclic array. The data type for *A0* is any of five Fortran intrinsic types: **integer; single precision, real; double precision, real; single precision, complex; and double precision, complex**. (Output)

## Optional Arguments

- LDA* — Leading dimension of *A* as specified in the calling program. If this argument is not present, *SIZE(A, 1)* is used. (Input)
- COLMAP* — Input logical which indicates whether the global array should be mapped in column major form or row major form. *COLMAP* set to *.TRUE.* will result in the array being mapped in column- major form while setting *COLMAP* to *.FALSE.* will result in the array being mapped in row major form. The default value of *COLMAP* is *.TRUE.* (Input)

## FORTRAN 90 Interface

Generic:     CALL *ScaLAPACK\_MAP* (*A*, *DESC\_A*, *A0* [, ...])

## Description

Subroutine *ScaLAPACK\_MAP* maps columns or rows of a global array on *MP\_RANK* = 0 to local distributed arrays so that the problem array is usable by a *ScaLAPACK* routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to place the data in the desired assumed-size arrays on the processors. The block sizes, contained in the array descriptor, determine the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to  $\lceil M \times N / (MB \times NB) \rceil$ . A temporary local buffer is allocated for staging the array data. It is of size *M* by *NB*, when mapping by columns, or *N* by *MB*, when mapping by rows.

## Example

See [ScaLAPACK\\_UNMAP](#).

---

# ScaLAPACK\_UNMAP



---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine unmaps array data from local distributed arrays to a global array. The data in the local arrays must have been stored in the two-dimensional block-cyclic form required by *ScaLAPACK* routines. All processors in the *BLACS* context call the routine.

## Required Arguments

*A0* — This is a local rank-1 or rank-2 array that contains this processor’s piece of the block-cyclic array. The data type for *A0* is any of five Fortran intrinsic types: **integer; single precision, real; double precision, real; single precision, complex; or double precision, complex**. (Input)

*DESC\_A* — An integer vector containing the nine parameters associated with the *ScaLAPACK* matrix descriptor for array *A*. See “[Usage Notes for ScaLAPACK Utilities](#)” for a description of the nine parameters. (Input)

*A* — Global rank-1 or rank-2 array which is to receive the array which had been mapped to the processor grid. The data type for *A* is any of five Fortran intrinsic types: **integer; single precision, real; double precision, real; single precision, complex; or double precision, complex**. *A* is only valid on  $MP\_RANK = 0$  after *ScaLAPACK\_UNMAP* has been called. (Output)

## Optional Arguments

*LDA* — Leading dimension of *A* as specified in the calling program. If this argument is not present,  $SIZE(A, 1)$  is used. (Input)

*COLMAP* — Input logical which indicates whether the global array should be mapped in column major form or row major form. *COLMAP* set to *.TRUE.* will result in the array being mapped in column major form while setting *COLMAP* to *.FALSE.* will result in the array being mapped in row major form. The default value of *COLMAP* is *.TRUE.* (Input)

## FORTRAN 90 Interface

Generic:    CALL ScaLAPACK\_UNMAP (A0, DESC\_A, A [, ...])

## Description

Subroutine *ScaLAPACK\_UNMAP* unmaps columns or rows of local distributed arrays to a global array on  $MP\_RANK = 0$ . It uses the two-dimensional block-cyclic array descriptor for the matrix

to retrieve the data from the assumed-size arrays on the processors. The block sizes, contained in the array descriptor, determine the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to  $\lceil M \times N / (MB \times NB) \rceil$ . A temporary local buffer is allocated for staging the array data. It is of size  $M$  by  $NB$ , when mapping by columns, or  $N$  by  $MB$ , when mapping by rows.

### Example: Distributed Linear Solver with IMSL ScaLAPACK Interface

The program `SCPKMP_EX1` illustrates solving a system of linear-algebraic equations,  $Ax = b$  by calling routine `LSLRG`, an IMSL routine which interfaces with a *ScaLAPACK* routine. The right-hand side is produced by defining  $A$  and  $y$  to have random values. Then the matrix-vector product  $b = Ay$  is computed. The problem size is such that the residuals,  $x - y \approx 0$  are checked on `MP_RANK = 0`. IMSL routine `ScaLAPACK_SETUP` is called to define the process grid and provide further information identifying each process. IMSL routine `ScaLAPACK_MAP` is called to map the global arrays to local distributed arrays. Then `LSLRG` is called to compute the approximate solution,  $x$ .

```

program scpkmp_ex1
! This is Example 1 for ScaLAPACK_MAP and ScaLAPACK_UNMAP.
! A linear system is solved with an IMSL routine which
! interfaces with ScaLAPACK and is checked.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT
USE LSLRG_INT

IMPLICIT NONE

INCLUDE "mpif.h"
INTEGER, PARAMETER :: N=9
INTEGER MXLDA, MXCOL, INFO, DESC_A(9), DESC_X(9)

LOGICAL :: GRID1D = .TRUE., NSQUARE = .TRUE.
real(kind(ld0)) :: ERROR=0d0, SIZE_Y
real(kind(ld0)), allocatable, dimension(:, :) :: A, B(:,) , &
 X(:,), Y(:,), A0, B0(:,), X0(:)

 MP_NPROCS=MP_SETUP()

! Set up a 1D processor grid and define its context ID, MP_ICTXT
CALL SCALAPACK_SETUP(N, N, NSQUARE, GRID1D)
! Get the array descriptor entities MXLDA, and MXCOL
CALL SCALAPACK_GETDIM(N, N, MP_MB, MP_NB, MXLDA, MXCOL)
! Set up the array descriptors
CALL DESCINIT(DESC_A, N, N, MP_MB, MP_NB, 0, 0, MP_ICTXT, &
 MXLDA, INFO)
CALL DESCINIT(DESC_X, N, 1, MP_MB, 1, 0, 0, MP_ICTXT, &
 MXLDA, INFO)
! Allocate space for local arrays
ALLOCATE(A0(MXLDA, MXCOL), B0(MXLDA), X0(MXLDA))

! A root process is used to create the matrix data for the test.

```

```

IF(MP_RANK == 0) THEN
 ALLOCATE(A(N,N), B(N), X(N), Y(N))
 CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(Y)

! Compute the correct result.
 B=MATMUL(A,Y); SIZE_Y=SUM(ABS(Y))
END IF

! Map the input arrays to the processor grid
 CALL SCALAPACK_MAP(A, DESC_A, A0)
 CALL SCALAPACK_MAP(B, DESC_X, B0)

! Compute the distributed product solution to A x = b.
 CALL LSLRG(A0, B0, X0)

! Put the result on the root node.
 Call ScaLAPACK_UNMAP(X0, DESC_X, X)

IF(MP_RANK == 0) THEN
! Check the residuals for size.
 B=X-Y
 ERROR=SUM(ABS(B))/SIZE_Y
END IF
! See to any error messages.
 call elpop("Mp_Setup")
IF(ERROR <= SQRT(EPSILON(ERROR)) .and. MP_RANK == 0) THEN
 write(*,*) &
 " Example 1 for ScaLAPACK_MAP and ScaLAPACK_UNMAP is correct."
END IF

! Deallocate storage arrays.
 IF (MP_RANK == 0) DEALLOCATE(A, B, X, Y)
 DEALLOCATE(A0, B0, X0)

! Exit from using this process grid.
 CALL SCALAPACK_EXIT(MP_ICTXT)
! Shut down MPI
 MP_NPROCS = MP_SETUP('FINAL')
END

```

## Output

Example 1 for ScaLAPACK\_MAP and ScaLAPACK\_UNMAP is correct.

---

## ScaLAPACK\_EXIT




---

For a detailed description of MPI Requirements see “[Using ScaLAPACK Enhanced Routines](#)” in the Introduction of this manual.

---

This routine exits ScaLAPACK mode for the IMSL Library routines. All processors in the *BLACS* context call the routine.

## Required Arguments

*ICTXT* — The BLACS context ID to which the processor grid is associated. (Input)

## FORTRAN 90 Interface

Generic:     CALL ScaLAPACK\_EXIT (ICTXT)

## Description

Subroutine *ScaLAPACK\_EXIT* exits ScaLAPACK mode for the IMSL Library routines. The following actions occur when this routine is called:

- *BLACS\_GRIDEXIT* is called with the input BLACS context ID.
- The pointer to the grid ID, *MP\_PIGRID* is nullified.
- If the grid, *MP\_IGRID*, has been allocated, it is deallocated.
- *MP\_ICTXT* is reset to its default value, *HUGE(1)*.

---

# ERROR\_POST

Prints error messages that are generated by IMSL routines using *EPACK*.

## Required Argument

*EPACK* — (Input [/Output])

Derived type array of size *p* containing the array of message numbers and associated data for the messages. The definition of this derived type is packaged within the modules used as interfaces for each suite of routines. The declaration is:

```
type ?_error
 integer idummy; real(kind(?_)) rdummy
end type
```

The choice of “?” is either “s\_” or “d\_” depending on the accuracy of the data. This array gets additional messages and data from each routine that uses the “*epack=*” optional argument, provided *p* is large enough to hold data for a new message. The value *p* = 8 is sufficient to hold the longest single *terminal*, *fatal*, or *warning* message that an IMSL Fortran Library routine generates.

The location at entry *epack(1)%idummy* contains the number of data items for all messages. When the *error\_post* routine exits, this value is set to zero. Locations in array positions (2:) *%idummy* contain groups of integers consisting of a message number, the *error severity level*, then the required integer data for the message. Floating-point data, if required in the message, is passed in locations(:)%*rdummy*

matched with the starting point for integer data. The extent of the data for each message is determined by the requirements of the larger of each group of integer or floating-point values.

## Optional Arguments

`new_unit = nunit` (Input)

Unit number, of type integer, associated for reading the direct-access file of error messages for the IMSL Fortran 90 routines.

Default: `nunit = 4`

`new_path = path` (Input)

Pathname in the local file space, of type character\*64, needed for reading the direct-access file of error messages. Default string for `path` is defined during the installation procedure for certain IMSL Fortran Library routines.

## FORTRAN 90 Interface

Generic: `CALL ERROR_POST (EPACK [, ...])`

Specific: The specific interface names are `S_ERROR_POST` and `D_ERROR_POST`.

## Description

A default direct-access error message file (.daf file) is supplied with this product. This file is read by `error_post` using the contents of the derived type argument `epack`, containing the message number, error severity level, and associated data. The message is converted into character strings accepted by the error processor and then printed. The number of pending messages that print depends on the settings of the parameters `PRINT` and `STOP` in the [Reference Material](#) in the IMSL MATH/LIBRARY User's Manual. These values are initialized to defaults such that any Level 5 or Level 4 message causes a `STOP` within the error processor after a print of the text. To change these defaults so that more than one error message prints, use the routine `ERSET` documented and illustrated with examples in the [Reference Material](#) in the IMSL MATH/LIBRARY User's Manual. The method of using a message file to store the messages is required to support "shared-memory parallelism."

## Managing the Message File

For most applications of this product, there will be no need to manage this file. However, there are a few situations which may require changing or adding messages:

- New system-wide messages have been developed for applications using this Library.
- All or some of the existing messages need to be translated to another language
- A subset of users need to add a specific message file for their applications using this Library.

Following is information on changing the contents of the message file, and information on how to create and access a message file for a private application.

## Changing Messages

In order to change messages, two files are required:

- An editable message glossary, `messages.gls`, supplied with this product.
- A source program, `prepmess.f`, used to generate an executable which builds `messages.daf` from `messages.gls`.

To change messages, first make a backup copy of `messages.gls`. Use a text editor to edit `messages.gls`. The format of this file is a series of pairs of statements:

- `message_number=<nnnn>`
- `message='message string'`

(Note that neither of these lines should begin with a tab.)

The variable `<nnnn>` is an integer message number (see below for ranges and reserved message numbers).

The `'message string'` is any valid message string not to exceed 255 characters. If a message line is too long for a screen, the standard Fortran 90 concatenation operator `//` with the line continuation character `&` may be used to wrap the text.

Most strings have substitution parameters embedded within them. These may be in the following forms:

- `%(<i>n>)` for an integer substitution, where `n` is the `n`th integer output in this message.
- `%(<r>n>)` for single precision real number substitution, where `n` is the `n`th real number output in this message.
- `%(<d>n>)` for double precision real number substitution, where `n` is the `n`th double precision number output in this message.

New messages added to the system-wide error message file should be placed at the end of the file. Message numbers 5000 through 10000 have been reserved for user-added messages. Currently, messages 1 through 1400 are used by IMSL. Gaps in message number ranges are permitted; however, the message numbers must be in ascending order within the file. The message numbers used for each IMSL Fortran Library subroutine are documented in this manual and in online help.

If existing messages are being edited or translated, make sure not to alter the `message_number` lines. (This prevents conflicts with any new `messages.gls` file supplied with future versions of this Library.)

## Building a New Direct-access Message File

The `prepmess` executable must be available to complete the message changing process. For information on building the `prepmess` executable from `prepmess.f`, consult the installation guide for this product.

Once new messages have been placed in the `messages.gls` file, make a backup copy of the `messages.daf` file. Then remove `messages.daf` from the current directory. Now enter the following command:

```
prepmess > prepmess_output
```

A new `messages.daf` file is created. Edit the `prepmess_output` file and look near the end of the file for the new error messages. The `prepmess` program processes each message through the error message system as a validity check. There should be no `FATAL` error announcement within the `prepmess_output` file.

### Private Message Files

Users can create a private message file within their own messages. This file would generally be used by an application that calls this Library. Follow the steps outlined above to create a private `messages.gls` file. The user should then be given a copy of the `prepmess` executable. In the application code, call the `error_post` subprogram with the `new_unit/new_path` optional arguments. The new path should point to the directory in which the private `messages.daf` file resides.

## SHOW

Prints rank-1 or rank-2 arrays of numbers in a readable format.

### Required Arguments

*X* — Rank-1 or rank-2 array containing the numbers to be printed. (Input)

### Optional Arguments

`text` = CHARACTER (Input)

CHARACTER(LEN=\*) string used for labeling the array.

`image` = buffer (Output)

CHARACTER(LEN=\*) string used for an internal write buffer. With this argument present the output is converted to characters and packed. The lines are separated by an end-of-line sequence. The length of `buffer` is estimated by the line width in effect, time the number of lines for the array.

`iopt` = `iopt(:)` (Input)

Derived type array with the same precision as the input array; used for passing optional data to the routine. Use the `REAL(KIND(1E0))` precision for output of `INTEGER` arrays. The options are as follows:

| Packaged Options for <code>SHOW</code> |                                            |              |
|----------------------------------------|--------------------------------------------|--------------|
| Prefix is blank                        | Option Name                                | Option Value |
|                                        | <code>show_significant_digits_is_4</code>  | 1            |
|                                        | <code>show_significant_digits_is_7</code>  | 2            |
|                                        | <code>show_significant_digits_is_16</code> | 3            |
|                                        | <code>show_line_width_is_44</code>         | 4            |
|                                        | <code>show_line_width_is_72</code>         | 5            |
|                                        | <code>show_line_width_is_128</code>        | 6            |
|                                        | <code>show_end_of_line_sequence_is</code>  | 7            |

| Packaged Options for <code>SHOW</code> |                                         |    |
|----------------------------------------|-----------------------------------------|----|
|                                        | <code>show_starting_index_is</code>     | 8  |
|                                        | <code>show_starting_row_index_is</code> | 9  |
|                                        | <code>show_starting_col_index_is</code> | 10 |

```
iopt(IO) = show_significant_digits_is_4
```

```
iopt(IO) = show_significant_digits_is_7
```

```
iopt(IO) = show_significant_digits_is_16
```

These options allow more precision to be displayed. The default is 4D for each value. The other possible choices display 7D or 16D.

```
iopt(IO) = show_line_width_is_44
```

```
iopt(IO) = show_line_width_is_72
```

```
iopt(IO) = show_line_width_is_128
```

These options allow varying the output line width. The default is 72 characters per line. This allows output on many work stations or terminals to be read without wrapping of lines.

```
iopt(IO) = show_end-of_line_sequence_is
```

The sequence of characters ending a line when it is placed into the internal character buffer corresponding to the optional argument `'IMAGE = buffer'`. The value of `iopt(IO+1)%idummy` is the number of characters. These are followed, starting at `iopt(IO+2)%idummy`, by the *ASCII* codes of the characters themselves. The default is the single character, *ASCII* value 10 or *New Line*.

```
iopt(IO) = show_starting_index_is
```

This are used to reset the starting index for a rank-1 array to a value different from the default value, which is 1.

```
iopt(IO) = show_starting_row_index_is
```

```
iopt(IO) = show_starting_col_index_is
```

These are used to reset the starting row and column indices to values different from their defaults, each 1.

## FORTRAN 90 Interface

Generic: `CALL SHOW (X [, ...])`

Specific: The specific interface names are `S_SHOW` and `D_SHOW`.

## Description

The `show` routine is a generic subroutine interface to separate low-level subroutines for each data type and array shape. Output is directed to the unit number `IUNIT`. That number is obtained with the subroutine `UMACH`, *IMSL MATH/LIBRARY User's Manual*. Thus the user must open this unit in the calling program if it desired to be different from the standard output unit. If the optional argument `'IMAGE = buffer'` is present, the output is not sent to a file but to a character string within `buffer`. These characters are available to output or be used in the application.

## Fatal and Terminal Error Messages

See the `messages.gls` file for error messages for `SHOW`. These error messages are numbered 601–606; 611–617; 621–627; 631–636; 641–646.

## Example 1: Printing an Array

Array of random numbers for all the intrinsic data types are printed. For `REAL(KIND(1E0))` rank-1 arrays, the number of displayed digits is reset from the default value of 4 to the value 7 and the subscripts for the array are reset so they match their declared extent when printed. The output is not shown.

```
use show_int
use rand_int

implicit none

! This is Example 1 for SHOW.

integer, parameter :: n=7, m=3
real(kind(1e0)) s_x(-1:n), s_m(m,n)
real(kind(1d0)) d_x(n), d_m(m,n)
complex(kind(1e0)) c_x(n), c_m(m,n)
complex(kind(1d0)) z_x(n), z_m(m,n)
integer i_x(n), i_m(m,n)
type (s_options) options(3)

! The data types printed are real(kind(1e0)), real(kind(1d0)),
! complex(kind(1e0)), complex(kind(1d0)), and INTEGER.
! Fill with random numbers and then print the contents,
! in each case with a label.
s_x=rand(s_x); s_m=rand(s_m)
d_x=rand(d_x); d_m=rand(d_m)
c_x=rand(c_x); c_m=rand(c_m)
z_x=rand(z_x); z_m=rand(z_m)
i_x=100*rand(s_x(1:n)); i_m=100*rand(s_m)

call show (s_x, 'Rank-1, REAL')
call show (s_m, 'Rank-2, REAL')
call show (d_x, 'Rank-1, DOUBLE')
call show (d_m, 'Rank-2, DOUBLE')
call show (c_x, 'Rank-1, COMPLEX')
call show (c_m, 'Rank-2, COMPLEX')
call show (z_x, 'Rank-1, DOUBLE COMPLEX')
call show (z_m, 'Rank-2, DOUBLE COMPLEX')
call show (i_x, 'Rank-1, INTEGER')
```

```

 call show (i_m, 'Rank-2, INTEGER')

! Show 7 digits per number and according to the
! natural or declared size of the array.
 options(1)=show_significant_digits_is_7
 options(2)=show_starting_index_is
 options(3)= -1 ! The starting value.
 call show (s_x, &
'Rank-1, REAL with 7 digits, natural indexing', IOPT=options)
end

```

## Output

Example 1 for SHOW is correct.

## Additional Examples

### Example 2: Writing an Array to a Character Variable

This example prepares a rank-1 array for further processing, in this case delayed writing to the standard output unit. The indices and the amount of precision are reset from their defaults, as in Example 1. An end-of-line sequence of the characters CR-NL (*ASCII* 10,13) is used in place of the standard *ASCII* 10. This is not required for writing this array, but is included for an illustration of the option.

```

 use show_int
 use rand_int

 implicit none

! This is Example 2 for SHOW.
 integer, parameter :: n=7
 real(kind(1e0)) s_x(-1:n)
 type (s_options) options(7)
 CHARACTER (LEN=(72+2)*4) BUFFER
! The data types printed are real(kind(1e0)) random numbers.
 s_x=rand(s_x)

! Show 7 digits per number and according to the
! natural or declared size of the array.
! Prepare the output lines in array BUFFER.
! End each line with ASCII sequence CR-NL.
 options(1)=show_significant_digits_is_7

 options(2)=show_starting_index_is
 options(3)= -1 ! The starting value.

 options(4)=show_end_of_line_sequence_is
 options(5)= 2 ! Use 2 EOL characters.
 options(6)= 10 ! The ASCII code for CR.
 options(7)= 13 ! The ASCII code for NL.

 BUFFER= ' ' ! Blank out the buffer.

```

```

! Prepare the output in BUFFER.
call show (s_x, &
'Rank-1, REAL with 7 digits, natural indexing '//&
'internal BUFFER, CR-NL EOLs.', &
IMAGE=BUFFER, IOPT=options)

! Display BUFFER as a CHARACTER array. Discard blanks
! on the ends.
 WRITE(*, '(1x,A)') TRIM(BUFFER)

 end

```

## Output

Example 2 for SHOW is correct.

# WRRRN

Prints a real rectangular matrix with integer row and column labels.

## Required Arguments

**TITLE** — Character string specifying the title. (Input)  
 TITLE set equal to a blank character(s) suppresses printing of the title. Use “%/” within the title to create a new line. Long titles are automatically wrapped.

**A** — NRA by NCA matrix to be printed. (Input)

## Optional Arguments

**NRA** — Number of rows. (Input)  
 Default: NRA = SIZE (A, 1).

**NCA** — Number of columns. (Input)  
 Default: NCA = SIZE (A, 2).

**LDA** — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)  
 Default: LDA = SIZE (A, 1).

**ITRING** — Triangle option. (Input)  
 Default: ITRING = 0.

### ITRING Action

- |   |                                                             |
|---|-------------------------------------------------------------|
| 0 | Full matrix is printed.                                     |
| 1 | Upper triangle of A is printed, including the diagonal.     |
| 2 | Upper triangle of A excluding the diagonal of A is printed. |

- 1 Lower triangle of A is printed, including the diagonal.
- 2 Lower triangle of A excluding the diagonal of A is printed.

### **FORTRAN 90 Interface**

Generic: `CALL WRRRN (TITLE, A [, ...])`

Specific: The specific interface names are `S_WRRRN` and `D_WRRRN` for two dimensional arrays, and `S_WRRRN1D` and `D_WRRRN1D` for one dimensional arrays.

### **FORTRAN 77 Interface**

Single: `CALL WRRRN (TITLE, NRA, NCA, A, LDA, ITRING)`

Double: The double precision name is `DWRRRN`.

### **Description**

Routine `WRRRN` prints a real rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. `WRRRN` can restrict printing to the elements of the upper or lower triangles of matrices via the `ITRING` option. Generally, `ITRING ≠ 0` is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set `NRA` to the length of the array and set `NCA = 1`. For a row vector, set `NRA = 1` and set `NCA` to the length of the array. In both cases, set `LDA = NRA` and set `ITRING = 0`.

### **Comments**

1. A single D, E, or F format is chosen automatically in order to print 4 significant digits for the largest element of A in absolute value. Routine `WROPT` can be used to change the default format.
2. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking `WROPT`.
3. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT`.
4. Output is written to the unit specified by `UMACH` (see the [Reference Material](#)).

### **Example**

The following example prints all of a  $3 \times 4$  matrix A where  $a_{ij} = i + j/10$ .

```
USE WRRRN_INT

IMPLICIT NONE
INTEGER ITRING, LDA, NCA, NRA
```

```

PARAMETER (ITRING=0, LDA=10, NCA=4, NRA=3)
!
INTEGER I, J
REAL A(LDA,NCA)
!
DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = I + J*0.1
10 CONTINUE
20 CONTINUE
!
 Write A matrix.
CALL WRRRN ('A', A, NRA=NRA)
END

```

## Output

```

 A
 1 2 3 4
1 1.100 1.200 1.300 1.400
2 2.100 2.200 2.300 2.400
3 3.100 3.200 3.300 3.400

```

---

## WRRRL

Print a real rectangular matrix with a given format and labels.

### Required Arguments

**TITLE** — Character string specifying the title. (Input)  
**TITLE** set equal to a blank character(s) suppresses printing of the title.

**A** — NRA by NCA matrix to be printed. (Input)

**RLABEL** — CHARACTER \* (\*) vector of labels for rows of A. (Input)  
 If rows are to be numbered consecutively 1, 2, ..., NRA, use **RLABEL(1) = 'NUMBER'**. If no row labels are desired, use **RLABEL(1) = 'NONE'**. Otherwise, **RLABEL** is a vector of length NRA containing the labels.

**CLABEL** — CHARACTER \* (\*) vector of labels for columns of A. (Input)  
 If columns are to be numbered consecutively 1, 2, ..., NCA, use **CLABEL(1) = 'NUMBER'**. If no column labels are desired, use **CLABEL(1) = 'NONE'**. Otherwise, **CLABEL(1)** is the heading for the row labels, and either **CLABEL(2)** must be 'NUMBER' or 'NONE', or **CLABEL** must be a vector of length NCA + 1 with **CLABEL(1 + j)** containing the column heading for the j-th column.

### Optional Arguments

**NRA** — Number of rows. (Input)  
 Default: **NRA = SIZE(A,1)**.

**NCA** — Number of columns. (Input)  
Default: `NCA = SIZE (A,2)`.

**LDA** — Leading dimension of **A** exactly as specified in the dimension statement in the calling program. (Input)  
Default: `LDA = SIZE (A,1)`.

**ITRING** — Triangle option. (Input)  
Default: `ITRING = 0`.

**ITRING Action**

- 0 Full matrix is printed.
- 1 Upper triangle of **A** is printed, including the diagonal.
- 2 Upper triangle of **A** excluding the diagonal of **A** is printed.
- 1 Lower triangle of **A** is printed, including the diagonal.
- 2 Lower triangle of **A** excluding the diagonal of **A** is printed.

**FMT** — Character string containing formats. (Input)

If **FMT** is set to a blank character(s), the format used is specified by **WROPT**. Otherwise, **FMT** must contain exactly one set of parentheses and one or more edit descriptors. For example, `FMT = '(F10.3)'` specifies this **F** format for the entire matrix.

`FMT = '(2E10.3, 3F10.3)'` specifies an **E** format for columns 1 and 2 and an **F** format for columns 3, 4 and 5. If the end of **FMT** is encountered and if some columns of the matrix remain, format control continues with the first format in **FMT**. Even though the matrix **A** is real, an **I** format can be used to print the integer part of matrix elements of **A**. The most useful formats are special formats, called the **V** and **W** formats, that can be used to specify pretty formats automatically. Set `FMT = '(V10.4)'` if you want a single **D**, **E**, or **F** format selected automatically with field width 10 and with 4 significant digits. Set `FMT = '(W10.4)'` if you want a single **D**, **E**, **F**, or **I** format selected automatically with field width 10 and with 4 significant digits. While the **V** format prints trailing zeroes and a trailing decimal point, the **W** format does not. See Comment 4 for general descriptions of the **V** and **W** formats. **FMT** may contain only **D**, **E**, **F**, **G**, **I**, **V**, or **W** edit descriptors, e.g., the **X** descriptor is not allowed.

Default: `FMT = ' '`.

## FORTRAN 90 Interface

Generic: `CALL WRRRL (TITLE, A, RLABEL, CLABEL [, ...])`

Specific: The specific interface names are `S_WRRRL` and `D_WRRRL` for two dimensional arrays, and `S_WRRRL1D` and `D_WRRRL1D` for one dimensional arrays.

## FORTRAN 77 Interface

Single:       CALL WRRRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL, CLABEL)

Double:       The double precision name is DWRRRL.

## Description

Routine WRRRL prints a real rectangular matrix (stored in *A*) with row and column labels (specified by *RLABEL* and *CLABEL*, respectively) according to a given format (stored in *FMT*). WRRRL can restrict printing to the elements of upper or lower triangles of matrices via the *ITRING* option.

Generally, *ITRING* ≠ 0 is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set *NRA* to the length of the array and set *NCA* = 1. For a row vector, set *NRA* = 1 and set *NCA* to the length of the array. In both cases, set *LDA* = *NRA*, and set *ITRING* = 0.

## Comments

1.   Workspace may be explicitly provided, if desired, by use of W2RRL/DW2RRL. The reference is:

```
CALL W2RRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL,
 CLABEL, CHWK)
```

The additional argument is:

**CHWK** — CHARACTER \* 10 work vector of length *NCA*. This workspace is referenced only if all three conditions indicated at the beginning of this comment are met. Otherwise, *CHWK* is not referenced and can be a CHARACTER \* 10 vector of length one.

2.   The output appears in the following form:

|            | TITLE      |            |            |            |
|------------|------------|------------|------------|------------|
|            | CLABEL (1) | CLABEL (2) | CLABEL (3) | CLABEL (4) |
| RLABEL (1) | Xxxxxx     | Xxxxxx     | Xxxxxx     | Xxxxxx     |
| RLABEL (2) | Xxxxxx     | Xxxxxx     | Xxxxxx     | Xxxxxx     |

3.   Use “% /” within titles or labels to create a new line. Long titles or labels are automatically wrapped.
4.   For printing numbers whose magnitudes are unknown, the *G* format in FORTRAN is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The *V* and *W* formats are special formats used by this routine to select a *D*, *E*, *F*, or *I* format so that the decimal points will be aligned. The *V* and *W* formats are specified as *Vn.d* and *Wn.d*. Here, *n* is the field width and *d* is the number of significant digits generally printed. Valid values for *n* are 3, 4, ..., 40. Valid values for *d* are 1, 2, ..., *n* - 2. If *FMT* specifies one format and that format is a *V* or *W* format, all elements of the matrix *A* are examined to determine one FORTRAN format for

printing. If `FMT` specifies more than one format, FORTRAN formats are generated separately from each `V` or `W` format.

5. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT`.
6. Horizontal centering, method for printing large matrices, paging, method for printing NaN (not a number), printing a title on each page, and many other options can be selected by invoking `WROPT`.
7. Output is written to the unit specified by `UMACH` (see [Reference Material](#)).

## Example

The following example prints all of a  $3 \times 4$  matrix  $A$  where  $a_{ij} = (i + j/10)10^{j-3}$ .

```

USE WRRRL_INT

IMPLICIT NONE
INTEGER ITRING, LDA, NCA, NRA
PARAMETER (ITRING=0, LDA=10, NCA=4, NRA=3)
!
INTEGER I, J
REAL A(LDA,NCA)
CHARACTER CLABEL(5)*5, FMT*8, RLABEL(3)*5
!
DATA FMT/'(W10.6)'/
DATA CLABEL/' ', 'Col 1', 'Col 2', 'Col 3', 'Col 4'/
DATA RLABEL/'Row 1', 'Row 2', 'Row 3'/
!
DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = (I+J*0.1)*10.0**(J-3)
10 CONTINUE
20 CONTINUE
!
 Write A matrix.
CALL WRRRL ('A', A, RLABEL, CLABEL, NRA=NRA, FMT=FMT)
END

```

## Output

|       | A     |       |       |        |
|-------|-------|-------|-------|--------|
|       | Col 1 | Col 2 | Col 3 | Col 4  |
| Row 1 | 0.011 | 0.120 | 1.300 | 14.000 |
| Row 2 | 0.021 | 0.220 | 2.300 | 24.000 |
| Row 3 | 0.031 | 0.320 | 3.300 | 34.000 |

---

## WRIRN

Prints an integer rectangular matrix with integer row and column labels.

## Required Arguments

**TITLE** — Character string specifying the title. (Input)  
TITLE set equal to a blank character(s) suppresses printing of the title. Use “%/” within the title to create a new line. Long titles are automatically wrapped.

**MAT** — NRMAT by NCMAT matrix to be printed. (Input)

## Optional Arguments

**NRMAT** — Number of rows. (Input)  
Default: NRMAT = SIZE (MAT,1).

**NCMAT** — Number of columns. (Input)  
Default: NCMAT = SIZE (MAT,2).

**LDMAT** — Leading dimension of MAT exactly as specified in the dimension statement in the calling program. (Input)  
Default: LDMAT = SIZE (MAT,1).

**ITRING** — Triangle option. (Input)  
Default: ITRING = 0.

### ITRING Action

|    |                                                                 |
|----|-----------------------------------------------------------------|
| 0  | Full matrix is printed.                                         |
| 1  | Upper triangle of MAT is printed, including the diagonal.       |
| 2  | Upper triangle of MAT excluding the diagonal of MAT is printed. |
| -1 | Lower triangle of MAT is printed, including the diagonal.       |
| -2 | Lower triangle of MAT excluding the diagonal of MAT is printed. |

## FORTRAN 90 Interface

Generic: CALL WRIRN (TITLE, MAT [, ...])

Specific: The specific interface name is S\_WRIRN.

## FORTRAN 77 Interface

Single: CALL WRIRN (TITLE, NRMAT, NCMAT, MAT, LDMAT, ITRING)

## Description

Routine `WRIRN` prints an integer rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. `WRIRN` can restrict printing to elements of the upper and lower triangles of matrices via the `ITRING` option. Generally, `ITRING ≠ 0` is used with symmetric matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set `NRMAT` to the length of the array and set `NCMAT = 1`. For a row vector, set `NRMAT = 1` and set `NCMAT` to the length of the array. In both cases, set `LDMAT = NRMAT` and set `ITRING = 0`.

## Comments

1. All the entries in `MAT` are printed using a single `I` format. The field width is determined by the largest absolute entry.
2. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking `WROPT`.
3. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT`.
4. Output is written to the unit specified by `UMACH` (see [Reference Material](#)).

## Example

The following example prints all of a  $3 \times 4$  matrix  $A = \text{MAT}$  where  $a_{ij} = 10i + j$ .

```
USE WRIRN_INT

IMPLICIT NONE
INTEGER ITRING, LDMAT, NCMAT, NRMAT
PARAMETER (ITRING=0, LDMAT=10, NCMAT=4, NRMAT=3)
!
INTEGER I, J, MAT(LDMAT,NCMAT)
!
DO 20 I=1, NRMAT
 DO 10 J=1, NCMAT
 MAT(I,J) = I*10 + J
10 CONTINUE
20 CONTINUE
!
 Write MAT matrix.
CALL WRIRN ('MAT', MAT, NRMAT=NRMAT)
END
```

## Output

```
 MAT
 1 2 3 4
1 11 12 13 14
2 21 22 23 24
3 31 32 33 34
```

---

# WRIRL

Print an integer rectangular matrix with a given format and labels.

## Required Arguments

- TITLE** — Character string specifying the title. (Input)  
TITLE set equal to a blank character(s) suppresses printing of the title.
- MAT** — NRMAT by NCMAT matrix to be printed. (Input)
- RLABEL** — CHARACTER \* (\*) vector of labels for rows of MAT. (Input)  
If rows are to be numbered consecutively 1, 2, ..., NRMAT, use  
RLABEL(1) = 'NUMBER'. If no row labels are desired, use RLABEL(1) = 'NONE'.  
Otherwise, RLABEL is a vector of length NRMAT containing the labels.
- CLABEL** — CHARACTER \* (\*) vector of labels for columns of MAT. (Input)  
If columns are to be numbered consecutively 1, 2, ..., NCMAT, use  
CLABEL(1) = 'NUMBER'. If no column labels are desired, use CLABEL(1) = 'NONE'.  
Otherwise, CLABEL(1) is the heading for the row labels, and either CLABEL(2) must be  
'NUMBER' or 'NONE', or CLABEL must be a vector of length  
NCMAT + 1 with CLABEL(1 + j) containing the column heading for the j-th column.

## Optional Arguments

- NRMAT** — Number of rows. (Input)  
Default: NRMAT = SIZE (MAT,1).
- NCMAT** — Number of columns. (Input)  
Default: NCMAT = SIZE (MAT,2).
- LDMAT** — Leading dimension of MAT exactly as specified in the dimension statement in the calling program. (Input)  
Default: LDMAT = SIZE (MAT,1).
- ITRING** — Triangle option. (Input)  
Default: ITRING = 0.

### ITRING Action

- |    |                                                                 |
|----|-----------------------------------------------------------------|
| 0  | Full matrix is printed.                                         |
| 1  | Upper triangle of MAT is printed, including the diagonal.       |
| 2  | Upper triangle of MAT excluding the diagonal of MAT is printed. |
| -1 | Lower triangle of MAT is printed, including the diagonal.       |

-2 Lower triangle of `MAT` excluding the diagonal of `MAT` is printed.

**FMT** — Character string containing formats. (Input)

If `FMT` is set to a blank character(s), the format used is a single `I` format with field width determined by the largest absolute entry. Otherwise, `FMT` must contain exactly one set of parentheses and one or more `I` edit descriptors. For example, `FMT = ' (I10) '` specifies this `I` format for the entire matrix. `FMT = ' (2I10, 3I5) '` specifies an `I10` format for columns 1 and 2 and an `I5` format for columns 3, 4 and 5. If the end of `FMT` is encountered and if some columns of the matrix remain, format control continues with the first format in `FMT`. `FMT` may only contain the `I` edit descriptor, e.g., the `X` edit descriptor is not allowed.  
Default: `FMT = ' '`.

## FORTRAN 90 Interface

Generic: `CALL WRIRL (TITLE, MAT, RLABEL, CLABEL [, ...])`

Specific: The specific interface name is `S_WRIRL`.

## FORTRAN 77 Interface

Single: `CALL WRIRL (TITLE, NRMAT, NCMAT, MAT, LDMAT, ITRING, FMT, RLABEL, CLABEL)`

## Description

Routine `WRIRL` prints an integer rectangular matrix (stored in `MAT`) with row and column labels (specified by `RLABEL` and `CLABEL`, respectively), according to a given format (stored in `FMT`). `WRIRL` can restrict printing to the elements of upper or lower triangles of matrices via the `ITRING` option. Generally, `ITRING ≠ 0` is used with symmetric matrices. In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set `NRMAT` to the length of the array and set `NCMAT = 1`. For a row vector, set `NRMAT = 1` and set `NCMAT` to the length of the array. In both cases, set `LDMAT = NRMAT`, and set `ITRING = 0`.

## Comments

1. The output appears in the following form:

|           | TITLE     |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
|           | CLABEL(1) | CLABEL(2) | CLABEL(3) | CLABEL(4) |
| RLABEL(1) | Xxxxx     | xxxxx     | xxxxx     | xxxxx     |
| RLABEL(2) | Xxxxx     | xxxxx     | xxxxx     | xxxxx     |

2. Use “`% /`” within titles or labels to create a new line. Long titles or labels are automatically wrapped.
3. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT`.

4. Horizontal centering, a method for printing large matrices, paging, printing a title on each page, and many other options can be selected by invoking `WROPT`.
5. Output is written to the unit specified by `UMACH` (see the [Reference Material](#)).

## Example

The following example prints all of a  $3 \times 4$  matrix  $A = \text{MAT}$  where  $a_{ij} = 10i + j$ .

```

USE WRIRL_INT

IMPLICIT NONE
INTEGER ITRING, LDMAT, NCMAT, NRMAT

PARAMETER (ITRING=0, LDMAT=10, NCMAT=4, NRMAT=3)
!
INTEGER I, J, MAT(LDMAT,NCMAT)
CHARACTER CLABEL(5)*5, FMT*8, RLABEL(3)*5
!
DATA FMT/'(I2) '/
DATA CLABEL/' ', 'Col 1', 'Col 2', 'Col 3', 'Col 4'/
DATA RLABEL/'Row 1', 'Row 2', 'Row 3'/
!
DO 20 I=1, NRMAT
 DO 10 J=1, NCMAT
 MAT(I,J) = I*10 + J
10 CONTINUE
20 CONTINUE
!
 Write MAT matrix.
CALL WRIRL ('MAT', MAT, RLABEL, CLABEL, NRMAT=NRMAT)
END

```

## Output

|       | MAT   |       |       |       |
|-------|-------|-------|-------|-------|
|       | Col 1 | Col 2 | Col 3 | Col 4 |
| Row 1 | 11    | 12    | 13    | 14    |
| Row 2 | 21    | 22    | 23    | 24    |
| Row 3 | 31    | 32    | 33    | 34    |

---

## WRCRN

Prints a complex rectangular matrix with integer row and column labels.

### Required Arguments

- TITLE** — Character string specifying the title. (Input)  
 TITLE set equal to a blank character(s) suppresses printing of the title. Use “% /” within the title to create a new line. Long titles are automatically wrapped.
- A** — Complex NRA by NCA matrix to be printed. (Input)

## Optional Arguments

*NRA* — Number of rows. (Input)

Default: *NRA* = *SIZE* (*A*, 1).

*NCA* — Number of columns. (Input)

Default: *NCA* = *SIZE* (*A*, 2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement in the calling program. (Input)

Default: *LDA* = *SIZE* (*A*, 1).

*ITRING* — Triangle option. (Input)

Default: *ITRING* = 0.

### **ITRING Action**

- |    |                                                                           |
|----|---------------------------------------------------------------------------|
| 0  | Full matrix is printed.                                                   |
| 1  | Upper triangle of <i>A</i> is printed, including the diagonal.            |
| 2  | Upper triangle of <i>A</i> excluding the diagonal of <i>A</i> is printed. |
| -1 | Lower triangle of <i>A</i> is printed, including the diagonal.            |
| -2 | Lower triangle of <i>A</i> excluding the diagonal of <i>A</i> is printed. |

## FORTRAN 90 Interface

Generic: `CALL WRCRN (TITLE, A [, ...])`

Specific: The specific interface names are *S\_WRCRN* and *D\_WRCRN* for two dimensional arrays, and *S\_WRCRN1D* and *D\_WRCRN1D* for one dimensional arrays.

## FORTRAN 77 Interface

Single: `CALL WRCRN (TITLE, NRA, NCA, A, LDA, ITRING)`

Double: The double precision name is *DWRCRN*.

## Description

Routine *WRCRN* prints a complex rectangular matrix with the rows and columns labeled 1, 2, 3, and so on. *WRCRN* can restrict printing to the elements of the upper or lower triangles of matrices via the *ITRING* option. Generally, *ITRING* ≠ 0 is used with Hermitian matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set `NRA` to the length of the array, and set `NCA = 1`. For a row vector, set `NRA = 1`, and set `NCA` to the length of the array. In both cases, set `LDA = NRA`, and set `ITRING = 0`.

## Comments

1. A single `D`, `E`, or `F` format is chosen automatically in order to print 4 significant digits for the largest real or imaginary part in absolute value of all the complex numbers in `A`. Routine `WROPT` can be used to change the default format.
2. Horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number), and printing a title on each page can be selected by invoking `WROPT`.
3. A page width of 78 characters is used. Page width and page length can be reset by invoking subroutine `PGOPT`.
4. Output is written to the unit specified by `UMACH` (see [Reference Material](#)).

## Example

This example prints all of a  $3 \times 4$  complex matrix `A` with elements

$$a_{mn} = m + ni, \text{ where } i = \sqrt{-1}$$

```

USE WRCRN_INT

IMPLICIT NONE
INTEGER ITRING, LDA, NCA, NRA
PARAMETER (ITRING=0, LDA=10, NCA=4, NRA=3)
!
INTEGER I, J
COMPLEX A(LDA,NCA), CMPLX
INTRINSIC CMPLX
!
DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = CMPLX(I,J)
10 CONTINUE
20 CONTINUE
!
 Write A matrix.
CALL WRCRN ('A', A, NRA=NRA)
END

```

## Output

```

 A
 2
 3
 4
1 (1.000, 1.000) (1.000, 2.000) (1.000, 3.000) (1.000, 4.000)
2 (2.000, 1.000) (2.000, 2.000) (2.000, 3.000) (2.000, 4.000)
3 (3.000, 1.000) (3.000, 2.000) (3.000, 3.000) (3.000, 4.000)

```

---

# WRCRL

Prints a complex rectangular matrix with a given format and labels.

## Required Arguments

**TITLE** — Character string specifying the title. (Input)

TITLE set equal to a blank character(s) suppresses printing of the title.

**A** — Complex  $NRA$  by  $NCA$  matrix to be printed. (Input)

**RLABEL** — CHARACTER \* (\*) vector of labels for rows of A. (Input)

If rows are to be numbered consecutively 1, 2, ...,  $NRA$ , use  $RLABEL(1) = 'NUMBER'$ . If no row labels are desired, use  $RLABEL(1) = 'NONE'$ . Otherwise, **RLABEL** is a vector of length  $NRA$  containing the labels.

**CLABEL** — CHARACTER \* (\*) vector of labels for columns of A. (Input)

If columns are to be numbered consecutively 1, 2, ...,  $NCA$ , use  $CLABEL(1) = 'NUMBER'$ . If no column labels are desired, use  $CLABEL(1) = 'NONE'$ . Otherwise, **CLABEL(1)** is the heading for the row labels, and either **CLABEL(2)** must be 'NUMBER' or 'NONE', or **CLABEL** must be a vector of length  $NCA + 1$  with  $CLABEL(1 + j)$  containing the column heading for the  $j$ -th column.

## Optional Arguments

**NRA** — Number of rows. (Input)

Default:  $NRA = SIZE(A,1)$ .

**NCA** — Number of columns. (Input)

Default:  $NCA = SIZE(A,2)$ .

**LDA** — Leading dimension of A exactly as specified in the dimension statement in the calling program. (Input)

Default:  $LDA = SIZE(A, 1)$ .

**ITRING** — Triangle option. (Input)

Default:  $ITRING = 0$ .

| <b>ITRING</b> | <b>Action</b>                                               |
|---------------|-------------------------------------------------------------|
| 0             | Full matrix is printed.                                     |
| 1             | Upper triangle of A is printed, including the diagonal.     |
| 2             | Upper triangle of A excluding the diagonal of A is printed. |
| -1            | Lower triangle of A is printed, including the diagonal.     |

**FMT** — Character string containing formats. (Input)

If **FMT** is set to a blank character(s), the format used is specified by **WROPT**. Otherwise, **FMT** must contain exactly one set of parentheses and one or more edit descriptors. Because a complex number consists of two parts (a real and an imaginary part), two edit descriptors are used for printing a single complex number. **FMT = ' (E10.3, F10.3) '** specifies an **E** format for the real part and an **F** format for the imaginary part. **FMT = ' (F10.3) '** uses an **F** format for both the real and imaginary parts. If the end of **FMT** is encountered and if all columns of the matrix have not been printed, format control continues with the first format in **FMT**. Even though the matrix **A** is complex, an **I** format can be used to print the integer parts of the real and imaginary components of each complex number. The most useful formats are special formats, called the “**V** and **W** formats,” that can be used to specify pretty formats automatically. Set **FMT = ' (V10.4) '** if you want a single **D**, **E**, or **F** format selected automatically with field width 10 and with 4 significant digits. Set **FMT = ' (W10.4) '** if you want a single **D**, **E**, **F**, or **I** format selected automatically with field width 10 and with 4 significant digits. While the **V** format prints trailing zeroes and a trailing decimal point, the **W** format does not. See Comment 4 for general descriptions of the **V** and **W** formats. **FMT** may contain only **D**, **E**, **F**, **G**, **I**, **V**, or **W** edit descriptors, e.g., the **X** descriptor is not allowed.

Default: **FMT = ' '**.

## FORTRAN 90 Interface

Generic: `CALL WRCRL (TITLE, A, RLABEL, CLABEL[, ...])`

Specific: The specific interface names are `S_WRCRL` and `D_WRCRL` for two dimensional arrays, and `S_WRCRL1D` and `D_WRCRL1D` for one dimensional arrays.

## FORTRAN 77 Interface

Single: `CALL WRCRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL, CLABEL)`

Double: The double precision name is `DWRCRL`.

## Description

Routine `WRCRL` prints a complex rectangular matrix (stored in **A**) with row and column labels (specified by **RLABEL** and **CLABEL**, respectively) according to a given format (stored in **FMT**). Routine `WRCRL` can restrict printing to the elements of upper or lower triangles of matrices via the **ITRING** option. Generally, the **ITRING**  $\neq$  0 is used with Hermitian matrices.

In addition, one-dimensional arrays can be printed as column or row vectors. For a column vector, set **NRA** to the length of the array, and set **NCA** = 1. For a row vector, set **NRA** = 1, and set **NCA** to the length of the array. In both cases, set **LDA** = **NRA**, and set **ITRING** = 0.

## Comments

1. Workspace may be explicitly provided, if desired, by use of `W2CRL/DW2CRL`. The reference is:

```
CALL W2CRL (TITLE, NRA, NCA, A, LDA, ITRING, FMT, RLABEL,
 CLABEL, CHWK)
```

The additional argument is:

**CHWK** — CHARACTER \* 10 work vector of length 2 \* NCA. This workspace is referenced only if all three conditions indicated at the beginning of this comment are met. Otherwise, CHWK is not referenced and can be a CHARACTER \* 10 vector of length one.

2. The output appears in the following form:

```
 TITLE
CLABEL (1) CLABEL (2) CLABEL (3) CLABEL (4)
RLABEL (1) (xxxxxx,xxxxxx) (xxxxxx,xxxxxx) (xxxxxx,xxxxxx)
RLABEL (2) (xxxxxx,xxxxxx) (xxxxxx,xxxxxx) (xxxxxx,xxxxxx)
```

3. Use “% /” within titles or labels to create a new line. Long titles or labels are automatically wrapped.
4. For printing numbers whose magnitudes are unknown, the `G` format in FORTRAN is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The `V` and `W` formats are special formats used by this routine to select a `D`, `E`, `F`, or `I` format so that the decimal points will be aligned. The `V` and `W` formats are specified as `Vn.d` and `Wn.d`. Here,  $n$  is the field width, and  $d$  is the number of significant digits generally printed. Valid values for  $n$  are 3, 4, ..., 40. Valid values for  $d$  are 1, 2, ...,  $n - 2$ . If `FMT` specifies one format and that format is a `V` or `W` format, all elements of the matrix `A` are examined to determine one FORTRAN format for printing. If `FMT` specifies more than one format, FORTRAN formats are generated separately from each `V` or `W` format.
5. A page width of 78 characters is used. Page width and page length can be reset by invoking `PGOPT`.
6. Horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number), printing a title on each page, and may other options can be selected by invoking `WROPT`.
7. Output is written to the unit specified by `UMACH` (see the [Reference Material](#)).

## Example

The following example prints all of a  $3 \times 4$  matrix  $A$  with elements

$$a_{mn} = (m + .123456) + ni, \text{ where } i = \sqrt{-1}$$

```

USE WRCRL_INT

IMPLICIT NONE

INTEGER ITRING, LDA, NCA, NRA
PARAMETER (ITRING=0, LDA=10, NCA=4, NRA=3)
!
INTEGER I, J
COMPLEX A(LDA,NCA), CMPLX
CHARACTER CLABEL(5)*5, FMT*8, RLABEL(3)*5
INTRINSIC CMPLX
!
DATA FMT/'(W12.6)'/
DATA CLABEL/' ', 'Col 1', 'Col 2', 'Col 3', 'Col 4'/
DATA RLABEL/'Row 1', 'Row 2', 'Row 3'/
!
DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = CMPLX(I,J) + 0.123456
 10 CONTINUE
20 CONTINUE
!
 Write A matrix.
CALL WRCRL ('A', A, RLABEL, CLABEL, NRA=NRA, FMT=FMT)
END

```

## Output

```

 A
 Col 1 Col 2
Row 1 (1.12346, 1.00000) (1.12346, 2.00000)
Row 2 (2.12346, 1.00000) (2.12346, 2.00000)
Row 3 (3.12346, 1.00000) (3.12346, 2.00000)

 Col 3 Col 4
Row 1 (1.12346, 3.00000) (1.12346, 4.00000)
Row 2 (2.12346, 3.00000) (2.12346, 4.00000)
Row 3 (3.12346, 3.00000) (3.12346, 4.00000)

```

---

## WROPT

Sets or retrieves an option for printing a matrix.

### Required Arguments

*IOPT* — Indicator of option type. (Input)

| <b>IOPT</b> | <b>Description of Option Type</b>                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1, 1        | Horizontal centering or left justification of matrix to be printed                                                                                                             |
| -2, 2       | Method for printing large matrices                                                                                                                                             |
| -3, 3       | Paging                                                                                                                                                                         |
| -4, 4       | Method for printing NaN (not a number), and negative and positive machine infinity.                                                                                            |
| -5, 5       | Title option                                                                                                                                                                   |
| -6, 6       | Default format for real and complex numbers                                                                                                                                    |
| -7, 7       | Spacing between columns                                                                                                                                                        |
| -8, 8       | Maximum horizontal space reserved for row labels                                                                                                                               |
| -9, 9       | Indentation of continuation lines for row labels                                                                                                                               |
| -10, 10     | Hot zone option for determining line breaks for row labels                                                                                                                     |
| -11, 11     | Maximum horizontal space reserved for column labels                                                                                                                            |
| -12, 12     | Hot zone option for determining line breaks for column labels                                                                                                                  |
| -13, 13     | Hot zone option for determining line breaks for titles                                                                                                                         |
| -14, 14     | Option for the label that appears in the upper left hand corner that can be used as a heading for the row numbers or a label for the column headings for $WR^{*} * N$ routines |
| -15, 15     | Option for skipping a line between invocations of $WR^{*} * N$ routines, provided a new page is not to be issued                                                               |
| -16, 16     | Option for vertical alignment of the matrix values relative to the associated row labels that occupy more than one line                                                        |

| <b>IOPT</b> | <b>Description of Option Type</b>                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0           | Reset all the current settings saved in internal variables back to their last setting made with an invocation of WROPT with ISCOPE = 1. (This option is used internally by routines printing a matrix and is not useful otherwise.) |

If IOPT is negative, ISETNG and ISCOPE are input and are saved in internal variables. If IOPT is positive, ISETNG is output and receives the currently active setting for the option (if ISCOPE = 0) or the last global setting for the option (if ISCOPE = 1). If IOPT = 0, ISETNG and ISCOPE are not referenced.

**ISETNG** — Setting for option selected by IOPT. (Input, if IOPT is negative; output, if IOPT is positive; not referenced if IOPT = 0)

| <b>IOPT</b> | <b>ISETNG</b> | <b>Meaning</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -1, 1       | 0             | Matrix is left justified                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|             | 1             | Matrix is centered horizontally on page                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| -2, 2       | 0             | A complete row is printed before the next row is printed. Wrapping is used if necessary.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|             | <i>m</i>      | Here, <i>m</i> is a positive integer. Let $n_1$ be the maximum number of columns beginning with column 1 that fit across the page (as determined by the widths of the printing formats). First, columns 1 through $n_1$ are printed for rows 1 through <i>m</i> . Let $n_2$ be the maximum number of columns beginning with column $n_1 + 1$ that fit across the page. Second, columns $n_1 + 1$ through $n_1 + n_2$ are printed for rows 1 through <i>m</i> . This continues until the last columns are printed for rows 1 through <i>m</i> . Printing continues in this fashion for the next <i>m</i> rows, etc. |
| -3, 3       | -2            | Printing begins on the next line, and no paging occurs.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|             | -1            | Paging is on. Every invocation of a WR*** routine begins on a new page, and paging occurs within each invocation as is needed                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|             | 0             | Paging is on. The first invocation of a WR*** routine begins on a new page, and subsequent paging occurs as is needed. With this option, every invocation of a WR*** routine ends with a call to WROPT to reset this option to <i>k</i> , a positive integer giving the number of lines printed on the current page.                                                                                                                                                                                                                                                                                               |

| <b>IOPT</b> | <b>ISETNG</b> | <b>Meaning</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -1, 1       | 0             | Matrix is left justified                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|             | $k$           | Here, $k$ is a positive integer. Paging is on, and $k$ lines have been printed on the current page. If $k$ is less than the page length IPAGE (see PGOPT), then IPAGE - $k$ lines are printed before a new page instruction is issued. If $k$ is greater than or equal to IPAGE, then the first invocation of a WR*** routine begins on a new page. In any case, subsequent paging occurs as is needed. With this option, every invocation of a WR*** routine ends with a call to WROPT to reset the value of $k$ . |
| -4, 4       | 0             | NaN is printed as a series of decimal points, negative machine infinity is printed as a series of minus signs, and positive machine infinity is printed as a series of plus signs.                                                                                                                                                                                                                                                                                                                                  |
|             | 1             | NaN is printed as a series of blank characters, negative machine infinity is printed as a series of minus signs, and positive machine infinity is printed as a series of plus signs.                                                                                                                                                                                                                                                                                                                                |
|             | 2             | NaN is printed as "NaN," negative machine infinity is printed as "-Inf" and positive machine infinity is printed as "Inf."                                                                                                                                                                                                                                                                                                                                                                                          |
|             | 3             | NaN is printed as a series of blank characters, negative machine infinity is printed as "-Inf," and positive machine infinity is printed as "Inf."                                                                                                                                                                                                                                                                                                                                                                  |
| -5, 5       | 0             | Title appears only on first page.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|             | 1             | Title appears on the first page and all continuation pages.                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| -6, 6       | 0             | Format is (W10.4). See Comment 2.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|             | 1             | Format is (W12.6). See Comment 2.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|             | 2             | Format is (1PE12.5).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|             | 3             | Format is Vn.4 where the field width $n$ is determined. See Comment 2.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|             | 4             | Format is Vn.6 where the field width $n$ is determined. Comment 2.                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|             | 5             | Format is 1PE $n$ . $d$ where $n = d + 7$ , and $d + 1$ is the maximum number of significant digits.                                                                                                                                                                                                                                                                                                                                                                                                                |
| -7, 7       | $K_1$         | Number of characters left blank between columns. $K_1$ must be between 0 and 5, inclusively.                                                                                                                                                                                                                                                                                                                                                                                                                        |
| -8, 8       | $K_2$         | Maximum width (in characters) reserved for row labels. $K_2 = 0$ means use the default.                                                                                                                                                                                                                                                                                                                                                                                                                             |

| <b>IOPT</b> | <b>ISETNG</b>  | <b>Meaning</b>                                                                                                                                                                                                           |
|-------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -1, 1       | 0              | Matrix is left justified                                                                                                                                                                                                 |
| -9, 9       | K <sub>3</sub> | Number of characters used to indent continuation lines for row labels. K <sub>3</sub> must be between 0 and 10, inclusively.                                                                                             |
| -10, 10     | K <sub>4</sub> | Width (in characters) of the hot zone where line breaks in row labels can occur. K <sub>4</sub> = 0 means use the default. K <sub>4</sub> must not exceed 50.                                                            |
| -11, 11     | K <sub>5</sub> | Maximum width (in characters) reserved for column labels. K <sub>5</sub> = 0 means use the default.                                                                                                                      |
| -12, 12     | K <sub>6</sub> | Width (in characters) of the hot zone where line breaks in column labels can occur. K <sub>6</sub> = 0 means use the default. K <sub>6</sub> must not exceed 50.                                                         |
| -13, 13     | K <sub>7</sub> | Width (in characters) of the hot zone where line breaks in titles can occur. K <sub>7</sub> must be between 1 and 50, inclusively.                                                                                       |
| -14         | 0              | There is no label in the upper left hand corner.                                                                                                                                                                         |
|             | 1              | The label in the upper left hand corner is "Component" if a row vector or column vector is printed; the label is "Row/Column" if both the number of rows and columns are greater than one; otherwise, there is no label. |
| -15         | 0              | A blank line is printed on each invocation of a WR**N routine before the matrix title provided a new page is not to be issued.                                                                                           |
|             | 1              | A blank line is not printed on each invocation of a WR**N routine before the matrix title.                                                                                                                               |
| -16, 16     | 0              | The matrix values are aligned vertically with the last line of the associated row label for the case IOPT = 2 and ISET is positive.                                                                                      |
|             | 1              | The matrix values are aligned vertically with the first line of the associated row label.                                                                                                                                |

**ISCOPE** — Indicator of the scope of the option. (Input if IOPT is nonzero; not referenced if IOPT = 0)

**ISCOPE Action**

- 0 Setting is temporarily active for the next invocation of a WR\*\*\* matrix printing routine.
- 1 Setting is active until it is changed by another invocation of WROPT.

## FORTRAN 90 Interface

Generic:     CALL WROPT (IOPT, ISETNG, ISCOPE)

Specific:    The specific interface name is WROPT.

## FORTRAN 77 Interface

Single:      CALL WROPT (IOPT, ISETNG, ISCOPE)

## Description

Routine WROPT allows the user to set or retrieve an option for printing a matrix. The options controlled by WROPT include the following: horizontal centering, a method for printing large matrices, paging, method for printing NaN (not a number) and positive and negative machine infinities, printing titles, default formats for numbers, spacing between columns, maximum widths reserved for row and column labels, indentation of row labels that continue beyond one line, widths of hot zones for breaking of labels and titles, the default heading for row labels, whether to print a blank line between invocations of routines, and vertical alignment of matrix entries with respect to row labels continued beyond one line. (NaN and positive and negative machine infinities can be retrieved by AMACH and DMACH that are documented in the section “[Machine-Dependent Constants](#)” in the Reference Material.) Options can be set globally (ISCOPE = 1) or temporarily for the next call to a printing routine (ISCOPE = 0).

## Comments

1. This program can be invoked repeatedly before using a WR\*\*\* routine to print a matrix. The matrix printing routines retrieve these settings to determine the printing options. It is not necessary to call WROPT if a default value of a printing option is desired. The defaults are as follows.

| IOPT | Default Value for ISET | Meaning                                                                                                                    |
|------|------------------------|----------------------------------------------------------------------------------------------------------------------------|
| 1    | 0                      | Left justified                                                                                                             |
| 2    | 1000000                | Number lines before wrapping                                                                                               |
| 3    | -2                     | No paging                                                                                                                  |
| 4    | 2                      | NaN is printed as “NaN,” negative machine infinity is printed as “-Inf” and positive machine infinity is printed as “Inf.” |
| 5    | 0                      | Title only on first page.                                                                                                  |
| 6    | 3                      | Default format is Vn.4.                                                                                                    |
| 7    | 2                      | 2 spaces between columns.                                                                                                  |
| 8    | 0                      | Maximum row label width MAXRLW = 2 *<br>IPAGEW/3 if matrix has one column;<br>MAXRLW = IPAGEW/4 otherwise.                 |

| IOPT | Default Value for ISET | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 9    | 3                      | 3 character indentation of row labels continued beyond one line.                                                                                                                                                                                                                                                                                                                                                                         |
| 10   | 0                      | Width of row label hot zone is $\text{MAXRLW}/3$ characters.                                                                                                                                                                                                                                                                                                                                                                             |
| 11   | 0                      | Maximum column label width<br>$\text{MAXCLW} = \min\{\max(\text{NW} + \text{NW}/2, 15), 40\}$<br>for integer and real matrices, where $\text{NW}$ is the field width for the format corresponding to the particular column.<br>$\text{MAXCLW} = \min\{\max(\text{NW} + \text{NW}/2, 15), 83\}$ for complex matrices, where $\text{NW}$ is the sum of the two field widths for the formats corresponding to the particular column plus 3. |
| 12   | 0                      | Width of column label hot zone is $\text{MAXCLW}/3$ characters.                                                                                                                                                                                                                                                                                                                                                                          |
| 13   | 10                     | Width of hot zone for titles is 10 characters.                                                                                                                                                                                                                                                                                                                                                                                           |
| 14   | 0                      | There is no label in the upper left hand corner.                                                                                                                                                                                                                                                                                                                                                                                         |
| 15   | 0                      | Blank line is printed.                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 16   | 0                      | The matrix values are aligned vertically with the last line of the associated row label.                                                                                                                                                                                                                                                                                                                                                 |

For  $\text{IOPT} = 8$ , the default depends on the current value for the page width,  $\text{IPAGEW}$  (see [PGOPT](#)).

- The  $\text{v}$  and  $\text{w}$  formats are special formats that can be used to select a  $\text{D}$ ,  $\text{E}$ ,  $\text{F}$ , or  $\text{I}$  format so that the decimal points will be aligned. The  $\text{v}$  and  $\text{w}$  formats are specified as  $\text{Vn.d}$  and  $\text{Wn.d}$ . Here,  $n$  is the field width and  $d$  is the number of significant digits generally printed. Valid values for  $n$  are 3, 4, ..., 40. Valid values for  $d$  are 1, 2, ...,  $n - 2$ . While the  $\text{v}$  format prints trailing zeroes and a trailing decimal point, the  $\text{w}$  format does not.

### Example

The following example illustrates the effect of  $\text{WROPT}$  when printing a  $3 \times 4$  real matrix  $A$  with  $\text{WRRRN}$  where  $a_{ij} = i + j/10$ . The first call to  $\text{WROPT}$  sets horizontal printing so that the matrix is first printed horizontally centered on the page. In the next invocation of  $\text{WRRRN}$ , the left-justification option has been set via routine  $\text{WROPT}$  so the matrix is left justified when printed. Finally, because the scope of left justification was only for the next call to a printing routine, the last call to  $\text{WRRRN}$  results in horizontally centered printing.

```
USE WROPT_INT
USE WRRRN_INT

IMPLICIT NONE
```

```

 INTEGER ITRING, LDA, NCA, NRA
 PARAMETER (ITRING=0, LDA=10, NCA=4, NRA=3)
!
 INTEGER I, IOPT, ISCOPE, ISETNG, J
 REAL A(LDA,NCA)
!
 DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = I + J*0.1
10 CONTINUE
20 CONTINUE
!
! Activate centering option.
! Scope is global.
 IOPT = -1
 ISETNG = 1
 ISCOPE = 1
!
 CALL WROPT (IOPT, ISETNG, ISCOPE)
! Write A matrix.
 CALL WRRRN ('A', A, NRA=NRA)
! Activate left justification.
! Scope is local.
 IOPT = -1
 ISETNG = 0
 ISCOPE = 0
 CALL WROPT (IOPT, ISETNG, ISCOPE)
 CALL WRRRN ('A', A, NRA=NRA)
 CALL WRRRN ('A', A, NRA=NRA)
 END

```

## Output

```

 A
 1 2 3 4
 1 1.100 1.200 1.300 1.400
 2 2.100 2.200 2.300 2.400
 3 3.100 3.200 3.300 3.400

 A
 1 2 3 4
1 1.100 1.200 1.300 1.400
2 2.100 2.200 2.300 2.400
3 3.100 3.200 3.300 3.400

 A
 1 2 3 4
 1 1.100 1.200 1.300 1.400
 2 2.100 2.200 2.300 2.400
 3 3.100 3.200 3.300 3.400

```

---

## PGOPT

Sets or retrieves page width and length for printing.

## Required Arguments

*IOPT* — Page attribute option. (Input)

| <i>IOPT</i> | Description of Attribute |
|-------------|--------------------------|
|-------------|--------------------------|

|       |             |
|-------|-------------|
| -1, 1 | Page width. |
|-------|-------------|

|       |              |
|-------|--------------|
| -2, 2 | Page length. |
|-------|--------------|

Negative values of *IOPT* indicate the setting *IPAGE* is input. Positive values

of *IOPT* indicate the setting *IPAGE* is output.

*IPAGE* — Value of page attribute. (Input, if *IOPT* is negative; output, if *IOPT* is positive.)

| <i>IOPT</i> | Description of Attribute | Settings for <i>IPAGE</i> |
|-------------|--------------------------|---------------------------|
|-------------|--------------------------|---------------------------|

|       |                            |             |
|-------|----------------------------|-------------|
| -1, 1 | Page width (in characters) | 10, 11, ... |
|-------|----------------------------|-------------|

|       |                        |             |
|-------|------------------------|-------------|
| -2, 2 | Page length (in lines) | 10, 11, ... |
|-------|------------------------|-------------|

## FORTRAN 90 Interface

Generic:    CALL *PGOPT* (*IOPT*, *IPAGE*)

Specific:   The specific interface name is *PGOPT*.

## FORTRAN 77 Interface

Single:     CALL *PGOPT* (*IOPT*, *IPAGE*)

## Description

Routine *PGOPT* is used to set or retrieve the page width or the page length for routines that perform printing.

## Example

The following example illustrates the use of *PGOPT* to set the page width at 20 characters. Routine *WRRRN* is then used to print a  $3 \times 4$  matrix *A* where  $a_{ij} = i + j/10$ .

```
USE PGOPT_INT
USE WRRRN_INT

IMPLICIT NONE
INTEGER ITRING, LDA, NCA, NRA
PARAMETER (ITRING=0, LDA=3, NCA=4, NRA=3)
!
INTEGER I, IOPT, IPAGE, J
REAL A(LDA, NCA)
```

```

!
 DO 20 I=1, NRA
 DO 10 J=1, NCA
 A(I,J) = I + J*0.1
10 CONTINUE
20 CONTINUE
!
 IOPT = -1
 IPAGE = 20
 CALL PGOPT (IOPT, IPAGE)
!
 CALL WRRRN ('A', A)
 END

```

Set page width.

Print the matrix A.

## Output

```

 A
 1 2
1 1.100 1.200
2 2.100 2.200
3 3.100 3.200

 3 4
1 1.300 1.400
2 2.300 2.400
3 3.300 3.400

```

---

## PERMU

Rearranges the elements of an array as specified by a permutation.

### Required Arguments

*X* — Real vector of length *N* containing the array to be permuted. (Input)

*IPERMU* — Integer vector of length *N* containing a permutation  
*IPERMU*(1), ..., *IPERMU*(*N*) of the integers 1, ..., *N*. (Input)

*XPERMU* — Real vector of length *N* containing the array *X* permuted. (Output)  
 If *X* is not needed, *X* and *XPERMU* can share the same storage locations.

### Optional Arguments

*N* — Length of the arrays *X* and *XPERMU*. (Input)  
 Default: *N* = SIZE (*IPERMU*,1).

*IPATH* — Integer flag. (Input)  
 Default: *IPATH* = 1.  
*IPATH* = 1 means *IPERMU* represents a forward permutation, i.e., *X*(*IPERMU*(*I*)) is

moved to XPERMU(I). IPATH = 2 means IPERMU represents a backward permutation, i.e., X(I) is moved to XPERMU (IPERMU(I)).

## FORTRAN 90 Interface

Generic: CALL PERMU (X, IPERMU, XPERMU [, ...])

Specific: The specific interface names are S\_PERMU and D\_PERMU.

## FORTRAN 77 Interface

Single: CALL PERMU (N, X, IPERMU, IPATH, XPERMU)

Double: The double precision name is DPERMU.

## Description

Routine PERMU rearranges the elements of an array according to a permutation vector. It has the option to do both forward and backward permutations.

## Example

This example rearranges the array X using IPERMU; forward permutation is performed.

```
USE PERMU_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER IPATH, N
PARAMETER (IPATH=1, N=4)
!
INTEGER IPERMU(N), J, NOUT
REAL X(N), XPERMU(N)
!
! Set values for X, IPERMU
!
! X = (5.0 6.0 1.0 4.0)
! IPERMU = (3 1 4 2)
!
DATA X/5.0, 6.0, 1.0, 4.0/, IPERMU/3, 1, 4, 2/
!
! Permute X into XPERMU
CALL PERMU (X, IPERMU, XPERMU)
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Print results
WRITE (NOUT,99999) (XPERMU(J),J=1,N)
!
99999 FORMAT (' The output vector is:', /, 10(1X,F10.2))
END
```

## Output

The Output vector is:  
1.00      5.00      4.00      6.00

---

# PERMA

Permutes the rows or columns of a matrix.

## Required Arguments

*A* — *NRA* by *NCA* matrix to be permuted. (Input)

*IPERMU* — Vector of length *K* containing a permutation *IPERMU*(1), ..., *IPERMU*(*K*) of the integers 1, ..., *K* where *K* = *NRA* if the rows of *A* are to be permuted and *K* = *NCA* if the columns of *A* are to be permuted. (Input)

*APER* — *NRA* by *NCA* matrix containing the permuted matrix. (Output)  
If *A* is not needed, *A* and *APER* can share the same storage locations.

## Optional Arguments

*NRA* — Number of rows. (Input)  
Default: *NRA* = *SIZE* (*A*,1).

*NCA* — Number of columns. (Input)  
Default: *NCA* = *SIZE* (*A*,2).

*LDA* — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDA* = *SIZE* (*A*, 1).

*IPATH* — Option parameter. (Input)  
*IPATH* = 1 means the rows of *A* will be permuted. *IPATH* = 2 means the columns of *A* will be permuted.  
Default: *IPATH* = 1.

*LDAPER* — Leading dimension of *APER* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *LDAPER* = *SIZE* (*APER*,1).

## FORTRAN 90 Interface

Generic:     CALL PERMA (*A*, *IPERMU*, *APER* [, ...])

Specific:    The specific interface names are *S\_PERMA* and *D\_PERMA*.

## FORTRAN 77 Interface

Single:      CALL PERMA (NRA, NCA, A, LDA, IPERMU, IPATH, APER, LDAPER)

Double:     The double precision name is DPERMA.

## Description

Routine PERMA interchanges the rows or columns of a matrix using a permutation vector such as the one obtained from routines [SVRBP](#) or [SVRGP](#).

The routine PERMA permutes a column (row) at a time by calling [PERMU](#). This process is continued until all the columns (rows) are permuted. On completion, let  $B = APER$  and  $p_i = IPERMU(I)$ , then

$$B_{ij} = A_{p_i j}$$

for all  $i, j$ .

## Comments

1.    Workspace may be explicitly provided, if desired, by use of [P2RMA](#)/[DP2RMA](#). The reference is:

```
CALL P2RMA (NRA, NCA, A, LDA, IPERMU, IPATH, APER, LDAPER,
 WORK)
```

The additional argument is:

**WORK** — Real work vector of length NCA.

## Example

This example permutes the columns of a matrix A.

```
USE PERMA_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER IPATH, LDA, LDAPER, NCA, NRA
PARAMETER (IPATH=2, LDA=3, LDAPER=3, NCA=5, NRA=3)
!
INTEGER I, IPERMU(5), J, NOUT
REAL A(LDA,NCA), APER(LDAPER,NCA)
! Set values for A, IPERMU
! A = (3.0 5.0 1.0 2.0 4.0)
! (3.0 5.0 1.0 2.0 4.0)
! (3.0 5.0 1.0 2.0 4.0)
!
! IPERMU = (3 4 1 5 2)
!
DATA A/3*3.0, 3*5.0, 3*1.0, 3*2.0, 3*4.0/, IPERMU/3, 4, 1, 5, 2/
! Perform column permutation on A,
! giving APER
```

```

 CALL PERMA (A, IPERMU, APER, IPATH=IPATH)
! Get output unit number
 CALL UMACH (2, NOUT)
! Print results
 WRITE (NOUT,99999) ((APER(I,J),J=1,NCA),I=1,NRA)
!
99999 FORMAT (' The output matrix is:', /, 3(5F8.1,/))
 END

```

## Output

```

The Output matrix is:
1.0 2.0 3.0 4.0 5.0
1.0 2.0 3.0 4.0 5.0
1.0 2.0 3.0 4.0 5.0

```

---

## SORT\_REAL

Sorts a rank-1 array of real numbers  $x$  so the  $y$  results are algebraically nondecreasing,  
 $y_1 \leq y_2 \leq \dots y_n$ .

### Required Arguments

**X** — Rank-1 array containing the numbers to be sorted. (Output)

**Y** — Rank-1 array containing the sorted numbers. (Output)

### Optional Arguments

**NSIZE** =  $n$  (Input)  
 Uses the sub-array of size  $n$  for the numbers.  
 Default value:  $n = \text{SIZE}(x)$

**IPERM** =  $\text{iperms}$  (Input/Output)  
 Applies interchanges of elements that occur to the entries of  $\text{iperms}(\cdot)$ . If the values  $\text{iperms}(i)=i, i=1, n$  are assigned prior to call, then the output array is moved to its proper order by the subscripted array assignment  $y = x(\text{iperms}(1:n))$ .

**ICYCLE** =  $\text{icycle}$  (Output)  
 Permutations applied to the input data are converted to cyclic interchanges. Thus, the output array  $y$  is given by the following elementary interchanges, where  $:=$  denotes a swap:

```

j = icycle(i)
y(j) := y(i), i = 1,n

```

**IOPT** =  $\text{iop}(\cdot)$  (Input)  
 Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

| Packaged Options for <code>SORT_REAL</code> |                                     |              |
|---------------------------------------------|-------------------------------------|--------------|
| Option Prefix = ?                           | Option Name                         | Option Value |
| <code>s_, d_</code>                         | <code>Sort_real_scan_for_NaN</code> | 1            |

```
iopt(IO) = ?_options(?_sort_real_scan_for_NaN, ?_dummy)
```

Examines each input array entry to find the first value such that

```
isNaN(x(i)) == .true.
```

See the `isNaN()` function, [Chapter 10](#).

Default: Does not scan for NaNs.

## FORTRAN 90 Interface

Generic: `CALL SORT_REAL (X, Y [, ...])`

Specific: The specific interface names are `S_SORT_REAL` and `D_SORT_REAL`.

## Description

For a detailed description, see the “[Description](#)” section of routine `SVRGN`, which appears later in this chapter.

## Fatal and Terminal Error Messages

See the `messages.gls` file for error messages for `SORT_REAL`. These error messages are numbered 561–567; 581–587.

## Example 1: Sorting an Array

An array of random numbers is obtained. The values are sorted so they are nondecreasing.

```

use sort_real_int
use rand_gen_int

implicit none

! This is Example 1 for SORT_REAL.

integer, parameter :: n=100
real(kind(1e0)), dimension(n) :: x, y

! Generate random data to sort.
call rand_gen(x)

! Sort the data so it is non-decreasing.
call sort_real(x, y)

! Check that the sorted array is not decreasing.
if (count(y(1:n-1) > y(2:n)) == 0) then
 write (*,*) 'Example 1 for SORT_REAL is correct.'
end if

end

```

## Output

Example 1 for SORT\_REAL is correct.

## Additional Examples

### Example 2: Sort and Final Move with a Permutation

A set of  $n$  random numbers is sorted so the results are nonincreasing. The columns of an  $n \times n$  random matrix are moved to the order given by the permutation defined by the interchange of the entries. Since the routine sorts the results to be algebraically nondecreasing, the array of negative values is used as input. Thus, the negative value of the sorted output order is nonincreasing. The optional argument “`iperm=`” records the final order and is used to move the matrix columns to that order. This example illustrates the principle of sorting record *keys*, followed by direct movement of the records to sorted order.

```
 use sort_real_int
 use rand_gen_int

 implicit none

! This is Example 2 for SORT_REAL.

 integer i
 integer, parameter :: n=100
 integer ip(n)
 real(kind(1e0)) a(n,n), x(n), y(n), temp(n*n)

! Generate a random array and matrix of values.
 call rand_gen(x)
 call rand_gen(temp)
 a = reshape(temp, (/n,n/))

! Initialize permutation to the identity.
 do i=1, n
 ip(i) = i
 end do

! Sort using negative values so the final order is
! non-increasing.
 call sort_real(-x, y, iperm=ip)

! Final movement of keys and matrix columns.
 y = x(ip(1:n))
 a = a(:,ip(1:n))

! Check the results.
 if (count(y(1:n-1) < y(2:n)) == 0) then
 write (*,*) 'Example 2 for SORT_REAL is correct.'
 end if

 end
```

## Output

Example 2 for SORT\_REAL is correct.

---

# SVRGN

Sorts a real array by algebraically increasing value.

## Required Arguments

**RA** — Vector of length  $N$  containing the array to be sorted. (Input)

**RB** — Vector of length  $N$  containing the sorted array. (Output)  
If **RA** is not needed, **RA** and **RB** can share the same storage locations.

## Optional Arguments

**N** — Number of elements in the array to be sorted. (Input)  
Default:  $N = \text{SIZE}(\text{RA}, 1)$ .

## FORTRAN 90 Interface

Generic:    CALL SVRGN (RA, RB [, ...])

Specific:   The specific interface names are S\_SVRGN and D\_SVRGN.

## FORTRAN 77 Interface

Single:     CALL SVRGN (N, RA, RB)

Double:     The double precision name is DSVRGN.

## Description

Routine *SVRGN* sorts the elements of an array,  $A$ , into ascending order by algebraic value. The array  $A$  is divided into two parts by picking a central element  $T$  of the array. The first and last elements of  $A$  are compared with  $T$  and exchanged until the three values appear in the array in ascending order. The elements of the array are rearranged until all elements greater than or equal to the central element appear in the second part of the array and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the array. On completion,  $A_j \leq A_i$  for  $j < i$ . For more details, see Singleton (1969), Griffin and Redish (1970), and Petro (1970).

## Example

This example sorts the 10-element array **RA** algebraically.

```

USE SVRGN_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER N, NOUT, J
PARAMETER (N=10)
REAL RA(N), RB(N)
!
! Set values for RA
! RA = (-1.0 2.0 -3.0 4.0 -5.0 6.0 -7.0 8.0 -9.0 10.0)
!
! DATA RA/-1.0, 2.0, -3.0, 4.0, -5.0, 6.0, -7.0, 8.0, -9.0, 10.0/
! Sort RA by algebraic value into RB
CALL SVRGN (RA, RB)
!
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99999) (RB(J), J=1,N)
!
99999 FORMAT (' The output vector is:', /, 10(1X,F5.1))
END

```

## Output

```

The Output vector is:
-9.0 -7.0 -5.0 -3.0 -1.0 2.0 4.0 6.0 8.0 10.0

```

---

## SVRGP

Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

### Required Arguments

**RA** — Vector of length *N* containing the array to be sorted. (Input)

**RB** — Vector of length *N* containing the sorted array. (Output)  
If *RA* is not needed, *RA* and *RB* can share the same storage locations.

**IPERM** — Vector of length *N*. (Input/Output)  
On input, *IPERM* should be initialized to the values 1, 2, ..., *N*. On output, *IPERM* contains a record of permutations made on the vector *RA*.

### Optional Arguments

**N** — Number of elements in the array to be sorted. (Input)  
Default: *N* = *SIZE* (*IPERM*,1).

### FORTRAN 90 Interface

Generic:    CALL SVRGP (RA, RB, IPERM [, ...])

Specific: The specific interface names are `S_SVRGP` and `D_SVRGP`.

## FORTRAN 77 Interface

Single: `CALL SVRGP (N, RA, RB, IPERM)`

Double: The double precision name is `DSVRGP`.

## Description

Routine `SVRGP` sorts the elements of an array,  $A$ , into ascending order by algebraic value, keeping a record in  $P$  of the permutations to the array  $A$ . That is, the elements of  $P$  are moved in the same manner as are the elements in  $A$  as  $A$  is being sorted. The routine `SVRGP` uses the algorithm discussed in [SVRGN](#). On completion,  $A_j \leq A_i$  for  $j < i$ .

## Comments

For wider applicability, integers (1, 2, ..., N) that are to be associated with `RA(I)` for  $I = 1, 2, \dots, N$  may be entered into `IPERM(I)` in any order. Note that these integers must be unique.

## Example

This example sorts the 10-element array `RA` algebraically.

```
USE SVRGP_INT
USE UMACH_INT

IMPLICIT NONE
!
! Declare variables
INTEGER N, NOUT, J
PARAMETER (N=10)
REAL RA(N), RB(N)
INTEGER IPERM(N)
!
! Set values for RA and IPERM
! RA = (10.0 -9.0 8.0 -7.0 6.0 5.0 4.0 -3.0 -2.0 -1.0)
!
! IPERM = (1 2 3 4 5 6 7 8 9 10)
!
DATA RA/10.0, -9.0, 8.0, -7.0, 6.0, 5.0, 4.0, -3.0, -2.0, -1.0/
DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
!
! Sort RA by algebraic value into RB
CALL SVRGP (RA, RB, IPERM)
!
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99998) (RB(J),J=1,N)
WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT (' The output vector is:', /, 10(1X,F5.1))
99999 FORMAT (' The permutation vector is:', /, 10(1X,I5))
END
```

## Output

The output vector is:

-9.0 -7.0 -3.0 -2.0 -1.0 4.0 5.0 6.0 8.0 10.0

The permutation vector is:

2 4 8 9 10 7 6 5 3 1

---

# SVIGN

Sorts an integer array by algebraically increasing value.

## Required Arguments

*IA* — Integer vector of length *N* containing the array to be sorted. (Input)

*IB* — Integer vector of length *N* containing the sorted array. (Output)  
If *IA* is not needed, *IA* and *IB* can share the same storage locations.

## Optional Arguments

*N* — Number of elements in the array to be sorted. (Input)  
Default: *N* = SIZE (*IA*,1).

## FORTRAN 90 Interface

Generic: CALL SVIGN (*IA*, *IB* [, ...])

Specific: The specific interface name is `S_SVIGN`.

## FORTRAN 77 Interface

Single: CALL SVIGN (*N*, *IA*, *IB*)

## Description

Routine `SVIGN` sorts the elements of an integer array, *A*, into ascending order by algebraic value. The routine `SVIGN` uses the algorithm discussed in [SVRGN](#). On completion,  $A_j \leq A_i$  for  $j < i$ .

## Example

This example sorts the 10-element array *IA* algebraically.

```
USE SVIGN_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N, NOUT, J
```

```

PARAMETER (N=10)
INTEGER IA(N), IB(N)
!
! Set values for IA
! IA = (-1 2 -3 4 -5 6 -7 8 -9 10)
!
DATA IA/-1, 2, -3, 4, -5, 6, -7, 8, -9, 10/
! Sort IA by algebraic value into IB
CALL SVIGN (IA, IB)
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99999) (IB(J),J=1,N)
!
99999 FORMAT (' The output vector is:', /, 10(1X,I5))
END

```

## Output

```

The Output vector is:
-9 -7 -5 -3 -1 2 4 6 8 10

```

---

# SVIGP

Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

## Required Arguments

**IA** — Integer vector of length *N* containing the array to be sorted. (Input)

**IB** — Integer vector of length *N* containing the sorted array. (Output)  
If **IA** is not needed, **IA** and **IB** can share the same storage locations.

**IPERM** — Vector of length *N*. (Input/Output)  
On input, **IPERM** should be initialized to the values 1, 2, ..., *N*. On output, **IPERM** contains a record of permutations made on the vector **IA**.

## Optional Arguments

**N** — Number of elements in the array to be sorted. (Input)  
Default: *N* = SIZE(**IPERM**,1).

## FORTRAN 90 Interface

Generic: CALL SVIGP (IA, IB, IPERM [, ...])

Specific: The specific interface name is S\_SVIGP.

## FORTRAN 77 Interface

Single: CALL SVIGP (N, IA, IB, IPERM)

## Description

Routine `SVIGP` sorts the elements of an integer array,  $A$ , into ascending order by algebraic value, keeping a record in  $P$  of the permutations to the array  $A$ . That is, the elements of  $P$  are moved in the same manner as are the elements in  $A$  as  $A$  is being sorted. The routine `SVIGP` uses the algorithm discussed in `SVRGN`. On completion,  $A_j \leq A_i$  for  $j < i$ .

## Comments

For wider applicability, integers (1, 2, ..., N) that are to be associated with  $IA(I)$  for  $I = 1, 2, \dots, N$  may be entered into  $IPERM(I)$  in any order. Note that these integers must be unique.

## Example

This example sorts the 10-element array  $IA$  algebraically.

```
USE SVIGP_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N, J, NOUT
PARAMETER (N=10)
INTEGER IA(N), IB(N), IPERM(N)
! Set values for IA and IPERM
! IA = (10 -9 8 -7 6 5 4 -3 -2 -1)
!
! IPERM = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
!
DATA IA/10, -9, 8, -7, 6, 5, 4, -3, -2, -1/
DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
! Sort IA by algebraic value into IB
CALL SVIGP (IA, IB, IPERM)
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99998) (IB(J),J=1,N)
WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT (' The output vector is:', /, 10(1X,I5))
99999 FORMAT (' The permutation vector is:', /, 10(1X,I5))
END
```

## Output

```
The Output vector is:
-9 -7 -3 -2 -1 4 5 6 8 10

The permutation vector is:
 2 4 8 9 10 7 6 5 3 1
```

---

# SVRBN

Sorts a real array by nondecreasing absolute value.

## Required Arguments

*RA* — Vector of length *N* containing the array to be sorted. (Input)

*RB* — Vector of length *N* containing the sorted array. (Output)  
If *RA* is not needed, *RA* and *RB* can share the same storage locations.

## Optional Arguments

*N* — Number of elements in the array to be sorted. (Input)  
Default: *N* = SIZE (*RA*,1).

## FORTRAN 90 Interface

Generic: CALL SVRBN (*RA*, *RB* [, ...])

Specific: The specific interface names are S\_SVRBN and D\_SVRBN.

## FORTRAN 77 Interface

Single: CALL SVRBN (*N*, *RA*, *RB*)

Double: The double precision name is DSVRBN.

## Description

Routine SVRBN sorts the elements of an array, *A*, into ascending order by absolute value. The routine SVRBN uses the algorithm discussed in SVRGN. On completion,  $|A_j| \leq |A_i|$  for  $j < i$ .

## Example

This example sorts the 10-element array *RA* by absolute value.

```
USE SVRBN_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N, J, NOUT
PARAMETER (N=10)
REAL RA(N), RB(N)
! Set values for RA
! RA = (-1.0 3.0 -4.0 2.0 -1.0 0.0 -7.0 6.0 10.0 -7.0)
!
DATA RA/-1.0, 3.0, -4.0, 2.0, -1.0, 0.0, -7.0, 6.0, 10.0, -7.0/
! Sort RA by absolute value into RB
CALL SVRBN (RA, RB)
```

```

! Print results
 CALL UMACH (2,NOUT)
 WRITE (NOUT, 99999) (RB(J),J=1,N)
!
99999 FORMAT (' The output vector is :', /, 10(1X,F5.1))
 END

```

## Output

```

The Output vector is :
0.0 -1.0 -1.0 2.0 3.0 -4.0 6.0 -7.0 -7.0 10.0

```

---

# SVRBP

Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

## Required Arguments

**RA** — Vector of length  $N$  containing the array to be sorted. (Input)

**RB** — Vector of length  $N$  containing the sorted array. (Output)  
If **RA** is not needed, **RA** and **RB** can share the same storage locations.

**IPERM** — Vector of length  $N$ . (Input/Output)  
On input, **IPERM** should be initialized to the values 1, 2, ...,  $N$ . On output, **IPERM** contains a record of permutations made on the vector **IA**.

## Optional Arguments

**N** — Number of elements in the array to be sorted. (Input)  
Default:  $N = \text{SIZE}(\text{IPERM}, 1)$ .

## FORTRAN 90 Interface

Generic:    CALL SVRBP (RA, RB, IPERM [, ...])

Specific:    The specific interface names are `S_SVRBP` and `D_SVRBP`.

## FORTRAN 77 Interface

Single:     CALL SVRBP (N, RA, RB, IPERM)

Double:     The double precision name is `DSVRBP`.

## Description

Routine SVRBP sorts the elements of an array,  $A$ , into ascending order by absolute value, keeping a record in  $P$  of the permutations to the array  $A$ . That is, the elements of  $P$  are moved in the same manner as are the elements in  $A$  as  $A$  is being sorted. The routine SVRBP uses the algorithm discussed in SVRGN. On completion,  $A_j \leq A_i$  for  $j < i$ .

## Comments

For wider applicability, integers (1, 2, ..., N) that are to be associated with  $RA(I)$  for  $I = 1, 2, \dots, N$  may be entered into  $IPERM(I)$  in any order. Note that these integers must be unique.

## Example

This example sorts the 10-element array RA by absolute value.

```
USE SVRBP_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N, J, NOUT, I
PARAMETER (N=10)
REAL RA(N), RB(N)
INTEGER IPERM(N)

! Set values for RA and IPERM
! RA = (10.0 9.0 8.0 7.0 6.0 5.0 -4.0 3.0 -2.0 1.0)
!
! IPERM = (1 2 3 4 5 6 7 8 9 10)
!
DATA RA/10.0, 9.0, 8.0, 7.0, 6.0, 5.0, -4.0, 3.0, -2.0, 1.0/
DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
! Sort RA by absolute value into RB
CALL SVRBP (RA, RB, IPERM)

! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99998) (RB(J),J=1,N)
WRITE (NOUT, 99999) (IPERM(I),I=1,N)

!
99998 FORMAT (' The output vector is:', /, 10(1X,F5.1))
99999 FORMAT (' The permutation vector is:', /, 10(1X,I5))
END
```

## Output

```
The output vector is:
1.0 -2.0 3.0 -4.0 5.0 6.0 7.0 8.0 9.0 10.0
The permutation vector is:
10 9 8 7 6 5 4 3 2 1
```

---

# SVIBN

Sorts an integer array by nondecreasing absolute value.

## Required Arguments

**IA** — Integer vector of length  $N$  containing the array to be sorted. (Input)

**IB** — Integer vector of length  $N$  containing the sorted array. (Output)  
If **IA** is not needed, **IA** and **IB** can share the same storage locations.

## Optional Arguments

**N** — Number of elements in the array to be sorted. (Input)  
Default:  $N = \text{SIZE}(\text{IA}, 1)$ .

## FORTRAN 90 Interface

Generic:     CALL SVIBN (IA, IB [, ...])

Specific:    The specific interface name is S\_SVIBN.

## FORTRAN 77 Interface

Single:      CALL SVIBN (N, IA, IB)

## Description

Routine `SVIBN` sorts the elements of an integer array,  $A$ , into ascending order by absolute value. This routine `SVIBN` uses the algorithm discussed in [SVRGN](#). On completion,  $A_j \leq A_i$  for  $j < i$ .

## Example

This example sorts the 10-element array **IA** by absolute value.

```
USE SVIBN_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER I, J, NOUT, N
PARAMETER (N=10)
INTEGER IA(N), IB(N)
! Set values for IA
! IA = (-1 3 -4 2 -1 0 -7 6 10 -7)
!
DATA IA/-1, 3, -4, 2, -1, 0, -7, 6, 10, -7/
! Sort IA by absolute value into IB
CALL SVIBN (IA, IB)
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99999) (IB(J),J=1,N)
!
99999 FORMAT (' The output vector is:', /, 10(1X,I5))
END
```

## Output

The Output vector is:

0    -1    -1    2    3    -4    6    -7    -7    10

---

# SVIBP

Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

## Required Arguments

*IA* — Integer vector of length *N* containing the array to be sorted. (Input)

*IB* — Integer vector of length *N* containing the sorted array. (Output)  
If *IA* is not needed, *IA* and *IB* can share the same storage locations.

*IPERM* — Vector of length *N*. (Input/Output)  
On input, *IPERM* should be initialized to the values 1, 2, ..., *N*. On output, *IPERM* contains a record of permutations made on the vector *IA*.

## Optional Arguments

*N* — Number of elements in the array to be sorted. (Input)  
Default: *N* = SIZE(*IA*,1).

## FORTRAN 90 Interface

Generic:    CALL SVIBP (*IA*, *IB*, *IPERM* [, ...])

Specific:    The specific interface name is *S\_SVIBP*.

## FORTRAN 77 Interface

Single:    CALL SVIBP (*N*, *IA*, *IB*, *IPERM*)

## Description

Routine *SVIBP* sorts the elements of an integer array, *A*, into ascending order by absolute value, keeping a record in *P* of the permutations to the array *A*. That is, the elements of *P* are moved in the same manner as are the elements in *A* as *A* is being sorted. The routine *SVIBP* uses the algorithm discussed in [SVRGN](#). On completion,  $A_j \leq A_i$  for  $j < i$ .

## Comments

For wider applicability, integers (1, 2, ..., *N*) that are to be associated with *IA*(*I*) for *I* = 1, 2, ..., *N* may be entered into *IPERM*(*I*) in any order. Note that these integers must be unique.

## Example

This example sorts the 10-element array `IA` by absolute value.

```
USE SVIBP_INT
USE UMACH_INT

IMPLICIT NONE
! Declare variables
INTEGER N, U, NOUT, J
PARAMETER (N=10)
INTEGER IA(N), IB(N), IPERM(N)
! Set values for IA
! IA = (10 9 8 7 6 5 -4 3 -2 1)
!
! IPERM = (1 2 3 4 5 6 7 8 9 10)
!
DATA IA/10, 9, 8, 7, 6, 5, -4, 3, -2, 1/
DATA IPERM/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
! Sort IA by absolute value into IB
CALL SVIBP (IA, IB, IPERM)
! Print results
CALL UMACH (2,NOUT)
WRITE (NOUT, 99998) (IB(J),J=1,N)
WRITE (NOUT, 99999) (IPERM(J),J=1,N)
!
99998 FORMAT (' The output vector is:', /, 10(1X,I5))
99999 FORMAT (' The permutation vector is:', /, 10(1X,I5))
END
```

## Output

```
The Output vector is:
1 -2 3 -4 5 6 7 8 9 10

The permutation vector is:
10 9 8 7 6 5 4 3 2 1
```

---

# SRCH

Searches a sorted vector for a given scalar and return its index.

## Required Arguments

**VALUE** — Scalar to be searched for in `Y`. (Input)

**X** — Vector of length `N * INCX`. (Input)

`Y` is obtained from `X` for `I = 1, 2, ..., N` by  $Y(I) = X(1 + (I - 1) * INCX)$ . `Y(1), Y(2), ..., Y(N)` must be in ascending order.

**INDEX** — Index of *Y* pointing to *VALUE*. (Output)

If *INDEX* is positive, *VALUE* is found in *Y*. If *INDEX* is negative, *VALUE* is not found in *Y*.

| <b>INDEX</b>       | <b>Location of VALUE</b>           |
|--------------------|------------------------------------|
| 1 thru <i>N</i>    | $VALUE = Y(INDEX)$                 |
| -1                 | $VALUE < Y(1)$ or $N = 0$          |
| - <i>N</i> thru -2 | $Y(-INDEX - 1) < VALUE < Y(INDEX)$ |
| -( <i>N</i> + 1)   | $VALUE > Y(N)$                     |

### Optional Arguments

***N*** — Length of vector *Y*. (Input)

Default:  $N = (SIZE(X,1)) / INCX$ .

***INCX*** — Displacement between elements of *X*. (Input)

*INCX* must be greater than zero.

Default:  $INCX = 1$ .

### FORTRAN 90 Interface

Generic: `CALL SRCH (VALUE, X, INDEX [, ...])`

Specific: The specific interface names are `S_SRCH` and `D_SRCH`.

### FORTRAN 77 Interface

Single: `CALL SRCH (N, VALUE, X, INCX, INDEX)`

Double: The double precision name is `DSRCH`.

### Description

Routine `SRCH` searches a real vector *x* (stored in *X*), whose *n* elements are sorted in ascending order for a real number *c* (stored in *VALUE*). If *c* is found in *x*, its index *i* (stored in *INDEX*) is returned so that  $x_i = c$ . Otherwise, a negative number *i* is returned for the index. Specifically,

|                        |                              |
|------------------------|------------------------------|
| if $1 \leq i \leq n$   | then $x_i = c$               |
| if $i = -1$            | then $c < x_1$ or $n = 0$    |
| if $-n \leq i \leq -2$ | then $x_{-i-1} < c < x_{-i}$ |
| if $i = -(n + 1)$      | then $c > x_n$               |

The argument `INCX` is useful if a row of a matrix, for example, row number `I` of a matrix `X`, must be searched. The elements of row `I` are assumed to be in ascending order. In this case, set `INCX` equal to the leading dimension of `X` exactly as specified in the dimension statement in the calling program. With `X` declared

```
REAL X(LDX,N)
```

the invocation

```
CALL SRCH (N, VALUE, X(I,1), LDX, INDEX)
```

returns an index that will reference a column number of `X`.

Routine `SRCH` performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

### Example

This example searches a real vector sorted in ascending order for the value 653.0. The problem is discussed by Knuth (1973, pages 407–409).

```
USE SRCH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=16)
!
INTEGER INDEX, NOUT
REAL VALUE, X(N)
!
DATA X/61.0, 87.0, 154.0, 170.0, 275.0, 426.0, 503.0, 509.0, &
 512.0, 612.0, 653.0, 677.0, 703.0, 765.0, 897.0, 908.0/
!
VALUE = 653.0
CALL SRCH (VALUE, X, INDEX)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'INDEX = ', INDEX
END
```

### Output

```
INDEX = 11
```

---

## ISRCH

Searches a sorted integer vector for a given integer and return its index.

### Required Arguments

*IVALUE* — Scalar to be searched for in `IY`. (Input)

**IX** — Vector of length  $N * INCX$ . (Input)  
 IY is obtained from IX for  $I = 1, 2, \dots, N$  by  
 $IY(I) = IX(1 + (I - 1) * INCX)$ . IY(1), IY(2), ..., IY(N) must be in ascending order.

**INDEX** — Index of IY pointing to IVALUE. (Output)  
 If INDEX is positive, IVALUE is found in IY. If INDEX is negative, IVALUE is not found in IY.

| <b>INDEX</b> | <b>Location of VALUE</b>               |
|--------------|----------------------------------------|
| 1 thru N     | $IVALUE = IY(INDEX)$                   |
| -1           | $IVALUE < IY(1)$ or $N = 0$            |
| -N thru -2   | $IY(-INDEX - 1) < IVALUE < IY(-INDEX)$ |
| -(N + 1)     | $IVALUE > Y(N)$                        |

### Optional Arguments

**N** — Length of vector IY. (Input)  
 Default:  $N = SIZE(IX,1) / INCX$ .

**INCX** — Displacement between elements of IX. (Input)  
 INCX must be greater than zero.  
 Default:  $INCX = 1$ .

### FORTRAN 90 Interface

Generic: `CALL ISRCH (IVALUE, IX, INDEX [, ...])`

Specific: The specific interface name is `S_ISRCH`.

### FORTRAN 77 Interface

Single: `CALL ISRCH (N, IVALUE, IX, INCX, INDEX)`

### Description

Routine `ISRCH` searches an integer vector  $x$  (stored in `IX`), whose  $n$  elements are sorted in ascending order for an integer  $c$  (stored in `IVALUE`). If  $c$  is found in  $x$ , its index  $i$  (stored in `INDEX`) is returned so that  $x_i = c$ . Otherwise, a negative number  $i$  is returned for the index. Specifically,

|                        |                              |
|------------------------|------------------------------|
| if $1 \leq i \leq n$   | Then $x_i = c$               |
| if $i = -1$            | Then $c < x_1$ or $n = 0$    |
| if $-n \leq i \leq -2$ | Then $x_{-i-1} < c < x_{-i}$ |
| if $i = -(n + 1)$      | Then $c > x_n$               |

The argument `INCX` is useful if a row of a matrix, for example, row number `I` of a matrix `IX`, must be searched. The elements of row `I` are assumed to be in ascending order. Here, set `INCX` equal to the leading dimension of `IX` exactly as specified in the dimension statement in the calling program. With `IX` declared

```
INTEGER IX(LDIX,N)
```

the invocation

```
CALL ISRCH (N, IVALUE, IX(I,1), LDIX, INDEX)
```

returns an index that will reference a column number of `IX`.

The routine `ISRCH` performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

### Example

This example searches an integer vector sorted in ascending order for the value 653. The problem is discussed by Knuth (1973, pages 407–409).

```

USE ISRCH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=16)
!
INTEGER INDEX, NOUT
INTEGER IVALUE, IX(N)
!
DATA IX/61, 87, 154, 170, 275, 426, 503, 509, 512, 612, 653, 677, &
 703, 765, 897, 908/
!
IVALUE = 653
CALL ISRCH (IVALUE, IX, INDEX)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'INDEX = ', INDEX
END

```

### Output

```
INDEX = 11
```

---

## SSRCH

Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

## Required Arguments

*N* — Length of vector *CHY*. (Input)

Default:  $N = \text{SIZE}(\text{CHX}, 1) / \text{INCX}$ .

*STRING* — Character string to be searched for in *CHY*. (Input)

*CHX* — Vector of length  $N * \text{INCX}$  containing character strings. (Input)

*CHY* is obtained from *CHX* for  $I = 1, 2, \dots, N$  by  $\text{CHY}(I) = \text{CHX}(1 + (I - 1) * \text{INCX})$ .  
*CHY*(1), *CHY*(2), ..., *CHY*(*N*) must be in ascending ASCII order.

*INCX* — Displacement between elements of *CHX*. (Input)

*INCX* must be greater than zero.

Default:  $\text{INCX} = 1$ .

*INDEX* — Index of *CHY* pointing to *STRING*. (Output)

If *INDEX* is positive, *STRING* is found in *CHY*. If *INDEX* is negative, *STRING* is not found in *CHY*.

| <b>INDEX</b>       | <b>Location of STRING</b>                                                   |
|--------------------|-----------------------------------------------------------------------------|
| 1 thru <i>N</i>    | $\text{STRING} = \text{CHY}(\text{INDEX})$                                  |
| -1                 | $\text{STRING} < \text{CHY}(1)$ or $N = 0$                                  |
| - <i>N</i> thru -2 | $\text{CHY}(-\text{INDEX} - 1) < \text{STRING} < \text{CHY}(-\text{INDEX})$ |
| -( <i>N</i> + 1)   | $\text{STRING} > \text{CHY}(N)$                                             |

## FORTRAN 90 Interface

Generic:    CALL SSRCH (N, STRING, CHX, INCX, INDEX)

Specific:   The specific interface name is SSRCH.

## FORTRAN 77 Interface

Single:     CALL SSRCH (N, STRING, CHX, INCX, INDEX)

## Description

Routine *SSRCH* searches a vector of character strings *x* (stored in *CHX*), whose *n* elements are sorted in ascending ASCII order, for a character string *c* (stored in *STRING*). If *c* is found in *x*, its index *i* (stored in *INDEX*) is returned so that  $x_i = c$ . Otherwise, a negative number *i* is returned for the index. Specifically,

|                      |                           |
|----------------------|---------------------------|
| if $1 \leq i \leq n$ | Then $x_i = c$            |
| if $i = -1$          | Then $c < x_1$ or $n = 0$ |

|                        |                              |
|------------------------|------------------------------|
| if $-n \leq i \leq -2$ | Then $x_{-i-1} < c < x_{-i}$ |
| if $i = -(n + 1)$      | Then $c > x_n$               |

Here, “<” and “>” are in reference to the ASCII collating sequence. For comparisons made between character strings  $c$  and  $x_i$  with different lengths, the shorter string is considered as if it were extended on the right with blanks to the length of the longer string. (SSRCH uses FORTRAN intrinsic functions LLT and LGT.)

The argument INCX is useful if a row of a matrix, for example, row number I of a matrix CHX, must be searched. The elements of row I are assumed to be in ascending ASCII order. In this case, set INCX equal to the leading dimension of CHX exactly as specified in the dimension statement in the calling program. With CHX declared

```
CHARACTER * 7 CHX(LDCHX,N)
```

the invocation

```
CALL SSRCH (N, STRING, CHX(I,1), LDCHX, INDEX)
```

returns an index that will reference a column number of CHX.

The routine SSRCH performs a binary search. The routine is an implementation of algorithm *B* discussed by Knuth (1973, pages 407–411).

## Example

This example searches a CHARACTER \* 2 vector containing 9 character strings, sorted in ascending ASCII order, for the value 'CC'.

```

USE SSRCH_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N, INCX
PARAMETER (N=9)

!
INTEGER INDEX, NOUT
CHARACTER CHX(N)*2, STRING*2
!
DATA CHX/'AA', 'BB', 'CC', 'DD', 'EE', 'FF', 'GG', 'HH', &
 'II'/
!
INCX = 1
STRING = 'CC'
CALL SSRCH (N, STRING, CHX, INCX, INDEX)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'INDEX = ', INDEX
END

```

## Output

```
INDEX = 3
```

---

# ACHAR

This function returns a character given its ASCII value.

## Function Return Value

*ACHAR* — CHARACTER \* 1 string containing the character in the *I*-th position of the ASCII collating sequence. (Output)

## Required Arguments

*I* — Integer ASCII value of the character desired. (Input)  
*I* must be greater than or equal to zero and less than or equal to 127.

## FORTRAN 90 Interface

Generic:    ACHAR (I)

Specific:   The specific interface name is ACHAR.

## FORTRAN 77 Interface

Single:     ACHAR (I)

## Description

Routine ACHAR returns the character of the input ASCII value. The input value should be between 0 and 127. If the input value is out of range, the value returned in ACHAR is machine dependent.

## Example

This example returns the character of the ASCII value 65.

```
 USE ACHAR_INT
 USE UMACH_INT

! IMPLICIT NONE
! INTEGER I, NOUT
!
! CALL UMACH (2, NOUT)
! Get character for ASCII value
! of 65 ('A')
 I = 65
 WRITE (NOUT,99999) I, ACHAR(I)
!
99999 FORMAT (' For the ASCII value of ', I2, ', the character is : ', &
 A1)
 END
```

## Output

For the ASCII value of 65, the character is : A

---

# IACHAR

This function returns the integer ASCII value of a character argument.

## Function Return Value

*IACHAR* — Integer ASCII value for *CH*. (Output)

The character *CH* is in the *IACHAR*-th position of the ASCII collating sequence.

## Required Arguments

*CH* — Character argument for which the integer ASCII value is desired. (Input)

## FORTRAN 90 Interface

Generic:    *IACHAR* (*CH*)

Specific:   The specific interface name is *IACHAR*.

## FORTRAN 77 Interface

### Description

Routine *IACHAR* returns the ASCII value of the input character.

Single:    *IACHAR* (*CH*)

### Example

This example gives the ASCII value of character A.

```
USE IACHAR_INT

IMPLICIT NONE
INTEGER NOUT
CHARACTER CH
!
CALL UMACH (2, NOUT)
! Get ASCII value for the character
! 'A'.
CH = 'A'
WRITE (NOUT,99999) CH, IACHAR(CH)
!
99999 FORMAT (' For the character ', A1, ' the ASCII value is : ', &
 I3)
END
```

## Output

For the character A the ASCII value is : 65

---

# ICASE

This function returns the ASCII value of a character converted to uppercase.

## Function Return Value

*ICASE* — Integer ASCII value for *CH* without regard to the case of *CH*. (Output)  
Routine *ICASE* returns the same value as *IACHAR* for all but lowercase letters. For these, it returns the *IACHAR* value for the corresponding uppercase letter.

## Required Arguments

*CH* — Character to be converted. (Input)

## FORTRAN 90 Interface

Generic: `ICASE (CH)`

Specific: The specific interface name is *ICASE*.

## FORTRAN 77 Interface

Single: `ICASE (CH)`

## Description

Routine *ICASE* converts a character to its integer ASCII value. The conversion is case insensitive; that is, it returns the ASCII value of the corresponding uppercase letter for a lowercase letter.

## Example

This example shows the case insensitive conversion.

```
USE ICASE_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
CHARACTER CHR
!
! CALL UMACH (2, NOUT) Get output unit number
!
! Get ASCII value for the character
! 'a'.
CHR = 'a'
WRITE (NOUT,99999) CHR, ICASE(CHR)
!
99999 FORMAT (' For the character ', A1, ' the ICASE value is : ', &
```

```
 I3)
END
```

## Output

For the character a the ICASE value is : 65

---

# IICSR

This function compares two character strings using the ASCII collating sequence but without regard to case.

## Function Return Value

**IICSR** — Comparison indicator. (Output)

Let *USTR1* and *USTR2* be the uppercase versions of *STR1* and *STR2*, respectively. The following table indicates the relationship between *USTR1* and *USTR2* as determined by the ASCII collating sequence.

| <b>IICSR</b> | <b>Meaning</b>                     |
|--------------|------------------------------------|
| -1           | <i>USTR1</i> precedes <i>USTR2</i> |
| 0            | <i>USTR1</i> equals <i>USTR2</i>   |
| 1            | <i>USTR1</i> follows <i>USTR2</i>  |

## Required Arguments

**STR1** — First character string. (Input)

**STR2** — Second character string. (Input)

## FORTRAN 90 Interface

Generic:    **IICSR** (*STR1*, *STR2*)

Specific:   The specific interface name is **IICSR**.

## FORTRAN 77 Interface

Single:     **IICSR** (*STR1*, *STR2*)

## Description

Routine `IICSR` compares two character strings. It returns `-1` if the first string is less than the second string, `0` if they are equal, and `1` if the first string is greater than the second string. The comparison is case insensitive.

## Comments

If the two strings, `STR1` and `STR2`, are of unequal length, the shorter string is considered as if it were extended with blanks to the length of the longer string.

## Example

This example shows different cases on comparing two strings.

```
USE IICSR_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
CHARACTER STR1*6, STR2*6
! Get output unit number
CALL UMACH (2, NOUT)
! Compare String1 and String2
! String1 is 'bigger' than String2
STR1 = 'Abc 1'
STR2 = ' '
WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
! String1 is 'equal' to String2
STR1 = 'AbC'
STR2 = 'ABC'
WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
! String1 is 'smaller' than String2
STR1 = 'ABC'
STR2 = 'aBC 1'
WRITE (NOUT,99999) STR1, STR2, IICSR(STR1,STR2)
!
99999 FORMAT (' For String1 = ', A6, ' and String2 = ', A6, &
 ' IICSR = ', I2, '/')
END
```

## Output

```
For String1 = Abc 1 and String2 = IICSR = 1
For String1 = AbC and String2 = ABC IICSR = 0
For String1 = ABC and String2 = aBC 1 IICSR = -1
```

---

# IIDEX

This function determines the position in a string at which a given character sequence begins without regard to case.

## Function Return Value

**IIDEX** — Position in `CHRSTR` where `KEY` begins. (Output)  
If `KEY` occurs more than once in `CHRSTR`, the starting position of the first occurrence is returned. If `KEY` does not occur in `CHRSTR`, then `IIDEX` returns a zero.

## Required Arguments

**CHRSTR** — Character string to be searched. (Input)  
**KEY** — Character string that contains the key sequence. (Input)

## FORTRAN 90 Interface

Generic:     **IIDEX** (`CHRSTR`, `KEY`)  
Specific:    The specific interface name is `IIDEX`.

## FORTRAN 77 Interface

Single:     **IIDEX** (`CHRSTR`, `KEY`)

## Description

Routine `IIDEX` searches for a key string in a given string and returns the index of the starting element at which the key character string begins. It returns 0 if there is no match. The comparison is case insensitive. For a case-sensitive version, use the FORTRAN 77 intrinsic function `INDEX`.

## Comments

If the length of `KEY` is greater than the length `CHRSTR`, `IIDEX` returns a zero.

## Example

This example locates a key string.

```
USE IIDEX_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
CHARACTER KEY*5, STRING*10
! Get output unit number
CALL UMACH (2, NOUT)
```

```

! Locate KEY in STRING
 STRING = 'a1b2c3d4e5'
 KEY = 'C3d4E'
 WRITE (NOUT,99999) STRING, KEY, IINDEX (STRING,KEY)
!
 KEY = 'F'
 WRITE (NOUT,99999) STRING, KEY, IINDEX (STRING,KEY)
!
99999 FORMAT (' For STRING = ', A10, ' and KEY = ', A5, ' IINDEX = ', I2, &
 /)
 END

```

## Output

```

For STRING = a1b2c3d4e5 and KEY = C3d4E IINDEX = 5
For STRING = a1b2c3d4e5 and KEY = F IINDEX = 0

```

---

# CVTSI

Converts a character string containing an integer number into the corresponding integer form.

## Required Arguments

**STRING** — Character string containing an integer number. (Input)

**NUMBER** — The integer equivalent of STRING. (Output)

## FORTRAN 90 Interface

Generic:    CALL CVTSI (STRING, NUMBER)

Specific:   The specific interface name is CVTSI.

## FORTRAN 77 Interface

Single:     CALL CVTSI (STRING, NUMBER)

## Description

Routine CVTSI converts a character string containing an integer to an INTEGER variable. Leading and trailing blanks in the string are ignored. If the string contains something other than an integer, a terminal error is issued. If the string contains an integer larger than can be represented by an INTEGER variable as determined from routine IMACH (see the [Reference Material](#)), a terminal error is issued.

## Example

The string “12345” is converted to an INTEGER variable.

```

 USE CVTSI_INT

```

```

USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT, NUMBER
CHARACTER STRING*10
!
DATA STRING/'12345'/
!
CALL CVTSI (STRING, NUMBER)
!
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'NUMBER = ', NUMBER
END

```

## Output

```
NUMBER = 12345
```

---

# CPSEC

This function returns CPU time used in seconds.

## Function Return Value

*CPSEC* — CPU time used (in seconds) since first call to *CPSEC*. (Output)

## Required Arguments

None

## FORTRAN 90 Interface

Generic: CPSEC ()

Specific: The specific interface name is *CPSEC*.

## FORTRAN 77 Interface

Single: CPSEC (1)

## Comments

1. The first call to *CPSEC* returns 0.0.
2. The accuracy of this routine depends on the hardware and the operating system. On some systems, identical runs can produce timings differing by more than 10 percent.

---

# TIMDY

Gets time of day.

## Required Arguments

**I***HO***UR** — Hour of the day. (Output)  
IHOURL is between 0 and 23 inclusive.

**M***I***N****U****T****E** — Minute within the hour. (Output)  
MINUTE is between 0 and 59 inclusive.

**I***S***E****C** — Second within the minute. (Output)  
ISEC is between 0 and 59 inclusive.

## FORTRAN 90 Interface

Generic:    CALL TIMDY (IHOURL, MINUTE, ISEC)

Specific:   The specific interface name is TIMDY.

## FORTRAN 77 Interface

Single:     CALL TIMDY (IHOURL, MINUTE, ISEC)

## Description

Routine TIMDY is used to retrieve the time of day.

## Example

The following example uses TIMDY to return the current time. Obviously, the output is dependent upon the time at which the program is run.

```
USE TIMDY_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IHOURL, IMIN, ISEC, NOUT
!
CALL TIMDY (IHOURL, IMIN, ISEC)
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'Hour:Minute:Second = ', IHOURL, ':', IMIN, &
 ':', ISEC
IF (IHOURL .EQ. 0) THEN
 WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
 ' second(s) past midnight.'
ELSE IF (IHOURL .LT. 12) THEN
 WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
 ' second(s) past ', IHOURL, ' am.'
ELSE IF (IHOURL .EQ. 12) THEN
 WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
```

```

 ' second(s) past noon.'
ELSE
 WRITE (NOUT,*) 'The time is ', IMIN, ' minute(s), ', ISEC, &
 ' second(s) past ', I HOUR-12, ' pm.'
END IF
END
END

```

## Output

```

Hour:Minute:Second = 14 : 34 : 30
The time is 34 minute(s), 30 second(s) past 2 pm.

```

---

## TDATE

Gets today's date.

### Required Arguments

*IDAY* — Day of the month. (Output)  
*IDAY* is between 1 and 31 inclusive.

*MONTH* — Month of the year. (Output)  
*MONTH* is between 1 and 12 inclusive.

*IYEAR* — Year. (Output)  
 For example, *IYEAR* = 1985.

### FORTRAN 90 Interface

Generic:    CALL TDATE (IDAY, MONTH, IYEAR)

Specific:   The specific interface name is TDATE.

### FORTRAN 77 Interface

Single:     CALL TDATE (IDAY, MONTH, IYEAR)

### Description

Routine TDATE is used to retrieve today's date. Obviously, the output is dependent upon the date the program is run.

### Example

The following example uses TDATE to return today's date.

```

USE TDATE_INT
USE UMACH_INT

```

```

 IMPLICIT NONE
 INTEGER IDAY, IYEAR, MONTH, NOUT
!
 CALL TDATE (IDAY, MONTH, IYEAR)
 CALL UMACH (2, NOUT)
 WRITE (NOUT,*) 'Day-Month-Year = ', IDAY, '-', MONTH, &
 '- ', IYEAR
 END

```

## Output

```
Day-Month-Year = 7 - 7 - 2006
```

# NDAYS

This function computes the number of days from January 1, 1900, to the given date.

## Function Return Value

*NDAYS* — Function value. (Output)

If *NDAYS* is negative, it indicates the number of days prior to January 1, 1900.

## Required Arguments

*IDAY* — Day of the input date. (Input)

*MONTH* — Month of the input date. (Input)

*IYEAR* — Year of the input date. (Input)

1950 would correspond to the year 1950 A.D. and 50 would correspond to year 50 A.D.

## FORTRAN 90 Interface

Generic: `NDAYS (IDAY, MONTH, IYEAR)`

Specific: The specific interface name is `NDAYS`.

## FORTRAN 77 Interface

Single: `NDAYS (IDAY, MONTH, IYEAR)`

## Description

Function `NDAYS` returns the number of days from January 1, 1900, to the given date. The function `NDAYS` returns negative values for days prior to January 1, 1900. A negative `IYEAR` can be used to specify B.C. Input dates in year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, `NDAYS` issues a terminal error.

## Comments

1. Informational error

Type Code

1 1 The Julian calendar, the first modern calendar, went into use in 45 B.C. No calendar prior to 45 B.C. was as universally used nor as accurate as the Julian. Therefore, it is assumed that the Julian calendar was in use prior to 45 B.C.

2. The number of days from one date to a second date can be computed by two references to `NDAYS` and then calculating the difference.
3. The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. `NDAYS` makes the proper adjustment for the change in calendars.

## Example

The following example uses `NDAYS` to compute the number of days from January 15, 1986, to February 28, 1986:

```
USE NDAYS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IDAY, IYEAR, MONTH, NDAY0, NDAY1, NOUT
!
IDAY = 15
MONTH = 1
IYEAR = 1986
NDAY0 = NDAYS(IDAY, MONTH, IYEAR)
IDAY = 28
MONTH = 2
IYEAR = 1986
NDAY1 = NDAYS(IDAY, MONTH, IYEAR)
CALL UMACH(2, NOUT)
WRITE(NOUT,*) 'Number of days = ', NDAY1 - NDAY0
END
```

## Output

```
Number of days = 44
```

---

## NDYIN

Gives the date corresponding to the number of days since January 1, 1900.

### Required Arguments

*NDAYS* — Number of days since January 1, 1900. (Input)

*IDAY* — Day of the input date. (Output)

*MONTH* — Month of the input date. (Output)

*IYEAR* — Year of the input date. (Output)

1950 would correspond to the year 195 A.D. and -50 would correspond to year 50 B.C.

## **FORTRAN 90 Interface**

Generic:     CALL NDYIN (NDAYS, IDAY, MONTH, IYEAR)

Specific:    The specific interface name is NDYIN.

## **FORTRAN 77 Interface**

Single:     CALL NDYIN (NDAYS, IDAY, MONTH, IYEAR)

## **Description**

Routine NDYIN computes the date corresponding to the number of days since January 1, 1900. For an input value of NDAYS that is negative, the date computed is prior to January 1, 1900. The routine NDYIN is the inverse of NDAYS.

## **Comments**

The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. Routine NDYIN makes the proper adjustment for the change in calendars.

## **Example**

The following example uses NDYIN to compute the date for the 100th day of 1986. This is accomplished by first using NDAYS to get the “day number” for December 31, 1985.

```
USE NDYIN_INT
USE NDAYS_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IDAY, IYEAR, MONTH, NDAYO, NOUT, NDAYO
!
NDAYO = NDAYS(31,12,1985)
CALL NDYIN (NDAYO+100, IDAY, MONTH, IYEAR)
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'Day 100 of 1986 is (day-month-year) ', IDAY, &
 '- ', MONTH, '- ', IYEAR
END
```

## **Output**

## IDYWK

This function computes the day of the week for a given date.

### Function Return Value

*IDYWK* — Function value. (Output)

The value of *IDYWK* ranges from 1 to 7, where 1 corresponds to Sunday and 7 corresponds to Saturday.

### Required Arguments

*IDAY* — Day of the input date. (Input)

*MONTH* — Month of the input date. (Input)

*IYEAR* — Year of the input date. (Input)

1950 would correspond to the year 1950 A.D. and 50 would correspond to year 50 A.D.

### FORTRAN 90 Interface

Generic: `IDYWK (IDAY, MONTH, IYEAR)`

Specific: The specific interface name is *IDYWK*.

### FORTRAN 77 Interface

Single: `IDYWK (IDAY, MONTH, IYEAR)`

### Description

Function *IDYWK* returns an integer code that specifies the day of week for a given date. Sunday corresponds to 1, Monday corresponds to 2, and so forth.

A negative *IYEAR* can be used to specify B.C. Input dates in year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, *IDYWK* issues a terminal error.

### Comments

1. Informational error

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                                                                                      |
|---|---|--------------------------------------------------------------------------------------------------------------------------------------|
| 1 | 1 | The Julian calendar, the first modern calendar, went into use in 45 B.C. No calendar prior to 45 B.C. was as universally used nor as |
|---|---|--------------------------------------------------------------------------------------------------------------------------------------|

accurate as the Julian. Therefore, it is assumed that the Julian calendar was in use prior to 45 B.C.

2. The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use. Function `IDYWK` makes the proper adjustment for the change in calendars.

### Example

The following example uses `IDYWK` to return the day of the week for February 24, 1963.

```
USE IDYWK_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER IDAY, IYEAR, MONTH, NOUT
!
IDAY = 24
MONTH = 2
IYEAR = 1963
CALL UMACH (2, NOUT)
WRITE (NOUT,*) 'IDYWK (index for day of week) = ', &
 IDYWK (IDAY,MONTH,IYEAR)
END
```

### Output

```
IDYWK (index for day of week) = 1
```

---

## VERML

This function obtains IMSL MATH/LIBRARY-related version, system and serial numbers.

### Function Return Value

*VERML* — CHARACTER string containing information. (Output)

### Required Arguments

*ISELCT* — Option for the information to retrieve. (Input)

**ISELCT**    **VERML**

- 1            IMSL MATH/LIBRARY version number
- 2            Operating system (and version number) for which the library was produced.
- 3            Fortran compiler (and version number) for which the library was produced.
- 4            IMSL MATH/LIBRARY serial number

## **FORTRAN 90 Interface**

Generic: VERML ( ISELCT )

Specific: The specific interface name is VERML.

## **FORTRAN 77 Interface**

Single: VERML ( ISELCT )

## **Example**

In this example, we print all of the information returned by VERML on a particular machine. The output is omitted because the results are system dependent.

```
USE UMACH_INT
USE VERML_INT

IMPLICIT NONE
INTEGER ISELCT, NOUT
CHARACTER STRING(4)*50, TEMP*32
!
STRING(1) = '(' IMSL MATH/LIBRARY Version Number: ', A) '
STRING(2) = '(' Operating System ID Number: ', A) '
STRING(3) = '(' Fortran Compiler Version Number: ', A) '
STRING(4) = '(' IMSL MATH/LIBRARY Serial Number: ', A) '
!
 Print the versions and numbers.
CALL UMACH (2, NOUT)
DO 10 ISELCT=1, 4
 TEMP = VERML(ISELCT)
 WRITE (NOUT, STRING(ISELCT)) TEMP
10 CONTINUE
END
```

## **Output**

```
IMSL MATH/LIBRARY Version Number: IMSL Fortran Numerical Library, Version 6.0.0
Operating System ID Number: Solaris Version 10
Fortran Compiler Version Number: Sun Fortran 95 8.1 2005/01/07 (Workshop 10.0)
IMSL MATH/LIBRARY Serial Number: 999999
```

---

# **RAND\_GEN**

Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

## **Required Argument**

*X* — Rank-1 array containing the random numbers. (Output)

## Optional Arguments

IRND = IRND (Output)

Rank-1 integer array. These integers are the internal results of the Generalized Feedback Shift Register (GFSR) algorithm. The values are scaled to yield the floating-point array  $x$ . The output array entries are between 1 and  $2^{31} - 1$  in value.

ISTATE\_IN = ISTATE\_IN (Input)

Rank-1 integer array of size  $3p + 2$ , where  $p = 521$ , that defines the ensuing state of the GFSR generator. It is used to reset the internal tables to a previously defined state. It is the result of a previous use of the “ISTATE\_OUT=” optional argument.

ISTATE\_OUT = ISTATE\_OUT (Output)

Rank-1 integer array of size  $3p + 2$  that describes the current state of the GFSR generator. It is normally used to later reset the internal tables to the state defined following a return from the GFSR generator. It is the result of a use of the generator without a user initialization, or it is the result of a previous use of the optional argument “ISTATE\_IN=” followed by updates to the internal tables from newly generated values. Example 2 illustrates use of ISTATE\_IN and ISTATE\_OUT for setting and then resetting RAND\_GEN so that the sequence of integers, irnd, is repeatable.

IOPT = IOPT(:) (Input[/Output])

Derived type array with the same precision as the array  $x$ ; used for passing optional data to rand\_gen. The options are as follows:

| Packaged Options for RAND_GEN |                            |              |
|-------------------------------|----------------------------|--------------|
| Option Prefix = ?             | Option Name                | Option Value |
| s_, d_                        | Rand_gen_generator_seed    | 1            |
| s_, d_                        | Rand_gen_LCM_modulus       | 2            |
| s_, d_                        | Rand_gen_use_Fushimi_start | 3            |

IOPT(IO) = ?\_options(?\_rand\_gen\_generator\_seed, ?\_dummy)

Sets the initial values for the GFSR. The present value of the seed, obtained by default from the real-time clock as described below, swaps places with  $iopt(IO + 1)\%idummy$ . If the seed is set before any current usage of RAND\_GEN, the exchanged value will be zero.

IOPT(IO) = ?\_options(?\_rand\_gen\_LCM\_modulus, ?\_dummy)

IOPT(IO+1) = ?\_options(modulus, ?\_dummy)

Sets the initial values for the GFSR. The present value of the LCM, with default value  $k = 16807$ , swaps places with  $iopt(IO+1)\%idummy$ .

IOPT(IO) = ?\_options(?\_rand\_gen\_use\_Fushimi\_start, ?\_dummy)

Starts the GFSR sequence as suggested by Fushimi (1990). The default starting sequence is with the LCM recurrence described below.

## FORTRAN 90 Interface

Generic:    CALL RAND\_GEN (X [, ...])

Specific:    The specific interface names are S\_RAND\_GEN and D\_RAND\_GEN.

## Description

This GFSR algorithm is based on the recurrence

$$x_t = x_{t-3p} \oplus x_{t-3q}$$

where  $a \oplus b$  is the exclusive OR operation on two integers  $a$  and  $b$ . This operation is performed until  $\text{SIZE}(x)$  numbers have been generated. The subscripts in the recurrence formula are computed modulo  $3p$ . These numbers are converted to floating point by effectively multiplying the positive integer quantity

$$x_t \cup 1$$

by a scale factor slightly smaller than  $1./(\text{huge}(1))$ . The values  $p = 521$  and  $q = 32$  yield a sequence with a period approximately

$$2^p > 10^{156.8}$$

The default initial values for the sequence of integers  $\{x_t\}$  are created by a congruential generator starting with an odd integer seed

$$m = v + |count| \cap \left( 2^{\text{bit\_size}(1)} - 1 \right) \cup 1$$

obtained by the Fortran 90 real-time clock routine:

```
CALL SYSTEM_CLOCK(COUNT=count,CLOCK_RATE=CLRATE)
```

An error condition is noted if the value of  $\text{CLRATE}=0$ . This indicates that the processor does not have a functioning real-time clock. In this exceptional case a starting seed must be provided by the user with the optional argument “ $\text{i\_opt}=\text{”}$  and option number  $\text{?\_rand\_generator\_seed}$ . The value  $v$  is the current clock for this day, in milliseconds. This value is obtained using the date routine:

```
CALL DATE_AND_TIME(VALUES=values)
```

and converting  $\text{values}(5:8)$  to milliseconds.

The LCM generator initializes the sequence  $\{x_t\}$  using the following recurrence:

$$m \leftarrow m \times k, \text{ mod}(\text{huge}(1)/2)$$

The default value of  $k = 16807$ . Using the optional argument “ $\text{i\_opt}=\text{”}$  and the packaged option number  $\text{?\_rand\_gen\_LCM\_modulus}$ ,  $k$  can be given an alternate value. The option number  $\text{?\_rand\_gen\_generator\_seed}$  can be used to set the initial value of  $m$  instead of using the asynchronous value given by the system clock. This is illustrated in Example 2. If the default choice of  $m$  results in an unsatisfactory starting sequence or it is necessary to duplicate the

sequence, then it is recommended that users set the initial seed value to one of their own choosing. Resetting the seed complicates the usage of the routine.

This software is based on Fushimi (1990), who gives a more elaborate starting sequence for the {xt}. The starting sequence suggested by Fushimi can be used with the option number `?_rand_gen_use_Fushimi_start`. Fushimi's starting process is more expensive than the default method, and it is equivalent to starting in another place of the sequence with period  $2^p$ .

## Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `RAND_GEN`. These error messages are numbered 521–528; 541–548.

## Example 1: Running Mean and Variance

An array of random numbers is obtained. The sample mean and variance are computed. These values are compared with the same quantities computed using a stable method for the running means and variances, sequentially moving through the data. Details about the running mean and variance are found in Henrici (1982, pp. 21–23).

```
use rand_gen_int

implicit none

! This is Example 1 for RAND_GEN.

integer i
integer, parameter :: n=1000
real(kind(1e0)), parameter :: one=1e0, zero=0e0
real(kind(1e0)) x(n), mean_1(0:n), mean_2(0:n), s_1(0:n), s_2(0:n)

! Obtain random numbers.
call rand_gen(x)

! Calculate each partial mean.
do i=1,n
 mean_1(i) = sum(x(1:i))/i
end do

! Calculate each partial variance.
do i=1,n
 s_1(i)=sum((x(1:i)-mean_1(i))**2)/i
end do

mean_2(0)=zero
mean_2(1)=x(1)
s_2(0:1)=zero

! Alternately calculate each running mean and variance,
! handling the random numbers once.
do i=2,n
 mean_2(i)=((i-1)*mean_2(i-1)+x(i))/i
 s_2(i) = (i-1)*s_2(i-1)/i+(mean_2(i)-x(i))**2/(i-1)
end do

! Check that the two sets of means and variances agree.
```

```

if (maxval(abs(mean_1(1:)-mean_2(1:))/mean_1(1:)) <= &
 sqrt(epsilon(one))) then
 if (maxval(abs(s_1(2:)-s_2(2:))/s_1(2:)) <= &
 sqrt(epsilon(one))) then
 write (*,*) 'Example 1 for RAND_GEN is correct.'
 end if
end if

end

```

## Output

Example 1 for RAND\_GEN is correct.

## Additional Examples

### Example 2: Seeding, Using, and Restoring the Generator

```

use rand_gen_int

implicit none

! This is Example 2 for RAND_GEN.

integer i
integer, parameter :: n=34, p=521
real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
integer irndi(n), i_out(3*p+2), hidden_message(n)
real(kind(1e0)) x(n), y(n)
type(s_options) :: iopti(2)=s_options(0,zero)
character*34 message, returned_message

! This is the message to be hidden.
message = 'SAVE YOURSELF. WE ARE DISCOVERED!'

! Start the generator with a known seed.
iopti(1) = s_options(s_rand_gen_generator_seed,zero)
iopti(2) = s_options(123,zero)
call rand_gen(x, iopt=iopti)

! Save the state of the generator.
call rand_gen(x, istate_out=i_out)

! Get random integers.
call rand_gen(y, irnd=irndi)

! Hide text using collating sequence subtracted from integers.
do i=1, n
 hidden_message(i) = irndi(i) - ichar(message(i:i))
end do

! Reset generator to previous state and generate the previous
! random integers.
call rand_gen(x, irnd=irndi, istate_in=i_out)

```

```

! Subtract hidden text from integers and convert to character.
 do i=1, n
 returned_message(i:i) = char(irndi(i) - hidden_message(i))
 end do

! Check the results.
 if (returned_message == message) then

 write (*,*) 'Example 2 for RAND_GEN is correct.'
 end if

end

```

## Output

Example 2 for RAND\_GEN is correct.

### Example 3: Generating Strategy with a Histogram

We generate random integers but with the frequency as in a histogram with  $n_{bins}$  slots. The generator is initially used a large number of times to demonstrate that it is making choices with the same shape as the histogram. This is not required to generate samples. The program next generates a summary set of integers according to the histogram. These are not repeatable and are representative of the histogram in the sense of looking at 20 integers during generation of a large number of samples.

```

 use rand_gen_int
 use show_int

 implicit none

! This is Example 3 for RAND_GEN.

 integer i, i_bin, i_map, i_left, i_right
 integer, parameter :: n_work=1000
 integer, parameter :: n_bins=10
 integer, parameter :: scale=1000
 integer, parameter :: total_counts=100
 integer, parameter :: n_samples=total_counts*scale
 integer, dimension(n_bins) :: histogram= &
 (/4, 6, 8, 14, 20, 17, 12, 9, 7, 3 /)
 integer, dimension(n_work) :: working=0
 integer, dimension(n_bins) :: distribution=0
 integer break_points(0:n_bins)
 real(kind(1e0)) rn(n_samples)
 real(kind(1e0)), parameter :: tolerance=0.005

 integer, parameter :: n_samples_20=20
 integer rand_num_20(n_samples_20)
 real(kind(1e0)) rn_20(n_samples_20)

! Compute the normalized cumulative distribution.
 break_points(0)=0

```

```

do i=1,n_bins
 break_points(i)=break_points(i-1)+histogram(i)
end do

break_points=break_points*n_work/total_counts

! Obtain uniform random numbers.
 call rand_gen(rn)

! Set up the secondary mapping array.
do i_bin=1,n_bins
 i_left=break_points(i_bin-1)+1
 i_right=break_points(i_bin)
 do i=i_left, i_right
 working(i)=i_bin
 end do
end do

! Map the random numbers into the 'distribution' array.
! This is made approximately proportional to the histogram.
do i=1,n_samples
 i_map=nint(rn(i)*(n_work-1)+1)
 distribution(working(i_map))= &
 distribution(working(i_map))+1
end do

! Check the agreement between the distribution of the
! generated random numbers and the original histogram.
write (*, '(A)', advance='no') 'Original: '
write (*, '(10I6)') histogram*scale
write (*, '(A)', advance='no') 'Generated:'
write (*, '(10I6)') distribution

if (maxval(abs(histogram(1:)*scale-distribution(1:))) &
 <= tolerance*n_samples) then
 write(*, '(A/)') 'Example 3 for RAND_GEN is correct.'
end if

! Generate 20 integers in 1, 10 according to the distribution
! induced by the histogram.
 call rand_gen(rn_20)

! Map from the uniform distribution to the induced distribution.
do i=1,n_samples_20
 i_map=nint(rn_20(i)*(n_work-1)+1)
 rand_num_20(i)=working(i_map)
end do

 call show(rand_num_20,&
'Twenty integers generated according to the histogram:')
end

```

## Output

Example 3 for RAND\_GEN is correct.

#### Example 4: Generating with a Cosine Distribution

We generate random numbers based on the continuous distribution function

$$p(x) = (1 + \cos(x)) / 2\pi, -\pi \leq x \leq \pi$$

Using the cumulative

$$q(x) = \int_{-\pi}^x p(t) dt = 1/2 + (x + \sin(x)) / 2\pi$$

we generate the samples by obtaining uniform samples  $u$ ,  $0 < u < 1$  and solve the equation

$$q(x) - u = 0, -\pi < x < \pi$$

These are evaluated in vector form, that is all entries at one time, using Newton's method:

$$x \leftarrow x - dx, dx = (q(x) - u) / p(x)$$

An iteration counter forces the loop to terminate, but this is not often required although it is an important detail.

```
use rand_gen_int
use show_int
use Numerical_Libraries

IMPLICIT NONE

! This is Example 4 for RAND_GEN.

integer i, i_map, k
integer, parameter :: n_bins=36
integer, parameter :: offset=18
integer, parameter :: n_samples=10000
integer, parameter :: n_samples_30=30
integer, parameter :: COUNT=15

real(kind(1e0)) probabilities(n_bins)
real(kind(1e0)), dimension(n_bins) :: counts=0.0
real(kind(1e0)), dimension(n_samples) :: rn, x, f, fprime, dx
real(kind(1e0)), dimension(n_samples_30) :: rn_30, &
 x_30, f_30, fprime_30, dx_30
real(kind(1e0)), parameter :: one=1e0, zero=0e0, half=0.5e0
real(kind(1e0)), parameter :: tolerance=0.01
real(kind(1e0)) two_pi, omega

! Initialize values of 'two_pi' and 'omega'.
two_pi=2.0*const(('/pi'/))
omega=two_pi/n_bins

! Compute the probabilities for each bin according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
do i=1,n_bins
 probabilities(i)=(sin(omega*(i-offset)) &
```

```

 -sin(omega*(i-offset-1))+omega)/two_pi
 end do

! Obtain uniform random numbers in (0,1).
 call rand_gen(rn)

! Use Newton's method to solve the nonlinear equation:
! accumulated_distribution_function - random_number = 0.
 x=zero; k=0
 solve_equation: do
 f=(sin(x)+x)/two_pi+half-rn
 fprime=(one+cos(x))/two_pi
 dx=f/fprime
 x=x-dx; k=k+1
 if (maxval(abs(dx)) <= sqrt(epsilon(one)) &
 .or. k > COUNT) exit solve_equation
 end do solve_equation

! Map the random numbers 'x' array into the 'counts' array.
 do i=1,n_samples
 i_map=int(x(i)/omega+offset)+1
 counts(i_map)=counts(i_map)+one
 end do

! Normalize the counts array.
 counts=counts/n_samples

! Check that the generated random numbers are indeed
! based on the original distribution.
 if (maxval(abs(counts(1:)-probabilities(1:))) &
 <= tolerance) then
 write(*,'(a/)') 'Example 4 for RAND_GEN is correct.'
 end if

! Generate 30 random numbers in (-pi,pi) according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
 call rand_gen(rn_30)

 x_30=0.0; k=0
 solve_equation_30: do
 f_30=(sin(x_30)+x_30)/two_pi+half-rn_30
 fprime_30=(one+cos(x_30))/two_pi
 dx_30=f_30/fprime_30
 x_30=x_30-dx_30
 if (maxval(abs(dx_30)) <= sqrt(epsilon(one)) &
 .or. k > COUNT) exit solve_equation_30
 end do solve_equation_30

 write(*,'(A)') 'Thirty random numbers generated ', &
 'according to the probability density ', &
 'pdf(x)=(cos(x)+1)/(2*pi), -pi<x<pi:'

 call show(x_30)
end

```

## Output

Example 4 for RAND\_GEN is correct.

---

# RNGET

Retrieves the current value of the seed used in the IMSL random number generators.

## Required Arguments

**ISEED** — The seed of the random number generator. (Output)

ISEED is in the range (1, 2147483646).

## FORTRAN 90 Interface

Generic:    CALL RNGET (ISEED)

Specific:   The specific interface name is RNGET.

## FORTRAN 77 Interface

Single:     CALL RNGET (ISEED)

## Description

Routine RNGET retrieves the current value of the “seed” used in the IMSL random number generators. A reason for doing this would be to restart a simulation, using RNSSET to reset the seed.

## Example

The following FORTRAN statements illustrate the use of RNGET:

```
 INTEGER ISEED
! CALL RNSET(123457) Call RNSET to initialize the seed.
! CALL RNSET(123457)
! Do some simulations.
! ...
! CALL RNGET(ISEED)
! Save ISEED. If the simulation is to be continued
! in a different program, ISEED should be output,
! possibly to a file.
! ...
! ...
! When the simulations begun above are to be
! restarted, restore ISEED to the value obtained
! above and use as input to RNSET.
! CALL RNSET(ISEED)
! Now continue the simulations.
! ...
! ...
```

---

# RNSET

Initializes a random seed for use in the IMSL random number generators.

## Required Arguments

*ISEED* — The seed of the random number generator. (Input)

*ISEED* must be in the range (0, 2147483646). If *ISEED* is zero, a value is computed using the system clock; and, hence, the results of programs using the IMSL random number generators will be different at different times.

## FORTRAN 90 Interface

Generic:     CALL RNSET (ISEED)

Specific:    The specific interface name is RNSET .

## FORTRAN 77 Interface

Single:     CALL RNSET (ISEED)

## Description

Routine `RNSET` is used to initialize the seed used in the IMSL random number generators. If the seed is not initialized prior to invocation of any of the routines for random number generation by calling `RNSET`, the seed is initialized via the system clock. The seed can be reinitialized to a clock-dependent value by calling `RNSET` with *ISEED* set to 0.

The effect of `RNSET` is to set some values in a FORTRAN `COMMON` block that is used by the random number generators.

A common use of `RNSET` is in conjunction with `RNGET` to restart a simulation.

## Example

The following FORTRAN statements illustrate the use of `RNSET`:

```
 INTEGER ISEED
! Call RNSET to initialize the seed via the
! system clock.
 CALL RNSET(0)
! Do some simulations.
 ...
 ...
! Obtain the current value of the seed.
 CALL RNGET(ISEED)
! If the simulation is to be continued in a
! different program, ISEED should be output,
! possibly to a file.
 ...
 ...
```

```

! When the simulations begun above are to be
! restarted, restore ISEED to the value
! obtained above, and use as input to RNSET.
 CALL RNSET(ISEED)
! Now continue the simulations.
 ...
 ...

```

---

## RNOPT

Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.

### Required Arguments

**IOPT** — Indicator of the generator. (Input)

The random number generator is either a multiplicative congruential generator with modulus  $2^{31} - 1$  or a GFSR generator. IOPT is used to choose the multiplier and whether or not shuffling is done, or is used to choose the GFSR method, or is used to choose the Mersenne Twister generator.

#### IOPT Generator

- 1 The multiplier 16807 is used.
- 2 The multiplier 16807 is used with shuffling.
- 3 The multiplier 397204094 is used.
- 4 The multiplier 397204094 is used with shuffling.
- 5 The multiplier 950706376 is used.
- 6 The multiplier 950706376 is used with shuffling.
- 7 GFSR, with the recursion  $X_t = X_{t-1563} \oplus X_{t-96}$  is used.
- 8 A 32-bit Mersenne Twister generator is used. The real and double random numbers are generated from 32-bit integers.
- 9 A 64-bit Mersenne Twister generator is used. The real and double random numbers are generated from 64-bit integers. This ensures that all bits of both float and double are random.

### FORTRAN 90 Interface

Generic:    CALL RNOPT (IOPT)

Specific:   The specific interface name is RNOPT.

## FORTRAN 77 Interface

Single:      `CALL RNOPT (IOPT)`

### Description

The uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling or a GFSR method, or the Mersenne Twister method. Routine `RNOPT` determines which method is used; and in the case of a multiplicative congruential method, it determines the value of the multiplier and whether or not to use shuffling. The description of `RNUN` may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (see Lewis, Goodman, and Miller, 1969). This is the generator formerly known as `GGUBS` in the IMSL Library. It is the “minimal standard generator” discussed by Park and Miller (1988).

Both of the Mersenne Twister generators have a period of  $2^{19937} - 1$  and a 624-dimensional equi-distribution property. See Matsumoto et al. 1998 for details.

The IMSL Mersenne Twister generators are derived from code copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved. It is subject to the following notice:

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The IMSL 32-bit Mersenne Twister generator is based on the Matsumoto and Nishimura code ‘mt19937ar’ and the 64-bit code is based on ‘mt19937-64’.

### Example

The FORTRAN statement

```
CALL RNOPT(1)
```

would select the simple multiplicative congruential generator with multiplier 16807. Since this is the same as the default, this statement would have no effect unless `RNOPT` had previously been called in the same program to select a different generator.

---

## RNIN32

Initializes the 32-bit Mersenne Twister generator using an array.

## Required Arguments

**KEY**— Integer array of length `LEN` used to initialize the 32-bit Mersenne Twister generator.  
(Input)

## Optional Arguments

**LEN** — Length of the array key. (Input)

## FORTRAN 90 Interface

Generic:    `CALL RNIN32 (KEY [, ...])`

Specific:   The specific interface name is `S_RNIN32`.

## FORTRAN 77 Interface

Single:     `CALL RNIN32 (KEY, LEN)`

## Description

By default, the Mersenne Twister random number generator is initialized using the current seed value (see [RNGET](#)). The seed is limited to one integer for initialization. This function allows an arbitrary length array to be used for initialization. This subroutine completely replaces the use of the seed for initialization of the 32-bit Mersenne Twister generator.

## Example

See routine [RNGE32](#).

---

# RNGE32

Retrieves the current table used in the 32-bit Mersenne Twister generator.

## Required Arguments

**MTABLE** — Integer array of length 625 containing the table used in the 32-bit Mersenne Twister generator. (Output)

## FORTRAN 90 Interface

Generic:    `CALL RNGE32 (MTABLE)`

Specific:   The specific interface name is `RNGE32`

## FORTRAN 77 Interface

Single:     `CALL RNGE32 (MTABLE)`

## Description

The values in the table contain the state of the 32-bit Mersenne Twister random number generator. The table can be used by RNSE32 to set the generator back to this state.

## Example

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore, the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
USE RNIN32_INT
USE RNSE32_INT
USE RNSET_INT
USE UMACH_INT
USE RNUN_INT
IMPLICIT NONE
INTEGER I, ISEED, NOUT
INTEGER INIT(4)
DATA INIT/291,564,837,1110/
DATA ISEED/123457/
INTEGER NR
REAL R(5)
INTEGER MTABLE(625)
CHARACTER CLABEL(5)*5, FMT*8, RLABEL(3)*5
RLABEL(1)='NONE'
CLABEL(1)='NONE'
DATA FMT/'(W10.4)'/
NR=5
CALL UMACH(2, NOUT)
ISEED = 123457
CALL RNOPT(8)
CALL RNSET(ISEED)
CALL RNUN(R)
CALL WRRRL('FIRST STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! REINITIALIZE MERSENNE TWISTER SERIES WITH AN ARRAY
CALL RNIN32(INIT)
! SAVE THE STATE OF THE SERIES
CALL RNSE32(MTABLE)
CALL RNUN(R)
CALL WRRRL('SECOND STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! RESTORE THE STATE OF THE TABLE
CALL RNSE32(MTABLE)
CALL RNUN(R)
CALL WRRRL('THIRD STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! RESET THE SERIES - IT WILL REINITIALIZE FROM THE SEED
MTABLE(1)=1000
CALL RNSE32(MTABLE)
```

```

CALL RNUN(R)
CALL WRRRL('FOURTH STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
END

```

## Output

|        |        |                      |        |        |  |
|--------|--------|----------------------|--------|--------|--|
|        |        | First stream output  |        |        |  |
| 0.4347 | 0.3522 | 0.0139               | 0.2091 | 0.4956 |  |
|        |        | Second stream output |        |        |  |
| 0.2486 | 0.2226 | 0.1111               | 0.9563 | 0.9846 |  |
|        |        | Third stream output  |        |        |  |
| 0.2486 | 0.2226 | 0.1111               | 0.9563 | 0.9846 |  |
|        |        | Fourth stream output |        |        |  |
| 0.4347 | 0.3522 | 0.0139               | 0.2091 | 0.4956 |  |

---

## RNSE32

Sets the current table used in the 32-bit Mersenne Twister generator.

### Required Arguments

*MTABLE* — Integer array of length 625 containing the table used in the 32-bit Mersenne Twister generator. (Input)

### FORTRAN 90 Interface

Generic:    CALL RNSE32 (MTABLE)

Specific:   The specific interface name is RNSE32

### FORTRAN 77 Interface

Single:     CALL RNSE32 (MTABLE)

### Description

The values in *MTABLE* are the state of the 32-bit Mersenne Twister random number generator obtained by a call to *RNGE32*. The values in the table can be used to restore the state of the generator.

Alternatively, if *MTABLE* [1] > 625 then the generator is set to its original, uninitialized, state.

### Example

See routine [RNGE32](#).

---

## RNIN64

Initializes the 64-bit Mersenne Twister generator using an array.

## Required Arguments

**KEY**— Integer(kind=8) array of length `LEN` used to initialize the 64-bit Mersenne Twister generator. (Input)

## Optional Arguments

**LEN** — Length of the array key. (Input)

## FORTRAN 90 Interface

Generic:     `CALL RNIN64 (KEY [, ...])`

Specific:    The specific interface name is `S_RNIN64`.

## FORTRAN 77 Interface

Single:      `CALL RNIN64 (KEY, LEN)`

## Description

By default, the Mersenne Twister random number generator is initialized using the current seed value (see [RNGET](#)). The seed is limited to one integer for initialization. This function allows an arbitrary length array to be used for initialization. This subroutine completely replaces the use of the seed for initialization of the 64-bit Mersenne Twister generator.

---

# RNGE64

Retrieves the current table used in the 64-bit Mersenne Twister generator.

## Required Arguments

**MTABLE** — Integer(kind=8) array of length 313 containing the table used in the 64-bit Mersenne Twister generator. (Output)

## FORTRAN 90 Interface

Generic:     `CALL RNGE64 (MTABLE)`

Specific:    The specific interface name is `RNGE64`

## FORTRAN 77 Interface

Single:      `CALL RNGE64 (MTABLE)`

## Description

The values in the table contain the state of the 64-bit Mersenne Twister random number generator. The table can be used by `RNSE64` to set the generator back to this state.

## Example

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore, the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
USE RNIN64 _INT
USE RNGE64 _INT
USE RNSET _INT
USE UMACH _INT
USE RNUN _INT
IMPLICIT NONE
INTEGER I, ISEED, NOUT
INTEGER(KIND=8) INIT(4)
DATA INIT/291,564,837,1110/
DATA ISEED/123457/
INTEGER NR
REAL R(5)
INTEGER(KIND=8) MTABLE(313)
CHARACTER CLABEL(5)*5, FMT*8, RLABEL(3)*5
RLABEL(1)='NONE'
CLABEL(1)='NONE'
DATA FMT/'(W10.4)'/
NR=5
CALL UMACH(2, NOUT)
ISEED = 123457
CALL RNOPT(9)
CALL RNSET(ISEED)
CALL RNUN(R)
CALL WRRRL('FIRST STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! REINITIALIZE MERSENNE TWISTER SERIES WITH AN ARRAY
CALL RNIN64(INIT)
! SAVE THE STATE OF THE SERIES
CALL RNGE64(MTABLE)
CALL RNUN(R)
CALL WRRRL('SECOND STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! RESTORE THE STATE OF THE TABLE
CALL RNSE64(MTABLE)
CALL RNUN(R)
CALL WRRRL('THIRD STREAM OUTPUT',1,5,R,1,0, &
 FMT, RLABEL, CLABEL)
! RESET THE SERIES - IT WILL REINITIALIZE FROM THE SEED
MTABLE(1)=1000
CALL RNSE64(MTABLE)
CALL RNUN(R)
CALL WRRRL('FOURTH STREAM OUTPUT',1,5,R,1,0, &
```

```
END FMT, RLABEL, CLABEL)
```

## Output

```
 First stream output
0.5799 0.9401 0.7102 0.1640 0.5457
 Second stream output
0.4894 0.7397 0.5725 0.0863 0.7588
 Third stream output
0.4894 0.7397 0.5725 0.0863 0.7588
 Fourth stream output
0.5799 0.9401 0.7102 0.1640 0.5457
```

---

## RNSE64

Sets the current table used in the 64-bit Mersenne Twister generator.

### Required Arguments

*MTABLE* — Integer (kind=8) array of length 313 containing the table used in the 64-bit Mersenne Twister generator. (Input)

### FORTRAN 90 Interface

Generic:    CALL RNSE64 (MTABLE)

Specific:   The specific interface name is RNSE64

### FORTRAN 77 Interface

Single:     CALL RNSE64 (MTABLE)

### Description

The values in *MTABLE* are the state of the 64-bit Mersenne Twister random number generator obtained by a call to *RNGE64*. The values in the table can be used to restore the state of the generator. Alternatively, if *MTABLE* [1] > 313 then the generator is set to its original, uninitialized, state.

### Example

See function [RNGE64](#).

---

## RNUNF

This function generates a pseudorandom number from a uniform (0, 1) distribution.

## Function Return Value

*RNUNF* — Function value, a random uniform (0, 1) deviate. (Output)  
See Comment 1.

## Required Arguments

None

## FORTRAN 90 Interface

Generic: `RNUNF ( )`

Specific: The specific interface names are `S_RNUNF` and `D_RNUNF`.

## FORTRAN 77 Interface

Single: `RNUNF ( )`

Double: The double precision name is `DRNUNF`.

## Description

Routine `RNUNF` is the function form of `RNUN`. The routine `RNUNF` generates pseudorandom numbers from a uniform (0, 1) distribution. The algorithm used is determined by `RNOPT`. The values returned by `RNUNF` are positive and less than 1.0.

If several uniform deviates are needed, it may be more efficient to obtain them all at once by a call to `RNUN` rather than by several references to `RNUNF`.

## Comments

1. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

```
X = RNUNF (6)
Y = SQRT (X)
```

must be used rather than

```
Y = SQRT (RNUNF (6))
```

If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. Routine `RNSET` can be used to initialize the seed of the random number generator. The routine `RNOPT` can be used to select the form of the generator.
3. This function has a side effect: it changes the value of the seed, which is passed through a common block.

## Example

In this example, `RNUNF` is used to generate five pseudorandom uniform numbers. Since `RNOPT` is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```
 USE RNUNF_INT
 USE RNSET_INT
 USE UMACH_INT

 IMPLICIT NONE
 INTEGER I, ISEED, NOUT
 REAL R(5)
!
 CALL UMACH (2, NOUT)
 ISEED = 123457
 CALL RNSET (ISEED)
 DO 10 I=1, 5
 R(I) = RNUNF()
10 CONTINUE
 WRITE (NOUT,99999) R
99999 FORMAT (' Uniform random deviates: ', 5F8.4)
 END
```

## Output

```
Uniform random deviates: 0.9662 0.2607 0.7663 0.5693 0.8448
```

---

# RNUN

Generates pseudorandom numbers from a uniform (0, 1) distribution.

## Required Arguments

*R* — Vector of length `NR` containing the random uniform (0, 1) deviates. (Output)

## Optional Arguments

*NR* — Number of random numbers to generate. (Input)

Default: `NR = SIZE (R,1)`.

## FORTRAN 90 Interface

Generic:    `CALL RNUN (R [, ...])`

Specific:    The specific interface names are `S_RNUN` and `D_RNUN`.

## FORTRAN 77 Interface

Single:    `CALL RNUN (NR, R)`

Double:      The double precision name is DRNUN.

## Description

Routine RNUN generates pseudorandom numbers from a uniform (0,1) distribution using either a multiplicative congruential method or a generalized feedback shift register (GFSR) method, or the Mersenne Twister generator. The form of the multiplicative congruential generator is

$$x_i \equiv cx_{i-1} \bmod(2^{31} - 1)$$

Each  $x_i$  is then scaled into the unit interval (0,1). The possible values for  $c$  in the IMSL generators are 16807, 397204094, and 950706376. The selection is made by the routine RNOPT. The choice of 16807 will result in the fastest execution time. If no selection is made explicitly, the routines use the multiplier 16807.

The user can also select a shuffled version of the multiplicative congruential generators. In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each  $x_i$  from the simple generator, the low-order bits of  $x_i$  are used to select a random integer,  $j$ , from 1 to 128. The  $j$ -th entry in the table is then delivered as the random number; and  $x_i$ , after being scaled into the unit interval, is inserted into the  $j$ -th position in the table.

The GFSR method is based on the recursion  $X_t = X_{t-1563} \oplus X_{t-96}$ . This generator, which is different from earlier GFSR generators, was proposed by Fushimi (1990), who discusses the theory behind the generator and reports on several empirical tests of it.

Mersenne Twister(MT) is a pseudorandom number generating algorithm developed by Makoto Matsumoto and Takuji Nishimura in 1996-1997. MT has far longer period and far higher order of equidistribution than any other implemented generators. The values returned in R by RNUN are positive and less than 1.0. Values in R may be smaller than the smallest relative spacing, however. Hence, it may be the case that some value  $R(i)$  is such that  $1.0 - R(i) = 1.0$ .

Deviates from the distribution with uniform density over the interval (A, B) can be obtained by scaling the output from RNUN. The following statements (in single precision) would yield random deviates from a uniform (A, B) distribution:

```
CALL RNUN (NR, R)
CALL SSCAL (NR, B-A, R, 1)
CALL SADD (NR, A, R, 1)
```

## Comments

The routine RNSET can be used to initialize the seed of the random number generator. The routine RNOPT can be used to select the form of the generator.

## Example

In this example, RNUN is used to generate five pseudorandom uniform numbers. Since RNOPT is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```

USE RNUN_INT
USE RNSET_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER ISEED, NOUT, NR
REAL R(5)
!
CALL UMACH (2, NOUT)
NR = 5
ISEED = 123457
CALL RNSET (ISEED)
CALL RNUN (R)
WRITE (NOUT,99999) R
99999 FORMAT (' Uniform random deviates: ', 5F8.4)
END

```

## Output

```
Uniform random deviates: 0.9662 0.2607 0.7663 0.5693 0.8448
```

---

## FAURE\_INIT

Shuffled Faure sequence initialization.

### Required Arguments

**NDIM** — The dimension of the hyper-rectangle. (Input)

**STATE** — An `IMSL_FAURE` pointer for the derived type created by the call to `FAURE_INIT`. The output contains information about the sequence. Use `?_IMSL_FAURE` as the type, where `?_` is `S_` or `D_` depending on precision. (Output)

### Optional Arguments

**NBASE** — The base of the Faure sequence. (Input)

Default: The smallest prime number greater than or equal to `NDIM`.

**NSKIP** — The number of points to be skipped at the beginning of the Faure sequence. (Input)

Default:  $\lfloor \text{base}^{m/2-1} \rfloor$ , where  $m = \lfloor \log B / \log \text{base} \rfloor$  and  $B$  is the largest machine representable integer.

### FORTRAN 90 Interface

Generic: `CALL FAURE_INIT (NDIM, STATE [, ...])`

Specific: The specific interface names are `S_FAURE_INIT` and `D_FAURE_INIT`.

---

## FAURE\_FREE

Frees the structure containing information about the Faure sequence.

### Required Arguments

*STATE* — An `IMSL_FAURE` pointer containing the structure created by the call to `FAURE_INIT`. (Input/Output)

### FORTRAN 90 Interface

Generic: `CALL FAURE_FREE (STATE)`

Specific: The specific interface names are `S_FAURE_FREE` and `D_FAURE_FREE`.

---

## FAURE\_NEXT

Computes a shuffled Faure sequence.

### Required Arguments

*STATE* — An `IMSL_FAURE` pointer containing the structure created by the call to `FAURE_INIT`. The structure contains information about the sequence. The structure should be freed using `FAURE_FREE` after it is no longer needed. (Input/Output)

*NEXT\_PT* — Vector of length `NDIM` containing the next point in the shuffled Faure sequence, where `NDIM` is the dimension of the hyper-rectangle specified in `FAURE_INIT`. (Output)

### Optional Arguments

*IMSL\_RETURN\_SKIP* — Returns the current point in the sequence. The sequence can be restarted by calling `FAURE_INIT` using this value for `NSKIP`, and using the same value for `NDIM`. (Input)

### FORTRAN 90 Interface

Generic: `CALL FAURE_NEXT (STATE, NEXT_PT [, ...])`

Specific: The specific interface names are `S_FAURE_NEXT` and `D_FAURE_NEXT`.

### Description

The routines `FAURE_INIT` and `FAURE_NEXT` are used to generate shuffled Faure sequence of low discrepancy  $n$ -dimensional points. Low discrepancy series fill an  $n$ -dimensional cube more

uniformly than psuedo-random sequences, and are used in multivariate quadrature, simulation, and global optimization. Because of this uniformity, use of low discrepancy series is generally more efficient than psuedo-random series for multivariate Monte Carlo methods. See the IMSL routine QMC (Chapter 4, Integration and Differentiation) for a discussion of quasi-Monte Carlo quadrature based on low discrepancy series.

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set  $x_1, \dots, x_n \in [0, 1]^d$ ,  $d \geq 1$ , is defined

$$D_n^{(d)} = \sup_E \left| \frac{A(E; n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of  $[0, 1]^d$  of the form

$$E = \left[ 0, t_1 \right) \times \dots \times \left[ 0, t_d \right), \quad 0 \leq t_j \leq 1, \quad 1 \leq j \leq d,$$

$\lambda$  is the Lebesque measure, and  $A(E; n)$  is the number of the  $x_j$  contained in  $E$ .

The sequence  $x_1, x_2, \dots$  of points  $[0, 1]^d$  is a low-discrepancy sequence if there exists a constant  $c(d)$ , depending only on  $d$ , such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all  $n > 1$ .

Generalized Faure sequences can be defined for any prime base  $b \geq d$ . The lowest bound for the discrepancy is obtained for the smallest prime  $b \geq d$ , so the optional argument NBASE defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence  $x_1, x_2, \dots$ , is computed as follows:

Write the positive integer  $n$  in its  $b$ -ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where  $a_i(n)$  are integers,  $0 \leq a_i(n) < b$ .

The  $j$ -th coordinate of  $x_n$  is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n) b^{-k-1}, \quad 1 \leq j \leq d$$

The generator matrix for the series,  $c_{kd}^{(j)}$ , is defined to be

$$c_{kd}^{(j)} = j^{d-k} c_{kd}$$

and  $c_{kd}$  is an element of the Pascal matrix,

$$c_{kd} = \begin{cases} \frac{d!}{c!(d-c)!} & k \leq d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the  $b$ -ary Gray code. The function  $G(n)$  maps the positive integer  $n$  into the integer given by its  $b$ -ary expansion.

The sequence computed by this function is  $x(G(n))$ , where  $x$  is the generalized Faure sequence.

### Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that `FAURE_INIT` is used to create a structure that holds the state of the sequence. Each call to `FAURE_NEXT` returns the next point in the sequence and updates the `IMSL_FAURE` structure. The final call to `FAURE_FREE` frees data items, stored in the structure, that were allocated by `FAURE_INIT`.

```

use faure_int
implicit none
type (s_imsl_faure), pointer :: state
real(kind(1e0)) :: x(3)
integer,parameter :: ndim=3
integer :: k
!
! CREATE THE STRUCTURE THAT HOLDS
! THE STATE OF THE SEQUENCE.
call faure_init(ndim, state)
! GET THE NEXT POINT IN THE SEQUENCE
do k=1,5
 call faure_next(state, x)
 write(*,'(3F15.3)') x(1), x(2), x(3)
enddo
! FREE DATA ITEMS STORED IN
! state STRUCTURE
call faure_free(state)
end

```

### Output

|       |       |       |
|-------|-------|-------|
| 0.334 | 0.493 | 0.064 |
| 0.667 | 0.826 | 0.397 |
| 0.778 | 0.270 | 0.175 |

|       |       |       |
|-------|-------|-------|
| 0.111 | 0.604 | 0.509 |
| 0.445 | 0.937 | 0.842 |

---

## IUMAG

This routine handles MATH/LIBRARY and STAT/LIBRARY type INTEGER options.

### Required Arguments

*PRODNM* — Product name. Use either “MATH” or “STAT.” (Input)

*ICHP* — Chapter number of the routine that uses the options. (Input)

*IACT* — 1 if user desires to “get” or read options, or 2 if user desires to “put” or write options. (Input)

*NUMOPT* — Size of IOPTS. (Input)

*IOPTS* — Integer array of size NUMOPT containing the option numbers to “get” or “put.” (Input)

*IVALS* — Integer array containing the option values. These values are arrays corresponding to the individual options in IOPTS in sequential order. The size of IVALS is the sum of the sizes of the individual options. (Input/Output)

### FORTRAN 90 Interface

Generic:     CALL IUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, IVALS)

Specific:    The specific interface name is IUMAG.

### FORTRAN 77 Interface

Single:     CALL IUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, IVALS)

### Description

The Options Manager routine IUMAG reads or writes INTEGER data for some MATH/LIBRARY and STAT/LIBRARY codes. See Atchison and Hanson (1991) for more complete details.

There are MATH/LIBRARY routines in Chapters 1, 2, and 5 that now use IUMAG to communicate optional data from the user.

### Comments

1. Users can normally avoid reading about options when first using a routine that calls IUMAG.

2. Let  $I$  be any value between 1 and `NUMOPT`. A negative value of `IOPTS(I)` refers to option number  $-IOPTS(I)$  but with a different effect: For a “get” operation, the default values are returned in `IVALS`. For a “put” operation, the default values replace the current values. In the case of a “put,” entries of `IVALS` are not allocated by the user and are not used by `IUMAG`.
3. Both positive and negative values of `IOPTS` can be used.
4. INTEGER Options
  - 1 If the value is positive, print the next activity for any library routine that uses the Options Manager codes `IUMAG`, `SUMAG`, or `DUMAG`. Each printing step decrements the value if it is positive.  
Default value is 0.
  - 2 If the value is 2, perform error checking in `IUMAG`, `SUMAG`, and `DUMAG` such as the verifying of valid option numbers and the validity of input data. If the value is 1, do not perform error checking.  
Default value is 2.
  - 3 This value is used for testing the installation of `IUMAG` by other IMSL software.  
Default value is 3.

### Example

The number of iterations allowed for the constrained least squares solver `LCLSQ` that calls `L2LSQ` is changed from the default value of  $\max(nra, nca)$  to the value 6. The default value is restored after the call to `LCLSQ`. This change has no effect on the solution. It is used only for illustration. The first two arguments required for the call to `IUMAG` are defined by the product name, “MATH,” and chapter number, 1, where `LCLSQ` is documented. The argument `IACT` denotes a write or “put” operation. There is one option to change so `NUMOPT` has the value 1. The arguments for the option number, 14, and the new value, 6, are defined by reading the documentation for `LCLSQ`.

```

USE IUMAG_INT
USE LCLSQ_INT
USE UMACH_INT
USE SNRM2_INT

IMPLICIT NONE

!
! Solve the following in the least squares sense:
! 3x1 + 2x2 + x3 = 3.3
! 4x1 + 2x2 + x3 = 2.3
! 2x1 + 2x2 + x3 = 1.3
! x1 + x2 + x3 = 1.0
!
! Subject to: x1 + x2 + x3 <= 1
! 0 <= x1 <= .5
! 0 <= x2 <= .5
! 0 <= x3 <= .5
!
! -----

```

```

! Declaration of variables
!
INTEGER ICHP, IPUT, LDA, LDC, MCON, NCA, NEWMAX, NRA, NUMOPT
PARAMETER (ICHP=1, IPUT=2, MCON=1, NCA=3, NEWMAX=14, NRA=4, &
 NUMOPT=1, LDA=NRA, LDC=MCON)
!
INTEGER IOPT(1), IRTYPE(MCON), IVAL(1), NOUT
REAL A(LDA,NCA), B(NRA), BC(MCON), C(LDC,NCA), RES(NRA), &
 RESNRM, XLB(NCA), XSOL(NCA), XUB(NCA)
! Data initialization
!
DATA A/3.0E0, 4.0E0, 2.0E0, 1.0E0, 2.0E0, 2.0E0, 2.0E0, 1.0E0, &
 1.0E0, 1.0E0, 1.0E0, 1.0E0/, B/3.3E0, 2.3E0, 1.3E0, 1.0E0/, &
 C/3*1.0E0/, BC/1.0E0/, IRTYPE/1/, XLB/3*0.0E0/, XUB/3*.5E0/
! -----
!
! Reset the maximum number of
!
! iterations to use in the solver.
! The value 14 is the option number.
! The value 6 is the new maximum.
IOPT(1) = NEWMAX
IVAL(1) = 6
CALL IUMAG ('math', ICHP, IPUT, NUMOPT, IOPT, IVAL)
! -----
!
! Solve the bounded, constrained
! least squares problem.
!
CALL LCLSQ (A, B, C, BC, B, IRTYPE, XLB, XUB, XSOL, RES=RES)
!
! Compute the 2-norm of the residuals.
RESNRM = SNRM2(NRA,RES,1)
!
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) XSOL, RES, RESNRM
! -----
!
! Reset the maximum number of
! iterations to its default value.
! This is not required but is
! recommended programming practice.
IOPT(1) = -IOPT(1)
CALL IUMAG ('math', ICHP, IPUT, NUMOPT, IOPT, IVAL)
! -----
!
99999 FORMAT (' The solution is ', 3F9.4, '//, ' The residuals ', &
 'evaluated at the solution are ',/, 18X, 4F9.4, '//, &
 ' The norm of the residual vector is ', F8.4)
!
END

```

## Output

```
The solution is 0.5000 0.3000 0.2000

The residuals evaluated at the solution are
 -1.0000 0.5000 0.5000 0.0000

The norm of the residual vector is 1.2247
```

---

## UMAG

This routine handles MATH/LIBRARY and STAT/LIBRARY type REAL and double precision options.

### Required Arguments

*PRODNM* — Product name. Use either “MATH” or “STAT.” (Input)

*ICHP* — Chapter number of the routine that uses the options. (Input)

*IACT* — 1 if user desires to “get” or read options, or 2 if user desires to “put” or write options. (Input)

*IOPTS* — Integer array of size NUMOPT containing the option numbers to “get” or “put.” (Input)

*SVALS* — Array containing the option values. These values are arrays corresponding to the individual options in IOPTS in sequential order. The size of SVALS is the sum of the sizes of the individual options. (Input/Output)

### Optional Arguments

*NUMOPT* — Size of IOPTS. (Input)  
Default: NUMOPT = SIZE (IOPTS,1).

### FORTRAN 90 Interface

Generic: CALL UMAG (PRODNM, ICHP, IACT, IOPTS, SVALS [, ...])

Specific: The specific interface names are S\_UMAG and D\_UMAG.

### FORTRAN 77 Interface

Single: CALL SUMAG (PRODNM, ICHP, IACT, NUMOPT, IOPTS, SVALS)

Double: The double precision name is DUMAG.

## Description

The Options Manager routine `SUMAG` reads or writes `REAL` data for some `MATH/LIBRARY` and `STAT/LIBRARY` codes. See Atchison and Hanson (1991) for more complete details. There are `MATH/LIBRARY` routines in Chapters 1 and 5 that now use `SUMAG` to communicate optional data from the user.

## Comments

1. Users can normally avoid reading about options when first using a routine that calls `SUMAG`.
2. Let `I` be any value between 1 and `NUMOPT`. A negative value of `IOPTS(I)` refers to option number `-IOPTS(I)` but with a different effect: For a “get” operation, the default values are returned in `SVALS`. For a “put” operation, the default values replace the current values. In the case of a “put,” entries of `SVALS` are not allocated by the user and are not used by `SUMAG`.
3. Both positive and negative values of `IOPTS` can be used.
4. Floating Point Options
  - 1 This value is used for testing the installation of `SUMAG` by other IMSL software. Default value is `3.0E0`.

## Example

The rank determination tolerance for the constrained least squares solver `LCLSQ` that calls `L2LSQ` is changed from the default value of `SQRT(AMACH(4))` to the value 0.01. The default value is restored after the call to `LCLSQ`. This change has no effect on the solution. It is used only for illustration. The first two arguments required for the call to `SUMAG` are defined by the product name, “`MATH`,” and chapter number, 1, where `LCLSQ` is documented. The argument `IACT` denotes a write or “put” operation. There is one option to change so `NUMOPT` has the value 1. The arguments for the option number, 2, and the new value, `0.01E+0`, are defined by reading the documentation for `LCLSQ`.

```
USE UMAG_INT
USE LCLSQ_INT
USE UMACH_INT
USE SNRM2_INT

IMPLICIT NONE

!
! Solve the following in the least squares sense:
! 3x1 + 2x2 + x3 = 3.3
! 4x1 + 2x2 + x3 = 2.3
! 2x1 + 2x2 + x3 = 1.3
! x1 + x2 + x3 = 1.0
!
! Subject to: x1 + x2 + x3 <= 1
! 0 <= x1 <= .5
```

```

! 0 <= x2 <= .5
! 0 <= x3 <= .5
!
!-----
!
! Declaration of variables
!
INTEGER ICHP, IPUT, LDA, LDC, MCON, NCA, NEWTOL, NRA, NUMOPT
PARAMETER (ICHP=1, IPUT=2, MCON=1, NCA=3, NEWTOL=2, NRA=4, &
 NUMOPT=1, LDA=NRA, LDC=MCON)
!
INTEGER IOPT(1), IRTYPE(MCON), NOUT
REAL A(LDA,NCA), B(NRA), BC(MCON), C(LDC,NCA), RES(NRA), &
 RESNRM, SVAL(1), XLB(NCA), XSOL(NCA), XUB(NCA)
!
! Data initialization
!
DATA A/3.0E0, 4.0E0, 2.0E0, 1.0E0, 2.0E0, 2.0E0, 2.0E0, 1.0E0, &
 1.0E0, 1.0E0, 1.0E0, 1.0E0/, B/3.3E0, 2.3E0, 1.3E0, 1.0E0/, &
 C/3*1.0E0/, BC/1.0E0/, IRTYPE/1/, XLB/3*0.0E0/, XUB/3*.5E0/
!-----
!
! Reset the rank determination
! tolerance used in the solver.
! The value 2 is the option number.
! The value 0.01 is the new tolerance.
!
IOPT(1) = NEWTOL
SVAL(1) = 0.01E+0
CALL UMAG ('math', ICHP, IPUT, IOPT, SVAL)
!-----
!
! Solve the bounded, constrained
! least squares problem.
!
CALL LCLSQ (A, B, C, BC, BC, IRTYPE, XLB, XUB, XSOL, RES=RES)
! Compute the 2-norm of the residuals.
RESNRM = SNRM2(NRA,RES,1)
! Print results
CALL UMACH (2, NOUT)
WRITE (NOUT,99999) XSOL, RES, RESNRM
!-----
!
! Reset the rank determination
! tolerance to its default value.
! This is not required but is
! recommended programming practice.
!
IOPT(1) = -IOPT(1)
CALL UMAG ('math', ICHP, IPUT, IOPT, SVAL)
!-----
!
99999 FORMAT (' The solution is ', 3F9.4, '//, ' The residuals ', &
 'evaluated at the solution are ', /, 18X, 4F9.4, '//, &
 ' The norm of the residual vector is ', F8.4)
!
END

```

## Output

```
The solution is 0.5000 0.3000 0.2000

The residuals evaluated at the solution are
 -1.0000 0.5000 0.5000 0.0000

The norm of the residual vector is 1.2247
```

---

## DUMAG

See [UMAG](#).

---

## PLOTP

Prints a plot of up to 10 sets of points.

### Required Arguments

*X* — Vector of length *N*DATA containing the values of the independent variable. (Input)

*A* — Matrix of dimension *N*DATA by *N*FUN containing the *N*FUN sets of dependent variable values. (Input)

*SYMBOL* — CHARACTER string of length *N*FUN. (Input)  
SYMBOL(*I* : *I*) is the symbol used to plot function *I*.

*XTITLE* — CHARACTER string used to label the *x*-axis. (Input)

*YTITLE* — CHARACTER string used to label the *y*-axis. (Input)

*TITLE* — CHARACTER string used to label the plot. (Input)

### Optional Arguments

*N*DATA — Number of independent variable data points. (Input)  
Default: *N*DATA = SIZE (*X*,1).

*N*FUN — Number of sets of points. (Input)  
*N*FUN must be less than or equal to 10.  
Default: *N*FUN = SIZE (*A*,2).

*L*DA — Leading dimension of *A* exactly as specified in the dimension statement of the calling program. (Input)  
Default: *L*DA = SIZE (*A*, 1).

**INC** — Increment between elements of the data to be used. (Input)

PLOTP plots  $X(1 + (I - 1) * INC)$  for  $I = 1, 2, \dots, NDATA$ .

Default:  $INC = 1$ .

**RANGE** — Vector of length four specifying minimum  $x$ , maximum  $x$ , minimum  $y$  and maximum  $y$ . (Input)

PLOTP will calculate the range of the axis if the minimum and maximum of that range are equal.

Default:  $RANGE = 1.e0$ .

## FORTRAN 90 Interface

Generic: `CALL PLOTP (X, A, SYMBOL, XTITLE, YTITLE, TITLE [, ...])`

Specific: The specific interface names are `S_PLOTP` and `D_PLOTP`.

## FORTRAN 77 Interface

Single: `CALL PLOTP (NDATA, NFUN, X, A, LDA, INC, RANGE, SYMBOL, XTITLE, YTITLE, TITLE)`

Double: The double precision name is `DPLOTP`.

## Description

Routine PLOTP produces a line printer plot of up to ten sets of points superimposed upon the same plot. A character “M” is printed to indicate multiple points. The user may specify the  $x$  and  $y$ -axis plot ranges and plotting symbols. Plot width and length may be reset in advance by calling [PGOPT](#).

## Comments

1. Informational errors

| Type | Code |  |
|------|------|--|
|------|------|--|

|   |   |                                                                   |
|---|---|-------------------------------------------------------------------|
| 3 | 7 | NFUN is greater than 10. Only the first 10 functions are plotted. |
|---|---|-------------------------------------------------------------------|

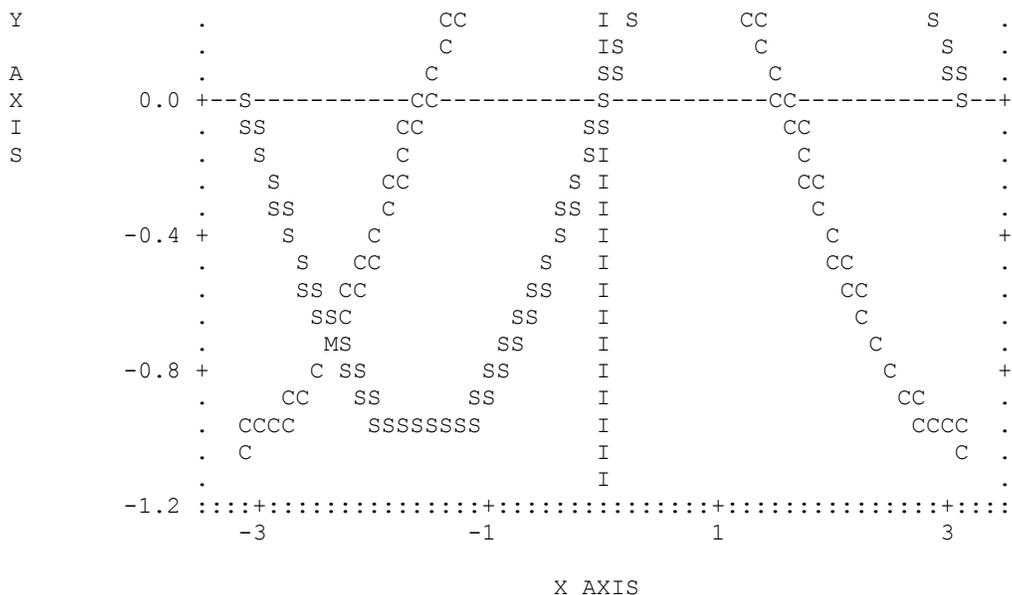
|   |   |                                                            |
|---|---|------------------------------------------------------------|
| 3 | 8 | TITLE is too long. TITLE is truncated from the right side. |
|---|---|------------------------------------------------------------|

|   |   |                                                              |
|---|---|--------------------------------------------------------------|
| 3 | 9 | YTITLE is too long. YTITLE is truncated from the right side. |
|---|---|--------------------------------------------------------------|

|   |    |                                                                                                                                                                                                |
|---|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | 10 | XTITLE is too long. XTITLE is truncated from the right side. The maximum number of characters allowed depends on the page width and the page length. See Comment 5 below for more information. |
|---|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

2. YTITLE and TITLE are automatically centered.
3. For multiple plots, the character M is used if the same print position is shared by two or more data sets.
4. Output is written to the unit specified by UMACH (see [Reference Material](#)).





## PRIME

Decomposes an integer into its prime factors.

### Required Arguments

*N* — Integer to be decomposed. (Input)

*NPF* — Number of different prime factors of  $ABS(N)$ . (Output)  
If *N* is equal to -1, 0, or 1, *NPF* is set to 0.

*IPF* — Integer vector of length 13. (Output)  
*IPF(I)* contains the prime factors of the absolute value of *N*, for  $I = 1, \dots, NPF$ . The remaining  $13 - NPF$  locations are not used.

*IEXP* — Integer vector of length 13. (Output)  
*IEXP(I)* is the exponent of *IPF(I)*, for  $I = 1, \dots, NPF$ . The remaining  $13 - NPF$  locations are not used.

*IPW* — Integer vector of length 13. (Output)  
*IPW(I)* contains the quantity  $IPF(I)**IEXP(I)$ , for  $I = 1, \dots, NPF$ . The remaining  $13 - NPF$  locations are not used.

### FORTRAN 90 Interface

Generic:    CALL PRIME (N, NPF, IPF, IPW)

Specific:   The specific interface name is PRIME.

## FORTRAN 77 Interface

Single:      CALL PRIME (N, NPF, IPF, IEXP, IPW)

## Description

Routine `PRIME` decomposes an integer into its prime factors. The number to be factored,  $N$ , may not have more than 13 distinct factors. The smallest number with more than 13 factors is about  $1.3 \times 10^{16}$ . Most computers do not allow integers of this size.

The routine `PRIME` is based on a routine by Brenner (1973).

## Comments

The output from `PRIME` should be interpreted in the following way:  
 $ABS(N) = IPF(1)**IEXP(1) * \dots * IPF(NPF)**IEXP(NPF)$ .

## Example

This example factors the integer  $144 = 2^4 3^2$ .

```
USE PRIME_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER N
PARAMETER (N=144)
!
INTEGER IEXP(13), IPF(13), IPW(13), NOUT, NPF
! Get prime factors of 144
CALL PRIME (N, NPF, IPF, IEXP, IPW)
! Get output unit number
CALL UMACH (2, NOUT)
! Print results
WRITE (NOUT,99999) N, IPF(1), IPF(2), IEXP(1), IEXP(2), IPW(1), &
IPW(2), NPF
!
99999 FORMAT (' The prime factors for', I5, ' are: ', /, 10X, 2I6, // &
' IEXP =', 2I6, /, ' IPW =', 2I6, /, ' NPF =', I6, /)
END
```

## Output

```
The prime factors for 144 are:
 2 3
```

```
IEXP = 4 2
IPW = 16 9
NPF = 2
```

---

# CONST

This function returns the value of various mathematical and physical constants.

## Function Return Value

*CONST* — Value of the constant. (Output)  
See Comment 1.

## Required Arguments

*NAME* — Character string containing the name of the desired constant. (Input)  
See Comment 3 for a list of valid constants.

## FORTRAN 90 Interface

Generic:    CONST (NAME)

Specific:   The specific interface names are S\_CONST and D\_CONST.

## FORTRAN 77 Interface

Single:     CONST (NAME)

Double:     The double precision name is DCONST.

## Description

Routine `CONST` returns the value of various mathematical and physical quantities. For all of the physical values, the Systeme International d'Unites (SI) are used.

The reference for constants are indicated by the code in [ ] Comment above.

[1] Cohen and Taylor (1986)

[2] Liepman (1964)

[3] Precomputed mathematical constants

The constants marked with an `E` before the [ ] are exact (to machine precision).

To change the units of the values returned by `CONST`, see `CUNIT`.

## Comments

1. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

```
X = CONST ('PI')
Y = COS (x)
```

must be used rather than

$Y = \text{COS}(\text{CONST}('PI'))$ .

If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. The case of the character string in `NAME` does not matter. The names “PI”, “Pi”, “pi”, and “`pi`” are equivalent.
3. The units of the physical constants are in `SI` units (meter kilogram-second).
4. The names allowed are as follows:

| <b>Name</b>      | <b>Description</b>     | <b>Value</b>                   | <b>Ref.</b> |
|------------------|------------------------|--------------------------------|-------------|
| AMU              | Atomic mass unit       | $1.6605402E - 27 \text{ kg}$   | [1]         |
| ATM              | Standard atm pressure  | $1.01325E + 5N/m^2E$           | [2]         |
| AU               | Astronomical unit      | $1.496E + 11m$                 | [ ]         |
| Avogadro         | Avogadro's number      | $6.0221367E + 231/\text{mole}$ | [1]         |
| Boltzman         | Boltzman's constant    | $1.380658E - 23J/K$            | [1]         |
| C                | Speed of light         | $2.997924580E + 8m/sE$         | [1]         |
| Catalan          | Catalan's constant     | $0.915965 \dots E$             | [3]         |
| E                | Base of natural logs   | $2.718\dots E$                 | [3]         |
| ElectronCharge   | Electron change        | $1.60217733E - 19c$            | [1]         |
| ElectronMass     | Electron mass          | $9.1093897E - 31 \text{ kg}$   | [1]         |
| ElectronVolt     | Electron volt          | $1.60217733E - 19J$            | [1]         |
| Euler            | Euler's constant gamma | $0.577 \dots E$                | [3]         |
| Faraday          | Faraday constant       | $9.6485309E + 4C/\text{mole}$  | [1]         |
| FineStructure    | fine structure         | $7.29735308E - 3$              | [1]         |
| Gamma            | Euler's constant       | $0.577 \dots E$                | [3]         |
| Gas              | Gas constant           | $8.314510J/\text{mole}/k$      | [1]         |
| Gravity          | Gravitational constant | $6.67259E - 11N * m^2/kg^2$    | [1]         |
| Hbar             | Planck constant / 2 pi | $1.05457266E - 34J * s$        | [1]         |
| PerfectGasVolume | Std vol ideal gas      | $2.241383E - 2m^3/\text{mole}$ | [*]         |
| Pi               | Pi                     | $3.141 \dots E$                | [3]         |
| Planck           | Planck's constant $h$  | $6.6260755E - 34J * s$         | [1]         |
| ProtonMass       | Proton mass            | $1.6726231E - 27 \text{ kg}$   | [1]         |
| Rydberg          | Rydberg's constant     | $1.0973731534E + 7/m$          | [1]         |
| SpeedLight       | Speed of light         | $2.997924580E + 8m/s E$        | [1]         |
| StandardGravity  | Standard $g$           | $9.80665m/s^2E$                | [2]         |

| Name             | Description           | Value                                        | Ref. |
|------------------|-----------------------|----------------------------------------------|------|
| StandardPressure | Standard atm pressure | 1.01325E + 5N/m <sup>2</sup> E               | [2]  |
| StefanBoltzmann  | Stefan-Boltzman       | 5.67051E - 8W/K <sup>4</sup> /m <sup>2</sup> | [1]  |
| WaterTriple      | Triple point of water | 2.7316E + 2K E                               | [2]  |

### Example

In this example, Euler's constant  $\gamma$  is obtained and printed. Euler's constant is defined to be

$$\gamma = \lim_{n \rightarrow \infty} \left[ \sum_{k=1}^{n-1} \frac{1}{k} - \ln n \right]$$

```

USE CONST_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
REAL GAMA

! Get output unit number
CALL UMACH (2, NOUT)

! Get gamma
GAMA = CONST('GAMMA')

! Print gamma
WRITE (NOUT,*) 'GAMMA = ', GAMA
END

```

### Output

GAMMA = 0.5772157

For another example, see [CUNIT](#).

## CUNIT

Converts  $X$  in units  $XUNITS$  to  $Y$  in units  $YUNITS$ .

### Required Arguments

$X$  — Value to be converted. (Input)

$XUNITS$  — Character string containing the name of the units for  $X$ . (Input)  
See [Comments](#) for a description of units allowed.

$Y$  — Value in  $YUNITS$  corresponding to  $X$  in  $XUNITS$ . (Output)

$YUNITS$  — Character string containing the name of the units for  $Y$ . (Input)  
See [Comments](#) for a description of units allowed.

## **FORTRAN 90 Interface**

Generic:     CALL CUNIT (X, XUNITS, Y, YUNITS [, ...])

Specific:    The specific interface names are S\_CUNIT and D\_CUNIT.

## **FORTRAN 77 Interface**

Single:      CALL CUNIT (X, XUNITS, Y, YUNITS)

Double:     The double precision name is DCUNIT.

## **Description**

Routine CUNIT converts a value expressed in one set of units to a value expressed in another set of units.

The input and output units are checked for consistency unless the input unit is "SI". SI means the Systeme International d'Unites. This is the meter-kilogram-second form of the metric system. If the input units are "SI", then the input is assumed to be expressed in the SI units consistent with the output units.

## **Comments**

1. Strings XUNITS and YUNITS have the form  $U_1 * U_2 * \dots * U_m / V_1 \dots V_n$ , where  $U_i$  and  $V_i$  are the names of basic units or are the names of basic units raised to a power. Examples are, "METER \* KILOGRAM/SECOND", "M \* KG/S", "METER", or "M/KG<sup>2</sup>".
2. The case of the character string in XUNITS and YUNITS does not matter. The names "METER", "Meter" and "meter" are equivalent.
3. If XUNITS is "SI", then X is assumed to be in the standard international units corresponding to YUNITS. Similarly, if YUNITS is "SI", then Y is assumed to be in the standard international units corresponding to XUNITS.
4. The basic unit names allowed are as follows:

Units of time

day, hour = hr, min = minute, s = sec = second, year

Units of frequency

Hertz = Hz

Units of mass

AMU, g = gram, lb = pound, ounce = oz, slug

Units of distance

Angstrom, AU, feet = foot = ft, in = inch, m = meter = metre, micron,  
mile, mill, parsec, yard

Units of area

acre

Units of volume

l = liter = litre

Units of force

dyne, N = Newton, poundal

Units of energy

BTU(thermochemical), Erg, J = Joule

Units of work

W = watt

Units of pressure

ATM = atmosphere, bar, Pascal

Units of temperature

degC = Celsius, degF = Fahrenheit, degK = Kelvin

Units of viscosity

poise, stoke

Units of charge

Abcoulomb, C = Coulomb, statcoulomb

Units of current

A = ampere, abampere, statampere,

Units of voltage

Abvolt, V = volt

Units of magnetic induction

T = Tesla, Wb = Weber

Other units

l, farad, mole, Gauss, Henry, Maxwell, Ohm

The following metric prefixes may be used with the above units. Note that the one or two letter prefixes may only be used with one letter unit abbreviations.

|   |       |          |
|---|-------|----------|
| A | Atto  | 1.E - 18 |
| F | Femto | 1.E - 15 |
| P | Pico  | 1.E - 12 |
| N | Nano  | 1.E - 9  |

|    |        |                                                |
|----|--------|------------------------------------------------|
| U  | Micro  | 1.E-6                                          |
| M  | Milli  | 1.E-3                                          |
| C  | Centi  | 1.E-2                                          |
| D  | Deci   | 1.E-1                                          |
| DK | Deca   | 1.E+2                                          |
| K  | Kilo   | 1.E+3                                          |
|    | Myriad | 1.E+4 (no single letter prefix; M means milli) |
|    | Mega   | 1.E+6 (no single letter prefix; M means milli) |
| G  | Giga   | 1.E+9                                          |
| T  | Tera   | 1.E+12                                         |

#### 5. Informational error

Type Code

3 8 A conversion of units of mass to units of force was required for consistency.

### Example

The routine `CONST` is used to obtain the speed on light,  $c$ , in SI units. `CUNIT` is then used to convert  $c$  to mile/second and to parsec/year. An example involving substitution of force for mass is required in conversion of Newtons/Meter<sup>2</sup> to Pound/Inch<sup>2</sup>.

```

USE CONST_INT
USE CUNIT_INT
USE UMACH_INT

IMPLICIT NONE
INTEGER NOUT
REAL CMH, CMS, CPY, CPSI
!
! Get output unit number
CALL UMACH (2, NOUT)
!
! Get speed of light in SI (m/s)
CMS = CONST('SpeedLight')
WRITE (NOUT,*) 'Speed of Light = ', CMS, ' meter/second'
!
! Get speed of light in mile/second
CALL CUNIT (CMS, 'SI', CMH, 'Mile/Second')
WRITE (NOUT,*) 'Speed of Light = ', CMH, ' mile/second'
!
! Get speed of light in parsec/year
CALL CUNIT (CMS, 'SI', CPY, 'Parsec/Year')
WRITE (NOUT,*) 'Speed of Light = ', CPY, ' Parsec/Year'
!
! Convert Newton/Meter**2 to
! Pound/Inch**2.
CALL CUNIT(1.E0, 'Newton/Meter**2', CPSI, &
 'Pound/Inch**2')
WRITE(NOUT,*) ' Atmospheres, in Pound/Inch**2 = ',CPSI
END

```

### Output

```
Speed of Light = 299792440.0 meter/second
Speed of Light = 186282.39 mile/second
Speed of Light = 0.3063872 Parsec/Year
```

```
*** WARNING ERROR 8 from CUNIT. A conversion of units of mass to units of
*** force was required for consistency.
Atmospheres, in Pound/Inch**2 = 1.4503773E-4
```

---

## HYPOT

This function computes  $\text{SQRT}(A^2 + B^2)$  without underflow or overflow.

### Function Return Value

*HYPOT* —  $\text{SQRT}(A^2 + B^2)$ . (Output)

### Required Arguments

*A* — First parameter. (Input)

*B* — Second parameter. (Input)

### FORTRAN 90 Interface

Generic: `HYPOT (A, B)`

Specific: The specific interface names are `S_HYPOT` and `D_HYPOT`.

### FORTRAN 77 Interface

Single: `HYPOT (A, B)`

Double: The double precision name is `DHYPOT`.

### Description

Routine `HYPOT` is based on the routine `PYTHAG`, used in `EISPACK 3`. This is an update of the work documented in Garbow et al. (1972).

### Example

Computes

$$c = \sqrt{a^2 + b^2}$$

where  $a = 10^{20}$  and  $b = 2 \times 10^{20}$  without overflow.

```
USE HYPOT_INT
USE UMACH_INT

IMPLICIT NONE
```

```

! Declare variables
 INTEGER NOUT
 REAL A, B, C
!
 A = 1.0E+20
 B = 2.0E+20
 C = HYPOT(A,B)
! Get output unit number
 CALL UMACH (2, NOUT)
! Print the results
 WRITE (NOUT,'(A,1PE10.4)') ' C = ', C
 END

```

## Output

C = 2.2361E+20

---

## MP\_SETUP



Initializes or finalizes MPI.

### Function Return Value

Number of nodes, `MP_NPROCS`, in the communicator, `MP_LIBRARY_WORLD`. (Output)

Returned when `MP_SETUP` is called with no arguments:

`MP_NPROCS = MP_SETUP()`.

### Required Argument

None.

### Optional Arguments

**NOTE** — Character string `'Final'`. (Input)

With `'Final'` all pending error messages are sent from the nodes to the root and printed. If any node should STOP after printing messages, then `MPI_Finalize()` and a STOP are executed. Otherwise, only `MPI_Finalize()` is called. The character string `'Final'` is the only valid string for this argument.

**N** — Size of array to be allocated for timing. (Input)

When this argument is supplied, the array `MPI_NODE_PRIORITY` is allocated with `MP_PROCS` components. The matrix products `A .x. B` are timed individually at each node of the machine. The elapsed time is noted and sorted to determine the node priority order. `A` and `B` are allocated to size `N` by `N`, and initialized with random data. The priority order is finally broadcast to the other nodes.

## FORTRAN 90 Interface

```
MP_SETUP ([,...])
```

### Description

Following a call to the function `MP_SETUP()`, the module `MPI_node_int` will contain information about the number of processors, the rank of a processor, the communicator for IMSL Fortran Numerical Library, and the usage priority order of the node machines:

```
MODULE MPI_NODE_INT
 INTEGER, ALLOCATABLE :: MPI_NODE_PRIORITY(:)
 INTEGER, SAVE :: MP_LIBRARY_WORLD = huge(1)
 LOGICAL, SAVE :: MPI_ROOT_WORKS = .TRUE.
 INTEGER, SAVE :: MP_RANK = 0, MP_NPROCS = 1
END MODULE
```

When the function `MP_SETUP()` is called with no arguments, the following events occur:

- If MPI has not been initialized, it is first initialized. This step uses the routines `MPI_Initialized()` and possibly `MPI_Init()`. Users who choose not to call `MP_SETUP()` must make the required initialization call before using any IMSL Fortran Numerical Library code that relies on MPI for its execution. If the user's code calls an IMSL Fortran Numerical Library function utilizing the box data type and MPI has not been initialized, then the computations are performed on the root node. The only MPI routine always called in this context is `MPI_Initialized()`. The name `MP_SETUP` is pushed onto the subprogram or call stack.
- If `MP_LIBRARY_WORLD` equals its initial value (`=huge(1)`) then `MPI_COMM_WORLD`, the default MPI communicator, is duplicated and becomes its handle. This uses the routine `MPI_Comm_dup()`. Users can change the handle of `MP_LIBRARY_WORLD` as required by their application code. Often this issue can be ignored.
- The integers `MP_RANK` and `MP_NPROCS` are respectively the node's rank and the number of nodes in the communicator, `MP_LIBRARY_WORLD`. Their values require the routines `MPI_Comm_size()` and `MPI_Comm_rank()`. The default values are important when MPI is not initialized and a box data type is computed. In this case the root node is the only node and it will do all the work. No calls to MPI communication routines are made when `MP_NPROCS = 1` when computing the box data type functions. A program can temporarily assign this value to force box data type computation entirely at the root node. This is desirable for problems where using many nodes would be less efficient than using the root node exclusively.
- The array `MPI_NODE_PRIORITY(:)` is not allocated unless the user allocates it. The IMSL Fortran Numerical Library codes use this array for assigning tasks to processors, if it is allocated. If it is not allocated, the default priority of the nodes is `(0, 1, ..., MP_NPROCS-1)`. Use of the function call `MP_SETUP(N)` allocates the array, as explained below. Once the array is allocated its size is `MP_NPROCS`. The contents of the array is a permutation of the integers `0, ..., MP_NPROCS-1`. Nodes appearing at the start of the list are used first for parallel computing. A node other than the root can avoid any computing, except receiving the schedule, by setting the value `MPI_NODE_PRIORITY(I) < 0`. This

means that node `|MPI_NODE_PRIORITY(I)|` will be sent the task schedule but will not perform any significant work as part of box data type function evaluations.

- The LOGICAL flag `MPI_ROOT_WORKS` designates whether or not the root node participates in the major computation of the tasks. The root node communicates with the other nodes to complete the tasks but can be designated to do no other work. Since there may be only one processor, this flag has the default value `.TRUE.`, assuring that one node exists to do work. When more than one processor is available users can consider assigning `MPI_ROOT_WORKS=.FALSE.` This is desirable when the alternate nodes have equal or greater computational resources compared with the root node. [Parallel Example 4](#) illustrates this usage. A single problem is given a box data type, with one rack. The computing is done at the node, other than the root, with highest priority. This example requires more than one processor since the root does no work.

When the generic function `MP_SETUP(N)` is called, where `N` is a positive integer, a call to `MP_SETUP()` is first made, using no argument. Use just one of these calls to `MP_SETUP()`. This initializes the MPI system and the other parameters described above. The array `MPI_NODE_PRIORITY(:)` is allocated with size `MP_NPROCS`. Then DOUBLE PRECISION matrix products  $C = AB$ , where  $A$  and  $B$  are  $N$  by  $N$  matrices, are computed at each node and the elapsed time is recorded. These elapsed times are sorted and the contents of `MPI_NODE_PRIORITY(:)` are permuted in accordance with the shortest times yielding the highest priority. All the nodes in the communicator `MP_LIBRARY_WORLD` are timed. The array `MPI_NODE_PRIORITY(:)` is then broadcast from the root to the remaining nodes of `MP_LIBRARY_WORLD` using the routine `MPI_Bcast()`. Timing matrix products to define the node priority is relevant because the effort to compute  $C$  is comparable to that of many linear algebra computations of similar size. Users are free to define their own node priority and broadcast the array `MPI_NODE_PRIORITY(:)` to the alternate nodes in the communicator.

To print any IMSL Fortran Numerical Library error messages that have occurred at any node, and to finalize MPI, use the function call `MP_SETUP('Final')`. The case of the string 'Final' is not important. Any error messages pending will be discarded after printing on the root node. This is triggered by popping the name 'MP\_SETUP' from the subprogram stack or returning to Level 1 in the stack. Users can obtain error messages by popping the stack to Level 1 and still continuing with MPI calls. This requires executing call `elpop('MP_SETUP')`. To continue on after summarizing errors execute call `elpsh('MP_SETUP')`. More details about the error processor are found in Reference Material chapter of this manual.

Messages are printed by nodes from largest rank to smallest, which is the root node. Use of the routine `MPI_Finalize()` is made within `MP_SETUP('Final')`, which shuts down MPI. After `MPI_Finalize()` is called, the value of `MP_NPROCS = 0`. This flags that MPI has been initialized and terminated. It cannot be initialized again in the same program unit execution. No MPI routine is defined when `MP_NPROCS` has this value.

## Examples

### Parallel Example (parallel\_ex01.f90)

```
use linear_operators
use mpi_setup_int

implicit none
```

```

! This is the equivalent of Parallel Example 1 for .ix., with box data types
! and functions.

integer, parameter :: n=32, nr=4
real(kind(1e0)) :: one=1e0
real(kind(1e0)), dimension(n,n,nr) :: A, b, x, err(nr)

! Setup for MPI.
MP_NPROCS=MP_SETUP()

! Generate random matrices for A and b:
A = rand(A); b=rand(b)

! Compute the box solution matrix of Ax = b.
x = A .ix. b

! Check the results.
err = norm(b - (A .x. x)) / (norm(A)*norm(x)+norm(b))
if (ALL(err <= sqrt(epsilon(one))) .and. MP_RANK == 0) &
write (*,*) 'Parallel Example 1 is correct.'

! See to any error messages and quit MPI.
MP_NPROCS=MP_SETUP('Final')

end

```

### Parallel Example (parallel\_ex04.f90)

Here an alternate node is used to compute the majority of a single application, and the user does not need to make any explicit calls to MPI routines. The time-consuming parts are the evaluation of the eigenvalue-eigenvector expansion, the solving step, and the residuals. To do this, the rank-2 arrays are changed to a box data type with a unit third dimension. This uses parallel computing. The node priority order is established by the initial function call, `MP_SETUP(n)`. The root is restricted from working on the box data type by assigning `MPI_ROOT_WORKS=.false.` This example anticipates that the most efficient node, other than the root, will perform the heavy computing. Two nodes are required to execute.

```

use linear_operators
use mpi_setup_int

implicit none

! This is the equivalent of Parallel Example 4 for matrix exponential.
! The box dimension has a single rack.
integer, parameter :: n=32, k=128, nr=1
integer i
real(kind(1e0)), parameter :: one=1e0, t_max=one, delta_t=t_max/(k-1)
real(kind(1e0)) err(nr), sizes(nr), A(n,n,nr)
real(kind(1e0)) t(k), y(n,k,nr), y_prime(n,k,nr)
complex(kind(1e0)), dimension(n,nr) :: x(n,n,nr), z_0, &
z_1(n,nr,nr), y_0, d

! Setup for MPI. Establish a node priority order.
! Restrict the root from significant computing.
! Illustrates using the 'best' performing node that

```

```

! is not the root for a single task.
 MP_NPROCS=MP_SETUP(n)

 MPI_ROOT_WORKS=.false.

! Generate a random coefficient matrix.
 A = rand(A)

! Compute the eigenvalue-eigenvector decomposition
! of the system coefficient matrix on an alternate node.
 D = EIG(A, W=X)

! Generate a random initial value for the ODE system.
 y_0 = rand(y_0)

! Solve complex data system that transforms the initial
! values, X z_0=y_0.

 z_1= X .ix. y_0 ; z_0(:,nr) = z_1(:,nr,nr)

! The grid of points where a solution is computed:
 t = ((i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
! With the eigenvalue-eigenvector decomposition AX = XD, this
! is an evaluation of EXP(A t)y_0 = y(t).
 y = X .x.exp(spread(d(:,nr),2,k)*spread(t,1,n))*spread(z_0(:,nr),2,k)

! This is y', derived by differentiating y(t).
 y_prime = X .x. &
spread(d(:,nr),2,k)*exp(spread(d(:,nr),2,k)*spread(t,1,n))* &
 spread(z_0(:,nr),2,k)

! Check results. Is y' - Ay = 0?
 err = norm(y_prime-(A .x. y))
 sizes=norm(y_prime)+norm(A)*norm(y)
 if (ALL(err <= sqrt(epsilon(one))*sizes) .and. MP_RANK == 0) &
 write (*,*) 'Parallel Example 4 is correct.'

! See to any error messages and quit MPI.
 MP_NPROCS=MP_SETUP('Final')

end

```

# Reference Material

---

## Contents

|                                                |      |
|------------------------------------------------|------|
| Contents.....                                  | 1931 |
| Machine-Dependent Constants .....              | 1937 |
| Matrix Storage Modes.....                      | 1943 |
| Reserved Names .....                           | 1954 |
| Automatic Workspace Allocation .....           | 1955 |
| Deprecated Features and Renamed Routines ..... | 1955 |

---

## User Errors

IMSL routines attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, we recognize various levels of severity of errors, and we also consider the extent of the error in the context of the purpose of the routine; a trivial error in one situation may be serious in another. IMSL routines attempt to report as many errors as they can reasonably detect. Multiple errors present a difficult problem in error detection because input is interpreted in an uncertain context after the first error is detected.

### What Determines Error Severity

In some cases, the user's input may be mathematically correct, but because of limitations of the computer arithmetic and of the algorithm used, it is not possible to compute an answer accurately. In this case, the assessed degree of accuracy determines the severity of the error. In cases where the routine computes several output quantities, if some are not computable but most are, an error condition exists. The severity depends on an assessment of the overall impact of the error.

### Terminal errors

If the user's input is regarded as meaningless, such as  $N = -1$  when "N" is the number of equations, the routine prints a message giving the value of the erroneous input argument(s) and the reason for the erroneous input. The routine will then cause the user's program to stop. An error in which the user's input is meaningless is the most severe error and is called a *terminal error*. Multiple terminal error messages may be printed from a single routine.

### Informational errors

In many cases, the best way to respond to an error condition is simply to correct the input and rerun the program. In other cases, the user may want to take actions in the program itself based on errors that occur. An error that may be used as the basis for corrective action within the program is

called an *informational error*. If an informational error occurs, a user-retrievable code is set. A routine can return at most one informational error for a single reference to the routine. The codes for the informational error codes are printed in the error messages.

## Other errors

In addition to informational errors, IMSL routines issue error messages for which no user-retrievable code is set. Multiple error messages for this kind of error may be printed. These errors, which generally are not described in the documentation, include terminal errors as well as less serious errors. Corrective action within the calling program is not possible for these errors.

## Kinds of Errors and Default Actions

Five levels of severity of errors are defined in the MATH/LIBRARY. Each level has an associated PRINT attribute and a STOP attribute. These attributes have default settings (YES or NO), but they may also be set by the user. The purpose of having multiple error severity levels is to provide independent control of actions to be taken for errors of different severity. Upon return from an IMSL routine, exactly one error state exists. (A code 0 "error" is no informational error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except level 5 may be informational errors.

- Level 1: Note.** A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations. Default attributes: PRINT=NO, STOP=NO
- Level 2: Alert.** An *alert* indicates that the user should be advised about events occurring in the software. Default attributes: PRINT=NO, STOP=NO
- Level 3: Warning.** A *warning* indicates the existence of a condition that may require corrective action by the user or calling routine. A warning error may be issued because the results are accurate to only a few decimal places, because some of the output may be erroneous but most of the output is correct, or because some assumptions underlying the analysis technique are violated. Often no corrective action is necessary and the condition can be ignored. Default attributes: PRINT=YES, STOP=NO
- Level 4: Fatal.** A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling routine must take corrective action to recover. Default attributes: PRINT=YES, STOP=YES
- Level 5: Terminal.** A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors may also be caused by various programming errors impossible to diagnose correctly in FORTRAN. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the routine with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error because corrective action within the program is generally not reasonable. In normal usage, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur. Default attributes: PRINT=YES, STOP=YES

The user can set PRINT and STOP attributes by calling `ERSET` as described in “Routines for Error Handling.”

## Errors in Lower-Level Routines

It is possible that a user’s program may call an IMSL routine that in turn calls a nested sequence of lower-level IMSL routines. If an error occurs at a lower level in such a nest of routines and if the lower-level routine cannot pass the information up to the original user-called routine, then a traceback of the routines is produced. The only common situation in which this can occur is when an IMSL routine calls a user-supplied routine that in turn calls another IMSL routine.

## Routines for Error Handling

There are three ways in which the user may interact with the IMSL error handling system: (1) to change the default actions, (2) to retrieve the integer code of an informational error so as to take corrective action, and (3) to determine the severity level of an error. The routines to use are `ERSET`, `IERCD`, and `NIRTY`, respectively.

---

# ERSET

Change the default printing or stopping actions when errors of a particular error severity level occur.

## Required Arguments

*IERSVR* — Error severity level indicator. (Input)

If *IERSVR* = 0, actions are set for levels 1 to 5. If *IERSVR* is 1 to 5, actions are set for errors of the specified severity level.

*IPACT* — Printing action. (Input)

| <i>IPACT</i> | Action                            |
|--------------|-----------------------------------|
| -1           | Do not change current setting(s). |
| 0            | Do not print.                     |
| 1            | Print.                            |
| 2            | Restore the default setting(s).   |

*ISACT* — Stopping action. (Input)

| <i>ISACT</i> | Action |
|--------------|--------|
|--------------|--------|

|    |                                   |
|----|-----------------------------------|
| -1 | Do not change current setting(s). |
| 0  | Do not stop.                      |
| 1  | Stop.                             |
| 2  | Restore the default setting(s).   |

### **FORTRAN 90 Interface**

Generic:    CALL ERSET (IERSVR, IPACT, ISACT)

Specific:    The specific interface name is ERSET.

### **FORTRAN 77 Interface**

Single:     CALL ERSET (IERSVR, IPACT, ISACT)

---

## **IERCD and N1RTY**

The last two routines for interacting with the error handling system, IERCD and N1RTY, are INTEGER functions and are described in the following material.

IERCD retrieves the integer code for an informational error. Since it has no arguments, it may be used in the following way:

```
ICODE = IERCD()
```

The function retrieves the code set by the most recently called IMSL routine.

N1RTY retrieves the error type set by the most recently called IMSL routine. It is used in the following way:

```
ITYPE = N1RTY(1)
```

ITYPE = 1, 2, 4, and 5 correspond to error severity levels 1, 2, 4, and 5, respectively. ITYPE = 3 and ITYPE = 6 are both warning errors, error severity level 3. While ITYPE = 3 errors are informational errors (IERCD() ≠ 0), ITYPE = 6 errors are not informational errors (IERCD() = 0).

For software developers requiring additional interaction with the IMSL error handling system, see Aird and Howell (1991).

### **Examples**

#### **Changes to default actions**

Some possible changes to the default actions are illustrated below. The default actions remain in effect for the kinds of errors not included in the call to ERSET.

To turn off printing of warning error messages:

```
CALL ERSET (3, 0, -1)
```

To stop if warning errors occur:

```
CALL ERSET (3, -1, 1)
```

To print all error messages:

```
CALL ERSET (0, 1, -1)
```

To restore all default settings:

```
CALL ERSET (0, 2, 2)
```

## Use of informational error to determine program action

In the program segment below, the Cholesky factorization of a matrix is to be performed. If it is determined that the matrix is not nonnegative definite (and often this is not immediately obvious), the program is to take a different branch.

```
 .
 .
 .
 CALL LFTDS (A, FACT)
 IF (IERCD() .EQ. 2) THEN
! Handle matrix that is not nonnegative definite
 .
 .
 .
 END IF
```

## Examples of errors

The program below illustrates each of the different types of errors detected by the MATH/LIBRARY routines.

The error messages refer to the argument names that are used in the documentation for the routine, rather than the user's name of the variable used for the argument. In the message generated by IMSL routine LINRG in this example, reference is made to N, whereas in the program a literal was used for this argument.

```
 USE_IMSL_LIBRARIES
 INTEGER N
 PARAMETER (N=2)
!
! REAL A(N,N), AINV(N,N), B(N), X(N)
!
! DATA A/2.0, -3.0, 2.0, -3.0/
! DATA B/1.0, 2.0/
!
! Turn on printing and turn off
! stopping for all error types.
 CALL ERSET (0, 1, 0)
!
! Generate level 4 informational error.
 CALL LSARG (A, B, X)
!
! Generate level 5 terminal error.
 CALL LINRG (A, AINV, N = -1)
 END
```

## Output

```
*** FATAL ERROR 2 from LSARG. The input matrix is singular. Some of
*** the diagonal elements of the upper triangular matrix U of the
*** LU factorization are close to zero.

*** TERMINAL ERROR 1 from LINRG. The order of the matrix must be positive
*** while N = -1 is given.
```

## Example of traceback

The next program illustrates a situation in which a traceback is produced. The program uses the IMSL quadrature routines QDAG and QDAGS to evaluate the double integral

$$\int_0^1 \int_0^1 (x+y) dx dy = \int_0^1 g(y) dy$$

where

$$g(y) = \int_0^1 (x+y) dx = \int_0^1 f(x) dx, \text{ with } f(x) = x+y$$

Since both QDAG and QDAGS need 2500 numeric storage units of workspace, and since the workspace allocator uses some space to keep track of the allocations, 6000 numeric storage units of space are explicitly allocated for workspace. Although the traceback shows an error code associated with a terminal error, this code has no meaning to the user; the printed message contains all relevant information. It is not assumed that the user would take corrective action based on knowledge of the code.

```
 USE QDAGS_INT
!
! Specifications for local variables
 REAL A, B, ERRABS, ERREST, ERRREL, G, RESULT
 EXTERNAL G
!
! Set quadrature parameters
 A = 0.0
 B = 1.0
 ERRABS = 0.0
 ERRREL = 0.001
!
! Do the outer integral
 CALL QDAGS (G, A, B, RESULT, ERRABS, ERRREL, ERREST)
!
 WRITE (*,*) RESULT, ERREST
 END
!
 REAL FUNCTION G (ARGY)
 USE QDAG_INT
 REAL ARGY
!
 INTEGER IRULE
 REAL C, D, ERRABS, ERREST, ERRREL, F, Y
 COMMON /COMY/ Y
 EXTERNAL F
!
 Y = ARGY
```

```

C = 0.0
D = 1.0
ERRABS = 0.0
ERRREL = -0.001
IRULE = 1
!
CALL QDAG (F, C, D, G, ERRABS, ERRREL, IRULE, ERREST)
RETURN
END
!
REAL FUNCTION F (X)
REAL X
!
REAL Y
COMMON /COMY/ Y
!
F = X + Y
RETURN
END

```

## Output

```

*** TERMINAL ERROR 4 from Q2AG. The relative error desired ERRREL =
*** -1.000000E-03. It must be at least zero.
Here is a traceback of subprogram calls in reverse order:
Routine name Error type Error code

Q2AG 5 4 (Called internally)
QDAG 0 0
Q2AGS 0 0 (Called internally)
QDAGS 0 0
USER 0 0

```

---

## Machine-Dependent Constants

The function subprograms in this section return machine-dependent information and can be used to enhance portability of programs between different computers. The routines [IMACH](#), and [AMACH](#) describe the computer's arithmetic. The routine [UMACH](#) describes the input, output, and error output unit numbers.

---

## IMACH

This function retrieves machine integer constants that define the arithmetic used by the computer.

### Function Return Value

IMACH(1) = Number of bits per integer storage unit.

IMACH(2) = Number of characters per integer storage unit:

Integers are represented in  $M$ -digit, base  $A$  form as

$$\sigma \sum_{k=0}^M x_k A^k$$

where  $\sigma$  is the sign and  $0 \leq x_k < A$ ,  $k = 0, \dots, M$ .

Then,

IMACH(3) =  $A$ , the base.

IMACH(4) =  $M$ , the number of base- $A$  digits.

IMACH(5) =  $A^M - 1$ , the largest integer.

The machine model assumes that floating-point numbers are represented in normalized  $N$ -digit, base  $B$  form as

$$\sigma B^E \sum_{k=1}^N x_k B^{-k}$$

where  $\sigma$  is the sign,  $0 < x_1 < B$ ,  $0 \leq x_k < B$ ,  $k = 2, \dots, N$  and  $E_{\min} \leq E \leq E_{\max}$ . Then,

IMACH(6) =  $B$ , the base.

IMACH(7) =  $N_s$ , the number base- $B$ -digits in single precision.

IMACH(8) =  $E_{\min_s}$ , the smallest single precision exponent.

IMACH(9) =  $E_{\max_s}$ , the largest single precision exponent.

IMACH(10) =  $N_d$ , the number base- $B$ -digits in double precision.

IMACH(11) =  $E_{\min_d}$ , the smallest double precision exponent.

IMACH(12) =  $E_{\max_d}$ , largest double precision exponent.

## Required Arguments

$I$  — Index of the desired constant. (Input)

## FORTRAN 90 Interface

Generic:   IMACH (I)

Specific:   The specific interface name is IMACH.

## FORTRAN 77 Interface

Single:      `IMACH (I)`

---

# AMACH

The function subprogram `AMACH` retrieves machine constants that define the computer's single-precision or double precision arithmetic. Such floating-point numbers are represented in normalized  $N$ -digit, base  $B$  form as

$$\sigma B^E \sum_{k=1}^N x_k B^{-k}$$

where  $\sigma$  is the sign,  $0 < x_1 < B$ ,  $0 \leq x_k < B$ ,  $k = 2, \dots, N$  and

$$E_{\min} \leq E \leq E_{\max}$$

## Function Return Value

`AMACH(1)` =  $B^{E_{\min}-1}$ , the smallest normalized positive number.

`AMACH(2)` =  $B^{E_{\max}-1} (1 - B^{-N})$ , the largest number.

`AMACH(3)` =  $B^{-N}$ , the smallest relative spacing.

`AMACH(4)` =  $B^{1-N}$ , the largest relative spacing.

`AMACH(5)` =  $\log_{10}(B)$ .

`AMACH(6)` = NaN (non-signaling not a number).

`AMACH(7)` = positive machine infinity.

`AMACH(8)` = negative machine infinity.

See [Comment 1](#) for a description of the use of the generic version of this function.

See [Comment 2](#) for a description of *min*, *max*, and *N*.

## Required Arguments

*I* — Index of the desired constant. (Input)

## FORTRAN 90 Interface

Generic:      `AMACH (I)`

Specific:     The specific interface names are `S_AMACH` and `D_AMACH`.

## FORTRAN 77 Interface

Single:      AMACH (I)

Double:      The double precision name is DMACH.

## Comments

1. If the generic version of this function is used, the immediate result must be stored in a variable before use in an expression. For example:

```
X = AMACH (I)
Y = SQRT (X)
must be used rather than
Y = SQRT (AMACH (I)) .
```

If this is too much of a restriction on the programmer, then the specific name can be used without this restriction.

2. Note that for single precision  $B = \text{IMACH}(6)$ ,  $N = \text{IMACH}(7)$ .  
 $E_{\min} = \text{IMACH}(8)$ , and  $E_{\max} = \text{IMACH}(9)$ .  
For double precision  $B = \text{IMACH}(6)$ ,  $N = \text{IMACH}(10)$ .  
 $E_{\min} = \text{IMACH}(11)$ , and  $E_{\max} = \text{IMACH}(12)$ .
3. The IEEE standard for binary arithmetic (see IEEE 1985) specifies *quiet* NaN (not a number) as the result of various invalid or ambiguous operations, such as 0/0. The intent is that `AMACH(6)` return a *quiet* NaN. On IEEE format computers that do not support a quiet NaN, a special value near `AMACH(2)` is returned for `AMACH(6)`. On computers that do not have a special representation for infinity, `AMACH(7)` returns the same value as `AMACH(2)`.

---

## DMACH

See [AMACH](#).

---

## IFNAN(X)

This logical function checks if the argument `x` is NaN (not a number).

### Function Return Value

*IFNAN* - Logical function value. True is returned if the input argument is a NaN. Otherwise, False is returned. (Output)

### Required Arguments

*X* – Argument for which the test for NaN is desired. (Input)

## FORTRAN 90 Interface

Generic:    IFNAN (X)

Specific:   The specific interface names are S\_IFNAN and D\_IFNAN.

## FORTRAN 77 Interface

Single:     IFNAN (X)

Double:     The double precision name is DIFNAN.

## Example

```
USE IFNAN_INT
USE AMACH_INT
USE UMACH_INT
INTEGER NOUT
REAL X
!
CALL UMACH (2, NOUT)
!
X = AMACH(6)
IF (IFNAN(X)) THEN
 WRITE (NOUT,*) ' X is NaN (not a number).'
```

## Output

X is NaN (not a number).

## Description

The logical function `IFNAN` checks if the single or double precision argument `x` is NaN (not a number). The function `IFNAN` is provided to facilitate the transfer of programs across computer systems. This is because the check for NaN can be tricky and not portable across computer systems that do not adhere to the IEEE standard. For example, on computers that support the IEEE standard for binary arithmetic (see IEEE 1985), NaN is specified as a bit format not equal to itself. Thus, the check is performed as

```
IFNAN = X .NE. X
```

On other computers that do not use IEEE floating-point format, the check can be performed as:

```
IFNAN = X .EQ. AMACH(6)
```

The function `IFNAN` is equivalent to the specification of the function `Isnan` listed in the Appendix, (IEEE 1985). The above following example illustrates the use of `IFNAN`. If `x` is NaN, a message is printed instead of `X`. (Routine `UMACH`, which is described in the following section, is used to retrieve the output unit number for printing the message.)

---

# UMACH

Routine `UMACH` sets or retrieves the input, output, or error output device unit numbers.

## Required Arguments

*N* — Integer value indicating the action desired. If the value of *N* is negative, the input, output, or error output unit number is reset to `NUNIT`. If the value of *N* is positive, the input, output, or error output unit number is returned in `NUNIT`. See the table in argument `NUNIT` for legal values of *N*. (Input)

*NUNIT* — The unit number that is either retrieved or set, depending on the value of input argument *N*. (Input/Output)

The arguments are summarized by the following table:

| <b>N</b> | <b>Effect</b>                                              |
|----------|------------------------------------------------------------|
| 1        | Retrieves input unit number in <code>NUNIT</code> .        |
| 2        | Retrieves output unit number in <code>NUNIT</code> .       |
| 3        | Retrieves error output unit number in <code>NUNIT</code> . |
| -1       | Sets the input unit number to <code>NUNIT</code> .         |
| -2       | Sets the output unit number to <code>NUNIT</code> .        |
| -3       | Sets the error output unit number to <code>NUNIT</code> .  |

## FORTRAN 90 Interface

Generic:     `CALL UMACH (N, NUNIT)`

Specific:    The specific interface name is `UMACH`.

## FORTRAN 77 Interface

Single:     `CALL UMACH (N, NUNIT)`

## Description

Routine `UMACH` sets or retrieves the input, output, or error output device unit numbers. `UMACH` is set automatically so that the default FORTRAN unit numbers for standard input, standard output, and standard error are used. These unit numbers can be changed by inserting a call to `UMACH` at the beginning of the main program that calls `MATH/LIBRARY` routines. If these unit numbers are changed from the standard values, the user should insert an appropriate `OPEN` statement in the calling program.

## Example

In the following example, a terminal error is issued from the MATH/LIBRARY `AMACH` function since the argument is invalid. With a call to `UMACH`, the error message will be written to a local file named "CHECKERR".

```
USE AMACH_INT
USE UMACH_INT
INTEGER N, NUNIT
REAL X
!
! Set Parameter
N = 0
NUNIT = 9
!
CALL UMACH (-3, NUNIT)
OPEN (UNIT=NUNIT, FILE='CHECKERR')
X = AMACH(N)
END
```

## Output

The output from this example, written to "CHECKERR" is:  
\*\*\* TERMINAL ERROR 5 from AMACH. The argument must be between 1 and 8  
\*\*\* inclusive. N = 0

---

# Matrix Storage Modes

In this section, the word *matrix* will be used to refer to a mathematical object, and the word *array* will be used to refer to its representation as a FORTRAN data structure.

## General Mode

A *general* matrix is an  $N \times N$  matrix  $A$ . It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of  $A$ . It must be at least as large as  $N$ . IMSL general matrix subprograms only refer to values  $A_{ij}$  for  $i = 1, \dots, N$  and  $j = 1, \dots, N$ . The data type of a general array can be one of `REAL`, `DOUBLE PRECISION`, or `COMPLEX`. If your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX` can also be declared.

## Rectangular Mode

A *rectangular* matrix is an  $M \times N$  matrix  $A$ . It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of  $A$ . It must be at least as large as  $M$ . IMSL rectangular matrix subprograms only refer to values  $A_{ij}$  for  $i = 1, \dots, M$  and  $j = 1, \dots, N$ . The data type of a rectangular array can be `REAL`, `DOUBLE PRECISION`, or `COMPLEX`. If your FORTRAN compiler allows, you can declare the nonstandard data type `DOUBLE COMPLEX`.

## Symmetric Mode

A symmetric matrix is a square  $N \times N$  matrix  $A$ , such that  $A^T = A$ . ( $A^T$  is the transpose of  $A$ .) It is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of  $A$ . It must be at least as large as  $N$ . IMSL symmetric matrix subprograms only refer to the upper or to the lower half of  $A$  (i.e., to values  $A_{ij}$  for  $i = 1, \dots, N$  and  $j = i, \dots, N$ , or  $A_{ij}$  for  $j = 1, \dots, N$  and  $i = j, \dots, N$ ). The data type of a symmetric array can be one of `REAL` or `DOUBLE PRECISION`. Use of the upper half of the array is denoted in the BLAS that compute with symmetric matrices, see [Chapter 9, Basic Matrix/Vector Operations](#), using the `CHARACTER*1` flag `UPLO = 'U'`. Otherwise, `UPLO = 'L'` denotes that the lower half of the array is used.

## Hermitian Mode

A *Hermitian* matrix is a square  $N \times N$  matrix  $A$ , such that

$$\bar{A}^T = A$$

The matrix

$$\bar{A}$$

is the complex conjugate of  $A$  and

$$A^H \equiv \bar{A}^T$$

is the conjugate transpose of  $A$ . For Hermitian matrices,  $A^H = A$ . The matrix is stored in a FORTRAN array that is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of  $A$ . It must be at least as large as  $N$ . IMSL Hermitian matrix subprograms only refer to the upper or to the lower half of  $A$  (i.e., to values  $A_{ij}$  for  $i = 1, \dots, N$  and  $j = i, \dots, N$ , or  $A_{ij}$  for  $j = 1, \dots, N$  and  $i = j, \dots, N$ ). Use of the upper half of the array is denoted in the BLAS that compute with Hermitian matrices, see [Chapter 9, Basic Matrix/Vector Operations](#), using the `CHARACTER*1` flag `UPLO = 'U'`. Otherwise, `UPLO = 'L'` denotes that the lower half of the array is used. The data type of a Hermitian array can be `COMPLEX` or, if your FORTRAN compiler allows, the nonstandard data type `DOUBLE COMPLEX`.

## Triangular Mode

A *triangular* matrix is a square  $N \times N$  matrix  $A$  such that values  $A_{ij} = 0$  for  $i < j$  or  $A_{ij} = 0$  for  $i > j$ . The first condition defines a *lower* triangular matrix while the second condition defines an *upper* triangular matrix. A lower triangular matrix  $A$  is stored in the lower triangular part of a FORTRAN array  $A$ . An upper triangular matrix is stored in the upper triangular part of a FORTRAN array. Triangular matrices are called *unit* triangular whenever  $A_{jj} = 1$ ,  $j = 1, \dots, N$ . For unit triangular matrices, only the strictly lower or upper parts of the array are referenced. This is denoted in the BLAS that compute with triangular matrices, see [Chapter 9, Basic Matrix/Vector Operations](#), using the `CHARACTER*1` flag `DIAGNL = 'U'`. Otherwise, `DIAGNL = 'N'` denotes

that the diagonal array terms should be used. For unit triangular matrices, the diagonal terms are each used with the mathematical value 1. The array diagonal term does not need to be 1.0 in this usage. Use of the upper half of the array is denoted in the BLAS that compute with triangular matrices, see [Chapter 9, Basic Matrix/Vector Operations](#), using the CHARACTER\*1 flag UPLO = 'U'. Otherwise, UPLO = 'L' denotes that the lower half of the array is used. The data type of an array that contains a triangular matrix can be one of REAL, DOUBLE PRECISION, or COMPLEX. If your FORTRAN compiler allows, the nonstandard data type DOUBLE COMPLEX can also be declared.

## Band Storage Mode

A *band matrix* is an  $M \times N$  matrix  $A$  with all of its nonzero elements “close” to the main diagonal. Specifically, values  $A_{ij} = 0$  if  $i - j > NLCA$  or  $j - i > NUCA$ . The integers NLCA and NUCA are the *lower* and *upper* band widths. The integer  $m = NLCA + NUCA + 1$  is the total band width. The diagonals, other than the main diagonal, are called *codiagonals*. While any  $M \times N$  matrix is a band matrix, the band matrix mode is most useful only when the number of nonzero codiagonals is much less than  $m$ .

In the band storage mode, the NLCA lower codiagonals and NUCA upper codiagonals are stored in the rows of a FORTRAN array of dimension  $m \times N$ . The elements are stored in the same column of the array as they are in the matrix. The values  $A_{ij}$  inside the band width are stored in array positions  $(i - j + NUCA + 1, j)$ . This array is declared by the following statement:

```
DIMENSION A (LDA, N)
```

The parameter LDA is called the *leading dimension* of  $A$ . It must be at least as large as  $m$ . The data type of a band matrix array can be one of REAL, DOUBLE PRECISION, COMPLEX or, if your FORTRAN compiler allows, the nonstandard data type DOUBLE COMPLEX. Use of the CHARACTER\*1 flag TRANS='N' in the BLAS, see [Chapter 9, Basic Matrix/Vector Operations](#), specifies that the matrix  $A$  is used. The flag value

TRANS='T' uses  $A^T$

while

TRANS='C' uses  $\bar{A}^T$

For example, consider a real  $5 \times 5$  band matrix with 1 lower and 2 upper codiagonals, stored in the FORTRAN array declared by the following statements:

```
PARAMETER (N=5, NLCA=1, NUCA=2)
REAL A (NLCA+NUCA+1, N)
```

The matrix  $A$  has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{21} & A_{22} & A_{23} & A_{24} & 0 \\ 0 & A_{32} & A_{33} & A_{34} & A_{35} \\ 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

As a FORTRAN array, it is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{21} & A_{32} & A_{43} & A_{54} & \times \end{bmatrix}$$

The entries marked with an  $\times$  in the above array are not referenced by the IMSL band subprograms.

### Band Symmetric Storage Mode

A *band symmetric* matrix is a band matrix that is also symmetric. The band symmetric storage mode is similar to the band mode except only the lower or upper codiagonals are stored.

In the band symmetric storage mode, the `NCODA` upper codiagonals are stored in the rows of a FORTRAN array of dimension  $(\text{NCODA} + 1) \times N$ . The elements are stored in the same column of the array as they are in the matrix. Specifically, values  $A_{ij}, j \leq i$  inside the band are stored in array positions  $(i - j + \text{NCODA} + 1, j)$ . This is the storage mode designated by using the `CHARACTER*1` flag `UPLO = 'U'` in Level 2 BLAS that compute with band symmetric matrices, see [Chapter 9, Basic Matrix/Vector Operations](#). Alternatively,  $A_{ij}, j \leq i$ , inside the band, are stored in array positions  $(i - j + 1, j)$ . This is the storage mode designated by using the `CHARACTER*1` flag `UPLO = 'L'` in these Level 2 BLAS, see [Chapter 9, Basic Matrix/Vector Operations](#). The array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter `LDA` is called the *leading dimension* of  $A$ . It must be at least as large as `NCODA + 1`. The data type of a band symmetric array can be `REAL` or `DOUBLE PRECISION`.

For example, consider a real  $5 \times 5$  band matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N=5, NCODA=2)
REAL A(NCODA+1, N)
```

The matrix  $A$  has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{12} & A_{22} & A_{23} & A_{24} & 0 \\ A_{13} & A_{23} & A_{33} & A_{34} & A_{35} \\ 0 & A_{24} & A_{34} & A_{44} & A_{45} \\ 0 & 0 & A_{35} & A_{45} & A_{55} \end{bmatrix}$$

Since  $A$  is symmetric, the values  $A_{ij} = A_{ji}$ . In the FORTRAN array, it is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

The entries marked with an  $\times$  in the above array are not referenced by the IMSL band symmetric subprograms.

An alternate storage mode for band symmetric matrices is designated using the CHARACTER\*1 flag UPLO = 'L' in Level 2 BLAS that compute with band symmetric matrices, see [Chapter 9, Basic Matrix/Vector Operations](#). In that case, the example matrix is represented as

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{12} & A_{23} & A_{34} & A_{45} & \times \\ A_{13} & A_{24} & A_{35} & \times & \times \end{bmatrix}$$

### Band Hermitian Storage Mode

A *band Hermitian* matrix is a band matrix that is also Hermitian. The band Hermitian mode is a complex analogue of the band symmetric mode.

In the band Hermitian storage mode, the NCODA upper codiagonals are stored in the rows of a FORTRAN array of dimension  $(\text{NCODA} + 1) \times N$ . The elements are stored in the same column of the array as they are in the matrix. In the Level 2 BLAS, see [Chapter 9, Basic Matrix/Vector Operations](#), this is denoted by using the CHARACTER\*1 flag UPLO = 'U'. The array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter LDA is called the *leading dimension* of A. It must be at least as large as  $(\text{NCODA} + 1)$ . The data type of a band Hermitian array can be COMPLEX or, if your FORTRAN compiler allows, the nonstandard data type DOUBLE COMPLEX.

For example, consider a complex  $5 \times 5$  band matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N=5, NCODA = 2)
COMPLEX A(NCODA + 1, N)
```

The matrix A has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ \bar{A}_{12} & A_{22} & A_{23} & A_{24} & 0 \\ \bar{A}_{13} & \bar{A}_{23} & A_{33} & A_{34} & A_{35} \\ 0 & \bar{A}_{24} & \bar{A}_{34} & A_{44} & A_{45} \\ 0 & 0 & \bar{A}_{35} & \bar{A}_{45} & A_{55} \end{bmatrix}$$

where the value

$$\bar{A}_{ij}$$

is the complex conjugate of  $A_{ij}$ . This matrix represented as a FORTRAN array is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

The entries marked with an  $\times$  in the above array are not referenced by the IMSL band Hermitian subprograms.

An alternate storage mode for band Hermitian matrices is designated using the CHARACTER\*1 flag UPLO = 'L' in Level 2 BLAS that compute with band Hermitian matrices, see [Chapter 9, Basic Matrix/Vector Operations](#). In that case, the example matrix is represented as

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ \bar{A}_{12} & \bar{A}_{23} & \bar{A}_{34} & \bar{A}_{45} & \times \\ \bar{A}_{13} & \bar{A}_{24} & \bar{A}_{35} & \times & \times \end{bmatrix}$$

### Band Triangular Storage Mode

A *band triangular* matrix is a band matrix that is also triangular. In the band triangular storage mode, the NCODA codiagonals are stored in the rows of a FORTRAN array of dimension  $(\text{NCODA} + 1) \times N$ . The elements are stored in the same column of the array as they are in the matrix. For usage in the Level 2 BLAS, see Chapter 9, Programming Notes for BLAS, the CHARACTER\*1 flag DIAGNL has the same meaning as used in section “Triangular Storage Mode”. The flag UPLO has the meaning analogous with its usage in the section “Banded Symmetric Storage Mode”. This array is declared by the following statement:

```
DIMENSION A(LDA,N)
```

The parameter LDA is called the *leading dimension* of A. It must be at least as large as  $(\text{NCODA} + 1)$ .

For example, consider a  $5 \times 5$  band upper triangular matrix with 2 codiagonals. Its FORTRAN declaration is

```
PARAMETER (N = 5, NCODA = 2)
COMPLEX A(NCODA + 1, N)
```

The matrix A has the form

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ 0 & A_{22} & A_{23} & A_{24} & 0 \\ 0 & 0 & A_{33} & A_{34} & A_{35} \\ 0 & 0 & 0 & A_{44} & A_{45} \\ 0 & 0 & 0 & 0 & A_{55} \end{bmatrix}$$

This matrix represented as a FORTRAN array is

$$A = \begin{bmatrix} \times & \times & A_{13} & A_{24} & A_{35} \\ \times & A_{12} & A_{23} & A_{34} & A_{45} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \end{bmatrix}$$

This corresponds to the CHARACTER\*1 flags DIAGNL = 'N' and UPLO = 'U'. The matrix  $A^T$  is represented as the FORTRAN array

$$A = \begin{bmatrix} A_{11} & A_{22} & A_{33} & A_{44} & A_{55} \\ A_{12} & A_{23} & A_{34} & A_{45} & \times \\ A_{13} & A_{24} & A_{35} & \times & \times \end{bmatrix}$$

This corresponds to the CHARACTER\*1 flags DIAGNL = 'N' and UPLO = 'L'. In both examples, the entries indicated with an  $\times$  are not referenced by IMSL subprograms.

### Codiagonal Band Symmetric Storage Mode

This is an alternate storage mode for band symmetric matrices. It is not used by any of the BLAS, see [Chapter 9, Basic Matrix/Vector Operations](#). Storing data in a form transposed from the **Band Symmetric Storage Mode** maintains unit spacing between consecutive referenced array elements. This data structure is used to get good performance in the Cholesky decomposition algorithm that solves positive definite symmetric systems of linear equations  $Ax = b$ . The data type can be REAL or DOUBLE PRECISION. In the codiagonal band symmetric storage mode, the NCODA upper codiagonals and right-hand-side are stored in columns of this FORTRAN array. This array is declared by the following statement:

```
DIMENSION A(LDA, NCODA + 2)
```

The parameter LDA is the *leading positive dimension* of A. It must be at least as large as N + NCODA.

Consider a real symmetric  $5 \times 5$  matrix with 2 codiagonals

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{12} & A_{22} & A_{23} & A_{24} & 0 \\ A_{13} & A_{23} & A_{33} & A_{34} & A_{35} \\ 0 & A_{24} & A_{34} & A_{44} & A_{45} \\ 0 & 0 & A_{35} & A_{45} & A_{55} \end{bmatrix}$$

and a right-hand-side vector

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

A FORTRAN declaration for the array to hold this matrix and right-hand-side vector is

```
PARAMETER (N = 5, NCODA = 2, LDA = N + NCODA)
REAL A(LDA, NCODA + 2)
```

The matrix and right-hand-side entries are placed in the FORTRAN array A as follows:

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ A_{11} & \times & \times & b_1 \\ A_{22} & A_{12} & \times & b_2 \\ A_{33} & A_{23} & A_{13} & b_3 \\ A_{44} & A_{34} & A_{24} & b_4 \\ A_{55} & A_{45} & A_{35} & b_5 \end{bmatrix}$$

Entries marked with an  $\times$  do not need to be defined. Certain of the IMSL band symmetric subprograms will initialize and use these values during the solution process. When a solution is computed, the  $b_i, i = 1, \dots, 5$ , are replaced by  $x_i, i = 1, \dots, 5$ .

The nonzero  $A_{ij}, j \geq i$ , are stored in array locations  $A(j + \text{NCODA}, (j - i) + 1)$ . The right-hand-side entries  $b_j$  are stored in locations  $A(j + \text{NCODA}, \text{NCODA} + 2)$ . The solution entries  $x_j$  are returned in  $A(j + \text{NCODA}, \text{NCODA} + 2)$ .

### Codiagonal Band Hermitian Storage Mode

This is an alternate storage mode for band Hermitian matrices. It is not used by any of the BLAS, see [Chapter 9, Basic Matrix/Vector Operations](#). In the codiagonal band Hermitian storage mode, the real and imaginary parts of the  $2 * \text{NCODA} + 1$  upper codiagonals and right-hand-side are stored in columns of a FORTRAN array. Note that there is no explicit use of the COMPLEX or the nonstandard data type DOUBLE COMPLEX data type in this storage mode.

For *Hermitian* complex matrices,

$$A = U + \sqrt{-1}V$$

where  $U$  and  $V$  are real matrices. They satisfy the conditions  $U = U^T$  and  $V = -V^T$ . The right-hand-side

$$b = c + \sqrt{-1} d$$

where  $c$  and  $d$  are real vectors. The solution vector is denoted as

$$x = u + \sqrt{-1}v$$

where  $u$  and  $v$  are real. The storage is declared with the following statement

```
DIMENSION A(LDA, 2*NCODA + 3)
```

The parameter `LDA` is the *leading positive dimension* of  $A$ . It must be at least as large as  $N + \text{NCODA}$ .

The diagonal terms  $U_{jj}$  are stored in array locations  $A(j + \text{NCODA}, 1)$ . The diagonal  $V_{jj}$  are zero and are not stored. The nonzero  $U_{ij}, j > i$ , are stored in locations  $A(j + \text{NCODA}, 2 * (j - i))$ .

The nonzero  $V_{ij}$  are stored in locations  $A(j + \text{NCODA}, 2*(j - i) + 1)$ . The right side vector  $b$  is stored with  $c_j$  and  $d_j$  in locations  $A(j + \text{NCODA}, 2*\text{NCODA} + 2)$  and  $A(j + \text{NCODA}, 2*\text{NCODA} + 3)$  respectively. The real and imaginary parts of the solution,  $u_j$  and  $v_j$ , respectively overwrite  $c_j$  and  $d_j$ .

Consider a complex hermitian  $5 \times 5$  matrix with 2 codiagonals

$$A = \begin{bmatrix} U_{11} & U_{12} & U_{13} & 0 & 0 \\ U_{12} & U_{22} & U_{23} & U_{24} & 0 \\ U_{13} & U_{23} & U_{33} & U_{34} & U_{35} \\ 0 & U_{24} & U_{34} & U_{44} & U_{45} \\ 0 & 0 & U_{35} & U_{45} & U_{55} \end{bmatrix} + \sqrt{-1} \begin{bmatrix} 0 & V_{12} & V_{13} & 0 & 0 \\ -V_{12} & 0 & V_{23} & V_{24} & 0 \\ -V_{13} & -V_{23} & 0 & V_{34} & V_{35} \\ 0 & -V_{24} & -V_{34} & 0 & V_{45} \\ 0 & 0 & -V_{35} & -V_{45} & 0 \end{bmatrix}$$

and a right-hand-side vector

$$b = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} + \sqrt{-1} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix}$$

A FORTRAN declaration for the array to hold this matrix and right-hand-side vector is

```
PARAMETER (N = 5, NCODA = 2, LDA = N + NCODA)
REAL A(LDA, 2*NCODA + 3)
```

The matrix and right-hand-side entries are placed in the FORTRAN array  $A$  as follows:

$$A = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ U_{11} & \times & \times & \times & \times & c_1 & d_1 \\ U_{22} & U_{12} & V_{12} & \times & \times & c_2 & d_2 \\ U_{33} & U_{23} & V_{23} & U_{13} & V_{13} & c_3 & d_3 \\ U_{44} & U_{34} & V_{34} & U_{24} & V_{24} & c_4 & d_4 \\ U_{55} & U_{45} & V_{45} & U_{35} & V_{35} & c_5 & d_5 \end{bmatrix}$$

Entries marked with an  $\times$  do not need to be defined.

### Sparse Matrix Storage Mode

The sparse linear algebraic equation solvers in Chapter 1 accept the input matrix in *sparse storage mode*. This structure consists of INTEGER values N and NZ, the matrix dimension and the total number of nonzero entries in the matrix. In addition, there are two INTEGER arrays IROW(\*) and JCOL(\*) that contain unique matrix row and column coordinates where values are given. There is also an array A(\*) of values. All other entries of the matrix are zero. Each of the arrays IROW(\*), JCOL(\*), A(\*) must be of size NZ. The correspondence between matrix and array entries is given by

$$A_{\text{IROW}(i), \text{JCOL}(i)} = A(i), i = 1, \dots, \text{NZ}$$

The data type for A(\*) can be one of REAL, DOUBLE PRECISION, or COMPLEX. If your FORTRAN compiler allows, the nonstandard data type DOUBLE COMPLEX can also be declared.

For example, consider a real  $5 \times 5$  sparse matrix with 11 nonzero entries. The matrix A has the form

$$A = \begin{bmatrix} A_{11} & 0 & A_{13} & A_{14} & 0 \\ A_{21} & A_{22} & 0 & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & A_{43} & 0 & 0 \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

Declarations of arrays and definitions of the values for this sparse matrix are

```
PARAMETER (NZ = 11, N = 5)
DIMENSION IROW(NZ), JCOL(NZ), A(NZ)
DATA IROW /1, 1, 1, 2, 2, 3, 3, 3, 4, 5, 5/
DATA JCOL /1, 3, 4, 1, 2, 2, 3, 4, 3, 4, 5/
DATA A /A11, A13, A14, A21, A22, A32, A33, A34, & A43, A54, A55/
```

### Packed Symmetric Matrix Storage Mode

This structure contains either the upper or lower triangular portion of the symmetric data and is stored in an array of length  $ncol(ncol + 1)/2$ . For a matrix  $A$  and representative array  $a$ , the data is arranged sequentially column by column such that, for the upper triangular case,  $a(1)$  contains  $A_{11}$ ,  $a(2)$  contains  $A_{12}$ ,  $a(3)$  contains  $A_{22}$ , etc.

For example, consider the following real  $5 \times 5$  symmetric matrix  $A$

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

The array declaration for the upper triangle of  $A$  would be

```
DATA a /A11, A12, A22, A13, A23, A33, A14, A24, A34, A44, A15, A25, A35, A45, A55/
```

### Packed Triangular Matrix Storage Mode

This structure contains either the upper or lower triangular portion of a triangular matrix and is stored in an array of length  $ncol(ncol + 1)/2$ . For a matrix  $A$  and representative array  $a$ , the data is arranged sequentially column by column such that, for the upper triangular case,  $a(1)$  contains  $A_{11}$ ,  $a(2)$  contains  $A_{12}$ ,  $a(3)$  contains  $A_{22}$ , etc.

For example, consider the following real  $5 \times 5$  upper triangular matrix  $A$

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

The array declaration for the upper triangle of  $A$  would be

```
DATA a /A11, A12, A22, A13, A23, A33, A14, A24, A34, A44, A15, A25, A35, A45, A55/
```

### Packed Hermitian Matrix Storage Mode

This structure contains either the upper or lower triangular portion of a Hermitian matrix and is stored in an array of length  $ncol(ncol + 1)/2$ . For a matrix  $A$  and representative array  $a$ , the data is arranged sequentially column by column such that, for the upper triangular case,  $a(1)$  contains  $A_{11}$ ,  $a(2)$  contains  $A_{12}$ ,  $a(3)$  contains  $A_{22}$ , etc.

For example, consider the following  $5 \times 5$  Hermitian matrix  $A$

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \\ A_{51} & A_{52} & A_{53} & A_{45} & A_{55} \end{bmatrix}$$

The array declaration for the upper triangle of  $A$  would be

```
DATA a /A11,A12,A22,A13,A23,A33,A14,A24,A34,A44,A15,A25,A35,A45,A55/
```

## Reserved Names

When writing programs accessing the MATH/LIBRARY, the user should choose FORTRAN names that do not conflict with names of IMSL subroutines, functions, or named common blocks, such as the workspace common block `WORKSP` (see [Automatic Workspace Allocation](#)). The user needs to be aware of two types of name conflicts that can arise. The first type of name conflict occurs when a name (technically a *symbolic name*) is not uniquely defined within a program unit (either a main program or a subprogram). For example, such a name conflict exists when the name `RCURV` is used to refer both to a type `REAL` variable and to the IMSL subroutine `RCURV` in a single program unit. Such errors are detected during compilation and are easy to correct. The second type of name conflict, which can be more serious, occurs when names of program units and named common blocks are not unique. For example, such a name conflict would be caused by the user defining a subroutine named `WORKSP` and also referencing an MATH/LIBRARY subroutine that uses the named common block `WORKSP`. Likewise, the user must not define a subprogram with the same name as a subprogram in the MATH/LIBRARY, that is referenced directly by the user's program or is referenced indirectly by other MATH/LIBRARY subprograms.

The MATH/LIBRARY consists of many routines, some that are described in the *User's Manual* and others that are not intended to be called by the user and, hence, that are not documented. If the choice of names were completely random over the set of valid FORTRAN names, and if a program uses only a small subset of the MATH/LIBRARY, the probability of name conflicts is very small. Since names are usually chosen to be mnemonic, however, the user may wish to take some precautions in choosing FORTRAN names.

Many IMSL names consist of a root name that may have a prefix to indicate the type of the routine. For example, the IMSL single precision subroutine for fitting a polynomial by least squares has the name `RCURV`, which is the root name, and the corresponding IMSL double precision routine has the name `DRCURV`. Associated with these two routines are `R2URV` and `DR2URV`. `RCURV` is listed in the Alphabetical Index of Routines, but `DRCURV`, `R2URV`, and `DR2URV` are not. The user of `RCURV` must consider both names `RCURV` and `R2URV` to be reserved; likewise, the user of `DRCURV` must consider both names `DRCURV` and `DR2URV` to be reserved. The root names of *all* routines and named common blocks that are used by the MATH/LIBRARY and that do not have a numeral in the second position of the root name are listed in the Alphabetical Index of Routines. Some of the routines in this Index (such as the "Level 2 BLAS") are not intended to be called by the user and so are not documented.

The careful user can avoid any conflicts with IMSL names if the following rules are observed:

- Do not choose a name that appears in the Alphabetical Summary of Routines in the *User's Manual*, nor one of these names preceded by a D, S\_, D\_, C\_, or Z\_.
- Do not choose a name of three or more characters with a numeral in the second or third position.

These simplified rules include many combinations that are, in fact, allowable. However, if the user selects names that conform to these rules, no conflict will be encountered.

---

## Deprecated Features and Renamed Routines

### Automatic Workspace Allocation

FORTRAN subroutines that work with arrays as input and output often require extra arrays for use as workspace while doing computations or moving around data. IMSL routines generally do not require the user explicitly to allocate such arrays for use as workspace. On most systems the workspace allocation is handled transparently. The only limitation is the actual amount of memory available on the system.

On some systems the workspace is allocated out of a stack that is passed as a FORTRAN array in a named common block `WORKSP`. A very similar use of a workspace stack is described by Fox et al. (1978, pages 116–121). (For compatibility with older versions of the IMSL Libraries, space is allocated from the `COMMON` block, if possible.)

The arrays for workspace appear as arguments in lower-level routines. For example, the IMSL routine `LSARG` (in [Chapter 1, “Linear Systems”](#)), which solves systems of linear equations, needs arrays for workspace. `LSARG` allocates arrays from the common area, and passes them to the lower-level routine `L2ARG` which does the computations. In the “Comments” section of the documentation for `LSARG`, the amount of workspace is noted and the call to `L2ARG` is described. This scheme for using lower-level routines is followed throughout the IMSL Libraries. The names of these routines have a “2” in the second position (or in the third position in double precision routines having a “D” prefix). The user can provide workspace explicitly and call directly the “2-level” routine, which is documented along with the main routine. In a very few cases, the 2-level routine allows additional options that the main routine does not allow.

Prior to returning to the calling program, a routine that allocates workspace generally deallocates that space so that it becomes available for use in other routines.

### Changing the Amount of Space Allocated

*This section is relevant only to those systems on which the transparent workspace allocator is not available.*

By default, the total amount of space allocated in the common area for storage of numeric data is 5000 numeric storage units. (A numeric storage unit is the amount of space required to store an integer or a real number. By comparison, a double precision unit is twice this amount. Therefore the total amount of space allocated in the common area for storage of numeric data is 2500 double precision units.) This space is allocated as needed for `INTEGER`, `REAL`, or other numeric data. For larger problems in which the default amount of workspace is insufficient, the user can change the allocation by supplying the FORTRAN statements to define the array in the named common block and by informing the IMSL workspace allocation system of the new size of the common array. To request 7000 units, the statements are

```

COMMON /WORKSP/ RWKSP
REAL RWKSP(7000)
CALL IWKIN(7000)

```

If an IMSL routine attempts to allocate workspace in excess of the amount available in the common stack, the routine issues a fatal error message that indicates how much space is needed and prints statements like those above to guide the user in allocating the necessary amount. The program below uses IMSL routine `PERMA` to permute rows or columns of a matrix. This routine requires workspace equal to the number of columns, which in this example is too large. (Note that the work vector `RWKSP` must also provide extra space for bookkeeping.)

```

 USE_PERMA_INT
!
! Specifications for local variables
 INTEGER NRA, NCA, LDA, IPERMU(6000), IPATH
 REAL A(2,6000)
!
! Specifications for subroutines
!
 NRA = 2
 NCA = 6000
 LDA = 2
!
! Initialize permutation index
 DO 10 I = 1, NCA
 IPERMU(I) = NCA + 1 - I
10 CONTINUE
 IPATH = 2
 CALL PERMA (A, IPERMU, A, IPATH=IPATH)
 END

```

## Output

```

*** TERMINAL ERROR 10 from PERMA. Insufficient workspace for current
*** allocation(s). Correct by calling IWKIN from main program with
*** the three following statements: (REGARDLESS OF PRECISION)
*** COMMON /WORKSP/ RWKSP
*** REAL RWKSP(6018)
*** CALL IWKIN(6018)

*** TERMINAL ERROR 10 from PERMA. Workspace allocation was based on NCA =
*** 6000.

```

In most cases, the amount of workspace is dependent on the parameters of the problem so the amount needed is known exactly. In a few cases, however, the amount of workspace is dependent on the data (for example, if it is necessary to count all of the unique values in a vector), so the IMSL routine cannot tell in advance exactly how much workspace is needed. In such cases the error message printed is an estimate of the amount of space required.

## Character Workspace

Since character arrays cannot be equivalenced with numeric arrays, a separate named common block `WKSPCH` is provided for character workspace. In most respects this stack is managed in the same way as the numeric stack. The default size of the character workspace is 2000 character units. (A character unit is the amount of space required to store one character.) The routine analogous to `IWKIN` used to change the default allocation is `IWKCHN`.

The routines in the following list are being deprecated in Version 2.0 of MATH/LIBRARY. A deprecated routine is one that is no longer used by anything in the library but is being included in the product for those users who may be currently referencing it in their application. However, any future versions of MATH/LIBRARY will not include these routines. If any of these routines are being called within an application, it is recommended that you change your code or retain the deprecated routine before replacing this library with the next version. Most of these routines were called by users only when they needed to set up their own workspace. Thus, the impact of these changes should be limited.

|        |        |        |        |
|--------|--------|--------|--------|
| CZADD  | DE2LRH | DNCONF | E3CRG  |
| CZINI  | DE2LSB | DNCONG | E4CRG  |
| CZMUL  | DE3CRG | E2ASF  | E4ESF  |
| CZSTO  | DE3CRH | E2AHF  | E5CRG  |
| DE2AHF | DE3LSF | E2BHF  | E7CRG  |
| DE2ASF | DE4CRG | E2BSB  | G2CCG  |
| DE2BHF | DE4ESF | E2BSF  | G2CRG  |
| DE2BSB | DE5CRG | E2CCG  | G2LCG  |
| DE2BSF | DE7CRG | E2CCH  | G2LRG  |
| DE2CCG | DG2CCG | E2CHF  | G3CCG  |
| DE2CCH | DG2CRG | E2CRG  | G4CCG  |
| DE2CHF | DG2DF  | E2CRH  | G5CCG  |
| DE2CRG | DG2IND | E2CSB  | G7CRG  |
| DE2CRH | DG2LCG | E2EHF  | N0ONF  |
| DE2CSB | DG2LRG | E2ESB  | NCONF  |
| DE2EHF | DG3CCG | E2FHF  | NCONG  |
| DE2ESB | DG3DF  | E2FSB  | SDADD  |
| DE2FHF | DG4CCG | E2FSF  | SDINI  |
| DE2FSB | DG5CCG | E2LCG  | SDMUL  |
| DE2FSF | DG7CRG | E2LCH  | SDSTO  |
| DE2LCG | DHOUAP | E2LHF  | SHOUAP |
| DE2LCH | DHOUTR | E2LRG  | SHOUTR |
| DE2LHF | DIVPBS | E2LRH  |        |
| DE2LRG | DN0ONF | E2LSB  |        |

The following routines have been renamed due to naming conflicts with other software manufacturers.

CTIME — replaced with CPSEC

DTIME — replaced with TIMDY

PAGE — replaced with PGOPT



# Appendix A: GAMS Index

---

## Description

This index lists routines in MATH LIBRARY by a tree-structured classification scheme known as GAMS Version 2.0 (Boisvert, Howe, Kahaner, and Springmann (1990)). Only the GAMS classes that contain MATH/LIBRARY routines are included in the index. The page number for the documentation and the purpose of the routine appear alongside the routine name.

The first level of the full classification scheme contains the following major subject areas:

- A. Arithmetic, Error Analysis
- B. Number Theory
- C. Elementary and Special Functions
- D. Linear Algebra
- E. Interpolation
- F. Solution of Nonlinear Equations
- G. Optimization
- H. Differentiation and Integration
- I. Differential and Integral Equations
- J. Integral Transforms
- K. Approximation
- L. Statistics, Probability
- M. Simulation, Stochastic Modeling
- N. Data Handling
- O. Symbolic Computation
- P. Computational Geometry
- Q. Graphics
- R. Service Routines
- S. Software Development Tools

## Z. Other

There are seven levels in the classification scheme. Classes in the first level are identified by a capital letter as is given above. Classes in the remaining levels are identified by alternating letter-and-number combinations. A single letter (a-z) is used with the odd-numbered levels. A number (1–26) is used within the even-numbered levels.

---

# IMSL MATH LIBRARY

## A.....ARITHMETIC, ERROR ANALYSIS

A3.....Real

A3c.....Extended precision

DQADD . Adds a double-precision scalar to the accumulator in extended precision.

DQINI...Initializes an extended-precision accumulator with a double-precision scalar.

DQMUL...Multiplies double-precision scalars in extended precision.

DQSTO...Stores a double-precision approximation to an extended-precision scalar.

A4.....Complex

A4c.....Extended precision

ZQADD...Adds a double complex scalar to the accumulator in extended precision.

ZQINI...Initializes an extended-precision complex accumulator to a double complex scalar.

ZQMUL...Multiplies double complex scalars using extended precision.

ZQSTO...Stores a double complex approximation to an extended-precision complex scalar.

A6.....Change of representation

A6c.....Decomposition, construction

PRIME...Decomposes an integer into its prime factors.

## B.....NUMBER THEORY

PRIME . Decomposes an integer into its prime factors.

## C.....ELEMENTARY AND SPECIAL FUNCTIONS

C2.....Powers, roots, reciprocals

HYPOT...Computes  $\sqrt{a^2 + b^2}$  without underflow or overflow.

C19.....Other special functions

CONST...Returns the value of various mathematical and physical constants.

CUNIT . Converts  $X$  in units XUNITS to  $Y$  in units YUNITS.

D..... LINEAR ALGEBRA

D1..... Elementary vector and matrix operations

D1a..... Elementary vector operations

D1a1..... Set to constant

CSET..... Sets the components of a vector to a scalar, all complex.

ISET..... Sets the components of a vector to a scalar, all integer.

SSET..... Sets the components of a vector to a scalar, all single precision.

D1a2..... Minimum and maximum components

ICAMAX. Finds the smallest index of the component of a complex vector having maximum magnitude.

ICAMIN. Finds the smallest index of the component of a complex vector having minimum magnitude.

IIMAX... Finds the smallest index of the maximum component of a integer vector.

IIMIN... Finds the smallest index of the minimum of an integer vector.

ISAMAX Finds the smallest index of the component of a single-precision vector having maximum absolute value.

ISAMIN. Finds the smallest index of the component of a single-precision vector having minimum absolute value.

ISMAX... Finds the smallest index of the component of a single-precision vector having maximum value.

ISMIN... Finds the smallest index of the component of a single-precision vector having minimum value.

D1a3..... Norm

D1a3a...  $L_1$  (sum of magnitudes)

DISL1... Computes the 1-norm distance between two points.

SASUM... Sums the absolute values of the components of a single-precision vector.

SCASUM. Sums the absolute values of the real part together with the absolute values of the imaginary part of the components of a complex vector.

D1a3b...  $L_2$  (Euclidean norm)

DISL2... Computes the Euclidean (2-norm) distance between two points.

NORM2, CNORM2      Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions.

MNORM2, CMNORM2      Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions

NRM2, CNRM2        Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions.

SCNRM2. Computes the Euclidean norm of a complex vector.

SNRM2... Computes the Euclidean length or  $L_2$  norm of a single-precision vector.

D1a3c...  $L_\infty$  (maximum magnitude)

DISLI... Computes the infinity norm distance between two points.

ICAMAX. Finds the smallest index of the component of a complex vector having maximum magnitude.

ISAMAX. Finds the smallest index of the component of a single-precision vector having maximum absolute value.

D1a4..... Dot product (inner product)

CDOTC... Computes the complex conjugate dot product,  $\bar{x}^T y$ .

CDOTU.. Computes the complex dot product  $x^T y$ .

CZCDOT. Computes the sum of a complex scalar plus a complex conjugate dot product,  $a + \bar{x}^T y$ , using a double-precision accumulator.

CZDOTA. Computes the sum of a complex scalar, a complex dot product and the double-complex accumulator, which is set to the result  $ACC \leftarrow ACC + a + x^T y$ .

CZDOTC. Computes the complex conjugate dot product,  $\bar{x}^T y$ , using a double-precision accumulator.

CZDOTI. Computes the sum of a complex scalar plus a complex dot product using a double-complex accumulator, which is set to the result  $ACC \leftarrow a + x^T y$ .

CZDOTU. Computes the complex dot product  $x^T y$  using a double-precision accumulator.

CZUDOT. Computes the sum of a complex scalar plus a complex dot product,  $a + x^T y$ , using a double-precision accumulator.

DSDOT... Computes the single-precision dot product  $x^T y$  using a double precision accumulator.

SDDOTA. Computes the sum of a single-precision scalar, a single-precision dot product and the double-precision accumulator, which is set to the result  $ACC \leftarrow ACC + a + x^T y$ .

SDDOTI. Computes the sum of a single-precision scalar plus a single-precision dot product using a double-precision accumulator, which is set to the result  $ACC \leftarrow a + x^T y$ .

SDOT..... Computes the single-precision dot product  $x^T y$ .

SDSDOT. Computes the sum of a single-precision scalar and a single precision dot product,  $a + x^T y$ , using a double-precision accumulator.

D1a5..... Copy or exchange (swap)

CCOPY... Copies a vector  $x$  to a vector  $y$ , both complex.

CSWAP... Interchanges vectors  $x$  and  $y$ , both complex.

ICOPY... Copies a vector  $x$  to a vector  $y$ , both integer.

ISWAP... Interchanges vectors  $x$  and  $y$ , both integer.

SCOPY... Copies a vector  $x$  to a vector  $y$ , both single precision.

SSWAP... Interchanges vectors  $x$  and  $y$ , both single precision.

D1a6..... Multiplication by scalar

CSCAL... Multiplies a vector by a scalar,  $y \leftarrow ay$ , both complex.

CSSCAL. Multiplies a complex vector by a single-precision scalar,  $y \leftarrow ay$ .

CSVCAL. Multiplies a complex vector by a single-precision scalar and store the result in another complex vector,  $y \leftarrow ax$ .

CVCAL... Multiplies a vector by a scalar and store the result in another vector,  $y \leftarrow ax$ , all complex.

SSCAL... Multiplies a vector by a scalar,  $y \leftarrow ay$ , both single precision.

SVCAL... Multiplies a vector by a scalar and store the result in another vector,  $y \leftarrow ax$ , all single precision.

D1a7..... Triad ( $ax + y$  for vectors  $x$ ,  $y$  and scalar  $a$ )

CAXPY... Computes the scalar times a vector plus a vector,  $y \leftarrow ax + y$ , all complex.

SAXPY... Computes the scalar times a vector plus a vector,  $y \leftarrow ax + y$ , all single precision.

D1a8..... Elementary rotation (Givens transformation) (*search also class D1b10*)

CSROT... Applies a complex Givens plane rotation.

CSROTM. Applies a complex modified Givens plane rotation.

SROT..... Applies a Givens plane rotation in single precision.

SROTM... Applies a modified Givens plane rotation in single precision.

D1a10... Convolutions

RCONV... Computes the convolution of two real vectors.

VCONC... Computes the convolution of two complex vectors.

VCONR... Computes the convolution of two real vectors.

D1a11...Other vector operations

CADD . . Adds a scalar to each component of a vector,  $x \leftarrow x + a$ , all complex.

CSUB.....Subtracts each component of a vector from a scalar,  
 $x \leftarrow a - x$ , all complex.

DISL1...Computes the 1-norm distance between two points.

DISL2...Computes the Euclidean (2-norm) distance between two points.

DISLI...Computes the infinity norm distance between two points.

IADD.....Adds a scalar to each component of a vector,  $x \leftarrow x + a$ , all integer.

ISUB.....Subtracts each component of a vector from a scalar,  
 $x \leftarrow a - x$ , all integer.

ISUM.....Sums the values of an integer vector.

SADD.....Adds a scalar to each component of a vector,  $x \leftarrow x + a$ , all single precision.

SHPROD.Computes the Hadamard product of two single-precision vectors.

SPRDCT.Multiplies the components of a single-precision vector.

SSUB.....Subtracts each component of a vector from a scalar,  
 $x \leftarrow a - x$ , all single precision.

SSUM.....Sums the values of a single-precision vector.

SXYZ.....Computes a single-precision xyz product.

D1b.....Elementary matrix operations

CGERC...Computes the rank-one update of a complex general matrix:

$$A \leftarrow A + \alpha x \bar{y}^T .$$

CGERU...Computes the rank-one update of a complex general matrix:

$$A \leftarrow A + \alpha x y^T .$$

CHER.....Computes the rank-one update of an Hermitian matrix:

$$A \leftarrow A + \alpha x \bar{x}^T \text{ with } x \text{ complex and } \alpha \text{ real.}$$

CHER2 . Computes a rank-two update of an Hermitian matrix:

$$A \leftarrow A + \alpha x \bar{y}^T + \bar{\alpha} y \bar{x}^T .$$

CHER2K.Computes one of the Hermitian rank  $2k$  operations:

$$C \leftarrow \alpha A \bar{B}^T + \bar{\alpha} B \bar{A}^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T B + \bar{\alpha} B^T A + \beta C ,$$

where  $C$  is an  $n$  by  $n$  Hermitian matrix and  $A$  and  $B$  are  $n$  by  $k$  matrices in the first case and  $k$  by  $n$  matrices in the second case.

CHERK...Computes one of the Hermitian rank  $k$  operations:

$$C \leftarrow \alpha A \bar{A}^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T A + \beta C ,$$

where  $C$  is an  $n$  by  $n$  Hermitian matrix and  $A$  is an  $n$  by  $k$  matrix in the first case and a  $k$  by  $n$  matrix in the second case.

CSYR2K. Computes one of the symmetric rank  $2k$  operations:

$$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \text{ or } C \leftarrow \alpha A^T B + \alpha B^T A + \beta C,$$

where  $C$  is an  $n$  by  $n$  symmetric matrix and  $A$  and  $B$  are  $n$  by  $k$  matrices in the first case and  $k$  by  $n$  matrices in the second case.

CSYRK... Computes one of the symmetric rank  $k$  operations:

$$C \leftarrow \alpha AA^T + \beta C \text{ or } C \leftarrow \alpha A^T A + \beta C,$$

where  $C$  is an  $n$  by  $n$  symmetric matrix and  $A$  is an  $n$  by  $k$  matrix in the first case and a  $k$  by  $n$  matrix in the second case.

CTBSV... Solves one of the complex triangular systems:

$$x \leftarrow A^{-1}x, x \leftarrow (A^{-1})^T x, \text{ or } x \leftarrow (\bar{A}^T)^{-1} x,$$

where  $A$  is a triangular matrix in band storage mode.

CTRSM... Solves one of the complex matrix equations:

$$B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha (A^{-1})^T B, B \leftarrow \alpha B(A^{-1})^T,$$

$$B \leftarrow \alpha (\bar{A}^T)^{-1} B, \text{ or } B \leftarrow \alpha B(\bar{A}^T)^{-1}$$

where  $A$  is a triangular matrix.

CTRSV. Solves one of the complex triangular systems:

$$x \leftarrow A^{-1}x, x \leftarrow (A^{-1})^T x, \text{ or } x \leftarrow (\bar{A}^T)^{-1} x,$$

where  $A$  is a triangular matrix.

HRRRR... Computes the Hadamard product of two real rectangular matrices.

SGER... Computes the rank-one update of a real general matrix:

$$A \leftarrow A + \alpha xy^T.$$

SSYR... Computes the rank-one update of a real symmetric matrix:

$$A \leftarrow A + \alpha xx^T.$$

SSYR2. Computes the rank-two update of a real symmetric matrix:

$$A \leftarrow A + \alpha xy^T + \alpha yx^T.$$

SSYR2K Computes one of the symmetric rank  $2k$  operations:

$$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \text{ or } C \leftarrow \alpha A^T B + \alpha B^T A + \beta C,$$

where  $C$  is an  $n$  by  $n$  symmetric matrix and  $A$  and  $B$  are  $n$  by  $k$  matrices in the first case and  $k$  by  $n$  matrices in the second case.

SSYRK... Computes one of the symmetric rank  $k$  operations:

$$C \leftarrow \alpha AA^T + \beta C \text{ or } C \leftarrow \alpha A^T A + \beta C,$$

where  $C$  is an  $n$  by  $n$  symmetric matrix and  $A$  is an  $n$  by  $k$  matrix in the first case and a  $k$  by  $n$  matrix in the second case.

STBSV...Solves one of the triangular systems:

$$x \leftarrow A^{-1}x \text{ or } x \leftarrow (A^{-1})^T x,$$

where  $A$  is a triangular matrix in band storage mode.

STRSM . Solves one of the matrix equations:

where  $B$  is an  $m$  by  $n$  matrix and  $A$  is a triangular matrix.

STRSV . Solves one of the triangular linear systems:

$$x \leftarrow A^{-1}x \text{ or } x \leftarrow (A^{-1})^T x,$$

where  $A$  is a triangular matrix.

D1b2.....Norm

NR1CB...Computes the 1-norm of a complex band matrix in band storage mode.

NR1RB...Computes the 1-norm of a real band matrix in band storage mode.

NR1RR...Computes the 1-norm of a real matrix.

NR2RR...Computes the Frobenius norm of a real rectangular matrix.

NRIRR...Computes the infinity norm of a real matrix.

D1b3.....Transpose

TRNR...Transposes a rectangular matrix.

D1b4.....Multiplication by vector

BLINF...Computes the bilinear form  $x^T A y$ .

CGBMV...Computes one of the matrix-vector operations:

$$y \leftarrow \alpha A x + \beta y, y \leftarrow \alpha A^T x + \beta y, \text{ or } y \leftarrow \alpha \bar{A}^T + \beta y,$$

where  $A$  is a matrix stored in band storage mode.

CGEMV...Computes one of the matrix-vector operations:

$$y \leftarrow \alpha A x + \beta y, y \leftarrow \alpha A^T x + \beta y, \text{ or } y \leftarrow \alpha \bar{A}^T + \beta y,$$

CHBMV...Computes the matrix-vector operation

$$y \leftarrow \alpha A x + \beta y,$$

where  $A$  is an Hermitian band matrix in band Hermitian storage.

CHEMV...Computes the matrix-vector operation

$$y \leftarrow \alpha A x + \beta y,$$

where  $A$  is an Hermitian matrix.

CTBMV . Computes one of the matrix-vector operations:

$$x \leftarrow A x, x \leftarrow A^T x, \text{ or } x \leftarrow \bar{A}^T x,$$

where  $A$  is a triangular matrix in band storage mode.

CTRMV... Computes one of the matrix-vector operations:  
 $x \leftarrow Ax, x \leftarrow A^T x, \text{ or } x \leftarrow \bar{A}^T x,$   
 where  $A$  is a triangular matrix.

MUCBV. Multiplies a complex band matrix in band storage mode by a complex vector.

MUCRV... Multiplies a complex rectangular matrix by a complex vector.

MURBV... Multiplies a real band matrix in band storage mode by a real vector.

MURRV... Multiplies a real rectangular matrix by a vector.

SGBMV. Computes one of the matrix-vector operations:  
 $y \leftarrow \alpha Ax + \beta y, \text{ or } y \leftarrow \alpha A^T x + \beta y,$   
 where  $A$  is a matrix stored in band storage mode.

SGEMV... Computes one of the matrix-vector operations:  
 $y \leftarrow \alpha Ax + \beta y, \text{ or } y \leftarrow \alpha A^T x + \beta y,$

SSBMV. Computes the matrix-vector operation  
 $y \leftarrow \alpha Ax + \beta y,$   
 where  $A$  is a symmetric matrix in band symmetric storage mode.

SSYMV... Computes the matrix-vector operation  
 $y \leftarrow \alpha Ax + \beta y,$   
 where  $A$  is a symmetric matrix.

STBMV... Computes one of the matrix-vector operations:  
 $x \leftarrow Ax \text{ or } x \leftarrow A^T x,$   
 where  $A$  is a triangular matrix in band storage mode.

STRMV .. Computes one of the matrix-vector operations:  
 $x \leftarrow Ax \text{ or } x \leftarrow A^T x,$   
 where  $A$  is a triangular matrix.

D1b5..... Addition, subtraction

ACBCB... Adds two complex band matrices, both in band storage mode.

ARBRB. Adds two band matrices, both in band storage mode.

D1b6..... Multiplication

CGEMM... Computes one of the matrix-matrix operations:  
 $C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$   
 $+ \beta C, C \leftarrow \alpha A^T B^T + \beta C, C \leftarrow \alpha \bar{A} \bar{B}^T + \beta C,$   
 or  $C \leftarrow \alpha \bar{A}^T B + \beta C, C \leftarrow \alpha A^T \bar{B}^T + \beta C,$   
 $C \leftarrow \alpha \bar{A}^T B^T + \beta C, \text{ or } C \leftarrow \alpha \bar{A} \bar{B}^T + \beta C$

CHEMM . Computes one of the matrix-matrix operations:  
 $C \leftarrow \alpha AB + \beta C$  or  $C \leftarrow \alpha BA + \beta C$ ,  
 where  $A$  is an Hermitian matrix and  $B$  and  $C$  are  $m$  by  $n$  matrices.

CSYMM . Computes one of the matrix-matrix operations:  
 $C \leftarrow \alpha AB + \beta C$  or  $C \leftarrow \alpha BA + \beta C$ ,  
 where  $A$  is a symmetric matrix and  $B$  and  $C$  are  $m$  by  $n$  matrices.

CTRMM . Computes one of the matrix-matrix operations:  
 $B \leftarrow \alpha AB$ ,  $B \leftarrow \alpha A^T B$ ,  $B \leftarrow \alpha BA$ ,  $B \leftarrow \alpha BA^T$ ,  
 $B \leftarrow \alpha \bar{A}^T B$ , or  $B \leftarrow \alpha B \bar{A}^T$   
 where  $B$  is an  $m$  by  $n$  matrix and  $A$  is a triangular matrix.

MCRCR...Multiplies two complex rectangular matrices,  $AB$ .

MRRRR...Multiplies two real rectangular matrices,  $AB$ .

MXTXF...Computes the transpose product of a matrix,  $A^T A$ .

MXTYF...Multiplies the transpose of matrix  $A$  by matrix  $B$ ,  $A^T B$ .

MXYTF...Multiplies a matrix  $A$  by the transpose of a matrix  $B$ ,  $AB^T$ .

SGEMM...Compute one of the matrix-matrix operations:  
 $C \leftarrow \alpha AB + \beta C$ ,  $C \leftarrow \alpha A^T B + \beta C$ ,  $C \leftarrow \alpha AB^T$   
 $+ \beta C$ , or  $C \leftarrow \alpha A^T B^T + \beta C$

SSYMM...Computes one of the matrix-matrix operations:  
 $C \leftarrow \alpha AB + \beta C$  or  $C \leftarrow \alpha BA + \beta C$ ,  
 where  $A$  is a symmetric matrix and  $B$  and  $C$  are  $m$  by  $n$  matrices.

STRMM . Computes one of the matrix-matrix operations:  
 $B \leftarrow \alpha AB$ ,  $B \leftarrow \alpha A^T B$ , or  $B \leftarrow \alpha BA$ ,  $B \leftarrow \alpha BA^T$ ,  
 where  $B$  is an  $m$  by  $n$  matrix and  $A$  is a triangular matrix.

D1b7.....Matrix polynomial

POLRG . 1207 Evaluates a real general matrix polynomial.

D1b8.....Copy

CCBCB . Copies a complex band matrix stored in complex band storage mode.

CCGCG Copies a complex general matrix.

CRBRB...Copies a real band matrix stored in band storage mode.

CRGRG Copies a real general matrix.

D1b9.....Storage mode conversion

CCBCG...Converts a complex matrix in band storage mode to a complex matrix in full storage mode.

CCGCB... Converts a complex general matrix to a matrix in complex band storage mode.

CHBCB. Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.

CHFCG... Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.

CRBCB... Converts a real matrix in band storage mode to a complex matrix in band storage mode.

CRBRG... Converts a real matrix in band storage mode to a real general matrix.

CRGCG. Copies a real general matrix to a complex general matrix.

CRGRB... Converts a real general matrix to a matrix in band storage mode.

CRRCR... Copies a real rectangular matrix to a complex rectangular matrix.

CSBRB... Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.

CSFRG... Extends a real symmetric matrix defined in its upper triangle to its lower triangle.

D1b10... Elementary rotation (Givens transformation) (*search also class D1a8*)

SROTG... Constructs a Givens plane rotation in single precision.

SROTMG. Constructs a modified Givens plane rotation in single precision.

D2..... Solution of systems of linear equations (including inversion, *LU* and related decompositions)

D2a..... Real nonsymmetric matrices

LSLTO... Solves a real Toeplitz linear system.

D2a1 ..... General

LFCRG... Computes the *LU* factorization of a real general matrix and estimate its  $L_1$  condition number.

LFIRG... Uses iterative refinement to improve the solution of a real general system of linear equations.

LFSRG... Solves a real general system of linear equations given the *LU* factorization of the coefficient matrix.

LFTRG... Computes the *LU* factorization of a real general matrix.

LINRG... Computes the inverse of a real general matrix.

LSARG... Solves a real general system of linear equations with iterative refinement.

LSLRG... Solves a real general system of linear equations without iterative refinement.

LIN\_SOL\_GEN Solves a general system of linear equations  $Ax = b$ . Using optional arguments, any of several related computations can be performed. These extra tasks include computing the  $LU$  factorization of  $A$  using partial pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , and solving  $A^T x = b$  or  $Ax = b$  given the  $LU$  factorization of  $A$ .

#### D2a2.....Banded

LFCRB...Computes the  $LU$  factorization of a real matrix in band storage mode and estimate its  $L_1$  condition number.

LFIRB...Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.

LFSRB . Solves a real system of linear equations given the  $LU$  factorization of the coefficient matrix in band storage mode.

LFTRB . Computes the  $LU$  factorization of a real matrix in band storage mode.

LSARB...Solves a real system of linear equations in band storage mode with iterative refinement.

LSLRB...Solves a real system of linear equations in band storage mode without iterative refinement.

STBSV...Solves one of the triangular systems:  $x \leftarrow A^{-1}x$  or  $x \leftarrow (A^{-1})^T x$ , where  $A$  is a triangular matrix in band storage mode.

#### D2a2a ...Tridiagonal

LSLCR . Computes the  $LDU$  factorization of a real tridiagonal matrix  $A$  using a cyclic reduction algorithm.

LSLTR...Solves a real tridiagonal system of linear equations.

LIN\_SOL\_TRI Solves multiple systems of linear equations  $A_j x_j = y_j, j = 1, \dots, k$ . Each matrix  $A_j$  is tridiagonal with the same dimension,  $n$ : The default solution method is based on  $LU$  factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

TRI\_SOLVE A real, tri-diagonal, multiple system solver. Uses both cyclic reduction and Gauss elimination. Similar in function to `lin_sol_tri`.

#### D2a3.....Triangular

LFCRT...Estimates the condition number of a real triangular matrix.

LINRT...Computes the inverse of a real triangular matrix.

LSLRT . Solves a real triangular system of linear equations.

STRSM... Solves one of the matrix equations:

$$B \leftarrow \alpha A^{-1} B, B \leftarrow \alpha B A^{-1}, B \leftarrow \alpha (A^{-1})^T B$$
$$\text{or } B \leftarrow \alpha B (A^{-1})^T,$$

where  $B$  is an  $m$  by  $n$  matrix and  $A$  is a triangular matrix.

STRSV. Solves one of the triangular linear systems:

$$x \leftarrow A^{-1} x \text{ or } x \leftarrow (A^{-1})^T x$$

where  $A$  is a triangular matrix.

D2a4..... Sparse

LFSXG... Solves a sparse system of linear equations given the  $LU$  factorization of the coefficient matrix.

LFTXG... Computes the  $LU$  factorization of a real general sparse matrix.

LSLXG... Solves a sparse system of linear algebraic equations by Gaussian elimination.

GMRES... Uses restarted GMRES with reverse communication to generate an approximate solution of  $Ax = b$ .

D2b..... Real symmetric matrices

D2b1..... General

D2b1a... Indefinite

LCHRG... Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.

LFCSF... Computes the  $U D U^T$  factorization of a real symmetric matrix and estimate its  $L_1$  condition number.

LFISF... Uses iterative refinement to improve the solution of a real symmetric system of linear equations.

LFSSF... Solves a real symmetric system of linear equations given the  $U D U^T$  factorization of the coefficient matrix.

LFTSF... Computes the  $U D U^T$  factorization of a real symmetric matrix.

LSASF. Solves a real symmetric system of linear equations with iterative refinement.

LSLSF... Solves a real symmetric system of linear equations without iterative refinement.

LIN\_SOL\_SELF Solves a system of linear equations  $Ax = b$ , where  $A$  is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using symmetric pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , or computing

the solution of  $Ax = b$  given the factorization of  $A$ . An optional argument is provided indicating that  $A$  is positive definite so that the Cholesky decomposition can be used.

#### D2b1b...Positive definite

- LCHRG . Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.
- LFCDS...Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix and estimate its  $L_1$  condition number.
- LFIDS...Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.
- LFSDS . Solves a real symmetric positive definite system of linear equations given the  $R^T R$  Cholesky factorization of the coefficient matrix.
- LFTDS...Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix.
- LINDS . Computes the inverse of a real symmetric positive definite matrix.
- LSADS . Solves a real symmetric positive definite system of linear equations with iterative refinement.
- LSLDS . Solves a real symmetric positive definite system of linear equations without iterative refinement.
- LIN\_SOL\_SELF        Solves a system of linear equations  $Ax = b$ , where  $A$  is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using symmetric pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , or computing the solution of  $Ax = b$  given the factorization of  $A$ . An optional argument is provided indicating that  $A$  is positive definite so that the Cholesky decomposition can be used.

#### D2b2.....Positive definite banded

- LFCQS . Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode and estimate its  $L_1$  condition number.
- LFDQS...Computes the determinant of a real symmetric positive definite matrix given the  $R^T R$  Cholesky factorization of the band symmetric storage mode.
- LFIQS...Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.
- LFSQS...Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.

LFTQS... Computes the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.

LSAQS... Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.

LSLPB... Computes the  $R^T DR$  Cholesky factorization of a real symmetric positive definite matrix  $A$  in codiagonal band symmetric storage mode. Solve a system  $Ax = b$ .

LSLQS... Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement.

D2b4..... Sparse

JCGRC... Solves a real symmetric definite linear system using the Jacobi preconditioned conjugate gradient method with reverse communication.

LFSXD... Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

LNFXD... Computes the numerical Cholesky factorization of a sparse symmetrical matrix  $A$ .

LSCXD... Performs the symbolic Cholesky factorization for a sparse symmetric matrix using a minimum degree ordering or a userspecified ordering, and set up the data structure for the numerical Cholesky factorization.

LSLXD... Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.

PCGRC... Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication.

D2c..... Complex non-Hermitian matrices

LSLCC. Solves a complex circulant linear system.

LSLTC... Solves a complex Toeplitz linear system.

D2c1..... General

LFCCG... Computes the  $LU$  factorization of a complex general matrix and estimate its  $L_1$  condition number.

LFICG... Uses iterative refinement to improve the solution of a complex general system of linear equations.

LFSCG... Solves a complex general system of linear equations given the  $LU$  factorization of the coefficient matrix.

LFTCG... Computes the  $LU$  factorization of a complex general matrix.

LINCG. Computes the inverse of a complex general matrix.

LSACG... Solves a complex general system of linear equations with iterative refinement.

LSLCG...Solves a complex general system of linear equations without iterative refinement.

LIN\_SOL\_GEN       Solves a general system of linear equations  $Ax = b$ . Using optional arguments, any of several related computations can be performed. These extra tasks include computing the  $LU$  factorization of  $A$  using partial pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , and solving  $A^T x = b$  or  $Ax = b$  given the  $LU$  factorization of  $A$ .

#### D2c2.....Banded

CTBSV...Solves one of the complex triangular systems:

$$x \leftarrow A^{-1}x, x \leftarrow (A^{-1})^T x, \text{ or } x \leftarrow (\bar{A}^T)^{-1} x,$$

where  $A$  is a triangular matrix in band storage mode.

LFCCB...Computes the  $LU$  factorization of a complex matrix in band storage mode and estimate its  $L_1$  condition number.

LFICB...Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.

LFSCB...Solves a complex system of linear equations given the  $LU$  factorization of the coefficient matrix in band storage mode.

LFTCB...Computes the  $LU$  factorization of a complex matrix in band storage mode.

LSACB...Solves a complex system of linear equations in band storage mode with iterative refinement.

LSLCB...Solves a complex system of linear equations in band storage mode without iterative refinement.

#### D2c2a...Tridiagonal

LSLCQ . Computes the  $LDU$  factorization of a complex tridiagonal matrix  $A$  using a cyclic reduction algorithm.

LSLTQ...Solves a complex tridiagonal system of linear equations.

LIN\_SOL\_TRI       Solves multiple systems of linear equations  $A_j x_j = y_j, j = 1, \dots, k$ . Each matrix  $A_j$  is tridiagonal with the same dimension,  $n$ : The default solution method is based on  $LU$  factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

#### D2c3.....Triangular

CTRSM. Solves one of the complex matrix equations:

$$B \leftarrow \alpha A^{-1} B, B \leftarrow \alpha B A^{-1}, B \leftarrow \alpha (A^{-1})^T B, B \leftarrow \alpha B (A^{-1})^T, \\ B \leftarrow \alpha (\bar{A}^T)^{-1} B, \text{ or } B \leftarrow \alpha B (\bar{A}^T)^{-1}$$

where  $A$  is a triangular matrix.

CTRSV...Solves one of the complex triangular systems:

$$x \leftarrow A^{-1} x, x \leftarrow (A^{-1})^T x, \text{ or } x \leftarrow (\bar{A}^T)^{-1} x$$

where  $A$  is a triangular matrix.

LFCCT. Estimates the condition number of a complex triangular matrix.

LINCT...Computes the inverse of a complex triangular matrix.

LSLCT...Solves a complex triangular system of linear equations.

D2c4.....Sparse

LFSZG...Solves a complex sparse system of linear equations given the  $LU$  factorization of the coefficient matrix.

LFTZG...Computes the  $LU$  factorization of a complex general sparse matrix.

LSLZG...Solves a complex sparse system of linear equations by Gaussian elimination.

D2d.....Complex Hermitian matrices

D2d1.....General

D2d1a...Indefinite

LFCHF...Computes the  $U D U^H$  factorization of a complex Hermitian matrix and estimate its  $L_1$  condition number.

LFDFH...Computes the determinant of a complex Hermitian matrix given the  $U D U^H$  factorization of the matrix.

LFTHF...Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.

LFSHF...Solves a complex Hermitian system of linear equations given the  $U D U^H$  factorization of the coefficient matrix.

LFTHF...Computes the  $U D U^H$  factorization of a complex Hermitian matrix.

LSAHF...Solves a complex Hermitian system of linear equations with iterative refinement.

LSLHF...Solves a complex Hermitian system of linear equations without iterative refinement.

LIN\_SOL\_SELF Solves a system of linear equations  $Ax = b$ , where  $A$  is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing

and saving the factorization of  $A$  using symmetric pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , or computing the solution of  $Ax = b$  given the factorization of  $A$ . An optional argument is provided indicating that  $A$  is positive definite so that the Cholesky decomposition can be used.

#### D2d1b...Positive definite

LFCDH...Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix and estimate its  $L_1$  condition number.

LFI DH...Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.

LFS DH...Solves a complex Hermitian positive definite system of linear equations given the  $R^H R$  factorization of the coefficient matrix.

LFT DH...Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix.

LSADH...Solves a Hermitian positive definite system of linear equations with iterative refinement.

LSLDH...Solves a complex Hermitian positive definite system of linear equations without iterative refinement.

LIN\_SOL\_SELF      Solves a system of linear equations  $Ax = b$ , where  $A$  is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using symmetric pivoting, representing the determinant of  $A$ , computing the inverse matrix  $A^{-1}$ , or computing the solution of  $Ax = b$  given the factorization of  $A$ . An optional argument is provided indicating that  $A$  is positive definite so that the Cholesky decomposition can be used.

#### D2d2.....Positive definite banded

LFCQH...Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode and estimate its  $L_1$  condition number.

LFIQH...Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode.

LFSQH...Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode.

LFTQH...Computes the  $R^H R$  factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.

LSAQH...Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.

LSLQB. Computes the  $R^H DR$  Cholesky factorization of a complex hermitian positive-definite matrix  $A$  in codiagonal band hermitian storage mode. Solve a system  $Ax = b$ .

LSLQH...Solves a complex Hermitian positive definite system of linearequations in band Hermitian storage mode without iterative refinement.

D2d4..... Sparse

LFSZD...Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.

LNFDZD...Computes the numerical Cholesky factorization of a sparse Hermitian matrix  $A$ .

LSLZD...Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.

D3.....Determinants

D3a..... Real nonsymmetric matrices

D3a1..... General

LFDRG. Computes the determinant of a real general matrix given the  $LU$  factorization of the matrix.

D3a2..... Banded

LFDRB. Computes the determinant of a real matrix in band storage mode given the  $LU$  factorization of the matrix.

D3a3..... Triangular

LFDRT...Computes the determinant of a real triangular matrix.

D3b..... Real symmetric matrices

D3b1..... General

D3b1a... Indefinite

LFDSF...Computes the determinant of a real symmetric matrix given the  $U DU^T$  factorization of the matrix.

D3b1b... Positive definite

LFDDS. Computes the determinant of a real symmetric positive definite matrix given the  $R^H R$  Cholesky factorization of the matrix.

D3c..... Complex non-Hermitian matrices

D3c1..... General

LFDCG...Computes the determinant of a complex general matrix given the  $LU$  factorization of the matrix.

D3c2..... Banded

LFDCEB...Computes the determinant of a complex matrix given the  $LU$  factorization of the matrix in band storage mode.

D3c3.....Triangular

LFDCT . Computes the determinant of a complex triangular matrix.

D3d.....Complex Hermitian matrices

D3d1.....General

D3d1b...Positive definite

LFDDEH...Computes the determinant of a complex Hermitian positive definite matrix given the  $R^H R$  Cholesky factorization of the matrix.

D3d2.....Positive definite banded

LFDQEH...Computes the determinant of a complex Hermitian positive definite matrix given the  $R^H R$  Cholesky factorization in band Hermitian storage mode.

D4.....Eigenvalues, eigenvectors

ARPACK\_SYMMETRIC Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Ax = Bx$ .

ARPACK\_NONSYMMETRIC Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

ARPACK\_COMPLEX Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

D4a.....Ordinary eigenvalue problems ( $Ax = \lambda x$ )

ARPACK\_SYMMETRIC Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Ax = Bx$ .

ARPACK\_NONSYMMETRIC Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

D4a1 .....Real symmetric

ARPACK\_SYMMETRIC Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Ax = Bx$ .

EVASF...Computes the largest or smallest eigenvalues of a real symmetric matrix.

EVBSF...Computes selected eigenvalues of a real symmetric matrix.

EVCSF...Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.

EVESF...Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.

EVFSF...Computes selected eigenvalues and eigenvectors of a real symmetric matrix.

EVLSF... Computes all of the eigenvalues of a real symmetric matrix.

LIN\_EIG\_SELF      Computes the eigenvalues of a self-adjoint matrix,  $A$ .  
 Optionally, the eigenvectors can be computed. This gives the decomposition  $A = VDVT$ , where  $V$  is an  $n \times n$  orthogonal matrix and  $D$  is a real diagonal matrix.

D4a2..... Real nonsymmetric

ARPACK\_NONSYMMETRIC Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

EVCRG... Computes all of the eigenvalues and eigenvectors of a real matrix.

EVLRG... Computes all of the eigenvalues of a real matrix.

LIN\_EIG\_GEN      Computes the eigenvalues of an  $n \times n$  matrix,  $A$ .  
 Optionally, the eigenvectors of  $A$  or  $A^T$  are computed. Using the eigenvectors of  $A$  gives the decomposition  $AV = VE$ , where  $V$  is an  $n \times n$  complex matrix of eigenvectors, and  $E$  is the complex diagonal matrix of eigenvalues. Other options include the reduction of  $A$  to upper triangular or Schur form, reduction to block upper triangular form with  $2 \times 2$  or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

D4a3..... Complex Hermitian

ARPACK\_COMPLEX Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

EVAHF... Computes the largest or smallest eigenvalues of a complex Hermitian matrix.

EVBHF... Computes the eigenvalues in a given range of a complex Hermitian matrix.

EVCHF... Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.

EVEHF... Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.

EVFHF... Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.

EVLHF... Computes all of the eigenvalues of a complex Hermitian matrix.

LIN\_EIG\_SELF      Computes the eigenvalues of a self-adjoint matrix,  $A$ .  
 Optionally, the eigenvectors can be computed. This gives the decomposition  $A = VDVT$ , where  $V$  is an  $n \times n$  orthogonal matrix and  $D$  is a real diagonal matrix.

D4a4..... Complex non-Hermitian

ARPACK\_COMPLEX Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

EVCCG...Computes all of the eigenvalues and eigenvectors of a complex matrix.

EVLCG...Computes all of the eigenvalues of a complex matrix.

LIN\_EIG\_GEN        Computes the eigenvalues of an  $n \times n$  matrix,  $A$ .  
Optionally, the eigenvectors of  $A$  or  $A^T$  are computed. Using the  
eigenvectors of  $A$  gives the decomposition  
 $AV = VE$ , where  $V$  is an  $n \times n$  complex matrix of eigenvectors, and  $E$  is  
the complex diagonal matrix of eigenvalues. Other options include the  
reduction of  $A$  to upper triangular or Schur form, reduction to block  
upper triangular form with  $2 \times 2$  or unit sized diagonal block matrices,  
and reduction to upper Hessenberg form.

D4a6.....Banded

EVASB...Computes the largest or smallest eigenvalues of a real symmetric matrix  
in band symmetric storage mode.

EVBSB...Computes the eigenvalues in a given interval of a real symmetric matrix  
stored in band symmetric storage mode.

EVCSB...Computes all of the eigenvalues and eigenvectors of a real symmetric  
matrix in band symmetric storage mode.

EVEBSB...Computes the largest or smallest eigenvalues and the corresponding  
eigenvectors of a real symmetric matrix in band symmetric storage  
mode.

EVFSB...Computes the eigenvalues in a given interval and the corresponding  
eigenvectors of a real symmetric matrix stored in band symmetric  
storage mode.

EVLBSB...Computes all of the eigenvalues of a real symmetric matrix in band  
symmetric storage mode.

D4a7.....Sparse matrix eigenvalue problem

ARPACK\_SYMMETRIC    Computes some eigenvalues and eigenvectors of the  
generalized real symmetric eigenvalue problem  $Ax = Bx$ .

ARPACK\_NONSYMMETRIC    Compute some eigenvalues and eigenvectors of the  
generalized eigenvalue problem  $Ax = Bx$ .

ARPACK\_COMPLEX    Compute some eigenvalues and eigenvectors of the  
generalized eigenvalue problem  $Ax = Bx$ .

D4b.....Generalized eigenvalue problems (e.g.,  $Ax = \lambda Bx$ )

ARPACK\_SYMMETRIC    Computes some eigenvalues and eigenvectors of the  
generalized real symmetric eigenvalue problem  $Ax = Bx$ .

ARPACK\_NONSYMMETRIC    Compute some eigenvalues and eigenvectors of the  
generalized eigenvalue problem  $Ax = Bx$ .

ARPACK\_COMPLEX    Compute some eigenvalues and eigenvectors of the  
generalized eigenvalue problem  $Ax = Bx$ .

#### D4b1..... Real symmetric

ARPACK\_SYMMETRIC Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Ax = Bx$ .

GVCSPP... Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem  $Az = \lambda Bz$ , with  $B$  symmetric positive definite.

GVLSPP... Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem  $Az = \lambda Bz$ , with  $B$  symmetric positive definite.

LIN\_GEIG\_GEN Computes the generalized eigenvalues of an  $n \times n$  matrix pencil,  $Av \cong \lambda Bv$ . Optionally, the generalized eigenvectors are computed. If either of  $A$  or  $B$  is nonsingular, there are diagonal matrices  $\alpha$  and  $\beta$  and a complex matrix  $V$  computed such that  $AV\beta = BV\alpha$ .

#### D4b2..... Real general

ARPACK\_NONSYMMETRIC Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

GVCRG... Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem  $Az = \lambda Bz$ .

GVLRG... Computes all of the eigenvalues of a generalized real eigensystem  $Az = \lambda Bz$ .

LIN\_GEIG\_GEN Computes the generalized eigenvalues of an  $n \times n$  matrix pencil,  $Av \cong \lambda Bv$ . Optionally, the generalized eigenvectors are computed. If either of  $A$  or  $B$  is nonsingular, there are diagonal matrices  $\alpha$  and  $\beta$  and a complex matrix  $V$  computed such that  $AV\beta = BV\alpha$ .

#### D4b3..... Complex Hermitian generalized matrix eigenvalue problems

ARPACK\_COMPLEX Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

#### D4b4..... Complex general

ARPACK\_COMPLEX Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem  $Ax = Bx$ .

GVCCG... Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem  $Az = \lambda Bz$ .

GVLCG... Computes all of the eigenvalues of a generalized complex eigensystem  $Az = \lambda Bz$ .

LIN\_GEIG\_GEN Computes the generalized eigenvalues of an  $n \times n$  matrix pencil,  $Av \cong \lambda Bv$ . Optionally, the generalized eigenvectors are computed. If either of  $A$  or  $B$  is nonsingular, there are diagonal matrices  $\alpha$  and  $\beta$  and a complex matrix  $V$  computed such that  $AV\beta = BV\alpha$ .

D4c.....Associated operations

BALANC, CBSLANC Balances a general matrix before computing the eigenvalue-eigenvector decomposition.

EPICG...Computes the performance index for a complex eigensystem.

EPIHF...Computes the performance index for a complex Hermitian eigensystem.

EPIRG...Computes the performance index for a real eigensystem.

EPISB...Computes the performance index for a real symmetric eigensystem in band symmetric storage mode.

EPISF...Computes the performance index for a real symmetric eigensystem.

GPICG...Computes the performance index for a generalized complex eigensystem  $Az = \lambda Bz$ .

GPIRG...Computes the performance index for a generalized real eigensystem  $Az = \lambda Bz$ .

GPISP...Computes the performance index for a generalized real symmetric eigensystem problem.

PERFECT\_SHIFT Computes eigenvectors using actual eigenvalue as an explicit shift. Called by `lin_eig_self`.

PWK.....A rational QR algorithm for computing eigenvalues of real, symmetric tri-diagonal matrices. Called by `lin_svd` and `lin_eig_self`.

D4c2.....Compute eigenvalues of matrix in compact form

D4c2b...Hessenberg

EVCCH . Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.

EVCRH...Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.

EVLCH...Computes all of the eigenvalues of a complex upper Hessenberg matrix.

EVLRH...Computes all of the eigenvalues of a real upper Hessenberg matrix.

D5.....QR decomposition, Gram-Schmidt orthogonalization

LQERR...Accumulates the orthogonal matrix  $Q$  from its factored form given the  $QR$  factorization of a rectangular matrix  $A$ .

LQRRR...Computes the  $QR$  decomposition,  $AP = QR$ , using Householder transformations.

LQRSL...Computes the coordinate transformation, projection, and complete the solution of the least-squares problem  $Ax = b$ .

LSBRR...Solves a linear least-squares problem with iterative refinement.

LSQRR...Solves a linear least-squares problem without iterative refinement.

D6..... Singular value decomposition

ARPACK\_SVD Computes some singular values and left and right singular vectors of a real rectangular  $A_{M \times N} = USV^T$ .

LSVCR... Computes the singular value decomposition of a complex matrix.

LSVRR... Computes the singular value decomposition of a real matrix.

LIN\_SOL\_SVD Solves a rectangular least-squares system of linear equations  $Ax \cong b$  using singular value decomposition,  $A = USV^T$ . Using optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of  $A$ , the orthogonal  $m \times m$  and  $n \times n$  matrices  $U$  and  $V$ , and the  $m \times n$  diagonal matrix of singular values,  $S$ .

LIN\_SVD Computes the singular value decomposition (SVD) of a rectangular matrix,  $A$ . This gives the decomposition  $A = USV^T$ , where  $V$  is an  $n \times n$  orthogonal matrix,  $U$  is an  $m \times m$  orthogonal matrix, and  $S$  is a real, rectangular diagonal matrix.

D7..... Update matrix decompositions

D7b..... Cholesky

LDNCH... Downdates the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.

LUPCH... Updates the  $R^T R$  Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.

D7c..... QR

LUPQR... Computes an updated QR factorization after the rank-one matrix  $\alpha xy^T$  is added.

D9..... Singular, overdetermined or underdetermined systems of linear equations, generalized inverses

D9a..... Unconstrained

D9a1 ..... Least squares ( $L_2$ ) solution

BAND\_ACCUMALATION Accumulatez and solves banded least-squares problem using Householder transformations.

BAND\_SOLVE Accumulatez and solves banded least-squares problem using Householder transformations.

HOUSE\_HOLDER Accumulates and solves banded least-squares problem using Householder transformations.

LQRRR. Computes the QR decomposition,  $AP = QR$ , using Householder transformations.

- LQRRV...Computes the least-squares solution using Householder transformations applied in blocked form.
- LQRSL...Computes the coordinate transformation, projection, and complete the solution of the least-squares problem  $Ax = b$ .
- LSBRR...Solves a linear least-squares problem with iterative refinement.
- LSQRR...Solves a linear least-squares problem without iterative refinement.
- LIN\_SOL\_LSQ Solves a rectangular system of linear equations  $Ax \cong b$ , in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using column and row pivoting, representing the determinant of  $A$ , computing the generalized inverse matrix  $A^\dagger$ , or computing the least-squares solution of  $Ax \cong b$  or  $A^T y \cong d$  given the factorization of  $A$ . An optional argument is provided for computing the following unscaled covariance matrix:  $C = (A^T A)^{-1}$ .
- LIN\_SOL\_SVD Solves a rectangular least-squares system of linear equations  $Ax \cong b$  using singular value decomposition,  $A = USV^T$ . Using optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of  $A$ , the orthogonal  $m \times m$  and  $n \times n$  matrices  $U$  and  $V$ , and the  $m \times n$  diagonal matrix of singular values,  $S$ .
- D9b.....Constrained
- D9b1.....Least squares ( $L_2$ ) solution
- LCLSQ...Solves a linear least-squares problem with linear constraints.
- D9c.....Generalized inverses
- LSGRR...Computes the generalized inverse of a real matrix.
- LIN\_SOL\_LSQ Solves a rectangular system of linear equations  $Ax \cong b$ , in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of  $A$  using column and row pivoting, representing the determinant of  $A$ , computing the generalized inverse matrix  $A^\dagger$ , or computing the least-squares solution of  $Ax \cong b$  or  $A^T y \cong d$  given the factorization of  $A$ . An optional argument is provided for computing the following unscaled covariance matrix:  $C = (A^T A)^{-1}$ .
- E.....INTERPOLATION
- E1 .....Univariate data (curve fitting)
- E1a .....Polynomial splines (piecewise polynomials)
- BSINT...Computes the spline interpolant, returning the B-spline coefficients.
- CSAKM...Computes the Akima cubic spline interpolant.

CSCON... Computes a cubic spline interpolant that is consistent with the concavity of the data.

CSDEC... Computes the cubic spline interpolant with specified derivative endpoint conditions.

CSHER... Computes the Hermite cubic spline interpolant.

CSIEZ... Computes the cubic spline interpolant with the 'not-a-knot' condition and return values of the interpolant at specified points.

CSINT... Computes the cubic spline interpolant with the 'not-a-knot' condition.

CSPER... Computes the cubic spline interpolant with periodic boundary conditions.

QDVAL... Evaluates a function defined on a set of points using quadratic interpolation.

SPLEZ... Computes the values of a spline that either interpolates or fits user-supplied data.

SPLINE\_FITTING Solves constrained least-squares fitting of one-dimensional data by B-splines.

SPLINE\_SUPPORT B-spline function and derivative evaluation package.

E2 ..... Multivariate data (surface fitting)

E2a ..... Gridded

BS2IN... Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

BS3IN... Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.

QD2DR... Evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.

QD2VL... Evaluates a function defined on a rectangular grid using quadratic interpolation.

QD3DR... Evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation.

QD3VL... Evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.

SURFACE\_FITTING Solves constrained least-squares fitting of two-dimensional data by tensor products of B-splines.

SURFND Multidimensional interpolation and differentiation.

E2b ..... Scattered

SURF..... Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

SURFACE\_FAIRING Constrained weighted least-squares fitting of tensor product B-splines to discrete data, with covariance matrix and constraints at points.

E3 .....Service routines for interpolation

E3a .....Evaluation of fitted functions, including quadrature

E3a1 .....Function evaluation

BS1GD...Evaluates the derivative of a spline on a grid, given its B-spline representation.

BS2DR...Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS2GD...Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BS2VL . Evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS3GD...Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BS3VL...Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

BSVAL...Evaluates a spline, given its B-spline representation.

CSVAL.Evaluates a cubic spline.

PPVAL.Evaluates a piecewise polynomial.

QDDER...Evaluates the derivative of a function defined on a set of points using quadratic interpolation.

E3a2 .....Derivative evaluation

BS1GD...Evaluates the derivative of a spline on a grid, given its B-spline representation.

BS2DR...Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS2GD...Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BS3DR...Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.

BS3GD...Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.

BSDER...Evaluates the derivative of a spline, given its B-spline representation.

CS1GD...Evaluates the derivative of a cubic spline on a grid.

CSDER...Evaluates the derivative of a cubic spline.

PP1GD...Evaluates the derivative of a piecewise polynomial on a grid.

PPDER. Evaluates the derivative of a piecewise polynomial.

QDDER...Evaluates the derivative of a function defined on a set of points using quadratic interpolation.

E3a3 ..... Quadrature

BS2IG. Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.

BS3IG...Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.

BSITG...Evaluates the integral of a spline, given its B-spline representation.

CSITG. Evaluates the integral of a cubic spline.

E3b ..... Grid or knot generation

BSNAK...Computes the ‘not-a-knot’ spline knot sequence.

BSOPK...Computes the ‘optimal’ spline knot sequence.

E3c ..... Manipulation of basis functions (e.g., evaluation, change of basis)

BSCPP... Converts a spline in B-spline representation to piecewise polynomial representation.

F ..... SOLUTION OF NONLINEAR EQUATIONS

F1 ..... Single equation

F1a ..... Polynomial

F1a1 ..... Real coefficients

ZPLRC. Finds the zeros of a polynomial with real coefficients using Laguerre’s method.

ZPORC...Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.

F1a2 ..... Complex coefficients

ZPOCC. Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.

F1b ..... Nonpolynomial

ZANLY... Finds the zeros of a univariate complex function using Müller’s method.

ZUNI . . Finds a zero of a real univariate function.

ZBREN. Finds a zero of a real function that changes sign in a given interval.

ZREAL... Finds the real zeros of a real function using Müller’s method.

F2 ..... System of equations

NEQBF . Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.

NEQBJ...Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.

NEQNF . Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

NEQNJ...Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.

G.....OPTIMIZATION (search also classes K, L8)

G1.....Unconstrained

G1a.....Univariate

G1a1.....Smooth function

G1a1a..User provides no derivatives

UVMIF...Finds the minimum point of a smooth function of a single variable using only function evaluations.

G1a1b...User provides first derivatives

UVMID...Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.

G1a2.....General function (no smoothness assumed)

UVMGS...Finds the minimum point of a nonsmooth function of a single variable.

G1b.....Multivariate

G1b1.....Smooth function

G1b1a..User provides no derivatives

UMCGF...Minimizes a function of  $N$  variables using a conjugate gradient algorithm and a finite-difference gradient.

UMINF...Minimizes a function of  $N$  variables using a quasi-New method and a finite-difference gradient.

UNLSF...Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

G1b1b...User provides first derivatives

UMCGG...Minimizes a function of  $N$  variables using a conjugate gradient algorithm and a user-supplied gradient.

UMIDH...Minimizes a function of  $N$  variables using a modified Newton method and a finite-difference Hessian.

UMING...Minimizes a function of  $N$  variables using a quasi-New method and a user-supplied gradient.

UNLSJ... Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

G1b1c... User provides first and second derivatives

UMIAH... Minimizes a function of  $N$  variables using a modified Newton method and a user-supplied Hessian.

G1b2..... General function (no smoothness assumed)

UMPOL... Minimizes a function of  $N$  variables using a direct search polytope algorithm.

G2..... Constrained

G2a..... Linear programming

G2a1..... Dense matrix of constraints

DLPRS... Solves a linear programming problem via the revised simplex algorithm.

G2a2..... Sparse matrix of constraints

SLPRS... Solves a sparse linear programming problem via the revised simplex algorithm.

G2b..... Transportation and assignment problem

TRAN..... Solves a transportation problem.

G2e..... Quadratic programming

G2e1..... Positive definite Hessian (i.e., convex problem)

QPROG... Solves a quadratic programming problem subject to linear equality/inequality constraints.

G2h..... General nonlinear programming

G2h1..... Simple bounds

G2h1a... Smooth function

G2h1a1. User provides no derivatives

BCLSF... Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

BCONF... Minimizes a function of  $N$  variables subject to bounds the variables using a quasi-Newton method and a finite-difference gradient.

G2h1a2. User provides first derivatives

BCLSJ... Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

BCODH. Minimizes a function of  $N$  variables subject to bounds the variables using a modified Newton method and a finite-difference Hessian.

BCONG . Minimizes a function of  $N$  variables subject to bounds the variables using a quasi-Newton method and a user-supplied gradient.

G2h1a3. User provides first and second derivatives

BCOAH...Minimizes a function of  $N$  variables subject to bounds the variables using a modified Newton method and a user-supplied Hessian.

G2h1b...General function (no smoothness assumed)

BCPOL...Minimizes a function of  $N$  variables subject to bounds the variables using a direct search complex algorithm.

G2h2.....Linear equality or inequality constraints

G2h2a...Smooth function

G2h2a1 . User provides no derivatives

LCONF...Minimizes a general objective function subject to linear equality/inequality constraints.

G2h2a2. User provides first derivatives

LCONG...Minimizes a general objective function subject to linear equality/inequality constraints.

G2h3      Nonlinear constraints

G2h3b      Equality and inequality constraints

NNLPG...      Uses a sequential equality constrained QP method.

NNLPF...Uses a sequential equality constrained QP method.

G2h3b1      Smooth function and constraints

G2h3b1a      User provides no derivatives

G2h3b1b      User provides first derivatives of function and constraints

G4      Service routines

G4c      Check user-supplied derivatives

CHGRD...Checks a user-supplied gradient of a function.

CHHES...Checks a user-supplied Hessian of an analytic function.

CHJAC...Checks a user-supplied Jacobian of a system of equations with  $M$  functions in  $N$  unknowns.

G4d      Find feasible point

GGUES . Generates points in an  $N$ -dimensional space.

G4f      Other

CDGRD...Approximates the gradient using central differences.

DDJAC...Approximates the Jacobian of  $m$  functions in  $n$  unknowns using divided differences.

FDGRD... Approximates the gradient using forward differences.

FDHES... Approximates the Hessian using forward differences and function values.

FDJAC... Approximates the Jacobian of  $M$  functions in  $N$  unknowns using forward differences.

GDHES... Approximates the Hessian using forward differences and a user-supplied gradient.

**H      DIFFERENTIATION, INTEGRATION**

**H1      Numerical differentiation**

DERIV... Computes the first, second or third derivative of a user-supplied function.

**H2      Quadrature (numerical evaluation of definite integrals)**

**H2a      One-dimensional integrals**

**H2a1      Finite interval (general integrand)**

**H2a1a      Integrand available via user-defined procedure**

**H2a1a1      Automatic (user need only specify required accuracy)**

QDAG..... Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.

QDAG1D. Integrates a function with a possible internal or endpoint singularity.

QDAGS. Integrates a function (which may have endpoint singularities).

QDNG..... Integrates a smooth function using a nonadaptive rule.

**H2a2      Finite interval (specific or special type integrand including weight functions, oscillating and singular integrands, principal value integrals, splines, etc.).**

**H2a2a      Integrand available via user-defined procedure**

**H2a2a1      Automatic (user need only specify required accuracy)**

QDAGP... Integrates a function with singularity points given.

QDAWC... Integrates a function  $F(X)/(X - C)$  in the Cauchy principal value sense.

QDAWO... Integrates a function containing a sine or a cosine.

QDAWS... Integrates a function with algebraic-logarithmic singularities.

**H2a2b      Integrand available only on grid.**

**H2a2b1      Automatic (user need only specify required accuracy)**

BSITG... Evaluates the integral of a spline, given its  $B$ -spline representation.

**H2a3      Semi-infinite interval (including  $e^{-x}$  weight function)**

**H2a3a      Integrand available via user-defined procedure**

H2a3a1 Automatic (user need only specify required accuracy)

QDAGI...Integrates a function over an infinite or semi-infinite interval.

QDAWF...Computes a Fourier integral.

H2b.....Multidimensional integrals

H2b1.....One or more hyper-rectangular regions (including iterated integrals)

QMC.....Integrates a function over a hyperrectangle using a quasi-Monte Carlo method.

H2b1a...Integrand available via user-defined procedure

H2b1a1 .Automatic (user need only specify required accuracy)

QAND.....Integrates a function on a hyper-rectangle.

QDAG2D.Integrates a function of two variables with a possible internal or end point singularity.

QDAG3D.Integrates a function of three variables with a possible internal or endpoint singularity.

TWODQ...Computes a two-dimensional iterated integral.

H2b1b...Integrand available only on grid

H2b1b2.Nonautomatic

BS2IG...Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.

BS3IG...Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.

H2c.....Service routines (compute weight and nodes for quadrature formulas)

FQRUL...Computes a Fejér quadrature rule with various classical weight functions.

GQRCF...Computes a Gauss, Gauss-Radau or Gauss-Lobatto quadrature rule given the recurrence coefficients for the monic polynomials orthogonal with respect to the weight function.

GQRUL . Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

RECCF...Computes recurrence coefficients for various monic polynomials.

RECQR . Computes recurrence coefficients for monic polynomials given a quadrature rule.

I DIFFERENTIAL AND INTEGRAL EQUATIONS

I1 Ordinary differential equations (ODE's)

I1a Initial value problems

- I1a1 General, nonstiff or mildly stiff
  - I1a1a One-step methods (e.g., Runge-Kutta)
    - FEYNMAN\_KAC Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided.
    - IVMRK...Solves an initial-value problem  $y' = f(t, y)$  for ordinary differential equations using Runge-Kutta pairs of various orders.
    - IVPRK. Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.
  - I1a1b Multistep methods (e.g., Adams predictor-corrector)
    - IVOAM. Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method.
    - IVPAG. Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.
- I1a2 Stiff and mixed algebraic-differential equations
  - DASPG...Solves a first order differential-algebraic system of equations,  $g(t, y, y') = 0$ , using Petzold–Gear BDF method.
  - DAESL. Solves a first order differential-algebraic system of equations,  $g(t, y, y') = 0$ , possibly with additional constraints.
- I1b Multipoint boundary value problems
  - I1b2 Nonlinear
    - BVPFD...Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite-difference method with deferred corrections.
    - BVPMS. Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.
  - I1b3 Eigenvalue (e.g., Sturm-Liouville)
    - SLCNT...Calculates the indices of eigenvalues of a Sturm-Liouville problem with boundary conditions (at regular points) in a specified subinterval of the real line,  $[\alpha, \beta]$ .
    - SLEIG...Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form with boundary conditions (at regular points).
- I2 Partial differential equations
  - FEYNMAN\_KAC Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided.
  - I2a. Initial boundary value problems

FEYNMAN\_KAC Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided.

I2a1 Parabolic

FEYNMAN\_KAC Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided.

PDE\_1D\_MG Integrates an initial-value PDE problem with one space variable.

I2a1 One spatial dimension

MMOLCH Solves a system of partial differential equations of the form  $u_t = f(x, t, u, u_x, u_{xx})$  using the method of lines. The solution is represented with cubic Hermite polynomials.

I2b Elliptic boundary value problems

I2b1 Linear

I2b1a. Second order

I2b1a1 Poisson (Laplace) or Helmholtz equation

I2b1a1a Rectangular domain (or topologically rectangular in the coordinate system)

FPS2H Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uni mesh.

FPS3H Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

J INTEGRAL TRANSFORMS

J1 Trigonometric transforms including fast Fourier transforms

J1a One-dimensional

J1a1 Real

FFTRB Computes the real periodic sequence from its Fourier coefficients.

FFTRF Computes the Fourier coefficients of a real periodic sequence.

FFTRI Computes parameters needed by FFTRF and FFTRB.

J1a2 Complex

FAST-DFT Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array,  $x$ .

FFTCB...Computes the complex periodic sequence from its Fourier coefficients.

FFTCF...Computes the Fourier coefficients of a complex periodic sequence.

FFTCI...Computes parameters needed by FFTCF and FFTCB.

### J1a3 Sine and cosine transforms

FCOSI .. Computes parameters needed by FCOST.

FCOST... Computes the discrete Fourier cosine transformation of an even sequence.

FSINI ... Computes parameters needed by FSINT.

FSINT... Computes the discrete Fourier sine transformation of an odd sequence.

QCOSB... Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.

QCOSF... Computes the coefficients of the cosine Fourier transform with only odd wave numbers.

QCOSI . Computes parameters needed by QCOSF and QCOSB.

QSINB. Computes a sequence from its sine Fourier coefficients with only odd wave numbers.

QSINF... Computes the coefficients of the sine Fourier transform with only odd wave numbers.

QSINI... Computes parameters needed by QSINF and QSINB.

### J1b Multidimensional

FFT2B... Computes the inverse Fourier transform of a complex periodic two-dimensional array.

FFT2D... Computes Fourier coefficients of a complex periodic two-dimensional array.

FFT3B... Computes the inverse Fourier transform of a complex periodic three-dimensional array.

FFT3F... Computes Fourier coefficients of a complex periodic three-dimensional array.

FAST\_2DFT Computes the Discrete Fourier Transform (DFT) of a rank-2 complex array,  $x$ .

FAST\_3DFT Computes the Discrete Fourier Transform (DFT) of a rank-3 complex array,  $x$ .

### J2 Convolutions

CCONV... Computes the convolution of two complex vectors.

RCONV. Computes the convolution of two real vectors.

### J3 Laplace transforms

INLAP... Computes the inverse Laplace transform of a complex function.

SINLP... Computes the inverse Laplace transform of a complex function.

### K APPROXIMATION (search also class L8)

K1 Least squares ( $L_2$ ) approximation

K1a Linear least squares (search also classes D5, D6, D9)

K1a1 Unconstrained

K1a1a. Univariate data (curve fitting)

K1a1a1 Polynomial splines (piecewise polynomials)

BSLSQ...Computes the least-squares spline approximation, and return the B-spline coefficients.

BSVLS...Computes the variable knot B-spline least squares approximation to given data.

CONFIT...Computes the least-squares constrained spline approximation, returning the B-spline coefficients.

FRENCH\_CURVE Constrained weighted least-squares fitting of B-splines to discrete data, with covariance matrix and constraints at points.

K1a1a2 Polynomials

RCURV...Fits a polynomial curve using least squares.

K1a1a3 Other functions (e.g., trigonometric, user-specified)

FNLSQ...Compute a least-squares approximation with user-supplied basis functions.

K1a1b Multivariate data (surface fitting)

BSLS2...Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

BSLS3 . Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.

SURFACE\_FAIRING Constrained weighted least-squares fitting of tensor product B-splines to discrete data, with covariance matrix and constraints at points.

K1a2 Constrained

LIN\_SOL\_LSQ\_CON Routine for constrained linear-least squares based on a least-distance, dual algorithm.

LIN\_SOL\_LSQ\_INQ Routine for constrained linear-least squares based on a least-distance, dual algorithm.

LEAST\_PROJ\_DISTANCE Routine for constrained linear-least squares based on a least-distance, dual algorithm.

PARALLEL\_ & NONNEGATIVE\_LSQ Solves multiple systems of linear equations  $A_j x_j = y_j, j = 1, \dots, k$ . Each matrix  $A_j$  is tridiagonal with the same dimension,  $n$ : The default solution method is based on  $LU$  factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.

PARALLEL\_ & BOUNDED\_LSQ Parallel routines for simple bounded constrained linear-least squares based on a descent algorithm.

#### K1a2a ... Linear constraints

LCLSQ... Solves a linear least-squares problem with linear constraints.

PARALLEL\_NONNEGATIVE\_LSQ Solves a large least-squares system with non-negative constraints, using parallel computing.

PARALLEL\_BOUNDED\_LSQ Solves a large least-squares system with simple bounds, using parallel computing.

#### K1b Nonlinear least squares

##### K1b1 Unconstrained

##### K1b1a Smooth functions

##### K1b1a1 User provides no derivatives

UNLSF... Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

##### K1b1a2 User provides first derivatives

UNLSJ. Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

##### K1b2 Constrained

##### K1b2a Linear constraints

BCLSF... Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.

BCLSJ... Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

BCNLS... Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.

#### K2 Minimax ( $L_\infty$ ) approximation

RATCH. Computes a rational weighted Chebyshev approximation to a continuous function on an interval.

#### K5 Smoothing

CSSCV... Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.

CSSD... Smooths one-dimensional data by error detection.

CSSMH... Computes a smooth cubic spline approximation to noisy data.

#### K6 Service routines for approximation

K6a. Evaluation of fitted functions, including quadrature

K6a1 Function evaluation

BSVAL Evaluates a spline, given its B-spline representation.

CSVVAL Evaluates a cubic spline.

PPVAL Evaluates a piecewise polynomial.

K6a2 Derivative evaluation

BSDER Evaluates the derivative of a spline, given its B-spline representation.

CS1GD Evaluates the derivative of a cubic spline on a grid.

CSDER Evaluates the derivative of a cubic spline.

PP1GD Evaluates the derivative of a piecewise polynomial on a grid.

PPDER Evaluates the derivative of a piecewise polynomial.

K6a3 Quadrature

CSITG Evaluates the integral of a cubic spline.

PPITG Evaluates the integral of a piecewise polynomial.

K6c Manipulation of basis functions (e.g., evaluation, change of basis)

BSCPP Converts a spline in B-spline representation to piecewise polynomial representation.

L STATISTICS, PROBABILITY

L1 Data summarization

L1c. Multi-dimensional data

L1c1 Raw data

L1c1b Covariance, correlation

CCORL...Computes the correlation of two complex vectors.

RCORL . Computes the correlation of two real vectors.

L3 Elementary statistical graphics (*search also class Q*)

L3e. Multi-dimensional data

L3e3 Scatter diagrams

L3e3a Superimposed  $Y$  vs.  $X$

PLOTP Prints a plot of up to 10 sets of points.

L6 Random number generation

L6a. Univariate

RAND\_GEN Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

L6a21 Uniform (continuous, discrete), uniform order statistics

RNUN Generates pseudorandom numbers from a uniform (0, 1) distribution.

RNUNF Generates a pseudorandom number from a uniform (0, 1) distribution.

L6b Multivariate

L6b21 Linear L-1 (least absolute value) approximation random numbers

FAURE\_INIT Shuffles Faure sequence initialization.

FAURE\_FREE Frees the structure containing information about the Faure sequence.

FAURE\_NEXT Computes a shuffled Faure sequence.

L6c. Service routines (e.g., seed)

RNGET...Retrieves the current value of the seed used in the IMSL random number generators.

RNOPT...Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.

RNSET. Initializes a random seed for use in the IMSL random number generators.

RAND\_GEN Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

L8 Regression (search also classes D5, D6, D9, G, K)

L8a. Simple linear (e.g.,  $y = \beta_0 + \beta_1x + \varepsilon$ ) (*search also class L8h*)

L8a1 Ordinary least squares

FNLSQ. Computes a least-squares approximation with user-supplied basis functions.

L8a1a ...Parameter estimation

L8a1a1. Unweighted data

RLINE. Fits a line to a set of data points using least squares.

L8b. ....Polynomial (e.g.,  $y = \beta_0 + \beta_1x + \beta_2x^2 + \varepsilon$ ) (*search also class L8c*)

L8b1 ..... Ordinary least squares

L8b1b... Parameter estimation

L8b1b2. Using orthogonal polynomials

RCURV Fits a polynomial curve using least squares.

L8c Multiple linear (e.g.,  $y = \beta_0 + \beta_1x_1 + \dots + \beta_kx_k + \varepsilon$ )

L8c1 Ordinary least squares

L8c1b Parameter estimation (*search also class L8c1a*)

L8c1b1 Using raw data

LSBRR Solves a linear least-squares problem with iterative refinement.

LSQRR Solves a linear least-squares problem without iterative refinement.

## N DATA HANDLING

### N1 Input, output

PGOPT Sets or retrieves page width and length for printing.

WRCRL Prints a complex rectangular matrix with a given format and labels.

WRCRN...Prints a complex rectangular matrix with integer row and column labels.

WRIRL...Prints an integer rectangular matrix with a given format and labels.

WRIRN...Prints an integer rectangular matrix with integer row and column labels.

WROPT...Sets or retrieves an option for printing a matrix.

WRRRL...Prints a real rectangular matrix with a given format and labels.

WRRRN...Prints a real rectangular matrix with integer row and column labels.

SCALAPACK\_READ Reads matrix data from a file and place in a two-dimensional block-cyclic form on a process grid.

SCALAPACK\_WRITE Writes matrix data to a file, starting with a two-dimensional block-cyclic form on a process grid.

SHOW . . Prints rank-1 and rank-2 arrays with indexing and text.

### N3 Character manipulation

ACHAR...Returns a character given its ASCII value.

CVTSL...Converts a character string containing an integer number into the corresponding integer form.

IACHAR>Returns the integer ASCII value of a character argument.

ICASE . Returns the ASCII value of a character converted to uppercase.

IICSR...Compares two character strings using the ASCII collating sequence but without regard to case.

IIDEX...Determines the position in a string at which a given character sequence begins without regard to case.

### N4 Storage management (e.g., stacks, heaps, trees)

IWKGIN.Initializes bookkeeping locations describing the character workspace stack.

IWKIN...Initializes bookkeeping locations describing the workspace stack.

ScaLAPACK\_READ Moves data from a file to Block-Cyclic form, for use in ScaLAPACK.

ScaLAPACK\_WRITE Move data from Block-Cyclic form, following use in ScaLAPACK, to a file.

## N5 Searching

### N5b Insertion position

ISRCH. Searches a sorted integer vector for a given integer and return its index.

SRCH.....Searches a sorted vector for a given scalar and return its index.

SSRCH...Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

### N5c On a key

IINDEX...Determines the position in a string at which a given character sequence begins without regard to case.

ISRCH...Searches a sorted integer vector for a given integer and return its index.

SRCH.....Searches a sorted vector for a given scalar and return its index.

SSRCH...Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.

## N6 Sorting

### N6a Internal

#### N6a1 Passive (i.e., construct pointer array, rank)

##### N6a1a Integer

SVIBP...Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

SVIGP...Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

##### N6a1b Real

SVRBP. Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

SVRGP. Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

LIN\_SOL\_TRI Sorts a rank-1 array of real numbers  $x$  so the  $y$  results are algebraically nondecreasing,  $y_1 \leq y_2 \leq \dots y_n$ .

#### N6a2 Active

##### N6a2a Integer

SVIBN...Sorts an integer array by nondecreasing absolute value.

SVIBP...Sorts an integer array by nondecreasing absolute value and return the permutation that rearranges the array.

SVIGN. Sorts an integer array by algebraically increasing value.

SVIGP . Sorts an integer array by algebraically increasing value and return the permutation that rearranges the array.

#### N6a2b Real

SVRBN...Sorts a real array by nondecreasing absolute value.

SVRBP...Sorts a real array by nondecreasing absolute value and return the permutation that rearranges the array.

SVRGN...Sorts a real array by algebraically increasing value.

SVRGP...Sorts a real array by algebraically increasing value and return the permutation that rearranges the array.

#### N8 Permuting

PERMA...Permutes the rows or columns of a matrix.

PERMU...Rearranges the elements of an array as specified by a permutation.

#### Q GRAPHICS (search also classes L3)

PLOTP...Prints a plot of up to 10 sets of points.

#### R SERVICE ROUTINES

IDYWK...Computes the day of the week for a given date.

IUMAG...Sets or retrieves MATH/LIBRARY integer options.

NDAYS...Computes the number of days from January 1, 1900, to the given date.

NDYIN...Gives the date corresponding to the number of days since January 1, 1900.

SUMAG...Sets or retrieves MATH/LIBRARY single-precision options.

TDATE.Get stoday's date.

TIMDY.Gets time of day.

VERML...Obtains IMSL MATH/LIBRARY-related version, system and license numbers.

#### R1 Machine-dependent constants

AMACH...Retrieves single-precision machine constants.

IFNAN...Checks if a value is NaN (not a number).

IMACH Retrieves integer machine constants.

ISNAN . Detects an IEEE NaN (not-a-number).

NAN . . . Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN.

UMACH . Sets or retrieves input or output device unit numbers.

#### R3 Error handling

`BUILD_ERROR_STRUCTURE` Fills in flags, values and update the data structure for error conditions that occur in Library routines. Prepares the structure so that calls to routine `error_post` will display the reason for the error.

**R3b** Set unit number for error messages

`UMACH...` Sets or retrieves input or output device unit numbers.

**R3c** Other utilities

`ERROR_POST` Prints error messages that are generated by IMSL Library routines.

`ERSET` Sets error handler default print and stop actions.

`IERCDC` Retrieves the code for an informational error.

`N1RTY` Retrieves an error type for the most recently called IMSL routine.

**S** SOFTWARE DEVELOPMENT TOOLS

**S3** Dynamic program analysis tools

`CPSEC` .Returns CPU time used in seconds.



# Appendix B: Alphabetical Summary of Routines

---

## Routines

| Function/Page                                    | Purpose Statement                                                                                                                                            |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>A</b>                                         |                                                                                                                                                              |
| <a href="#">ACBCB</a> see page 1667              | Adds two complex band matrices, both in band storage mode.                                                                                                   |
| <a href="#">ACHAR</a> see page 1867              | Returns a character given its ASCII value.                                                                                                                   |
| <a href="#">AMACH</a> see page 1939              | Retrieves single-precision machine constants.                                                                                                                |
| <a href="#">ARBRB</a> see page 1665              | Adds two band matrices, both in band storage mode.                                                                                                           |
| <a href="#">ARPACK_COMPLEX</a> see page 685      | Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem $Ax = Bx$ .                                                                  |
| <a href="#">ARPACK_NONSYMMETRIC</a> see page 676 | Compute some eigenvalues and eigenvectors of the generalized eigenvalue problem $Ax = Bx$ .                                                                  |
| <a href="#">ARPACK_SYMMETRIC</a> see page 654    | Computes some eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Ax = Bx$ .                                                  |
| <a href="#">ARPACK_SVD</a> see page 451          | Computes some singular values and left and right singular vectors of a real rectangular $A_{M \times N} = USV^T$ .                                           |
| <b>B</b>                                         |                                                                                                                                                              |
| <a href="#">BCLSF</a> see page 1452              | Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian. |
| <a href="#">BCLSJ</a> see page 1459              | Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.     |
| <a href="#">BCNLS</a> see page 1466              | Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.                                                  |
| <a href="#">BCOAH</a> see page 1441              | Minimizes a function of N variables subject to bounds the variables using a modified Newton method and a user-                                               |

|                                     |                                                                                                                                                            |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                     | supplied Hessian.                                                                                                                                          |
| <a href="#">BCODH</a> see page 1434 | Minimizes a function of $N$ variables subject to bounds the variables using a modified Newton method and a finite-difference Hessian.                      |
| <a href="#">BCONF</a> see page 1420 | Minimizes a function of $N$ variables subject to bounds the variables using a quasi-Newton method and a finite-difference gradient.                        |
| <a href="#">BCONG</a> see page 1427 | Minimizes a function of $N$ variables subject to bounds the variables using a quasi-Newton method and a user-supplied gradient.                            |
| <a href="#">BCPOL</a> see page 1448 | Minimizes a function of $N$ variables subject to bounds the variables using a direct search complex algorithm.                                             |
| <a href="#">BLINF</a> see page 1653 | Computes the bilinear form $x^T Ay$ .                                                                                                                      |
| <a href="#">BS1GD</a> see page 786  | Evaluates the derivative of a spline on a grid, given its B-spline representation.                                                                         |
| <a href="#">BS2DR</a> see page 794  | Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.                                     |
| <a href="#">BS2GD</a> see page 797  | Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.                           |
| <a href="#">BS2IG</a> see page 801  | Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.                               |
| <a href="#">BS2IN</a> see page 771  | Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.                                          |
| <a href="#">BS2VL</a> see page 792  | Evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.                                                       |
| <a href="#">BS3DR</a> see page 807  | Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.                                   |
| <a href="#">BS3GD</a> see page 811  | Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.                         |
| <a href="#">BS3IG</a> see page 817  | Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation. |
| <a href="#">BS3IN</a> see page 776  | Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.                                        |
| <a href="#">BS3VL</a> see page 805  | Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation                                                      |
| <a href="#">BSCPP</a> see page 821  | Converts a spline in B-spline representation to piecewise polynomial representation.                                                                       |
| <a href="#">BSDER</a> see page 783  | Evaluates the derivative of a spline, given its B-spline representation.                                                                                   |

|                                     |                                                                                                                                                                                                  |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">BSINT</a> see page 761  | Computes the spline interpolant, returning the B-spline coefficients.                                                                                                                            |
| <a href="#">BSITG</a> see page 789  | Evaluates the integral of a spline, given its B-spline representation.                                                                                                                           |
| <a href="#">BSLS2</a> see page 889  | Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.                                                            |
| <a href="#">BSLS3</a> see page 894  | Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.                                                          |
| <a href="#">BSLSQ</a> see page 870  | Computes the least-squares spline approximation, and return the B-spline coefficients.                                                                                                           |
| <a href="#">BSNAK</a> see page 765  | Computes the ‘not-a-knot’ spline knot sequence.                                                                                                                                                  |
| <a href="#">BSOPK</a> see page 768  | Computes the ‘optimal’ spline knot sequence.                                                                                                                                                     |
| <a href="#">BSVAL</a> see page 782  | Evaluates a spline, given its B-spline representation.                                                                                                                                           |
| <a href="#">BSVLS</a> see page 874  | Computes the variable knot B-spline least squares approximation to given data.                                                                                                                   |
| <a href="#">BVPFD</a> see page 1037 | Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite-difference method with deferred corrections. |
| <a href="#">BVPMS</a> see page 1050 | Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.                                                              |

## C

|                                     |                                                                                                                                                                                           |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">CADD</a>                | Adds a scalar to each component of a vector, $x \leftarrow x + a$ , all complex.                                                                                                          |
| <a href="#">CAXPY</a>               | Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$ , all complex.                                                                                                    |
| <a href="#">CCBCB</a> see page 1619 | Copies a complex band matrix stored in complex band storage mode.                                                                                                                         |
| <a href="#">CCBCG</a> see page 1626 | Converts a complex matrix in band storage mode to a complex matrix in full storage mode.                                                                                                  |
| <a href="#">CCGCB</a> see page 1624 | Converts a complex general matrix to a matrix in complex band storage mode.                                                                                                               |
| <a href="#">CCGCG</a> see page 1616 | Copies a complex general matrix.                                                                                                                                                          |
| <a href="#">CCONV</a> see page 1294 | Computes the convolution of two complex vectors.                                                                                                                                          |
| <a href="#">CCOPY</a>               | Copies a vector $x$ to a vector $y$ , both complex.                                                                                                                                       |
| <a href="#">CCORL</a> see page 1304 | Computes the correlation of two complex vectors.                                                                                                                                          |
| <a href="#">CDGRD</a> see page 1536 | Approximates the gradient using central differences.                                                                                                                                      |
| <a href="#">CDOTC</a>               | Computes the complex conjugate dot product, $\bar{x}^T y$ .                                                                                                                               |
| <a href="#">CDOTU</a>               | Computes the complex dot product $x^T y$ .                                                                                                                                                |
| <a href="#">CGBMV</a>               | Computes one of the matrix-vector operations:<br>$y \leftarrow \alpha Ax + \beta y, \quad y \leftarrow \alpha A^T x + \beta y, \quad \text{or } y \leftarrow \alpha \bar{A}^T + \beta y,$ |

|                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                       | where $A$ is a matrix stored in band storage mode.                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <a href="#">CGEMM</a>                                 | <p>Computes one of the matrix-matrix operations:</p> $C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T + \beta C, C \leftarrow \alpha A^T B^T + \beta C, C \leftarrow \alpha \bar{A}^T B + \beta C, C \leftarrow \alpha A^T \bar{B}^T + \beta C,$ <p>or</p> $C \leftarrow \alpha \bar{A}^T B + \beta C, C \leftarrow \alpha A^T \bar{B}^T + \beta C, C \leftarrow \alpha \bar{A}^T B^T + \beta C, \text{ or } C \leftarrow \alpha \bar{A}^T \bar{B}^T + \beta C$ |
| <a href="#">CGEMV</a>                                 | <p>Computes one of the matrix-vector operations:</p> $y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, \text{ or } y \leftarrow \alpha \bar{A}^T x + \beta y$                                                                                                                                                                                                                                                                                                                             |
| <a href="#">CGERC</a>                                 | <p>Computes the rank-one update of a complex general matrix:</p> $A \leftarrow A + \alpha x \bar{y}^T.$                                                                                                                                                                                                                                                                                                                                                                                                         |
| <a href="#">CGERU</a>                                 | <p>Computes the rank-one update of a complex general matrix:</p> $A \leftarrow A + \alpha x y^T.$                                                                                                                                                                                                                                                                                                                                                                                                               |
| <a href="#">CHBCB</a> see page 1637                   | Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.                                                                                                                                                                                                                                                                                                                                                                              |
| <a href="#">CHBMV</a>                                 | <p>Computes the matrix-vector operation</p> $y \leftarrow \alpha Ax + \beta y, \text{ where } A \text{ is an Hermitian band matrix in band Hermitian storage.}$                                                                                                                                                                                                                                                                                                                                                 |
| <a href="#">CHECK_BUFFER_ALLOCATION</a> see page 1611 | Maintains buffer sizes on the NVIDIA device and performs one-time initialization.                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <a href="#">CHEMM</a>                                 | <p>Computes one of the matrix-matrix operations:</p> $C \leftarrow \alpha AB + \beta C \text{ or } C \leftarrow \alpha BA + \beta C,$ <p>where <math>A</math> is an Hermitian matrix and <math>B</math> and <math>C</math> are <math>m</math> by <math>n</math> matrices.</p>                                                                                                                                                                                                                                   |
| <a href="#">CHEMV</a>                                 | <p>Computes the matrix-vector operation</p> $y \leftarrow \alpha Ax + \beta y, \text{ where } A \text{ is an Hermitian matrix.}$                                                                                                                                                                                                                                                                                                                                                                                |
| <a href="#">CHER</a>                                  | <p>Computes the rank-one update of an Hermitian matrix:</p> $A \leftarrow A + \alpha x \bar{x}^T \text{ with } x \text{ complex and } \alpha \text{ real.}$                                                                                                                                                                                                                                                                                                                                                     |
| <a href="#">CHER2</a>                                 | <p>Computes a rank-two update of an Hermitian matrix:</p> $A \leftarrow A + \alpha x \bar{y}^T + \bar{\alpha} y \bar{x}^T.$                                                                                                                                                                                                                                                                                                                                                                                     |
| <a href="#">CHER2K</a>                                | <p>Computes one of the Hermitian rank <math>2k</math> operations:</p> $C \leftarrow \alpha \bar{A} \bar{B}^T + \bar{\alpha} B A^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T B + \bar{\alpha} \bar{B}^T A + \beta C,$ <p>, where <math>C</math> is an <math>n</math> by <math>n</math> Hermitian matrix and <math>A</math> and <math>B</math> are <math>n</math> by <math>k</math> matrices in the first case and <math>k</math> by <math>n</math> matrices in the second case.</p>                    |
| <a href="#">CHERK</a>                                 | <p>Computes one of the Hermitian rank <math>k</math> operations:</p> $C \leftarrow \alpha A \bar{A}^T + \beta C \text{ or } C \leftarrow \alpha \bar{A}^T A + \beta C,$ <p>where <math>C</math> is an <math>n</math> by <math>n</math> Hermitian matrix and <math>A</math> is an <math>n</math> by <math>k</math> matrix in</p>                                                                                                                                                                                 |

|                                     |                                                                                                                                 |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">CHFCG</a> see page 1634 | the first case and a $k$ by $n$ matrix in the second case.                                                                      |
| <a href="#">CHGRD</a> see page 1558 | Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.                                         |
| <a href="#">CHHES</a> see page 1561 | Checks a user-supplied gradient of a function.                                                                                  |
| <a href="#">CHJAC</a> see page 1565 | Checks a user-supplied Hessian of an analytic function.                                                                         |
| <a href="#">CHOL</a> see page 1743  | Checks a user-supplied Hessian of an analytic function.                                                                         |
| <a href="#">COND</a> see page 1746  | Checks a user-supplied Jacobian of a system of equations with $M$ functions in $N$ unknowns.                                    |
| <a href="#">CONFT</a> see page 879  | Computes the condition number of a matrix.                                                                                      |
| <a href="#">CONST</a> see page 1919 | Computes the condition number of a rectangular matrix, $A$ .                                                                    |
| <a href="#">CPSEC</a> see page 1874 | Computes the least-squares constrained spline approximation, returning the B-spline coefficients.                               |
| <a href="#">CRBCB</a> see page 1631 | Returns the value of various mathematical and physical constants.                                                               |
| <a href="#">CRBRB</a> see page 1617 | Returns CPU time used in seconds.                                                                                               |
| <a href="#">CRBRG</a> see page 1622 | Converts a real matrix in band storage mode to a complex matrix in band storage mode.                                           |
| <a href="#">CRGCG</a> see page 1627 | Copies a real band matrix stored in band storage mode.                                                                          |
| <a href="#">CRGRB</a> see page 1621 | Converts a real matrix in band storage mode to a real general matrix.                                                           |
| <a href="#">CRGRG</a> see page 1615 | Copies a real general matrix to a complex general matrix.                                                                       |
| <a href="#">CRRCR</a> see page 1629 | Converts a real general matrix to a matrix in band storage mode.                                                                |
| <a href="#">CS1GD</a> see page 753  | Copies a real general matrix.                                                                                                   |
| <a href="#">CSAKM</a> see page 740  | Copies a real rectangular matrix to a complex rectangular matrix.                                                               |
| <a href="#">CSBRB</a> see page 1635 | Evaluates the derivative of a cubic spline on a grid.                                                                           |
| <a href="#">CSCAL</a>               | Computes the Akima cubic spline interpolant.                                                                                    |
| <a href="#">CSCON</a> see page 742  | Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.    |
| <a href="#">CSDEC</a> see page 732  | Multiplies a vector by a scalar, $y \leftarrow ay$ , both complex.                                                              |
| <a href="#">CSDER</a> see page 750  | Computes a cubic spline interpolant that is consistent with the concavity of the data.                                          |
| <a href="#">CSET</a>                | Computes the cubic spline interpolant with specified derivative endpoint conditions.                                            |
| <a href="#">CSFRG</a> see page 1632 | Evaluates the derivative of a cubic spline.                                                                                     |
| <a href="#">CSHER</a> see page 737  | Sets the components of a vector to a scalar, all complex.                                                                       |
| <a href="#">CSIEZ</a> see page 727  | Extends a real symmetric matrix defined in its upper triangle to its lower triangle.                                            |
| <a href="#">CSINT</a> see page 729  | Computes the cubic spline interpolant with the ‘not-a-knot’ condition and return values of the interpolant at specified points. |
| <a href="#">CSITG</a> see page 756  | Computes the cubic spline interpolant with the ‘not-a-knot’ condition.                                                          |
| <a href="#">CSPER</a> see page 746  | Evaluates the integral of a cubic spline.                                                                                       |
|                                     | Computes the cubic spline interpolant with periodic boundary                                                                    |

|                    |                                                                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CSROT              | conditions.                                                                                                                                                                                                                                                                                                              |
| CSROTM             | Applies a complex Givens plane rotation.                                                                                                                                                                                                                                                                                 |
| CSSCAL             | Applies a complex modified Givens plane rotation.                                                                                                                                                                                                                                                                        |
| CSSCV see page 907 | Multiplies a complex vector by a single-precision scalar, $y \leftarrow ay$ .                                                                                                                                                                                                                                            |
| CSSD see page 900  | Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.                                                                                                                                                                                                   |
| CSSMH see page 904 | Smooths one-dimensional data by error detection.                                                                                                                                                                                                                                                                         |
| CSUB               | Computes a smooth cubic spline approximation to noisy data.                                                                                                                                                                                                                                                              |
| CSVAL see page 749 | Subtracts each component of a vector from a scalar, $x \leftarrow a - x$ , all complex.                                                                                                                                                                                                                                  |
| CSVCAL             | Evaluates a cubic spline.                                                                                                                                                                                                                                                                                                |
| CSWAP              | Multiplies a complex vector by a single-precision scalar and store the result in another complex vector, $y \leftarrow ax$ .                                                                                                                                                                                             |
| CSYMM              | Interchanges vectors $x$ and $y$ , both complex.                                                                                                                                                                                                                                                                         |
| CSYR2K             | Computes one of the matrix-matrix operations:<br>$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ ,<br>where $A$ is a symmetric matrix and $B$ and $C$ are $m$ by $n$ matrices.                                                                                                                  |
| CSYRK              | Computes one of the symmetric rank $2k$ operations:<br>$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$ ,<br>where $C$ is an $n$ by $n$ symmetric matrix and $A$ and $B$ are $n$ by $k$ matrices in the first case and $k$ by $n$ matrices in the second case. |
| CTBMV              | Computes one of the symmetric rank $k$ operations:<br>$C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ ,<br>where $C$ is an $n$ by $n$ symmetric matrix and $A$ is an $n$ by $k$ matrix in the first case and a $k$ by $n$ matrix in the second case.                                       |
| CTBSV              | Computes one of the matrix-vector operations:<br>$x \leftarrow Ax$ , $x \leftarrow A^T x$ , or $x \leftarrow \bar{A}^T x$ ,<br>where $A$ is a triangular matrix in band storage mode.                                                                                                                                    |
| CTRMM              | Solves one of the complex triangular systems:<br>$x \leftarrow A^{-1}x$ , $x \leftarrow (A^{-1})^T x$ , or $x \leftarrow (\bar{A}^T)^{-1} x$ ,<br>where $A$ is a triangular matrix in band storage mode.                                                                                                                 |
| CTRMV              | Computes one of the matrix-matrix operations:<br>$B \leftarrow \alpha AB$ , $B \leftarrow \alpha A^T B$ , $B \leftarrow \alpha BA$ , $B \leftarrow \alpha BA^T$ ,<br>$B \leftarrow \alpha \bar{A}^T B$ , or $B \leftarrow \alpha B \bar{A}^T$<br>where $B$ is an $m$ by $n$ matrix and $A$ is a triangular matrix.       |
| CTRSM              | Computes one of the matrix-vector operations:<br>$x \leftarrow Ax$ , $x \leftarrow A^T x$ , or $x \leftarrow \bar{A}^T x$ ,<br>where $A$ is a triangular matrix.                                                                                                                                                         |
|                    | Solves one of the complex matrix equations:                                                                                                                                                                                                                                                                              |

$$B \leftarrow \alpha A^{-1} B, B \leftarrow \alpha B A^{-1}, B \leftarrow \alpha (A^{-1})^T B, B \leftarrow \alpha B (A^{-1})^T, \\ B \leftarrow \alpha (\bar{A}^T)^{-1} B, \text{ or } B \leftarrow \alpha B (\bar{A}^T)^{-1}$$

where  $A$  is a triangular matrix

Solves one of the complex triangular systems:

$$x \leftarrow A^{-1} x, x \leftarrow (A^{-1})^T x, \text{ or } x \leftarrow (\bar{A}^T)^{-1} x,$$

where  $A$  is a triangular matrix.

[CTRVS](#)

[CUBLAS\\_GET](#) see page 1607

Returns the switchover value for a positional array argument for a specified BLAS routine.

[CUBLAS\\_SET](#) see page 1609

Sets the switchover value for an array used by a specified BLAS routine.

[CUDA\\_ERROR\\_PRINT](#) see page 1612

Prints error messages generated through the use of the CUDABLAS Library using the IMSL error handler.

[CUNIT](#) see page 1921

Converts  $X$  in units  $XUNITS$  to  $Y$  in units  $YUNITS$ .

[CVCAL](#)

Multiplies a vector by a scalar and store the result in another vector,  $y \leftarrow ax$ , all complex.

[CVTSI](#) see page 1873

Converts a character string containing an integer number into the corresponding integer form.

[CZCDOT](#)

Computes the sum of a complex scalar plus a complex conjugate dot product,  $a + \bar{x}^T y$ , using a double-precision accumulator.

[CZDOTA](#)

Computes the sum of a complex scalar, a complex dot product and the double-complex accumulator, which is set to the result  $ACC \leftarrow ACC + a + x^T y$ .

[CZDOTC](#)

Computes the complex conjugate dot product,  $\bar{x}^T y$ , using a double-precision accumulator.

[CZDOTI](#)

Computes the sum of a complex scalar plus a complex dot product using a double-complex accumulator, which is set to the result  $ACC \leftarrow a + x^T y$ .

[CZDOTU](#)

Computes the complex dot product  $x^T y$  using a double-precision accumulator.

[CZUDOT](#)

Computes the sum of a complex scalar plus a complex dot product,  $a + x^T y$ , using a double-precision accumulator.

## D

[DAESL](#) see page 1057

Solves a first order differential-algebraic system of equations,  $g(t, y, y') = 0$ , possibly with additional constraints.

[DDJAC](#) see page 1546

Approximates the Jacobian of  $m$  functions in  $n$  unknowns using divided differences

[DENSE\\_LP](#) see page 1488

Solves a linear programming problem.

|                                                                                      |                                                                                                                                                                           |
|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">DERIV</a> see page 995                                                   | Computes the first, second or third derivative of a user-supplied function.                                                                                               |
| <a href="#">DET</a> see page 1750                                                    | Computes the determinant of a rectangular matrix, $A$ .                                                                                                                   |
| <a href="#">DIAG</a> see page 1753                                                   | Constructs a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array.                                                                 |
| <a href="#">DIAGONALS</a> see page 1754                                              | Extracts a rank-1 array whose values are the diagonal terms of a rank-2 array argument.                                                                                   |
| <a href="#">DISL1</a> see page 1679                                                  | Computes the 1-norm distance between two points.                                                                                                                          |
| <a href="#">DISL2</a> see page 1677                                                  | Computes the Euclidean (2-norm) distance between two points.                                                                                                              |
| <a href="#">DISLI</a> see page 1681                                                  | Computes the infinity norm distance between two points.                                                                                                                   |
| <a href="#">DLPRS</a> see page 1494                                                  | Solves a linear programming problem via the revised simplex algorithm.                                                                                                    |
| <a href="#">DMACH</a> see page 1940                                                  | See AMACH.                                                                                                                                                                |
| <a href="#">DQADD</a> (See <a href="#">Extended Precision Arithmetic Chapter 9</a> ) | Adds a double-precision scalar to the accumulator in extended precision.                                                                                                  |
| <a href="#">DQINI</a> (See <a href="#">Extended Precision Arithmetic Chapter 9</a> ) | Initializes an extended-precision accumulator with a double-precision scalar.                                                                                             |
| <a href="#">DQMUL</a> (See <a href="#">Extended Precision Arithmetic Chapter 9</a> ) | Multiplies double-precision scalars in extended precision.                                                                                                                |
| <a href="#">DQSTO</a> (See <a href="#">Extended Precision Arithmetic Chapter 9</a> ) | Stores a double-precision approximation to an extended-precision scalar.                                                                                                  |
| <a href="#">DSDOT</a> (See <a href="#">Chapter 9</a> )                               | Computes the single-precision dot product $x^T y$ using a double precision accumulator. This routine handles MATH/LIBRARY and STAT/LIBRARY type DOUBLE PRECISION options. |
| <a href="#">DUMAG</a> see page 1914                                                  | This routine handles MATH/LIBRARY and STAT/LIBRARY type DOUBLE PRECISION options.                                                                                         |

## E

|                                          |                                                                                                     |
|------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <a href="#">EIG</a> see page 1755        | Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem. |
| <a href="#">EPICG</a> see page 562       | Computes the performance index for a complex eigensystem.                                           |
| <a href="#">EPIHF</a> see page 615       | Computes the performance index for a complex Hermitian eigensystem.                                 |
| <a href="#">EPIRG</a> see page 555       | Computes the performance index for a real eigensystem.                                              |
| <a href="#">EPISB</a> see page 596       | Computes the performance index for a real symmetric eigensystem in band symmetric storage mode.     |
| <a href="#">EPISF</a> see page 578       | Computes the performance index for a real symmetric eigensystem.                                    |
| <a href="#">ERROR_POST</a> see page 1810 | Prints error messages that are generated by IMSL routines using EPACK.                              |

|                                     |                                                                                                                                                   |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">ERSET</a> see page 1933 | Sets error handler default print and stop actions.                                                                                                |
| <a href="#">EVAHF</a> see page 604  | Computes the largest or smallest eigenvalues of a complex Hermitian matrix.                                                                       |
| <a href="#">EVASB</a> see page 585  | Computes the largest or smallest eigenvalues of a real symmetric matrix in band symmetric storage mode.                                           |
| <a href="#">EVASF</a> see page 568  | Computes the largest or smallest eigenvalues of a real symmetric matrix.                                                                          |
| <a href="#">EVBHF</a> see page 609  | Computes the eigenvalues in a given range of a complex Hermitian matrix.                                                                          |
| <a href="#">EVBSB</a> see page 591  | Computes the eigenvalues in a given interval of a real symmetric matrix stored in band symmetric storage mode.                                    |
| <a href="#">EVBSF</a> see page 573  | Computes selected eigenvalues of a real symmetric matrix.                                                                                         |
| <a href="#">EVCCG</a> see page 559  | Computes all of the eigenvalues and eigenvectors of a complex matrix.                                                                             |
| <a href="#">EVCCH</a> see page 623  | Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.                                                            |
| <a href="#">EVCHF</a> see page 601  | Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.                                                                   |
| <a href="#">EVCRG</a> see page 552  | Computes all of the eigenvalues and eigenvectors of a real matrix.                                                                                |
| <a href="#">EVCRH</a> see page 619  | Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.                                                               |
| <a href="#">EVCSB</a> see page 582  | Computes all of the eigenvalues and eigenvectors of a real symmetric matrix in band symmetric storage mode.                                       |
| <a href="#">EVCSF</a> see page 566  | Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.                                                                      |
| <a href="#">EVEHF</a> see page 606  | Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.                                    |
| <a href="#">EVESB</a> see page 588  | Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix in band symmetric storage mode.        |
| <a href="#">EVESF</a> see page 570  | Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.                                       |
| <a href="#">EVFHF</a> see page 612  | Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.                                       |
| <a href="#">EVFSB</a> see page 593  | Computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix stored in band symmetric storage mode. |
| <a href="#">EVFSF</a> see page 575  | Computes selected eigenvalues and eigenvectors of a real symmetric matrix.                                                                        |
| <a href="#">EVLCG</a> see page 557  | Computes all of the eigenvalues of a complex matrix.                                                                                              |
| <a href="#">EVLCH</a> see page 621  | Computes all of the eigenvalues of a complex upper Hessenberg matrix.                                                                             |
| <a href="#">EVLHF</a> see page 598  | Computes all of the eigenvalues of a complex Hermitian matrix.                                                                                    |
| <a href="#">EVLRG</a> see page 549  | Computes all of the eigenvalues of a real matrix.                                                                                                 |

|                                     |                                                                                            |
|-------------------------------------|--------------------------------------------------------------------------------------------|
| <a href="#">EVLRH</a> see page 617  | Computes all of the eigenvalues of a real upper Hessenberg matrix.                         |
| <a href="#">EVL SB</a> see page 580 | Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode. |
| <a href="#">EVL SF</a> see page 564 | Computes all of the eigenvalues of a real symmetric matrix.                                |
| <a href="#">EYE</a> see page 1759   | Creates a rank-2 square array whose diagonals are all the value one.                       |

## F

|                                           |                                                                                                                                                    |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">FAURE_FREE</a> see page 1905  | Frees the structure containing information about the Faure sequence.                                                                               |
| <a href="#">FAURE_INIT</a> see page 1904  | Shuffled Faure sequence initialization.                                                                                                            |
| <a href="#">FAURE_NEXT</a> see page 1905  | Computes a shuffled Faure sequence.                                                                                                                |
| <a href="#">FAST_DFT</a> see page 1220    | Computes the Discrete Fourier Transform of a rank-1 complex array, $x$ .                                                                           |
| <a href="#">FAST_2DFT</a> see page 1227   | Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, $x$ .                                                                    |
| <a href="#">FAST_3DFT</a> see page 1233   | Computes the Discrete Fourier Transform (2DFT) of a rank-3 complex array, $x$ .                                                                    |
| <a href="#">FCOSI</a> see page 1259       | Computes parameters needed by <code>FCOST</code> .                                                                                                 |
| <a href="#">FCOST</a> see page 1257       | Computes the discrete Fourier cosine transformation of an even sequence.                                                                           |
| <a href="#">FDGRD</a> see page 1538       | Approximates the gradient using forward differences.                                                                                               |
| <a href="#">FDHES</a> see page 1541       | Approximates the Hessian using forward differences and function values.                                                                            |
| <a href="#">FDJAC</a> see page 1555       | Approximates the Jacobian of $M$ functions in $N$ unknowns using forward differences.                                                              |
| <a href="#">FEYNMAN_KAC</a> see page 1128 | Solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are provided. |
| <a href="#">FFT</a> see page 1761         | The Discrete Fourier Transform of a complex sequence and its inverse transform.                                                                    |
| <a href="#">FFT_BOX</a> see page 1763     | The Discrete Fourier Transform of several complex or real sequences.                                                                               |
| <a href="#">FFT2B</a> see page 1277       | Computes the inverse Fourier transform of a complex periodic two-dimensional array.                                                                |
| <a href="#">FFT2D</a> see page 1274       | Computes Fourier coefficients of a complex periodic two-dimensional array.                                                                         |
| <a href="#">FFT3B</a> see page 1285       | Computes the inverse Fourier transform of a complex periodic three-dimensional array.                                                              |
| <a href="#">FFT3F</a> see page 1281       | Computes Fourier coefficients of a complex periodic three-dimensional array.                                                                       |
| <a href="#">FFTCB</a> see page 1248       | Computes the complex periodic sequence from its Fourier coefficients.                                                                              |
| <a href="#">FFTCF</a> see page 1245       | Computes the Fourier coefficients of a complex periodic                                                                                            |

|                                     |                                                                                                                                                                             |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">FFTCI</a> see page 1251 | sequence.                                                                                                                                                                   |
| <a href="#">FFTRB</a> see page 1240 | Computes parameters needed by <code>FFTCF</code> and <code>FFTCB</code> .                                                                                                   |
| <a href="#">FFTRF</a> see page 1236 | Computes the real periodic sequence from its Fourier coefficients.                                                                                                          |
| <a href="#">FFTRI</a> see page 1243 | Computes the Fourier coefficients of a real periodic sequence.                                                                                                              |
| <a href="#">FNLSQ</a> see page 865  | Computes parameters needed by <code>FFTRF</code> and <code>FFTRB</code> .                                                                                                   |
| <a href="#">FPS2H</a> see page 1188 | Computes a least-squares approximation with user-supplied basis functions.                                                                                                  |
| <a href="#">FPS3H</a> see page 1194 | Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the <code>HODIE</code> finite-difference scheme on a uni mesh. |
| <a href="#">FQRUL</a> see page 991  | Solves Poisson's or Helmholtz's equation on a three-dimensional box using a fast Poisson solver based on the <code>HODIE</code> finite-difference scheme on a uniform mesh. |
| <a href="#">FSINI</a> see page 1255 | Computes a Fejér quadrature rule with various classical weight functions.                                                                                                   |
| <a href="#">FSINT</a> see page 1253 | Computes parameters needed by <code>FSINT</code> .                                                                                                                          |
|                                     | Computes the discrete Fourier sine transformation of an odd sequence.                                                                                                       |

## G

|                                     |                                                                                                                                                                            |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">GDHES</a> see page 1543 | Approximates the Hessian using forward differences and a user-supplied gradient.                                                                                           |
| <a href="#">GGUES</a> see page 1569 | Generates points in an N-dimensional space.                                                                                                                                |
| <a href="#">GMRES</a> see page 440  | Uses restarted <code>GMRES</code> with reverse communication to generate an approximate solution of $Ax = b$ .                                                             |
| <a href="#">GPICG</a> see page 640  | Computes the performance index for a generalized complex eigensystem $Az = \lambda Bz$ .                                                                                   |
| <a href="#">GPIRG</a> see page 632  | Computes the performance index for a generalized real eigensystem $Az = \lambda Bz$ .                                                                                      |
| <a href="#">GPISP</a> see page 648  | Computes the performance index for a generalized real symmetric eigensystem problem.                                                                                       |
| <a href="#">GQRCF</a> see page 983  | Computes a Gauss, Gauss-Radau or Gauss-Lobatto quadrature rule given the recurrence coefficients for the monic polynomials orthogonal with respect to the weight function. |
| <a href="#">GQRUL</a> see page 979  | Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.                                                                   |
| <a href="#">GVCCG</a> see page 637  | Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem $Az = \lambda Bz$ .                                                                  |
| <a href="#">GVCRG</a> see page 629  | Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem $Az = \lambda Bz$ .                                                                     |
| <a href="#">GVCSP</a> see page 645  | Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$ .                                                  |

|                                               |                                                                                                                                                                          |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">GVLGG</a> see page 634            | with $B$ symmetric positive definite.                                                                                                                                    |
| <a href="#">GVLRG</a> see page 626            | Computes all of the eigenvalues of a generalized complex eigensystem $Az = \lambda Bz$ .                                                                                 |
| <a href="#">GVLSP</a> see page 642            | Computes all of the eigenvalues of a generalized real eigensystem $Az = \lambda Bz$ .                                                                                    |
|                                               | Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$ , with $B$ symmetric positive definite.                           |
| <b>H</b>                                      |                                                                                                                                                                          |
| <a href="#">HRRRR</a> see page 1651           | Computes the Hadamard product of two real rectangular matrices.                                                                                                          |
| <a href="#">HYPOT</a> see page 1925           | Computes $\sqrt{a^2 + b^2}$ without underflow or overflow.                                                                                                               |
| <b>I</b>                                      |                                                                                                                                                                          |
| <a href="#">IACHAR</a> see page 1868          | Returns the integer ASCII value of a character argument.                                                                                                                 |
| <a href="#">IADD</a>                          | Adds a scalar to each component of a vector, $x \leftarrow x + a$ , all integer. Finds the smallest index of the component of a complex vector having maximum magnitude. |
| <a href="#">ICAMAX</a>                        | Finds the smallest index of the component of a complex vector having minimum magnitude.                                                                                  |
| <a href="#">ICAMIN</a>                        | Returns the ASCII value of a character converted to uppercase.                                                                                                           |
| <a href="#">ICASE</a> see page 1869           | Copies a vector $x$ to a vector $y$ , both integer.                                                                                                                      |
| <a href="#">ICOPY</a>                         | Computes the day of the week for a given date.                                                                                                                           |
| <a href="#">IDYWK</a> see page 1880           | Retrieves the code for an informational error.                                                                                                                           |
| <a href="#">IERCD and NIRTY</a> see page 1934 | The inverse of the Discrete Fourier Transform of a complex sequence.                                                                                                     |
| <a href="#">IFFT</a> see page 1765            | The inverse of the Discrete Fourier Transform of a complex sequence.                                                                                                     |
| <a href="#">IFFT_BOX</a> see page 1767        | The inverse Discrete Fourier Transform of several complex or real sequences.                                                                                             |
| <a href="#">IFNAN(X)</a> see page 1940        | Checks if a value is NaN (not a number).                                                                                                                                 |
| <a href="#">IICSR</a> see page 1870           | Compares two character strings using the ASCII collating sequence but without regard to case.                                                                            |
| <a href="#">IIDEX</a> see page 1872           | Determines the position in a string at which a given character sequence begins without regard to case.                                                                   |
| <a href="#">IIMAX</a>                         | Finds the smallest index of the maximum component of a integer vector.                                                                                                   |
| <a href="#">IIMIN</a>                         | Finds the smallest index of the minimum of an integer vector.                                                                                                            |
| <a href="#">IMACH</a> see page 1937           | Retrieves integer machine constants.                                                                                                                                     |
| <a href="#">INLAP</a> see page 1309           | Computes the inverse Laplace transform of a complex                                                                                                                      |

|                                     |                                                                                                                                                                          |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                     | function.                                                                                                                                                                |
| <a href="#">ISAMAX</a>              | Finds the smallest index of the component of a single-precision vector having maximum absolute value.                                                                    |
| <a href="#">ISAMIN</a>              | Finds the smallest index of the component of a single-precision vector having minimum absolute value.                                                                    |
| <a href="#">ISET</a>                | Sets the components of a vector to a scalar, all integer.                                                                                                                |
| <a href="#">ISMAY</a>               | Finds the smallest index of the component of a single-precision vector having maximum value.                                                                             |
| <a href="#">ISMIN</a>               | Finds the smallest index of the component of a single-precision vector having minimum value.                                                                             |
| <a href="#">ISNAN</a> see page 1769 | This is a generic logical function used to test scalars or arrays for occurrence of an IEEE 754 Standard format of floating point (ANSI/IEEE 1985) NaN, or not-a-number. |
| <a href="#">ISRCH</a> see page 1862 | Searches a sorted integer vector for a given integer and return its index.                                                                                               |
| <a href="#">ISUB</a>                | Subtracts each component of a vector from a scalar, $x \leftarrow a - x$ , all integer.                                                                                  |
| <a href="#">ISUM</a>                | Sums the values of an integer vector.                                                                                                                                    |
| <a href="#">ISWAP</a>               | Interchanges vectors $x$ and $y$ , both integer.                                                                                                                         |
| <a href="#">IUMAG</a> see page 1908 | Sets or retrieves MATH/LIBRARY integer options.                                                                                                                          |
| <a href="#">IVMRK</a> see page 1011 | Solves an initial-value problem $y' = f(t, y)$ for ordinary differential equations using Runge-Kutta pairs of various orders.                                            |
| <a href="#">IVOAM</a> see page 1072 | Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method.                                 |
| <a href="#">IVPAG</a> see page 1021 | Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.                                                   |
| <a href="#">IVPRK</a> see page 1003 | Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.                                     |
| <b>J</b>                            |                                                                                                                                                                          |
| <a href="#">JCGRC</a> see page 437  | Solves a real symmetric definite linear system using the Jacobi preconditioned conjugate gradient method with reverse communication.                                     |
| <b>L</b>                            |                                                                                                                                                                          |
| <a href="#">LCHRG</a> see page 494  | Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.                                                           |
| <a href="#">LCLSQ</a> see page 467  | Solves a linear least-squares problem with linear constraints.                                                                                                           |
| <a href="#">LCONF</a> see page 1510 | Minimizes a general objective function subject to linear equality/inequality constraints.                                                                                |

|                                     |                                                                                                                                                                  |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">LCONG</a> see page 1516 | Minimizes a general objective function subject to linear equality/inequality constraints.                                                                        |
| <a href="#">LDNCH</a> see page 499  | Downdates the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed                                     |
| <a href="#">LFCCB</a> see page 333  | Computes the $LU$ factorization of a complex matrix in band storage mode and estimate its $L_1$ condition number.                                                |
| <a href="#">LFCCG</a> see page 128  | Computes the $LU$ factorization of a complex general matrix and estimate its $L_1$ condition number.                                                             |
| <a href="#">LFCCT</a> see page 169  | Estimates the condition number of a complex triangular matrix.                                                                                                   |
| <a href="#">LFCDH</a> see page 237  | Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix and estimate its $L_1$ condition number.                                      |
| <a href="#">LFCDS</a> see page 186  | Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix and estimate its $L_1$ condition number.                                |
| <a href="#">LFCFH</a> see page 265  | Computes the $U D U^H$ factorization of a complex Hermitian matrix and estimate its $L_1$ condition number.                                                      |
| <a href="#">LFCQH</a> see page 355  | Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode and estimate its $L_1$ condition number.       |
| <a href="#">LFCQS</a> see page 311  | Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode and estimate its $L_1$ condition number. |
| <a href="#">LFCRB</a> see page 290  | Computes the $LU$ factorization of a real matrix in band storage mode and estimate its $L_1$ condition number.                                                   |
| <a href="#">LFCRG</a> see page 93   | Computes the $LU$ factorization of a real general matrix and estimate its $L_1$ condition number.                                                                |
| <a href="#">LFCRT</a> see page 158  | Estimates the condition number of a real triangular matrix.                                                                                                      |
| <a href="#">LFCSF</a> see page 215  | Computes the $U D U^T$ factorization of a real symmetric matrix and estimate its $L_1$ condition number.                                                         |
| <a href="#">LFCDB</a> see page 344  | Computes the determinant of a complex matrix given the $LU$ factorization of the matrix in band storage mode.                                                    |
| <a href="#">LFCDG</a> see page 148  | Computes the determinant of a complex general matrix given the $LU$ factorization of the matrix.                                                                 |
| <a href="#">LFDCT</a> see page 173  | Computes the determinant of a complex triangular matrix.                                                                                                         |
| <a href="#">LFDDB</a> see page 258  | Computes the determinant of a complex Hermitian positive definite matrix given the $R^H R$ Cholesky factorization of the matrix.                                 |
| <a href="#">LFDDB</a> see page 204  | Computes the determinant of a real symmetric positive definite matrix given the $R^H R$ Cholesky factorization of the matrix.                                    |
| <a href="#">LFDHF</a> see page 276  | Computes the determinant of a complex Hermitian matrix                                                                                                           |

|                                     |                                                                                                                                                       |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                     | given the $UDU^H$ factorization of the matrix.                                                                                                        |
| <a href="#">LFDQH</a> see page 365  | Computes the determinant of a complex Hermitian positive definite matrix given the $R^H R$ Cholesky factorization in band Hermitian storage mode.     |
| <a href="#">LFDQS</a> see page 320  | Computes the determinant of a real symmetric positive definite matrix given the $R^T R$ Cholesky factorization of the band symmetric storage mode.    |
| <a href="#">LFDREB</a> see page 301 | Computes the determinant of a real matrix in band storage mode given the $LU$ factorization of the matrix.                                            |
| <a href="#">LFDREB</a> see page 113 | Computes the determinant of a real general matrix given the $LU$ factorization of the matrix.                                                         |
| <a href="#">LFDREB</a> see page 162 | Computes the determinant of a real triangular matrix.                                                                                                 |
| <a href="#">LFDREB</a> see page 226 | Computes the determinant of a real symmetric matrix given the $UDU^T$ factorization of the matrix.                                                    |
| <a href="#">LFDREB</a> see page 341 | Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.                                       |
| <a href="#">LFDREB</a> see page 143 | Uses iterative refinement to improve the solution of a complex general system of linear equations.                                                    |
| <a href="#">LFDREB</a> see page 252 | Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.                                |
| <a href="#">LFDREB</a> see page 199 | Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.                                   |
| <a href="#">LFDREB</a> see page 273 | Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.                                                  |
| <a href="#">LFDREB</a> see page 362 | Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode. |
| <a href="#">LFDREB</a> see page 318 | Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.    |
| <a href="#">LFDREB</a> see page 298 | Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.                                          |
| <a href="#">LFDREB</a> see page 108 | Uses iterative refinement to improve the solution of a real general system of linear equations.                                                       |
| <a href="#">LFDREB</a> see page 223 | Uses iterative refinement to improve the solution of a real symmetric system of linear equations.                                                     |
| <a href="#">LFDREB</a> see page 339 | Solves a complex system of linear equations given the $LU$ factorization of the coefficient matrix in band storage mode.                              |
| <a href="#">LFDREB</a> see page 138 | Solves a complex general system of linear equations given the $LU$ factorization of the coefficient matrix.                                           |
| <a href="#">LFDREB</a> see page 248 | Solves a complex Hermitian positive definite system of linear equations given the $R^H R$ factorization of the coefficient matrix.                    |
| <a href="#">LFDREB</a> see page 195 | Solves a real symmetric positive definite system of linear equations given the $R^T R$ Choleksy factorization of the                                  |

|                                    |                                                                                                                                                           |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                    | coefficient matrix.                                                                                                                                       |
| <a href="#">LFSHF</a> see page 271 | Solves a complex Hermitian system of linear equations given the $U D U^H$ factorization of the coefficient matrix.                                        |
| <a href="#">LFSQH</a> see page 360 | Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode. |
| <a href="#">LFSQS</a> see page 316 | Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.    |
| <a href="#">LFSRB</a> see page 296 | Solves a real system of linear equations given the $LU$ factorization of the coefficient matrix in band storage mode.                                     |
| <a href="#">LFSRG</a> see page 103 | Solves a real general system of linear equations given the $LU$ factorization of the coefficient matrix.                                                  |
| <a href="#">LFSSF</a> see page 221 | Solves a real symmetric system of linear equations given the $U D U^T$ factorization of the coefficient matrix.                                           |
| <a href="#">LFSXD</a> see page 408 | Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.                  |
| <a href="#">LFSXG</a> see page 377 | Solves a sparse system of linear equations given the $LU$ factorization of the coefficient matrix.                                                        |
| <a href="#">LFSZD</a> see page 421 | Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.               |
| <a href="#">LFSZG</a> see page 391 | Solves a complex sparse system of linear equations given the $LU$ factorization of the coefficient matrix.                                                |
| <a href="#">LFTCB</a> see page 336 | Computes the $LU$ factorization of a complex matrix in band storage mode.                                                                                 |
| <a href="#">LFTCG</a> see page 134 | Computes the $LU$ factorization of a complex general matrix.                                                                                              |
| <a href="#">LFTDH</a> see page 243 | Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix.                                                                       |
| <a href="#">LFTDS</a> see page 191 | Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix.                                                                 |
| <a href="#">LFTHF</a> see page 268 | Computes the $U D U^H$ factorization of a complex Hermitian matrix.                                                                                       |
| <a href="#">LFTQH</a> see page 358 | Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.                                        |
| <a href="#">LFTQS</a> see page 314 | Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.                                  |
| <a href="#">LFTRB</a> see page 293 | Computes the $LU$ factorization of a real matrix in band storage mode.                                                                                    |
| <a href="#">LFTRG</a> see page 99  | Computes the $LU$ factorization of a real general matrix.                                                                                                 |
| <a href="#">LFTSF</a> see page 218 | Computes the $U D U^T$ factorization of a real symmetric matrix.                                                                                          |
| <a href="#">LFTXG</a> see page 372 | Computes the $LU$ factorization of a real general sparse                                                                                                  |

|              |              |                                                                                                                         |
|--------------|--------------|-------------------------------------------------------------------------------------------------------------------------|
| LFTZG        | see page 385 | matrix.<br>Computes the $LU$ factorization of a complex general sparse matrix.                                          |
| LINCG        | see page 150 | Computes the inverse of a complex general matrix.                                                                       |
| LINCT        | see page 175 | Computes the inverse of a complex triangular matrix.                                                                    |
| LINDS        | see page 206 | Computes the inverse of a real symmetric positive definite matrix.                                                      |
| LINRG        | see page 115 | Computes the inverse of a real general matrix.                                                                          |
| LINRT        | see page 163 | Computes the inverse of a real triangular matrix.                                                                       |
| LIN_EIG_GEN  | see page 533 | Computes the eigenvalues of a self-adjoint matrix, $A$ .                                                                |
| LIN_EIG_SELF | see page 526 | Computes the eigenvalues of a self-adjoint matrix, $A$ .                                                                |
| LIN_GEIG_GEN | see page 542 | Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av = \lambda Bv$ .                              |
| LIN_SOL_GEN  | see page 10  | Solves a general system of linear equations $Ax = b$ .                                                                  |
| LIN_SOL_LSQ  | see page 27  | Solves a rectangular system of linear equations $Ax \cong b$ , in a least-squares sense.                                |
| LIN_SOL_SELF | see page 18  | Solves a system of linear equations $Ax = b$ , where $A$ is a self-adjoint matrix.                                      |
| LIN_SOL_SVD  | see page 36  | Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition.          |
| LIN_SOL_TRI  | see page 45  | Solves multiple systems of linear equations.                                                                            |
| LIN_SVD      | see page 57  | Computes the singular value decomposition (SVD) of a rectangular matrix, $A$ .                                          |
| LNFXD        | see page 403 | Computes the numerical Cholesky factorization of a sparse symmetrical matrix $A$ .                                      |
| LNFZD        | see page 416 | Computes the numerical Cholesky factorization of a sparse Hermitian matrix $A$ .                                        |
| LQERR        | see page 478 | Accumulates the orthogonal matrix $Q$ from its factored form given the $QR$ factorization of a rectangular matrix $A$ . |
| LQRRR        | see page 471 | Computes the $QR$ decomposition, $AP = QR$ , using Householder transformations.                                         |
| LQRRV        | see page 457 | Computes the least-squares solution using Householder transformations applied in blocked form.                          |
| LQRSL        | see page 483 | Computes the coordinate transformation, projection, and complete the solution of the least-squares problem $Ax = b$ .   |
| LSACB        | see page 327 | Solves a complex system of linear equations in band storage mode with iterative refinement.                             |
| LSACG        | see page 119 | Solves a complex general system of linear equations with iterative refinement.                                          |
| LSADH        | see page 227 | Solves a Hermitian positive definite system of linear equations with iterative refinement.                              |
| LSADS        | see page 177 | Solves a real symmetric positive definite system of linear equations with iterative refinement.                         |

|                                    |                                                                                                                                                                                                                 |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">LSAHF</a> see page 259 | Solves a complex Hermitian system of linear equations with iterative refinement.                                                                                                                                |
| <a href="#">LSAQH</a> see page 346 | Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.                                                                               |
| <a href="#">LSAQS</a> see page 303 | Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.                                                                                  |
| <a href="#">LSARB</a> see page 282 | Solves a real system of linear equations in band storage mode with iterative refinement.                                                                                                                        |
| <a href="#">LSARG</a> see page 83  | Solves a real general system of linear equations with iterative refinement.                                                                                                                                     |
| <a href="#">LSASF</a> see page 210 | Solves a real symmetric system of linear equations with iterative refinement.                                                                                                                                   |
| <a href="#">LSBRR</a> see page 463 | Solves a linear least-squares problem with iterative refinement.                                                                                                                                                |
| <a href="#">LSCXD</a> see page 399 | Performs the symbolic Cholesky factorization for a sparse symmetric matrix using a minimum degree ordering or a userspecified ordering, and set up the data structure for the numerical Cholesky factorization. |
| <a href="#">LSGRR</a> see page 514 | Computes the generalized inverse of a real matrix.                                                                                                                                                              |
| <a href="#">LSLCB</a> see page 330 | Solves a complex system of linear equations in band storage mode without iterative refinement.                                                                                                                  |
| <a href="#">LSLCC</a> see page 428 | Solves a complex circulant linear system.                                                                                                                                                                       |
| <a href="#">LSLCG</a> see page 123 | Solves a complex general system of linear equations without iterative refinement.                                                                                                                               |
| <a href="#">LSLCQ</a> see page 324 | Computes the <i>LDU</i> factorization of a complex tridiagonal matrix <i>A</i> using a cyclic reduction algorithm.                                                                                              |
| <a href="#">LSLCR</a> see page 279 | Computes the <i>LDU</i> factorization of a real tridiagonal matrix <i>A</i> using a cyclic reduction algorithm.                                                                                                 |
| <a href="#">LSLCT</a> see page 165 | Solves a complex triangular system of linear equations.                                                                                                                                                         |
| <a href="#">LSLDH</a> see page 232 | Solves a complex Hermitian positive definite system of linear equations without iterative refinement.                                                                                                           |
| <a href="#">LSLDS</a> see page 181 | Solves a real symmetric positive definite system of linear equations without iterative refinement.                                                                                                              |
| <a href="#">LSLHF</a> see page 262 | Solves a complex Hermitian system of linear equations without iterative refinement.                                                                                                                             |
| <a href="#">LSLPB</a> see page 308 | Computes the $R^T DR$ Cholesky factorization of a real symmetric positive definite matrix <i>A</i> in codiagonal band symmetric storage mode. Solve a system $Ax = b$ .                                         |
| <a href="#">LSLQB</a> see page 352 | Computes the $R^H DR$ Cholesky factorization of a complex hermitian positive-definite matrix <i>A</i> in codiagonal band hermitian storage mode. Solve a system $Ax = b$ .                                      |
| <a href="#">LSLQH</a> see page 349 | Solves a complex Hermitian positive definite system of linearequations in band Hermitian storage mode without iterative refinement.                                                                             |

|                                    |                                                                                                                                   |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">LSLQ</a> see page 305  | Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement. |
| <a href="#">LSLRB</a> see page 285 | Solves a real system of linear equations in band storage mode without iterative refinement.                                       |
| <a href="#">LSLRG</a> see page 87  | Solves a real general system of linear equations without iterative refinement.                                                    |
| <a href="#">LSLRT</a> see page 154 | Solves a real triangular system of linear equations.                                                                              |
| <a href="#">LSLSF</a> see page 213 | Solves a real symmetric system of linear equations without iterative refinement.                                                  |
| <a href="#">LSLTC</a> see page 426 | Solves a complex Toeplitz linear system.                                                                                          |
| <a href="#">LSLTO</a> see page 424 | Solves a real Toeplitz linear system.                                                                                             |
| <a href="#">LSLTQ</a> see page 322 | Solves a complex tridiagonal system of linear equations.                                                                          |
| <a href="#">LSLTR</a> see page 278 | Solves a real tridiagonal system of linear equations.                                                                             |
| <a href="#">LSLXD</a> see page 394 | Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.                         |
| <a href="#">LSLXG</a> see page 366 | Solves a sparse system of linear algebraic equations by Gaussian elimination.                                                     |
| <a href="#">LSLZD</a> see page 412 | Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.                           |
| <a href="#">LSLZG</a> see page 380 | Solves a complex sparse system of linear equations by Gaussian elimination.                                                       |
| <a href="#">LSQRR</a> see page 451 | Solves a linear least-squares problem without iterative refinement.                                                               |
| <a href="#">LSVCR</a> see page 510 | Computes the singular value decomposition of a complex matrix.                                                                    |
| <a href="#">LSVRR</a> see page 503 | Computes the singular value decomposition of a real matrix.                                                                       |
| <a href="#">LUPCH</a> see page 496 | Updates the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.         |
| <a href="#">LUPQR</a> see page 489 | Computes an updated $QR$ factorization after the rank-one matrix $\alpha xy^T$ is added.                                          |

## M

|                                        |                                                                                                                                                                                      |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">MCRCR</a> see page 1649    | Multiplies two complex rectangular matrices, $AB$ .                                                                                                                                  |
| <a href="#">MMOLCH</a> see page 1115   | Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials. |
| <a href="#">MP_SETUP</a> see page 1926 | Initializes or finalizes MPI.                                                                                                                                                        |
| <a href="#">MPS_FREE</a> see page 1485 | Deallocates the space allocated for the IMSL derived type <code>s_MPS</code> . This routine is usually used in conjunction with <code>READ_MPS</code> .                              |
| <a href="#">MRRRR</a> see page 1647    | Multiplies two real rectangular matrices, $AB$ .                                                                                                                                     |
| <a href="#">MUCBV</a> see page 1663    | Multiplies a complex band matrix in band storage mode by a complex vector.                                                                                                           |

|                                               |                                                                                                                                        |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">MUCRV</a> see page 1661           | Multiplies a complex rectangular matrix by a complex vector.                                                                           |
| <a href="#">MURBV</a> see page 1659           | Multiplies a real band matrix in band storage mode by a real vector.                                                                   |
| <a href="#">MURRV</a> see page 1657           | Multiplies a real rectangular matrix by a vector.                                                                                      |
| <a href="#">MXTXF</a> see page 1641           | Computes the transpose product of a matrix, $A^T A$ .                                                                                  |
| <a href="#">MXTYF</a> see page 1643           | Multiplies the transpose of matrix $A$ by matrix $B$ , $A^T B$ .                                                                       |
| <a href="#">MXYTF</a> see page 1645           | Multiplies a matrix $A$ by the transpose of a matrix $B$ , $AB^T$ .                                                                    |
| <b>N</b>                                      |                                                                                                                                        |
| <a href="#">NAN</a> see page 1770             | Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN.       |
| <a href="#">IERCD and NIRTY</a> see page 1934 | Retrieves an error type for the most recently called IMSL routine.                                                                     |
| <a href="#">NDAYS</a> see page 1877           | Computes the number of days from January 1, 1900, to the given date.                                                                   |
| <a href="#">NDYIN</a> see page 1878           | Gives the date corresponding to the number of days since January 1, 1900.                                                              |
| <a href="#">NEQBF</a> see page 1344           | Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.            |
| <a href="#">NEQBJ</a> see page 1350           | Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.                                     |
| <a href="#">NEQNF</a> see page 1337           | Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian. |
| <a href="#">NEQNJ</a> see page 1340           | Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.                         |
| <a href="#">NNLPF</a> see page 1522           | Uses a sequential equality constrained QP method.                                                                                      |
| <a href="#">NNLPG</a> see page 1528           | Uses a sequential equality constrained QP method.                                                                                      |
| <a href="#">NORM</a> see page 1771            | Computes the norm of a rank-1 or rank-2 array. For rank-3 arrays, the norms of each rank-2 array, in dimension 3, are computed.        |
| <a href="#">NR1CB</a> see page 1676           | Computes the 1-norm of a complex band matrix in band storage mode.                                                                     |
| <a href="#">NR1RB</a> see page 1674           | Computes the 1-norm of a real band matrix in band storage mode.                                                                        |
| <a href="#">NR1RR</a> see page 1671           | Computes the 1-norm of a real matrix.                                                                                                  |
| <a href="#">NR2RR</a> see page 1673           | Computes the Frobenius norm of a real rectangular matrix.                                                                              |
| <a href="#">NRIRR</a> see page 1670           | Computes the infinity norm of a real matrix.                                                                                           |

## O

### OPERATORS:

|                                    |                                                                                        |
|------------------------------------|----------------------------------------------------------------------------------------|
| <a href="#">.h.</a> see page 1726  | Computes transpose and conjugate transpose of a matrix.                                |
| <a href="#">.hx.</a> see page 1717 | Computes matrix-vector and matrix-matrix products.                                     |
| <a href="#">.i.</a> see page 1728  | Computes the inverse matrix, for square non-singular matrices.                         |
| <a href="#">.ix.</a> see page 1730 | Computes the inverse matrix times a vector or matrix for square non-singular matrices. |
| <a href="#">.t.</a> see page 1723  | Computes transpose and conjugate transpose of a matrix.                                |
| <a href="#">.tx.</a> see page 1711 | Computes matrix-vector and matrix-matrix products.                                     |
| <a href="#">.x.</a> see page 1707  | Computes matrix-vector and matrix-matrix products.                                     |
| <a href="#">.xh.</a> see page 1720 | Computes matrix-vector and matrix-matrix products.                                     |
| <a href="#">.xi.</a> see page 1740 | Computes the inverse matrix times a vector or matrix for square non-singular matrices. |
| <a href="#">.xt.</a> see page 1714 | Computes matrix-vector and matrix-matrix products.                                     |
| <a href="#">ORTH</a> see page 1774 | Orthogonalizes the columns of a rank-2 or rank-3 array.                                |

## P

|                                                                          |                                                                                                                             |
|--------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <a href="#">PCGRC</a> see page 431                                       | Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication. |
| <a href="#">PARALLEL_NONNEGATIVE_LS</a><br><a href="#">Q</a> see page 67 | Solves a linear, non-negative constrained least-squares system.                                                             |
| <a href="#">PARALLEL_BOUNDED_LSQ</a> see<br>page 75                      | Solves a linear least-squares system with bounds on the unknowns.                                                           |
| <a href="#">PDE_1D_MG</a> see page 1081                                  | Method of lines with Variable Griddings.                                                                                    |
| <a href="#">PERMA</a> see page 1844                                      | Permutates the rows or columns of a matrix.                                                                                 |
| <a href="#">PERMU</a> see page 1842                                      | Rearranges the elements of an array as specified by a permutation.                                                          |
| <a href="#">PGOPT</a> see page 1840                                      | Prints a plot of up to 10 sets of points.                                                                                   |
| <a href="#">PLOTB</a> see page 1914                                      | Prints a plot of up to 10 sets of points.                                                                                   |
| <a href="#">POLRG</a> see page 1655                                      | Evaluates a real general matrix polynomial.                                                                                 |
| <a href="#">PPIGD</a> see page 828                                       | Evaluates the derivative of a piecewise polynomial on a grid.                                                               |
| <a href="#">PPDER</a> see page 825                                       | Evaluates the derivative of a piecewise polynomial.                                                                         |
| <a href="#">PPITG</a> see page 831                                       | Evaluates the integral of a piecewise polynomial.                                                                           |
| <a href="#">PPVAL</a> see page 823                                       | Evaluates a piecewise polynomial.                                                                                           |
| <a href="#">PRIME</a> see page 1917                                      | Decomposes an integer into its prime factors.                                                                               |

## Q

|                                     |                                                          |
|-------------------------------------|----------------------------------------------------------|
| <a href="#">QAND</a> see page 973   | Integrates a function on a hyper-rectangle.              |
| <a href="#">QCOSB</a> see page 1270 | Computes a sequence from its cosine Fourier coefficients |

|                                     |                                                                                                                       |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
|                                     | with only odd wave numbers.                                                                                           |
| <a href="#">QCOSF</a> see page 1268 | Computes the coefficients of the cosine Fourier transform with only odd wave numbers.                                 |
| <a href="#">QCOSI</a> see page 1272 | Computes parameters needed by <a href="#">QCOSF</a> and <a href="#">QCOSEB</a> .                                      |
| <a href="#">QD2DR</a> see page 840  | Evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.                   |
| <a href="#">QD2VL</a> see page 838  | Evaluates a function defined on a rectangular grid using quadratic interpolation.                                     |
| <a href="#">QD3DR</a> see page 847  | Evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation. |
| <a href="#">QD3VL</a> see page 843  | Evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.                   |
| <a href="#">QDAG</a> see page 922   | Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.                                  |
| <a href="#">QDAGI</a> see page 935  | Integrates a function over an infinite or semi-infinite interval.                                                     |
| <a href="#">QDAGP</a> see page 925  | Integrates a function with singularity points given.                                                                  |
| <a href="#">QDAGID</a> see page 929 | Integrates a function with a possible internal or endpoint singularity.                                               |
| <a href="#">QDAG2D</a> see page 960 | Integrates a function of two variables with a possible internal or end point singularity.                             |
| <a href="#">QDAG3D</a> see page 966 | Integrates a function of three variables with a possible internal or endpoint singularity.                            |
| <a href="#">QDAGS</a> see page 918  | Integrates a function (which may have endpoint singularities).                                                        |
| <a href="#">QDAWC</a> see page 949  | Integrates a function $F(X)/(X - C)$ in the Cauchy principal value sense.                                             |
| <a href="#">QDAWF</a> see page 942  | Computes a Fourier integral.                                                                                          |
| <a href="#">QDAWO</a> see page 938  | Integrates a function containing a sine or a cosine.                                                                  |
| <a href="#">QDAWS</a> see page 946  | Integrates a function with algebraic-logarithmic singularities.                                                       |
| <a href="#">QDDER</a> see page 835  | Evaluates the derivative of a function defined on a set of points using quadratic interpolation.                      |
| <a href="#">QDNG</a> see page 953   | Integrates a smooth function using a nonadaptive rule.                                                                |
| <a href="#">QDVAL</a> see page 833  | Evaluates a function defined on a set of points using quadratic interpolation.                                        |
| <a href="#">QMC</a> see page 976    | Integrates a function over a hyperrectangle using a quasi-Monte Carlo method.                                         |
| <a href="#">QPROG</a> see page 1506 | Solves a quadratic programming problem subject to linear equality/inequality constraints.                             |
| <a href="#">QSINB</a> see page 1263 | Computes a sequence from its sine Fourier coefficients with only odd wave numbers.                                    |
| <a href="#">QSINF</a> see page 1261 | Computes the coefficients of the sine Fourier transform with only odd wave numbers.                                   |

[QSINI](#) see page 1266

Computes parameters needed by [QSINF](#) and [QSINB](#).

## R

[RAND](#) see page 1777

Computes a scalar, rank-1, rank-2 or rank-3 array of random numbers.

[RAND\\_GEN](#) see page 1882

Generates a rank-1 array of random numbers.

[RANK](#) see page 1779

Computes the mathematical rank of a rank-2 or rank-3 array.

[RATCH](#) see page 910

Computes a rational weighted Chebyshev approximation to a continuous function on an interval.

[RCONV](#) see page 1289

Computes the convolution of two real vectors.

[RCORL](#) see page 1299

Computes the correlation of two real vectors.

[RCURV](#) see page 861

Fits a polynomial curve using least squares.

[READ\\_MPS](#) see page 1475

Reads an MPS file containing a linear program problem or a quadratic programming problem.

[RECCF](#) see page 986

Computes recurrence coefficients for various monic polynomials.

[RECQR](#) see page 988

Computes recurrence coefficients for monic polynomials given a quadrature rule.

[RLINE](#) see page 858

Fits a line to a set of data points using least squares.

[RNGET](#) see page 1891

Retrieves the current value of the seed used in the IMSL random number generators.

[RNIN32](#) see page 1894

Initializes the 32-bit Merseene Twister generator using an array.

[RNGE32](#) see page 1895

Retrieves the current table used in the 32-bit Mersenne Twister generator.

[RNSE32](#) see page 1897

Sets the current table used in the 32-bit Mersenne Twister generator.

[RNIN64](#) see page 1897

Initializes the 32-bit Merseene Twister generator using an array.

[RNGE64](#) see page 1898

Retrieves the current table used in the 64-bit Mersenne Twister generator

[RNSE64](#) see page 1900

Sets the current table used in the 64-bit Mersenne Twister generator.

[RNOPT](#) see page 1893

Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.

[RNSET](#) see page 1892

Initializes a random seed for use in the IMSL random number generators.

[RNUN](#) see page 1902

Generates pseudorandom numbers from a uniform (0, 1) distribution.

[RNUNF](#) see page 1900

Generates a pseudorandom number from a uniform (0, 1) distribution.

## S

|                                                |                                                                                                                                                                                        |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">SADD</a>                           | Adds a scalar to each component of a vector, $x \leftarrow x + a$ , all single precision.                                                                                              |
| <a href="#">SASUM</a>                          | Sums the absolute values of the components of a single-precision vector.                                                                                                               |
| <a href="#">SAXPY</a>                          | Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$ , all single precision.                                                                                        |
| <a href="#">ScaLAPACK_EXIT</a> see page 1809   | Exits ScaLAPACK mode for the IMSL Library routines.                                                                                                                                    |
| <a href="#">ScaLAPACK_GETDIM</a> see page 1794 | Calculates the row and column dimensions of a local distributed array based on the size of the array to be distributed and the row and column blocking factors to be used.             |
| <a href="#">ScaLAPACK_MAP</a> see page 1805    | Maps array data from a global array to local arrays in the two-dimensional block-cyclic form required by <i>ScaLAPACK</i> routines.                                                    |
| <a href="#">ScaLAPACK_READ</a> see page 1795   | Reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by <i>ScaLAPACK</i> routines.                                                       |
| <a href="#">ScaLAPACK_SETUP</a> see page 1792  | Sets up a processor grid and calculates default values for use in mapping arrays to the processor grid                                                                                 |
| <a href="#">ScaLAPACK_UNMAP</a> see page 1807  | Unmaps array data from local distributed arrays to a global array.                                                                                                                     |
| <a href="#">ScaLAPACK_WRITE</a> see page 1797  | Writes the matrix data to a file.                                                                                                                                                      |
| <a href="#">SCASUM</a>                         | Sums the absolute values of the real part together with the absolute values of the imaginary part of the components of a complex vector.                                               |
| <a href="#">SCNRM2</a>                         | Computes the Euclidean norm of a complex vector.                                                                                                                                       |
| <a href="#">SCOPY</a>                          | Copies a vector $x$ to a vector $y$ , both single precision.                                                                                                                           |
| <a href="#">SDDOTA</a>                         | Computes the sum of a single-precision scalar, a single-precision dot product and the double-precision accumulator, which is set to the result $ACC \leftarrow ACC + a + x^T y$ .      |
| <a href="#">SDDOTI</a>                         | Computes the sum of a single-precision scalar plus a single-precision dot product using a double-precision accumulator, which is set to the result $ACC \leftarrow a + x^T y$ .        |
| <a href="#">SDOT</a>                           | Computes the single-precision dot product $x^T y$ .                                                                                                                                    |
| <a href="#">SDSDOT</a>                         | Computes the sum of a single-precision scalar and a single-precision dot product, $a + x^T y$ , using a double-precision accumulator.                                                  |
| <a href="#">SGBMV</a>                          | Computes one of the matrix-vector operations:<br>$y \leftarrow \alpha Ax + \beta y$ , or $y \leftarrow \alpha A^T x + \beta y$ ,<br>where $A$ is a matrix stored in band storage mode. |
| <a href="#">SGEMM</a>                          | Computes one of the matrix-matrix operations:                                                                                                                                          |

$$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T + \beta C, \text{ or } C \leftarrow \alpha A^T B^T + \beta C$$

|                                                 |                                                                                                                                                                                  |
|-------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">SGEMV</a>                           | Computes one of the matrix-vector operations:<br>$y \leftarrow \alpha Ax + \beta y$ , or $y \leftarrow \alpha A^T x + \beta y$ ,                                                 |
| <a href="#">SGER</a>                            | Computes the rank-one update of a real general matrix:<br>$A \leftarrow A + \alpha xy^T$ .                                                                                       |
| <a href="#">SHOW</a> see page 1813              | Prints rank-1 or rank-2 arrays of numbers in a readable format.                                                                                                                  |
| <a href="#">SHPROD</a>                          | Computes the Hadamard product of two single-precision vectors.                                                                                                                   |
| <a href="#">SINLP</a> see page 1311             | Computes the inverse Laplace transform of a complex function.                                                                                                                    |
| <a href="#">SLCNT</a> see page 1213             | Calculates the indices of eigenvalues of a Sturm-Liouville problem with boundary conditions (at regular points) in a specified subinterval of the real line, $[\alpha, \beta]$ . |
| <a href="#">SLEIG</a> see page 1201             | Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form with boundary conditions (at regular points).                  |
| <a href="#">SLPRS</a> see page 1497             | Solves a sparse linear programming problem via the revised simplex algorithm.                                                                                                    |
| <a href="#">SNRM2</a>                           | Computes the Euclidean length or $L_2$ norm of a single-precision vector.                                                                                                        |
| <a href="#">SORT_REAL</a> see page 1846         | Sorts a rank-1 array of real numbers $x$ so the $y$ results are algebraically nondecreasing, $y_1 \leq y_2 \leq \dots y_n$ .                                                     |
| <a href="#">SPLEZ</a> see page 758              | Computes the values of a spline that either interpolates or fits user-supplied data.                                                                                             |
| <a href="#">SPLINE_CONSTRAINTS</a> see page 702 | Returns the derived type array result.                                                                                                                                           |
| <a href="#">SPLINE_FITTING</a> see page 704     | Weighted least-squares fitting by B-splines to discrete One-Dimensional data is performed.                                                                                       |
| <a href="#">SPLINE_VALUES</a> see page 703      | Returns an array result, given an array of input                                                                                                                                 |
| <a href="#">SPRDC</a>                           | Multiplies the components of a single-precision vector.                                                                                                                          |
| <a href="#">SRCH</a> see page 1860              | Searches a sorted vector for a given scalar and return its index.                                                                                                                |
| <a href="#">SROT</a>                            | Applies a Givens plane rotation in single precision.                                                                                                                             |
| <a href="#">SROTG</a>                           | Constructs a Givens plane rotation in single precision.                                                                                                                          |
| <a href="#">SROTM</a>                           | Applies a modified Givens plane rotation in single precision.                                                                                                                    |
| <a href="#">SROTMG</a>                          | Constructs a modified Givens plane rotation in single precision.                                                                                                                 |
| <a href="#">SSBMV</a>                           | Computes the matrix-vector operation<br>$y \leftarrow \alpha Ax + \beta y$ ,<br>where $A$ is a symmetric matrix in band symmetric storage                                        |

|        |                                                                                                                                                                                                                                                                                                              |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | mode.                                                                                                                                                                                                                                                                                                        |
| SSCAL  | Multiplies a vector by a scalar, $y \leftarrow ay$ , both single precision.                                                                                                                                                                                                                                  |
| SSET   | Sets the components of a vector to a scalar, all single precision.                                                                                                                                                                                                                                           |
| SSPMV  | Performs the matrix-vector operation $y := \alpha * A * x + \beta * y$ .                                                                                                                                                                                                                                     |
| SSPR   | Performs the matrix-vector operation.                                                                                                                                                                                                                                                                        |
| SSPR2  | Performs the symmetric rank 2 operation.                                                                                                                                                                                                                                                                     |
| SSRCH  | see page 1864<br>Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.                                                                                                                                                                                      |
| SSUB   | Subtracts each component of a vector from a scalar, $x \leftarrow a - x$ , all single precision.                                                                                                                                                                                                             |
| SSUM   | Sums the values of a single-precision vector.                                                                                                                                                                                                                                                                |
| SSWAP  | Interchanges vectors $x$ and $y$ , both single precision.                                                                                                                                                                                                                                                    |
| SSYMM  | Computes one of the matrix-matrix operations:<br>$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ ,<br>where $A$ is a symmetric matrix and $B$ and $C$ are $m$ by $n$ matrices.                                                                                                      |
| SSYMV  | Computes the matrix-vector operation<br>$y \leftarrow \alpha Ax + \beta y$ ,<br>where $A$ is a symmetric matrix.                                                                                                                                                                                             |
| SSYR   | Computes the rank-one update of a real symmetric matrix:<br>$A \leftarrow A + \alpha xx^T$ .                                                                                                                                                                                                                 |
| SSYR2  | Computes the rank-two update of a real symmetric matrix:<br>$A \leftarrow A + \alpha xy^T + \alpha yx^T$ .                                                                                                                                                                                                   |
| SSYR2K | Computes one of the symmetric rank $2k$ operations:<br>$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A$<br>where $C$ is an $n$ by $n$ symmetric matrix and $A$ and $B$ are $n$ by $k$ matrices in the first case and $k$ by $n$ matrices in the second case. |
| SSYRK  | Computes one of the symmetric rank $k$ operations:<br>$C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ ,<br>where $C$ is an $n$ by $n$ symmetric matrix and $A$ is an $n$ by $k$ matrix in the first case and a $k$ by $n$ matrix in the second case.                           |
| STBMV  | Computes one of the matrix-vector operations:<br>$x \leftarrow Ax$ or $x \leftarrow A^T x$ ,<br><br>where $A$ is a triangular matrix in band storage mode.                                                                                                                                                   |
| STBSV  | Solves one of the triangular systems:<br>$x \leftarrow A^{-1}x$ or $x \leftarrow (A^{-1})^T x$ ,<br><br>where $A$ is a triangular matrix in band storage mode.                                                                                                                                               |
| STPMV  | Performs one of the matrix-vector operations.                                                                                                                                                                                                                                                                |
| STPSV  | Solves one of the systems of equations.                                                                                                                                                                                                                                                                      |

|                                                     |                                                                                                                                                                                            |
|-----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">STRMM</a>                               | Computes one of the matrix-matrix operations:<br>$B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, \text{ or } B \leftarrow \alpha BA, B \leftarrow \alpha BA^T$                         |
|                                                     | where $B$ is an $m$ by $n$ matrix and $A$ is a triangular matrix.                                                                                                                          |
| <a href="#">STRMV</a>                               | Computes one of the matrix-vector operations:<br>$x \leftarrow Ax \text{ or } x \leftarrow A^T x,$<br>where $A$ is a triangular matrix.                                                    |
| <a href="#">STRSM</a>                               | Solves one of the matrix equations:<br>$B \leftarrow \alpha A^{-1} B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha (A^{-1})^T B$<br><br>$\text{or } B \leftarrow \alpha B (A^{-1})^T,$ |
|                                                     | where $B$ is an $m$ by $n$ matrix and $A$ is a triangular matrix.                                                                                                                          |
| <a href="#">STRSV</a>                               | Solves one of the triangular linear systems:<br>$x \leftarrow A^{-1} x \text{ or } x \leftarrow (A^{-1})^T x$<br>where $A$ is a triangular matrix.                                         |
| <a href="#">SURF</a> see page 851                   | Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.                                                                           |
| <a href="#">SURFND</a> see page 855                 | Multidimensional interpolation and differentiation.                                                                                                                                        |
| <a href="#">SURFACE_CONSTRAINTS</a><br>see page 714 | Returns the derived type array result given optional input.                                                                                                                                |
| <a href="#">SURFACE_FITTING</a> see page 716        | Weighted least-squares fitting by tensor product B-splines to discrete two-dimensional data is performed.                                                                                  |
| <a href="#">SURFACE_VALUES</a> see page 715         | Returns a tensor product array result, given two arrays of independent variable values.                                                                                                    |
| <a href="#">SVCAL</a>                               | Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$ , all single precision.                                                                          |
| <a href="#">SVD</a> see page 1780                   | Computes the singular value decomposition of a rank-2 or rank-3 array, $A = USV^T$ .                                                                                                       |
| <a href="#">SVIBN</a> see page 1857                 | Sorts an integer array by nondecreasing absolute value.                                                                                                                                    |
| <a href="#">SVIGN</a> see page 1852                 | Sorts an integer array by algebraically increasing value.                                                                                                                                  |
| <a href="#">SVIGP</a> see page 1853                 | Sorts an integer array by algebraically increasing value and returns the permutation that rearranges the array.                                                                            |
| <a href="#">SVRBN</a> see page 1855                 | Sorts a real array by nondecreasing absolute value.                                                                                                                                        |
| <a href="#">SVRBP</a> see page 1856                 | Sorts a real array by nondecreasing absolute value and returns the permutation that rearranges the array.                                                                                  |
| <a href="#">SVRGN</a> see page 1849                 | Sorts a real array by algebraically increasing value.                                                                                                                                      |
| <a href="#">SVRGP</a> see page 1850                 | Sorts a real array by algebraically increasing value and returns the permutation that rearranges the array.                                                                                |
| <a href="#">SXYZ</a>                                | Computes a single-precision $xyz$ product.                                                                                                                                                 |

## T

|                                     |                                               |
|-------------------------------------|-----------------------------------------------|
| <a href="#">TDATE</a> see page 1876 | Gets today's date.                            |
| <a href="#">TIMDY</a> see page 1875 | Gets time of day.                             |
| <a href="#">TRAN</a> see page 1504  | Solves a transportation problem.              |
| <a href="#">TRNRR</a> see page 1639 | Transposes a rectangular matrix.              |
| <a href="#">TWODQ</a> see page 955  | Computes a two-dimensional iterated integral. |

## U

|                                     |                                                                                                                                     |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">UMACH</a> see page 1942 | Sets or retrieves input or output device unit numbers.                                                                              |
| <a href="#">UMAG</a> see page 1911  | Handles MATH/LIBRARY and STAT/LIBRARY type REAL and double precision options.                                                       |
| <a href="#">UMCGF</a> see page 1395 | Minimizes a function of N variables using a conjugate gradient algorithm and a finite-difference gradient.                          |
| <a href="#">UMCGG</a> see page 1399 | Minimizes a function of N variables using a conjugate gradient algorithm and a user-supplied gradient.                              |
| <a href="#">UMIAH</a> see page 1389 | Minimizes a function of N variables using a modified Newton method and a user-supplied Hessian.                                     |
| <a href="#">UMIDH</a> see page 1384 | Minimizes a function of N variables using a modified Newton method and a finite-difference Hessian.                                 |
| <a href="#">UMINF</a> see page 1372 | Minimizes a function of N variables using a modified Newton method and a finite-difference Hessian.                                 |
| <a href="#">UMING</a> see page 1377 | Minimizes a function of N variables using a quasi-New method and a finite-difference gradient.                                      |
| <a href="#">UMPOL</a> see page 1403 | Minimizes a function of N variables using a direct search polytope algorithm.                                                       |
| <a href="#">UNIT</a> see page 1783  | Normalizes the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.                                       |
| <a href="#">UNLSF</a> see page 1407 | Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.           |
| <a href="#">UNLSJ</a> see page 1413 | Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.               |
| <a href="#">UVMGS</a> see page 1369 | Finds the minimum point of a nonsmooth function of a single variable.                                                               |
| <a href="#">UVMID</a> see page 1365 | Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations. |
| <a href="#">UVMIF</a> see page 1362 | Finds the minimum point of a smooth function of a single variable using only function evaluations.                                  |

## V

|                                     |                                                  |
|-------------------------------------|--------------------------------------------------|
| <a href="#">VCONC</a> see page 1685 | Computes the convolution of two complex vectors. |
| <a href="#">VCONR</a> see page 1683 | Computes the convolution of two real vectors.    |

[VERML](#) see page 1881

Obtains IMSL MATH/LIBRARY-related version, system and license numbers.

## W

[WRCRL](#) see page 1830

Prints a complex rectangular matrix with a given format and labels.

[WRCRN](#) see page 1827

Prints a complex rectangular matrix with integer row and column labels.

[WRIRL](#) see page 1825

Prints an integer rectangular matrix with a given format and labels.

[WRIRN](#) see page 1822

Prints an integer rectangular matrix with integer row and column labels.

[WROPT](#) see page 1833

Sets or retrieves an option for printing a matrix.

[WRRRL](#) see page 1819

Prints a real rectangular matrix with a given format and labels.

[WRRRN](#) see page 1817

Prints a real rectangular matrix with integer row and column labels.

## Z

[ZANLY](#) see page 1325

Finds the zeros of a univariate complex function using Müller's method.

[ZBREN](#) see page 1331

Finds a zero of a real function that changes sign in a given interval.

[ZPLRC](#) see page 1320

Finds the zeros of a polynomial with real coefficients using Laguerre's method.

[ZPOCC](#) see page 1324

Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.

[ZPORC](#) see page 1322

Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.

[ZQADD](#)

Adds a double complex scalar to the accumulator in extended precision.

[ZQINI](#)

Initializes an extended-precision complex accumulator to a double complex scalar.

[ZQMUL](#)

Multiplies double complex scalars using extended precision.

[ZQSTO](#)

Stores a double complex approximation to an extended-precision complex scalar.

[ZREAL](#) see page 1334

Finds the real zeros of a real function using Müller's method.

[ZUNI](#) see page 1328

Finds a zero of a real univariate function.



# Appendix C: References

## **Adams et al.**

Adams et al. (2008), *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*, 561.

## **Aird and Howell**

Aird, Thomas J., and Byron W. Howell (1991), IMSL Technical Report 9103, IMSL, Houston.

## **Aird and Rice**

Aird, T.J., and J.R. Rice (1977), Systematic search in high dimensional sets, *SIAM Journal on Numerical Analysis*, **14**, 296–312.

## **Akima**

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, *Journal of the ACM*, **17**, 589–602.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, *ACM Transactions on Mathematical Software*, **4**, 148–159.

## **Anderson et al.**

Anderson, E., Bai, Z., Bishop, C., Blackford, S., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999), *LAPACK Users' Guide*, SIAM, 3<sup>rd</sup> ed., Philadelphia.

## **Arushanian et al.**

Arushanian, O.B., M.K. Samarin, V.V. Voevodin, E.E. Tyrtysnikov, B.S. Garbow, J.M. Boyle, W.R. Cowell, and K.W. Dritz (1983), *The TOEPLITZ Package Users' Guide*, Argonne National Laboratory, Argonne, Illinois.

## **Ashcraft**

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

### **Ashcraft et al.**

Ashcraft, C., R.Grimes, J. Lewis, B. Peyton, and H. Simon (1987), Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.*, **1**(4), 10–29.

### **Atkinson**

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

### **Atchison and Hanson**

Atchison, M.A., and R.J. Hanson (1991), *An Options Manager for the IMSL Fortran 77 Libraries*, Technical Report 9101, IMSL, Houston.

### **Bischof et al.**

Bischof, C., J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, D. Sorensen (1988), LAPACK Working Note #5: Provisional Contents, Argonne National Laboratory Report ANL-88-38, Mathematics and Computer Science.

### **Bjorck**

Bjorck, Ake (1967), Iterative refinement of linear least squares solutions I, *BIT*, **7**, 322–337.

Bjorck, Ake (1968), Iterative refinement of linear least squares solutions II, *BIT*, **8**, 8–30.

### **Boisvert (1984)**

Boisvert, Ronald (1984), A fourth order accurate fast direct method for the Helmholtz equation, *Elliptic Problem Solvers II*, (edited by G. Birkhoff and A. Schoenstadt), Academic Press, Orlando, Florida, 35–44.

### **Boisvert, Howe, and Kahaner**

Boisvert, Ronald F., Sally E. Howe, and David K. Kahaner (1985), GAMS: A framework for the management of scientific software, *ACM Transactions on Mathematical Software*, **11**, 313–355.

### **Boisvert, Howe, Kahaner, and Springmann**

Boisvert, Ronald F., Sally E. Howe, David K. Kahaner, and Jeanne L. Springmann (1990), *Guide to Available Mathematical Software*, NISTIR 90-4237, National Institute of Standards and Technology, Gaithersburg, Maryland.

### **Blackford et al.**

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C., (1997), *ScaLAPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA.

### **Brankin et al.**

Brankin, R.W., I. Gladwell, and L.F. Shampine, RKSUITE: a Suite of Runge-Kutta Codes for the Initial Value Problem for ODEs, Softreport 91-1, Mathematics Department, Southern Methodist University, Dallas, Texas, 1991.

### **Brenan, Campbell, and Petzold**

Brenan, K.E., S.L. Campbell, L.R. Petzold (1989), Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Elsevier Science Publ. Co.

### **Brenner**

Brenner, N. (1973), Algorithm 467: Matrix transposition in place [F1], *Communication of ACM*, **16**, 692–694.

### **Brent**

Brent, R.P. (1971), An algorithm with guaranteed convergence for finding a zero of a function, *The Computer Journal*, **14**, 422–425.

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

### **Brigham**

Brigham, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

### **Cheney**

Cheney, E.W. (1966), *Introduction to Approximation Theory*, McGraw-Hill, New York.

### **Cline et al.**

Cline, A.K., C.B. Moler, G.W. Stewart, and J.H. Wilkinson (1979), An estimate for the condition number of a matrix, *SIAM Journal of Numerical Analysis*, **16**, 368–375.

### **Cody, Fraser, and Hart**

Cody, W.J., W. Fraser, and J.F. Hart (1968), Rational Chebyshev approximation using linear equations, *Numerische Mathematik*, **12**, 242–251.

### **Cohen and Taylor**

Cohen, E. Richard, and Barry N. Taylor (1986), *The 1986 Adjustment of the Fundamental Physical Constants*, Codata Bulletin, Pergamon Press, New York.

### **Cooley and Tukey**

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.

### **Courant and Hilbert**

Courant, R., and D. Hilbert (1962), *Methods of Mathematical Physics, Volume II*, John Wiley & Sons, New York, NY.

### **Craven and Wahba**

Craven, Peter, and Grace Wahba (1979), Smoothing noisy data with spline functions, *Numerische Mathematik*, **31**, 377–403.

### **Crowe et al.**

Crowe, Keith, Yuan-An Fan, Jing Li, Dale Neaderhouser, and Phil Smith (1990), *A direct sparse linear equation solver using linked list storage*, IMSL Technical Report 9006, IMSL, Houston.

### **Crump**

Crump, Kenny S. (1976), Numerical inversion of Laplace transforms using a Fourier series approximation, *Journal of the Association for Computing Machinery*, **23**, 89–96.

### **Davis and Rabinowitz**

Davis, Philip F., and Philip Rabinowitz (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

### **de Boor**

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

### **de Hoog, Knight, and Stokes**

de Hoog, F.R., J.H. Knight, and A.N. Stokes (1982), An improved method for numerical inversion of Laplace transforms. *SIAM Journal on Scientific and Statistical Computing*, **3**, 357–366.

### **Demmel et al**

Demmel, J.W., Gilbert, J.R., and Li, X.S. (2003), *SuperLU User's Guide*, University of California, Berkeley, CA, Xerox Corporation.

### **Dennis and Schnabel**

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

### **Dongarra et al.**

Dongarra, J.J., and C.B. Moler, (1977) *EISPACK – A package for solving matrix eigenvalue problems*, Argonne National Laboratory, Argonne, Illinois.

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

Dongarra, J.J., J. DuCroz, S. Hammarling, R. J. Hanson (1988), An Extended Set of Fortran basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, **14**, 1–17.

Dongarra, J.J., J. DuCroz, S. Hammarling, I. Duff (1990), A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, **16**, 1–17.

### **Draper and Smith**

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, second edition, John Wiley & Sons, New York.

### **Du Croz et al.**

Du Croz, Jeremy, P. Mayes, G. and Radicati (1990), Factorization of band matrices using Level-3 BLAS, *Proceedings of CONPAR 90 VAPP IV, Lecture Notes in Computer Science*, Springer, Berlin, 222.

### **Duff and Reid**

Duff, I.S., and J.K. Reid (1983), The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, **9**, 302–325.

Duff, I.S., and J.K. Reid (1984), The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, **5**, 633–641.

### **Duff et al.**

Duff, I.S., A.M. Erisman, and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

### **Duff et al.**

Duff, Ian S., R. G. Grimes, and J. G. Lewis (1992) first ed, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, CERFACS, Toulouse Cedex, France.

### **Enright and Pryce**

Enright, W.H., and J.D. Pryce (1987), Two FORTRAN packages for assessing initial value methods, *ACM Transactions on Mathematical Software*, **13**, 1–22.

### **Fabijonas**

B. R. Fabijonas, Algorithm 838: Airy Functions, *ACM Transactions on Mathematical Software*, Vol. 30, No. 4, December 2004, Pages 491–501.

### **Fabijonas et al.**

B. R. Fabijonas, D. W. Lozier, and F. W. J. Olver Computation of Complex Airy Functions and Their Zeros Using Asymptotics and the Differential Equation, *ACM Transactions on Mathematical Software*, Vol. 30, No. 4, December 2004, 471–490.

### **Forsythe**

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74–88.

### **Fox, Hall, and Schryer**

Fox, P.A., A.D. Hall, and N.L. Schryer (1978), The PORT mathematical subroutine library, *ACM Transactions on Mathematical Software*, **4**, 104–126.

### **Garbow**

Garbow, B.S. (1978) CALGO Algorithm 535: The QZ algorithm to solve the generalized eigenvalue problem for complex matrices, *ACM Transactions on Mathematical Software*, **4**, 404–410.

### **Garbow et al.**

Garbow, B.S., J.M. Boyle, J.J. Dongarra, and C.B. Moler (1972), *Matrix eigensystem Routines: EISPACK Guide Extension*, Springer-Verlag, New York.

Garbow, B.S., J.M. Boyle, J.J. Dongarra, and C.B. Moler (1977), *Matrix Eigensystem Routines—EISPACK Guide Extension*, Springer-Verlag, New York.

Garbow, B.S., G. Giunta, J.N. Lyness, and A. Murli (1988), Software for an implementation of Weeks' method for the inverse Laplace transform problem, *ACM Transactions of Mathematical Software*, **14**, 163–170.

### **Gautschi**

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251–270.

### **Gautschi and Milovanovic**

Gautschi, Walter, and Gradimir V. Milovanovic (1985), Gaussian quadrature involving Einstein and Fermi functions with an application to summation of series, *Mathematics of Computation*, **44**, 177–190.

### **Gay**

Gay, David M. (1981), Computing optimal locally constrained steps, *SIAM Journal on Scientific and Statistical Computing*, **2**, 186–197.

Gay, David M. (1983), Algorithm 611: Subroutine for unconstrained minimization using a model/trust-region approach, *ACM Transactions on Mathematical Software*, **9**, 503– 524.

### **Gear**

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

### **Gear and Petzold**

Gear, C.W., and Linda R. Petzold (1984), ODE methods for the solutions of differential/algebraic equations, *SIAM Journal Numerical Analysis*, **21**, #4, 716.

### **George and Liu**

George, A., and J.W.H. Liu (1981), *Computer Solution of Large Sparse Positive-definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

### **Gill et al.**

Gill, Philip E., and Walter Murray (1976), *Minimization subject to bounds on the variables*, NPL Report NAC 72, National Physical Laboratory, England.

Gill, Philip E., Walter Murray, and Margaret Wright (1981), *Practical Optimization*, Academic Press, New York.

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

### **Goldfarb and Idnani**

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1–33.

### **Golub**

Golub, G.H. (1973), Some modified matrix eigenvalue problems, *SIAM Review*, **15**, 318–334.

### **Golub and Van Loan**

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1989), *Matrix Computations*, 2d ed., Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1996), *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, Maryland.

### **Golub and Welsch**

Golub, G.H., and J.H. Welsch (1969), Calculation of Gaussian quadrature rules, *Mathematics of Computation*, **23**, 221–230.

### **Gregory and Karney**

Gregory, Robert, and David Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, John Wiley & Sons, New York.

### **Griffin and Redish**

Griffin, R., and K.A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

### **Grosse**

Grosse, Eric (1980), Tensor spline approximation, *Linear Algebra and its Applications*, **34**, 29–41.

### **Guerra and Tapia**

Guerra, V., and R. A. Tapia (1974), *A local procedure for error detection and data smoothing*, MRC Technical Summary Report 1452, Mathematics Research Center, University of Wisconsin, Madison.

### **Hageman and Young**

Hageman, Louis A., and David M. Young (1981), *Applied Iterative Methods*, Academic Press, New York.

### **Hanson**

Hanson, Richard J. (1986), Least squares with bounds and linear constraints, *SIAM Journal Sci. Stat. Computing*, **7**, #3.

Hanson, Richard J. (1990), *A cyclic reduction solver for the IMSL Mathematics Library*, IMSL Technical Report 9002, IMSL, Houston.

### **Hanson et al.**

Hanson, Richard J., R. Lehoucq, J. Stolle, and A. Belmonte (1990), *Improved performance of certain matrix eigenvalue computations for the IMSL/MATH Library*, IMSL Technical Report 9007, IMSL, Houston.

### **Hartman**

Hartman, Philip (1964) *Ordinary Differential Equations*, John Wiley and Sons, New York, NY.

## Hausman

Hausman, Jr., R.F. (1971), *Function Optimization on a Line Segment by Golden Section*, Lawrence Radiation Laboratory, University of California, Livermore.

## Hindmarsh

Hindmarsh, A.C. (1974), *GEAR: Ordinary differential equation system solver*, Lawrence Livermore Laboratory Report UCID-30001, Revision 3.

## Hull et al.

Hull, T.E., W.H. Enright, and K.R. Jackson (1976), *User's guide for DVERK – A subroutine for solving non-stiff ODEs*, Department of Computer Science Technical Report 100, University of Toronto.

## IEEE

ANSI/IEEE Std 754-1985 (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, The IEEE, Inc., New York.

## IMSL (1991)

IMSL (1991), *IMSL STAT/LIBRARY User's Manual, Version 2.0*, IMSL, Houston.

## Irvine et al.

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129–151.

## Jenkins

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178–189.

## Jenkins and Traub

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, **7**, 545–566.

Jenkins, M.A., and J.F. Traub (1970), A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration, *Numerische Mathematik*, **14**, 252–263.

Jenkins, M.A., and J.F. Traub (1972), Zeros of a complex polynomial, *Communications of the ACM*, **15**, 97–99.

## Kennedy and Gentle

Kennedy, William J., Jr., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

## **Kershaw**

Kershaw, D. (1982), Solution of tridiagonal linear systems and vectorization of the ICCG algorithm on the Cray-1, *Parallel Computations*, Academic Press, Inc., 85-99.

## **Knuth**

Knuth, Donald E. (1973), *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Addison-Wesley Publishing Company, Reading, Mass.

## **Krogh**

Krogh, Fred T. (1970), Efficient Algorithms for Polynomial Interpolation and Numerical Differentiation, *Math. of Comp.* 24, **109**, 185-190.

Krogh, Fred T. (2005), *An Algorithm for Linear Programming*, <http://mathalacarte.com/fkrogh/pub/lp.pdf>, Tojunga, CA.

## **Lawson et al.**

Lawson, C.L., R.J. Hanson, D.R. Kincaid, and F.T. Krogh (1979), Basic linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software*, **5**, 308– 323.

## **Leavenworth**

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

## **Lehoucq et al.**

Lehoucq, R. B., Danny C. Sorenson, and Chao Yang (1998), *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia.

## **Levenberg**

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

## **Lewis et al.**

Lewis, P.A. W., A.S. Goodman, and J.M. Miller (1969), A pseudo-random number generator for the System/360, *IBM Systems Journal*, **8**, 136–146.

## **Liepman**

Liepman, David S. (1964), Mathematical constants, in *Handbook of Mathematical Functions*, Dover Publications, New York.

## **Liu**

Liu, J.W.H. (1986), On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, **12**, 249–264.

Liu, J.W.H. (1987), *A collection of routines for an implementation of the multifrontal method*, Technical Report CS-87-10, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1989), The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, **15**, 310–325.

Liu, J.W.H. (1990), The multifrontal method for sparse matrix solution: theory and practice, Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada.

## **Liu and Ashcraft**

Liu, J., and C. Ashcraft (1987), *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

## **Lyness and Giunta**

Lyness, J.N. and G. Giunta (1986), A modification of the Weeks Method for numerical inversion of the Laplace transform, *Mathematics of Computation*, **47**, 313–322.

## **Madsen and Sincovec**

Madsen, N.K., and R.F. Sincovec (1979), Algorithm 540: PDECOL, General collocation software for partial differential equations, *ACM Transactions on Mathematical Software*, **5**, #3, 326-351.

## **Marquardt**

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431–441.

## **Martin and Wilkinson**

Martin, R.S., and J.W. Wilkinson (1968), Reduction of the symmetric eigenproblem  $Ax = \lambda Bx$  and related problems to standard form, *Numerische Mathematik*, **11**, 99–119.

## **Matsumoto and Nishimure**

Makoto Matsumoto and Takuji Nishimura, *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998, Pages 3–30.

## **Micchelli et al.**

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth functions, *Numerische Mathematik*, **26**, 279–285

Micchelli, C.A., Philip W. Smith, John Swetits, and Joseph D. Ward (1985), Constrained  $L_p$  approximation, *Constructive Approximation*, **1**, 93–102.

### **Moler and Stewart**

Moler, C., and G.W. Stewart (1973), An algorithm for generalized matrix eigenvalue problems, *SIAM Journal on Numerical Analysis*, **10**, 241–256.

### **More et al.**

More, Jorge, Burton Garbow, and Kenneth Hillstom (1980), *User guide for MINPACK-1*, Argonne National Labs Report ANL-80-74, Argonne, Illinois.

### **Muller**

Muller, D.E. (1956), A method for solving algebraic equations using an automatic computer, *Mathematical Tables and Aids to Computation*, **10**, 208–215.

### **Murtagh**

Murtagh, Bruce A. (1981), *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York.

### **Murty**

Murty, Katta G. (1983), *Linear Programming*, John Wiley and Sons, New York.

### **Nelder and Mead**

Nelder, J.A., and R. Mead (1965), A simplex method for function minimization, *Computer Journal* **7**, 308–313.

### **Neter and Wasserman**

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

### **NVIDIA**

NVIDIA Corporation (©2005-2008), © All rights reserved. Portions of the NVIDIA **SGEMM** and **DGEMM** library routines were written by Vasily Volkov and are subject to the Modified Berkeley Software Distribution License. (©) 2007-09, Regents of the University of California.

### **Park and Miller**

Park, Stephen K., and Keith W. Miller (1988), Random number generators: good ones are hard to find, *Communications of the ACM*, **31**, 1192–1201.

## Parlett

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice–Hall, Inc., Englewood Cliffs, New Jersey.

## Patterson

Patterson, T.N.L. (1968), *The Optimum Addition Of Points To Quadrature Formulae*. *Mathematics of Comp*, **22**, 847-856.

## Pereyra

Pereyra, Victor (1978), PASVA3: An adaptive finite-difference FORTRAN program for first order nonlinear boundary value problems, in *Lecture Notes in Computer Science*, **76**, Springer-Verlag, Berlin, 67–88.

## Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

## Petzold

Petzold, L.R. (1982), A description of DASSL: A differential/ algebraic system solver, *Proceedings of the IMACS World Congress*, Montreal, Canada.

## Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. Uberhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

## Powell

Powell, M.J.D. (1977), Restart procedures for the conjugate gradient method, *Mathematical Programming*, **12**, 241–254.

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, in *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G.A. Watson), **630**, Springer-Verlag, Berlin, Germany, 144–157.

Powell, M.J.D. (1983), ZQPCVX a FORTRAN *subroutine for convex quadratic programming*, DAMTP Report NA17, Cambridge, England.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46-61.

Powell, M.J.D. (1988), *A tolerant algorithm for linearly constrained optimization calculations*, DAMTP Report NA17, University of Cambridge, England.

Powell, M.J.D. (1989), TOLMIN: *A fortran package for linearly constrained optimization calculations*, DAMTP Report NA2, University of Cambridge, England.

### **Pruess and Fulton**

Pruess, S. and C.T. Fulton (1993), Mathematical Software for Sturm-Liouville Problems, *ACM Transactions on Mathematical Software*, **17**, 3, 360–376.

### **Ralston**

Ralston, Anthony (1965), A First Course in Numerical Analysis, McGraw-Hill, New York.

### **Reinsch**

Reinsch, Christian H. (1967), Smoothing by spline functions, *Numerische Mathematik*, **10**, 177–183.

### **Rice**

Rice, J.R. (1983), *Numerical Methods, Software, and Analysis*, McGraw-Hill, New York.

### **Saad and Schultz**

Saad, Y., and M.H. Schultz (1986), GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.*, **7**, 856–869.

### **Schittkowski**

Schittkowski, K. (1987), *More test examples for nonlinear programming codes*, Springer-Verlag, Berlin, 74.

### **Schnabel**

Schnabel, Robert B. (1985), Finite Difference Derivatives – Theory and Practice, Report, National Bureau of Standards, Boulder, Colorado.

### **Schreiber and Van Loan**

Schreiber, R., and C. Van Loan (1989), A Storage–Efficient WY Representation for Products of Householder Transformations, *SIAM J. Sci. Stat. Comp.*, Vol. 10, No. 1, pp. 53-57, January (1989).

### **Scott et al.**

Scott, M.R., L.F. Shampine, and G.M. Wing (1969), Invariant Embedding and the Calculation of Eigenvalues for Sturm-Liouville Systems, *Computing*, **4**, 10–23.

### **Sewell**

Sewell, Granville (1982), *IMSL software for differential equations in one space variable*, IMSL Technical Report 8202, IMSL, Houston.

Sewell, Granville (2005), *Computational Methods of Linear Algebra*, section **4.6**, second edition, John Wiley & Sons, New York.

## **Shampine**

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179–180.

## **Shampine and Gear**

Shampine, L.F. and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1–17.

## **Sorensen**

Sorensen, D.C. (1992), *Implicit Application of Polynomial Filters in a K-step Arnoldi Method*, SIAM, J. Matrix Analysis and Applications, 13(1): 357-385.

## **Sincovec and Madsen**

Sincovec, R.F., and N.K. Madsen (1975), Software for nonlinear partial differential equations, *ACM Transactions on Mathematical Software*, **1**, #3, 232-260.

## **Singleton**

Singleton, R.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185–187.

## **Smith**

Smith, B.T. (1967), *ZERPOL, A Zero Finding Algorithm for Polynomials Using Laguerre's Method*, Department of Computer Science, University of Toronto.

## **Smith et al.**

Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler (1976), *Matrix Eigensystem Routines – EISPACK Guide*, Springer-Verlag, New York.

## **Spang**

Spang, III, H.A. (1962), A review of minimization techniques for non-linear functions, *SIAM Review*, **4**, 357–359.

## **Stewart**

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Stewart, G.W. (1976), The economical storage of plane rotations, *Numerische Mathematik*, **25**, 137–139.

## **Stoer**

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

## **Stroud and Secrest**

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

## **Titchmarsh**

Titchmarsh, E. Eigenfunction Expansions Associated with Second Order Differential Equations, Part I, 2d Ed., Oxford University Press, London, 1962.

## **Trench**

Trench, W.F. (1964), An algorithm for the inversion of finite Toeplitz matrices, *Journal of the Society for Industrial and Applied Mathematics*, **12**, 515–522.

## **Volkov**

Volkov, V. (2007-09), Portions of the NVIDIA **SGEMM** and **DGEMM** library routines were written by Vasily Volkov and are subject to the Modified Berkeley Software Distribution License as follows: Copyright (©) 2007-09, Regents of the University of California.

## **Walker**

Walker, H.F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM J. Sci. Stat. Comput.*, **9**, 152–163.

## **Washizu**

Washizu, K. (1968), *Variational Methods in Elasticity and Plasticity*, Pergamon Press, New York.

## **Watkins and Elsner**

Watkins, D.S., and L. Elsner (1990), Convergence of algorithms of decomposition type for the eigenvalue problem, *Linear Algebra and Applications* (to appear).

## **Weeks**

Weeks, W.T. (1966), Numerical inversion of Laplace transforms using Laguerre functions, *J. ACM*, **13**, 419–429.

## **Wilkinson**

Wilkinson, J.H., and Howinson, S., and Dewynne, J (1965), *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 635.

**Wilmot et al.**

Wilkinson, J.H. (1965), *The Mathematics of Financial Derivatives: A Student Introduction*, Cambridge University Press, NY, 41-57.



# Appendix D: Benchmarking or Timing Programs

---

## Scalar Program Descriptions

A set of benchmark programs is provided to allow the user to compare performance of certain routines with similar functionality. For example, the user may wish to compare the performance of `lin_sol_gen` with `LFTRG` and `LFSRG`. Since performance is dependent on problem size and platform, the user can run the `time_sol_gen` benchmark to determine which of these routines is likely to perform better with the user's specific configuration.

The benchmark programs are supplied with the product in the examples benchmark subdirectory and are summarized in [Table B](#). These programs call Fortran 90 array functions, in single and double precision, to compare the routines shown in columns A and B of [Table B](#). The main program reads single lines of input:

```
NSIZE NTRIES PREC "Description"
NSIZE NTRIES PREC "Description"
...
QUIT
```

The parameters of `NSIZE` and `NTRIES` appear in summary tables. The parameter `PREC` has values 1, 2 or 3. The choice depends on whether the user wants precision of single, double or both versions timed. The array functions return a summary table with these 6 values:

1. Average time
2. Standard deviation
3. Total time
4. `nsize`
5. `ntries`
6. Time Units/Sec.

As an example, the program `time_rand_gend` is compiled and linked with the single and double precision timing functions `s_rand_gen_bench` and `d_rand_gen_bench`.

The two lines of input are:

```
100000 5 3 "Random Number Benchmarks"
QUIT
```

This routine evaluates the elapsed time to compute 100,000 random numbers obtained with `rand_gen` and `rnun(drnun)`. The “Average” is the mean of the individual elapsed times for 5 calls to the routines, obtaining 100,000 random numbers in each call. The “St. Dev.” is the standard deviation for that “Average”. This value indicates the variability of the “Average”. In order for this value to provide any useful information it is necessary for `|NTRIES| > 1`. The value `|NTRIES| = 1` is acceptable, but only one time sample and no standard deviation is obtained. Values of `NTRIES > 0` result in the printing of results as shown in [Table A](#). The numbers in the table will vary depending on the machine and other factors that impact performance of Fortran codes.

| <b>Benchmark of <code>rand_gen</code> and <code>rnun</code>:</b> |            |            |                |
|------------------------------------------------------------------|------------|------------|----------------|
| <b>Date of benchmark, (Y, Mo, D, H, M, S): 2006 5 11 8 58 58</b> |            |            |                |
| 1                                                                | 3.6000E+00 | 3.2000E+00 | Average        |
| 2                                                                | 4.8990E-01 | 4.0000E-01 | St. Dev.       |
| 3                                                                | 1.8000E+01 | 1.6000E+01 | Total Ticks    |
| 4                                                                | 1.0000E+04 | 1.0000E+04 | Size           |
| 5                                                                | 5.0000E+00 | 5.0000E+00 | Repeats        |
| 6                                                                | 5.0000E+01 | 5.0000E+01 | Ticks per sec. |
| <b>Benchmark of <code>rand_gen</code> and <code>rnun</code>:</b> |            |            |                |
| <b>Date of benchmark, (Y, Mo, D, H, M, S): 2006 5 11 8 58 58</b> |            |            |                |
| 1                                                                | 2.8000E+00 | 3.2000E+00 | Average        |
| 2                                                                | 4.0000E-01 | 4.0000E-01 | St. Dev.       |
| 3                                                                | 1.4000E+01 | 1.6000E+01 | Total Ticks    |
| 4                                                                | 1.0000E+04 | 1.0000E+01 | Size           |
| 5                                                                | 5.0000E+00 | 5.0000E+00 | Repeats        |
| 6                                                                | 5.0000E+01 | 5.0000E+01 | Ticks per sec. |

*Table A: Benchmark Summary: `rand_gen`, `rnun`, (`drnun`)*

If `NTRIES < 0` the  $6 \times 2$  functions return the tabular values shown, with `|NTRIES|` samples. No printing is performed with `NTRIES < 0`.

To compute a related benchmark such as the rate “random numbers per second” for single precision `rand_gen`, separately calculate

```
rate = size × ticks per sec./average
= 104 × 50/3.6
= 138,889. numbers/sec.
= 0.139 million numbers/sec.
```

| Number | Program Units                                                       | Routines<br>Timed for Comparison |                                |
|--------|---------------------------------------------------------------------|----------------------------------|--------------------------------|
|        |                                                                     | A                                | B                              |
| 1      | time_dft.f90,<br>s_dft_bench.f90,<br>d_dft_bench.f90                | fast_dft                         | fftcf, fftcb<br>dfftcf, dfftcb |
| 2      | time_eig_gen.f90,<br>s_eig_gen_bench.f90,<br>d_eig_gen_bench.f90    | lin_eig_gen                      | e8crg, de8crg                  |
| 3      | time_eig_self.f90,<br>s_eig_self_bench.f90,<br>d_eig_self_bench.f90 | lin_eig_self                     | e5csf, de5csf                  |
| 4      | time_geig_gen.f90,<br>s_geig_gen_bench.f90,<br>d_geig_gen_bench.f90 | lin_geig_gen                     | g8crg, dg8crg                  |
| 5      | time_inv_chol.f90,<br>s_inv_chol_bench.f90,<br>d_inv_chol_bench.f90 | lin_sol_self                     | l2nds, dl2nds                  |
| 6      | time_inv_gen.f90,<br>s_inv_gen_bench.f90,<br>d_inv_gen_bench.f90    | lin_sol_gen                      | l2nrg, dl2nrg                  |
| 7      | time_inv_lsq.f90,<br>s_inv_lsq_bench.f90,<br>d_inv_lsq_bench.f90    | lin_sol_lsq                      | lsgrr, dlsgrr                  |
| 8      | time_inv_self.f90,<br>s_inv_self_bench.f90,<br>d_inv_self_bench.f90 | lin_sol_self                     | lftsf, lfssf<br>dlftsf, dlssf  |
| 9      | time_rand_gen.f90,<br>s_inv_rand_bench.f90,<br>d_inv_rand_bench.f90 | rand_gen                         | rnun, drnun                    |

*Table B: Scalar Benchmark Comparisons*

| Number | Program Units                                                       | Routines<br>Timed for Comparison |                                |
|--------|---------------------------------------------------------------------|----------------------------------|--------------------------------|
|        |                                                                     | A                                | B                              |
| 10     | time_sol_chol.f90,<br>s_inv_sol_chol.f90,<br>d_inv_sol_chol.f90     | lin_sol_self                     | lftds, lfds<br>dlftds, dlfsds  |
| 11     | time_sol_gen.f90,<br>s_sol_gen_bench.f90,<br>d_sol_gen_bench.f90    | lin_sol_gen                      | lftrg, lfsrg<br>dftrg, dlfsrg  |
| 12     | time_sol_lsq.f90,<br>s_sol_lsq_bench.f90,<br>d_sol_lsq_bench.f90    | lin_sol_lsq                      | l2rrv, dl2rrv                  |
| 13     | time_sol_self.f90,<br>s_sol_self_bench.f90,<br>d_sol_self_bench.f90 | lin_sol_self                     | lftsf, lfssf,<br>dlftsf, dlssf |
| 14     | time_svd.f90,<br>s_svd_bench.f90,<br>d_svd_bench.f90                | lin_svd                          | lsvrr, dlsvrr                  |
| 15     | time_tri.f90,<br>s_tri_bench.f90,<br>d_tri_bench.f90                | lin_sol_tri                      | lslcr, dlslcr                  |
| 16     | time_mult.f90<br>s_mult_bench.f90<br>d_mult_bench.f90               | A .x. B                          | matmul (D,E)                   |

Table B- continued: Scalar Benchmark Comparisons

Notes on the comparable problems:

1. Perform forward and backward DFT of a random complex sequence of size NSIZE.
2. Compute eigenexpansion of a random real matrix of dimension NSIZE × NSIZE.
3. Compute eigenexpansion of a random symmetric real matrix of dimension NSIZE × NSIZE.
4. Compute generalized eigenexpansion of a random matrix pencil of dimension NSIZE × NSIZE.
5. Compute the inverse of a positive definite real matrix of dimension NSIZE × NSIZE. Uses Cholesky method.
6. Compute the inverse of a general real random matrix of dimension NSIZE × NSIZE. Uses LU factorization.

7. Compute the generalized inverse of a general real random matrix of dimension  $(2 \times \text{NSIZE}) \times \text{NSIZE}$ . Uses QR factorization for `lin_sol_lsqr` and SVD for `LSGRR`.
8. Compute the inverse of a real, symmetric random matrix of dimension  $\text{NSIZE} \times \text{NSIZE}$ . Uses Aasen's decomposition for `lin_sol_self` and Bunch-Kaufman decomposition for `LFTSF`.
9. Generate `NSIZE` random numbers.
10. Solve a single system of linear equations with a positive definite real random matrix of dimension  $\text{NSIZE} \times \text{NSIZE}$ .
11. Solve a single system of linear equations with a general real random matrix of dimension  $\text{NSIZE} \times \text{NSIZE}$ .
12. Solve a single least-squares system of linear equations with a real random matrix of dimension  $(2 \times \text{NSIZE}) \times \text{NSIZE}$ .
13. Solve a single system of linear equations with a symmetric real random matrix of dimension  $\text{NSIZE} \times \text{NSIZE}$ .
14. Compute the full singular value decomposition of a general real random matrix of dimension  $\text{NSIZE} \times \text{NSIZE}$ .
15. Solve `NSIZE` systems of linear equations of a nonsymmetric  $\text{NSIZE} \times \text{NSIZE}$  tridiagonal matrix. Uses cyclic reduction.
16. Compute products of square matrices of size  $\text{NSIZE} \times \text{NSIZE}$ . Compare the IMSL defined operation `C = A .x. B` with `F = matmul(D,E)`. The arrays are assumed shape. Identical problems `A = D` and `B = E` are timed.
17. Compare times to use `SHOW()` for writing a random array of size `NSIZE` to a `CHARACTER` buffer vs. writing the same array to a scratch file.

---

## Parallel Program Descriptions

A set of parallel benchmark programs is shown in [Table D](#). These main programs call Fortran 90 box data type functions, in single and double precision. They compare our parallel allocation algorithm to a scalar sequential method. The main program reads single lines of input:

```
NSIZE NTRIES NRACKS PREC ROOT_WORKS "Description"
QUIT to Stop
```

Two initial lines of output echo the "Description" field, whether or not the root is working, and the number of processors in the MPI communicator. The parameters `NSIZE`, `NTRIES` and `NRACKS` appear in the summary tables. The parameter `PREC` has values 1, 2 or 3. The choice depends on whether the user wants precision of single, double or both versions timed. The array functions return a  $7 \times 2$  summary table of values. The (1:6, 1) and (1:6,2) elements of this array represent the results and parameters of the benchmark for the parallel and non-parallel versions. The (7,1) and

(7,2) elements of this array represent the ratio of the parallel to the scalar times and a first-order approximation to the variation in the ratio.

| <b>Parallel Box Version</b> | <b>Scalar Box Equivalent</b> |
|-----------------------------|------------------------------|
| 1. Average time             | Average time                 |
| 2. Standard deviation       | Standard deviation           |
| 3. Total Seconds            | Total Seconds                |
| 4. nsize                    | nsize                        |
| 5. nracks                   | nracks                       |
| 6. ntries                   | ntries                       |
| 7. Parallel/Scalar Ratio    | Variation in Ratio           |

As an example, the program `time_parallel_i` is compiled and linked with the single and double precision timing functions `s_parallel_i_bench` and `d_parallel_i_bench`.

This routine evaluates the time to compute 4 inverse matrices of size 600 by 600 using the defined operator `.i.` The “Average” is the mean of the individual elapsed times for 5 calls to the routines, obtaining 4 inverses in each call. The “St. Dev.” is the standard deviation for that “Average”. This value indicates the variability of the “Average”. In order for this value to provide any useful information it is necessary for  $|NTRIES| > 1$ . The value  $|NTRIES| = 1$  is acceptable, but only one time sample and no standard deviation is obtained. Values of  $NTRIES > 0$  result in the printing of results as shown in [Table C](#). The numbers in the table will vary depending on the machine and other factors that impact performance of Fortran codes. If  $NTRIES < 0$  the  $7 \times 2$  functions return the tabular values shown, with  $|NTRIES|$  samples. No printing is performed with  $NTRIES < 0$ .

| <b>Single precision benchmark of parallel .i. and non-parallel .i.:</b> |            |            |               |
|-------------------------------------------------------------------------|------------|------------|---------------|
| <b>Date of benchmark, (Y, Mo, D, H, M, S): 2006 5 11 8 58 58</b>        |            |            |               |
| 1                                                                       | 1.5815E+00 | 4.0241E+00 | Average       |
| 2                                                                       | 2.5031E-01 | 1.8035E-02 | St. Dev.      |
| 3                                                                       | 7.9077E+00 | 2.0121E+01 | Total Seconds |
| 4                                                                       | 5.0000E+01 | 5.0000E+01 | Size          |
| 5                                                                       | 5.0000E+00 | 5.0000E+00 | Racks per box |
| 6                                                                       | 5.0000E+00 | 5.0000E+00 | Repeats       |
| <b>Non-parallel/parallel averages and variation:</b>                    |            |            |               |
|                                                                         | 2.5444E+00 | 3.9129E-01 |               |

| <b>Double precision benchmark of parallel .i. and non-parallel .i.:</b> |            |            |               |
|-------------------------------------------------------------------------|------------|------------|---------------|
| <b>Date of benchmark, (Y, Mo, D, H, M, S): 2006 5 11 8 58 59</b>        |            |            |               |
| 1                                                                       | 1.6985D+00 | 4.0372D+00 | Average       |
| 2                                                                       | 9.8576D-01 | 2.3836D-02 | St. Dev.      |
| 3                                                                       | 8.4923D+00 | 2.0186D+01 | Total Seconds |
| 4                                                                       | 5.0000D+01 | 5.0000D+01 | Size          |
| 5                                                                       | 5.0000D+00 | 5.0000D+00 | Racks per box |
| 6                                                                       | 5.0000D+00 | 5.0000D+00 | Repeats       |
| <b>Non-parallel/parallel averages and variation:</b>                    |            |            |               |
|                                                                         | 2.3770D+00 | 1.2392D-01 |               |

Table C: Performance Summary: Box operator .i.

Below is a list of the performance evaluation programs that time the box data computations using parallel and non-parallel resources.

| Number | Program Units                                                                      | Function Timed |
|--------|------------------------------------------------------------------------------------|----------------|
| 1      | time_parallel_i.f90,<br>s_parallel_i_bench.f90,<br>d_parallel_i_bench.f90          | .i. A          |
| 2      | time_parallel_ix.f90,<br>s_parallel_ix_bench.f90,<br>d_parallel_ix_bench.f90       | A .ix. B       |
| 3      | time_parallel_xi.f90,<br>s_parallel_xi_bench.f90,<br>d_parallel_xi_bench.f90       | B .xi. A       |
| 4      | time_parallel_x.f90,<br>s_parallel_x_bench.f90,<br>d_parallel_x_bench.f90          | A .x. B        |
| 5      | time_parallel_tx.f90,<br>s_parallel_tx_bench.f90,<br>d_parallel_tx_bench.f90       | A .tx. B       |
| 6      | time_parallel_xt.f90,<br>s_parallel_xt_bench.f90,<br>d_parallel_xt_bench.f90       | A .xt. B       |
| 7      | time_parallel_hx.f90,<br>s_parallel_hx_bench.f90,<br>d_parallel_hx_bench.f90       | A .hx. B       |
| 8      | time_parallel_xh.f90,<br>s_parallel_xh_bench.f90,<br>d_parallel_xh_bench.f90       | A .xh. B       |
| 9      | time_parallel_chol.f90,<br>s_parallel_chol_bench.f90,<br>d_parallel_chol_bench.f90 | CHOL (A)       |
| 10     | time_parallel_cond.f90,<br>s_parallel_cond_bench.f90,<br>d_parallel_cond_bench.f90 | COND (A)       |
| 11     | time_parallel_rank.f90,<br>s_parallel_rank_bench.f90,<br>d_parallel_rank_bench.f90 | RANK (A)       |

| Number | Program Units                                                                      | Function Timed              |
|--------|------------------------------------------------------------------------------------|-----------------------------|
| 12     | time_parallel_det.f90,<br>s_parallel_det_bench.f90,<br>d_parallel_det_bench.f90    | DET (A)                     |
| 13     | time_parallel_orth.f90,<br>s_parallel_orth_bench.f90,<br>d_parallel_orth_bench.f90 | ORTH (A, R=R)               |
| 14     | time_parallel_svd.f90,<br>s_parallel_svd_bench.f90,<br>d_parallel_svd_bench.f90    | SVD (A, U=U, V=V)           |
| 15     | time_parallel_norm.f90,<br>s_parallel_norm_bench.f90,<br>d_parallel_norm_bench.f90 | NORM (A, TYPE=I)            |
| 16     | time_parallel_eig.f90,<br>s_parallel_eig_bench.f90,<br>d_parallel_eig_bench.f90    | EIG (A, W=W)                |
| 17     | time_parallel_fft.f90,<br>s_parallel_fft_bench.f90,<br>d_parallel_fft_bench.f90    | FFT_BOX (A)<br>IFFT_BOX (A) |

*Table D: Parallel and non-Parallel Box Comparisons*



# Product Support

---

## Contacting IMSL Support

Users within support warranty may contact Rogue Wave Software regarding the use of the IMSL Fortran Numerical Library. IMSL Support can consult on the following topics:

- Clarity of documentation
- Possible IMSL-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem

Not included in these topics are mathematical/statistical consulting and debugging of your program.

**Refer to the following for IMSL Product Support contact information:**

- <http://www.vni.com/tech/imsl/phone.php>.

The following describes the procedure for consultation with IMSL Support:

1. Include your IMSL license number
2. Include the product name and version number: IMSL Fortran Numerical Library Version 7.0
3. Include compiler and operating system version numbers
4. Include the name of the routine for which assistance is needed and a description of the problem



# Index

## 1

1-norm 1671, 1674, 1676, 1679

## 2

2DFT (Discrete Fourier Transform)  
1227, x

## 3

3DFT (Discrete Fourier Transform)  
x

## A

Aasen' s method 19, 20  
accuracy estimates of eigenvalues,  
example 541  
Adams i  
Adams method  
variable order 1072  
Adams-Moulton's method 1021  
adjoint eigenvectors, example 541  
adjoint matrix vi  
ainv= optional argument ix  
Akima interpolant 740  
algebraic-logarithmic singularities  
946  
ANSI i, 1770, 1771, xiii  
arguments, optional subprogram ix  
ARPACK  
Base Class, ARPACKBASE 651,  
653  
Object Oriented 653  
Type Extension 653  
array permutation 1842  
ASCII collating sequence 1870  
ASCII values 1867, 1868, 1869  
Avoiding Data Copy to Board 1602

## B

backward difference formulas 1062

band Hermitian storage mode 346,  
349, 355, 358, 360, 362, 365,  
1947  
band storage mode 282, 285, 290,  
298, 301, 327, 330, 333, 341,  
344, 1617, 1619, 1621, 1622,  
1624, 1626, 1631, 1637, 1659,  
1663, 1665, 1667, 1674, 1676,  
1945  
band symmetric storage mode 303,  
305, 311, 314, 316, 318, 320,  
322, 324, 327, 330, 333, 336,  
339, 341, 344, 346, 349, 352,  
355, 358, 360, 362, 365, 366,  
372, 377, 580, 582, 585, 588,  
591, 593, 596, 1635, 1946  
band triangular storage mode 1948  
Basic Linear Algebra Subprograms  
1576  
basis functions 865  
bidiagonal matrix 60  
bilinear form 1653  
Black-Scholes Equation  
American Put Pricing 1145  
Cash-or-Nothing Payoff, A Bet  
1150  
Convertible Bond Pricing 1154  
Greeks, Delta, Gamma, and Theta,  
Feynman-Kac 1145  
Vertical Spread Payoff 1150  
*BLACS* 1789  
BLAS 1576, 1577, 1588, 1589, 1590  
Level 1 1576, 1577  
Level 2 1588, 1589  
Level 3 1588, 1589, 1590  
block-cyclic decomposition  
reading, writing utility 1790  
boundary conditions 1037  
boundary value problem 53  
Brenan 54  
Broyden's update 1320  
B-spline coefficients 761, 870, 879  
B-spline representation 782, 783,  
786, 789, 821  
B-splines 695

## C

Campbell 54  
Cauchy principal value 916, 949  
central differences 1536  
changing messages 1812  
character arguments 1868  
character sequence 1872  
character string 1873  
character workspace 1956  
Chasing a Board 1606

- Chebyshev approximation 699, 910
- Chebyshev polynomials 31
- Cholesky
  - algorithm 21
  - decomposition 18, 532, 545
  - factorization 1743, 1744
  - method 22
- Cholesky decomposition 494
- Cholesky factorization 186, 191, 195, 204, 308, 311, 314, 320, 352, 365, 399, 403, 408, 416, 421, 424, 496, 499
- circulant linear system 428
- circulant matrices 8
- classical weight functions 979, 991
- codiagonal band hermitian storage mode 352
- codiagonal band Hermitian storage mode 1950
- codiagonal band symmetric storage mode 308, 1949
- coefficient matrix 296, 316, 339, 360, 377, 380, 385, 391, 394, 399, 403, 408, 412, 421, 424, 426, 428, 431, 437, 440, 451, 457, 463, 467, 471, 478, 483, 489, 494, 496, 503, 510, 514
- coefficients 1261, 1268
- column pivoting 494
- companion matrix 538
- Complex Eigenvectors, real matrices 679, 680
- complex function 1309, 1311
- complex periodic sequence 1245, 1248
- complex sparse Hermitian positive definite system 412, 421, 424
- complex sparse system 380, 391
- complex triangular system 165
- complex tridiagonal system 322
- complex vectors 1294, 1304
- computing
  - eigenvalues, example 528
  - the rank of A 36
  - the SVD 60
- computing eigenvalues, example 537
- Computing Initial Derivatives for DAE Systems 1063, 1066
- condition number 158, 169, 541
- conjugate gradient algorithm 1395, 1399
- conjugate gradient method 431, 437
- Constant elasticity of variance, CEV 1139
- Constraints
  - after Index Reduction 1060, 1066, 1069
  - Conservation Principles 1063, 1069
  - continuous Fourier transform 1219
  - continuous function 910
  - convolution 1289, 1294, 1683, 1685
  - convolutions, real or complex
    - periodic sequences 1226
  - coordinate transformation 483
  - correlation 1299, 1304
  - cosine 938
  - cosine Fourier coefficients 1270
  - cosine Fourier transform 1268
  - covariance matrix 22, 27, 29
  - CPU time 1874
  - crossvalidation 907
  - cross-validation with weighting, example 64
  - cubic spline 749, 750, 753, 756
  - cubic spline approximation 904, 907
  - cubic spline interpolant 727, 729, 732, 737, 740, 742, 746
  - cubic splines 697
  - cyclic reduction 45, 47
  - cyclic reduction algorithm 324
  - cyclical 2D data, linear trend 1230
  - cyclical data, linear trend 1223

**D**

  - DAE
    - Index of DAE System 1063
    - Reducing the Index 1063
  - DAE Solver 1057
  - DASPG
    - deprecated routine 1072
  - DASPG routine 54
  - data fitting
    - polynomial 30
    - two dimensional 33
  - data points 858
  - data, optional ix
  - date 1876, 1877, 1878, 1880
  - decomposition, singular value 36, xvii
  - degree of accuracy 1931
  - DENSE\_LP 1488
  - deprecated routines 1957
  - Deprecated Routines
    - DASPG 1072
    - MOLCH 1128
  - determinant 1751, viii
  - determinant of A 10
  - determinants 113, 148, 162, 163, 204, 226, 276, 301, 320, 344, 365
  - determinants 7

DFT (Discrete Fourier Transform)  
 1220, 1233  
 differential algebraic equations 1001  
 Differential Algebraic Equations 547  
 differential equations 1000, 1037  
 differential-algebraic equations 1057  
 differential-algebraic solver 54, 1057  
 diffusion equation 53  
 direct- access message file 1812  
 direct search complex algorithm  
 1448  
 direct search polytope algorithm  
 1403  
 discrete Fourier cosine  
 transformation 1257  
 discrete Fourier sine transformation  
 1253  
 discrete Fourier transform 1219,  
 1761, 1762, 1763, 1764, 1766,  
 1767, 1768, x, xii  
 inverse 1765, xii  
 dot product 1581, 1582  
 double precision i, 1687  
 DOUBLE PRECISION types v

## E

efficient solution method 539  
 eigensystem  
 complex 562, 634, 637, 640  
 Hermitian 615  
 real 555, 578, 626, 629, 632  
 symmetric 596, 648  
 eigenvalue 1755, 1757, viii  
 Eigenvalue problem  
 eigenvectors 651  
 generalized complex matrix 687  
 generalized real matrix 678  
 generalized symmetric 651, 655  
 standard complex matrix 687  
 standard real matrix 678  
 standard symmetric 651, 655  
 eigenvalue-eigenvector  
 decomposition 528, 532, 1755,  
 1757, viii  
 expansion (eigenexpansion) 529  
 eigenvalues 451, 549, 552, 557, 559,  
 564, 566, 568, 570, 573, 575,  
 580, 582, 585, 588, 591, 593,  
 598, 601, 604, 606, 609, 612,  
 617, 619, 621, 623, 626, 629,  
 634, 637, 642, 645, 654, 676,  
 685  
 eigenvalues, self-adjoint matrix 24,  
 526, 533, xvii  
 eigenvectors 51, 526, 530, 532, 533,  
 552, 559, 566, 570, 575, 582,  
 588, 593, 601, 606, 612, 619,

623, 629, 637, 645, 654, 676,  
 685  
 EISPACK xv  
 endpoint singularities 918  
 equality constraint, least squares 35  
 error detection 900  
 error handling xi, 1934  
 errors 1931, 1932, 1933  
 alert 1932  
 detection 1931  
 fatal 1932  
 informational 1931  
 multiple 1931  
 note 1932  
 printing error messages 1810  
 severity 1931  
 terminal 1931, 1932  
 warning 1932  
 Euclidean (2-norm) distance 1677  
 Euclidean length 1783  
 even sequence 1257  
 example  
 least-squares, by rows  
 distributed 72  
 linear constraints  
 distributed 77  
 linear inequalities  
 distributed 69  
 linear system  
 distributed, ScaLAPACK 1803,  
 1808  
 matrix product  
 distributed, PBLAS 1801  
 Newton's Method  
 distributed 80  
 transposing matrix  
 distributed 1798  
 Example  
 complex eigenvectors, complex  
 matrices 689  
 complex eigenvectors, real  
 matrices 680  
 Example  
 generalized symmetric matrix  
 shift and invert 664  
 real matrix  
 shift and invert 680  
 type extensions 680  
 symmetric matrix  
 matrix products 657  
 shift and invert 661  
 type extensions 661, 664  
 Example  
 complex matrix  
 type extensions 689  
 Example  
 complex matrix  
 products 689

- examples
  - accuracy estimates of eigenvalues 541
  - accurate least-squares solution with iterative refinement 25
  - analysis and reduction of a generalized eigensystem 532
  - complex polynomial equation Roots 538
  - computing eigenvalues 528, 537
  - computing eigenvectors with inverse iteration 530
  - computing generalized eigenvalues 545
  - computing the SVD 60
  - constraining a spline surface to be non-negative interpolation to data 725
  - constraining points using spline surface 723
  - convolution with Fourier Transform 1226
  - cross-validation with weighting 64
  - cyclical 2D data with a linear trend 1230
  - cyclical data with a linear trend 1223
  - eigenvalue-eigenvector expansion of a square matrix 529
  - evaluating the matrix exponential 15, 16
  - Generalized Singular Value Decomposition 62
  - generating strategy with a histogram 1887
  - generating with a Cosine distribution 1889
  - internal write of an array 1816
  - iterative refinement and use of partial pivoting 49
  - Laplace transform solution 42
  - larger data uncertainty 548
  - least squares with an equality constraint 35
  - least-squares solution of a rectangular system 39
  - linear least squares with a quadratic constraint 61
  - matrix inversion and determinant 14
  - natural cubic spline interpolation to data 705
  - parametric representation of a sphere 721
  - periodic curves 712
  - polar decomposition of a square matrix 39
  - printing an array 1815
  - reduction of an array of black and white 41
  - ridge regression 64
  - running mean and variance 1885
  - seeding, using, and restoring the generator 1886
  - selected eigenvectors of tridiagonal matrices 51
  - self-adjoint, positive definite generalized eigenvalue problem 546
  - several 2D transforms with initialization 1232
  - several transforms with initialization 1225
  - shaping a curve and its derivatives 707
  - solution of multiple tridiagonal systems 47
  - solving a linear least squares system of equations 21, 30
  - solving a linear system of equations 13
  - solving parametric linear systems with scalar change 539
  - sort and final move with a permutation 1848
  - sorting an array 1847
  - splines model a random number generator 709
  - system solving with Cholesky method 22
  - system solving with the generalized inverse 32
  - tensor product spline fitting of data 719
  - test for a regular matrix pencil 547
  - transforming array of random complex numbers 1223, 1230, 1235
  - tridiagonal matrix solving 53
  - two-dimensional data fitting 33
  - using inverse iteration for an eigenvector 24
- Examples
  - Linear ODE
    - User-Defined Linear Solver
- Constraints 1069
  - Swinging Pendulum
    - Constraints
- Index 1 System 1066
  - exclusive OR 1884
  - Expanded Matrix 670
  - extended precision arithmetic 1687

## F

factored secant update 1344, 1350  
factorization, LU 10  
Fast Fourier Transforms 1218  
Faure 1904, 1906, xli, x  
Faure sequence 1904, 1905, xli, x  
Fejer quadrature rule 991  
Feynman-Kac Differential Equation  
  Forcing or Source Term, Feynman-  
  Kac 1145  
FFT (Fast Fourier Transform) 1222,  
  1229, 1235  
finite difference gradient 1522  
finite-difference approximation  
  1337, 1344  
finite-difference gradient 1372,  
  1395, 1420  
finite-difference Hessian 1384  
finite-difference Jacobian 1407  
first derivative 995  
first derivative evaluations 1365  
FORTRAN 77  
  combining with Fortran 90 i  
Fortran 90  
  language i  
  rank-2 array ix  
  real-time clock 1884  
forward differences 1538, 1541,  
  1543, 1546, 1555  
Fourier coefficients 1236, 1240,  
  1245, 1248, 1274, 1281  
Fourier integral 942  
Fourier transform 1277, 1285  
Frobenius norm 1673  
full storage mode 1626  
Fushimi 1883, 1885

## G

Galerkin principle 54  
Gauss quadrature 917  
Gauss quadrature rule 979, 983  
Gaussian elimination 366, 372, 377,  
  380, 394, 412, 416  
Gauss-Kronrod rules 922  
Gauss-Lobatto quadrature rule 979,  
  983  
Gauss-Radau quadrature rule 979,  
  983  
Gear's BDF method 1021  
generalized  
  eigenvalue 532, 545, 1755, 1757,  
  viii  
  feedback shift register (GFSR)  
  1883  
  inverse  
  matrix 27, 28, 32

generalized inverse  
  system solving 32  
generator 1886, 1889  
getting started viii  
GFSR algorithm 1884  
Givens plane rotation 1585  
Givens transformations 1586, 1587  
globally adaptive scheme 922  
Golub 13, 21, 30, 35, 60, 62, 65, 528,  
  532, 537  
gradient 1536, 1538, 1543, 1558  
Gray code 1907  
GSVD 62

## H

Hadamard product 1583, 1651  
Hanson 528  
harmonic series 1223, 1230  
Helmholtz's equation 1188  
Helmholtz's equation 1194  
Hermite interpolant 737  
Hermite polynomials 1115  
Hermitian positive definite system  
  227, 232, 248, 252, 346, 349,  
  360, 362  
Hermitian system 259, 262, 271, 273  
Hessenberg matrix, upper 534, 538  
Hessian 1389, 1434, 1441, 1541,  
  1543, 1561  
High Performance Fortran  
  HPF 1789  
histogram 1887  
Horner's scheme 1656  
Householder 544  
Householder transformations 457,  
  471  
hyper-rectangle 973

## I

IEEE 1770, 1771, xiii  
Index of DAE System 1063  
infinite eigenvalues 545  
infinite interval 935  
infinity norm 1670  
infinity norm distance 1681  
informational errors 1931  
initialization, several 2D transforms  
  1232  
initialization, several transforms  
  1225  
initial-value problem 1003, 1011,  
  1021, 1072  
integer options 1908  
INTEGER types v  
integrals 756

integration 918, 922, 925, 929, 935,  
 938, 946, 949, 953, 960, 966,  
 973  
 interface block i  
 internal write 1816  
 interpolation 701  
   cubic spline 727, 729  
   quadratic 699  
   scattered data 699  
 inverse 10  
   iteration, computing eigenvectors  
     24, 51, 530  
   matrix ix, 11, 18, 19, 22  
     generalized 27, 28  
     transform 1221, 1228, 1233  
 inverse matrix 10  
 isNaN 1770, 1771  
 ISO i  
 iterated integral 955  
 Iterative Method 668  
 Iterative Method 451  
 iterative refinement ix, 6, 7, 49, 83,  
   108, 143, 177, 181, 186, 191,  
   195, 199, 204, 206, 210, 213,  
   223, 252, 273, 298, 318, 341,  
   346, 362, 451, 463  
 IVOAM  
   initial-value problem 1072  
 IVPAG routine 54

**J**

Jacobian 1320, 1337, 1340, 1344,  
   1350, 1413, 1452, 1459, 1546,  
   1555, 1565  
 Jenkins-Traub three-stage algorithm  
   1322

**K**

Kershaw 47

**L**

Laguerre's method 1320  
 LAPACK xv, 550, 553, 558, 560,  
   565, 567, 599, 602, 627, 630,  
   635  
 Laplace transform 1309, 1311  
 Laplace transform solution 42  
 larger data uncertainty, example 548  
 LDU factorization 324  
 least squares 21, 27, 33, 35, 36, 42,  
   43, 699, 858, 861, 879, 1223,  
   1230, xvii  
 least-squares approximation 865, 874  
 least-squares problem 483

least-squares solution 457  
 Lebesgue measure 1906  
 Left and right singular vectors 668,  
   671  
 Level 1 BLAS 1576, 1577  
 Level 2 BLAS 1588, 1589  
 NVIDIA  
   SGBMV, DGBMV, CGBMV,  
   ZGBMV 1590, 1592  
   SGEMV, DGEMV, CGEMV,  
   ZGEMV 1590, 1592  
   SGER, DGER 1591, 1595  
   SSYR, DSYR 1591, 1597  
 Level 3 BLAS 1588, 1589, 1590  
 NVIDIA  
   CHEMM, ZHEMM 1598  
   CHERK, ZHERK 1599  
   SGEMM, DGEMM, CGEMM,  
   ZGEMM 1598  
   SSYMM, DSYMM, CSYMM,  
   ZSYMM 1598  
   SSYR2K, DSYR2K 1599  
   SSYRK, DSYRK, CSYRK,  
   ZSYRK 1598  
   STRMM, DTRMM, CTRMM,  
   ZTRMM 1600  
   STRSM, DTRSM, CTRSM,  
   ZTRSM 1600  
 Levenberg-Marquardt algorithm  
   1358, 1407, 1413, 1452, 1459  
 library subprograms v  
 linear algebraic equations 366, 394  
 linear constraints 467  
 linear equality/inequality constraints  
   1510, 1516  
 linear equations 18  
   solving 83, 87, 103, 119, 123, 138,  
   165, 177, 181, 195, 199, 210,  
   213, 221, 223, 227, 232, 248,  
   252, 259, 262, 271, 273, 278,  
   282, 285, 296, 298, 303, 305,  
   316, 318, 322, 341, 346, 349,  
   360, 362, 377, 380, 391, 394,  
   408, 412, 421, 424, 431  
 linear least-squares problem 451,  
   463, 467  
 linear least-squares with non-  
   negativity constraints 67, 68,  
   69, 76  
 linear programming problem 1488,  
   1494, 1497  
 linear solutions  
   packaged options 11  
 linear trend, cyclical 2D data 1230  
 linear trend, cyclical data 1223

LINPACK xv, 550, 553, 558, 560,  
565, 567, 599, 602, 627, 630,  
635  
low-discrepancy 1907  
LU factorization 93, 99, 103, 113,  
128, 134, 138, 148, 290, 293,  
296, 301, 333, 336, 339, 344,  
372, 377, 385, 391  
LU factorization of A 10, 11, 18,  
1692

## M

machine-dependent constants 1937  
mathematical constants 1919  
matrices 1615, 1616, 1617, 1619,  
1621, 1622, 1624, 1626, 1627,  
1629, 1631, 1635, 1637, 1639,  
1647, 1649, 1657, 1659, 1661,  
1667, 1673, 1674, 1676, 1817,  
1819, 1822, 1825, 1827, 1830,  
1833  
adjoint vi  
complex 333, 336, 344, 510, 557,  
559, 1626, 1631  
band 1619, 1663, 1667, 1676  
general 128, 148, 150, 1616,  
1624, 1627  
general sparse 385  
Hermitian 237, 265, 268, 276,  
352, 355, 358, 365, 598, 601,  
604, 606, 609, 612, 1634, 1637  
rectangular 1629, 1649, 1661,  
1827, 1830  
tridiagonal 324  
upper Hessenberg 621, 623  
copying 1615, 1616, 1617, 1619,  
1627, 1629, 1635, 1637  
covariance 22, 27, 29  
general 1943  
Hermitian 1944  
inverse ix, 10, 11, 18, 19, 22  
generalized 27, 28, 32  
inversion and determinant 14  
multiplying 1645, 1647, 1649,  
1657, 1659, 1661  
orthogonal vi  
permutation 1844  
poorly conditioned 39  
printing 1817, 1819, 1822, 1825,  
1827, 1830, 1833  
real 290, 293, 301, 514, 549, 552,  
1622, 1631  
band 1617, 1659, 1674  
general 93, 99, 113, 115, 1615,  
1621, 1627  
general sparse 372

rectangular 1629, 1647, 1651,  
1657, 1673, 1817, 1819  
sparse 6  
symmetric 186, 191, 204, 206,  
215, 218, 226, 308, 311, 314,  
320, 496, 499, 564, 566, 568,  
570, 573, 575, 580, 582, 585,  
588, 591, 593, 1632, 1635  
tridiagonal 279  
upper Hessenberg 617, 619  
rectangular 1639, 1943  
sparse  
Hermitian 416  
symmetric 399  
symmetrical 403  
symmetric 494, 1944  
transposing 1639, 1641, 1643  
triangular 1944  
unitary vi  
upper Hessenberg 538  
matrix  
inversion 7  
types 5  
matrix pencil 545, 547  
matrix permutation 1844  
matrix storage modes 1943  
matrix/vector operations 1614  
matrix-matrix multiply 1598, 1600  
matrix-matrix solve 1600  
matrix-vector multiply 1592, 1593,  
1594, 1595  
Matrix-Vector Operations 671  
means 1885  
Mersenne Twister 1894, 1895, 1897,  
1898, 1900  
message file  
building new direct-access  
message file 1812  
changing messages 1812  
management 1811  
private message files 1813  
Metcalfe i  
method of lines 54, 1115  
minimization 1358, 1359, 1360,  
1362, 1365, 1369, 1372, 1377,  
1384, 1389, 1395, 1399, 1403,  
1420, 1427, 1434, 1441, 1448,  
1452, 1488, 1494, 1510, 1516,  
1522, 1528, 1536, 1538, 1541,  
1543, 1546, 1555, 1558, 1561,  
1565, 1569  
minimum degree ordering 399  
minimum point 1362, 1365, 1369  
mistake  
missing argument 1792  
Type, Kind or Rank  
TKR 1792

Modified Gram-Schmidt algorithm  
1775  
modified Powell hybrid algorithm  
1337, 1340  
MOLCH  
  deprecated routine 1128  
monic polynomials 986, 988  
Moore-Penrose 1728, 1729, 1731,  
  1740, 1741  
MPI 1698, 1927  
  parallelism xix  
Muller's method 1320, 1325  
multiple right sides 7  
multivariate functions 1358  
multivariate quadrature 917

## N

naming conventions iv  
NaN (Not a Number) 1770  
  quiet 1770  
  signaling 1770  
Newton algorithm 1358  
Newton method 1384, 1389, 1434,  
  1441  
Newton's method 42, 61  
noisy data 904, 907  
nonadaptive rule 953  
nonlinear equations 1337, 1340,  
  1344, 1350  
nonlinear least-squares problem  
  1358, 1407, 1413, 1452, 1459,  
  1466  
nonlinear programming 1522, 1528  
norm 1771  
Normal Matrix 670  
normalize 1783  
not-a-knot condition 727, 729  
Notes for BLAS Using NVIDIA  
  1576, 1601  
Number of Installed  
  Boards 1606  
numerical differentiation 918  
NVIDIA  
  Avoiding Data Copy to Board  
  1602  
  Chasing a Board 1606  
  Increased Performance 1576, 1601  
  Number of Installed  
  Boards 1606  
  Switching from Fortran to  
  NVIDIA BLAS 1602  
  Switchover Size 1601

## O

odd sequence 1253

odd wave numbers 1261, 1263,  
  1268, 1270  
optional argument ix  
optional data viii, ix  
optional subprogram arguments ix  
order one or two  
  system of ordinary differential  
  equations 1072  
ordinary differential equations 1000,  
  1003, 1011, 1021  
ordinary eigenvectors, example 541  
orthogonal  
  decomposition 60  
  factorization 30  
  matrix vi  
orthogonal matrix 478  
orthogonalized 51, 530  
overflow vi

## P

packed hermitian matrix storage  
  mode 1953  
packed symmetric matrix storage  
  mode 1953  
packed triangular matrix storage  
  mode 1953  
page length 1840  
page width 1840  
parameters 1243, 1251, 1255, 1259,  
  1266, 1272  
parametric linear systems with scalar  
  change 539  
parametric systems 539  
partial differential equations 1000,  
  1001, 1115  
Partial Expansion 671, 672  
partial pivoting 45, 47  
*PBLAS* 1789  
performance index 555, 562, 578,  
  596, 615, 632, 640, 648  
periodic boundary conditions 746  
permutation 1848  
Petzold 54  
physical constants 1919  
piecewise polynomial 695, 821, 823,  
  825, 828, 831  
piecewise-linear Galerkin 54  
pivoting  
  partial 10, 12, 19  
  row and column 27, 30  
  symmetric 18  
plane rotation 1586  
plots 1914  
Poisson solver 1188, 1194  
Poisson's equation 1188, 1194  
polar decomposition 39, 49

polynomial 1655  
     interpolation 855, xxvii  
 polynomial curve 861  
 prime factors 1917  
 printing 1840, 1914, 1933  
 printing an array, example 1815  
 printing arrays 1813  
 printing results xii  
 private message files 1813  
 programming conventions vi  
 pseudorandom number generators  
     1893  
 pseudorandom numbers 1900, 1902  
 PV-WAVE 1091

## Q

QR algorithm 60, 528  
     double-shifted 536  
*QR* decomposition 9, 471, 1751  
*QR* factorization 478, 489  
 quadratic interpolation 833, 835,  
     838, 840, 843, 847  
 quadratic polynomial interpolation  
     699  
 quadrature formulas 917  
 quadrature rule 988  
 quadruple precision 1687  
 quasi-Monte Carlo 976  
 quasi-Newton method 1372, 1377,  
     1420, 1427  
 quintic polynomial 851

## R

radial-basis functions 33  
 random complex numbers,  
     transforming an array 1223,  
     1230, 1235  
 random number generator 1895,  
     1896, 1897, 1898, 1899, 1900  
 random number generators 1891,  
     1892  
 random numbers 1882  
 rank-2k update 1599  
 rank-k update 1598, 1599  
 rank-one matrix 489, 496, 499  
 rank-one matrix update 1595, 1596,  
     1597  
 rank-two matrix update 1596, 1597  
 rational weighted Chebyshev  
     approximation 910  
 READ\_MPS 1475, 1485  
 real numbers, sorting 1846  
 real periodic sequence 1236, 1240  
 real sparse symmetric positive  
     definite system 408

real symmetric definite linear system  
     431, 437  
 real symmetric positive definite  
     system 177, 181, 195, 199,  
     303, 305, 316, 318  
 real symmetric system 210, 213, 221,  
     223  
 real triangular system 154  
 real tridiagonal system 278  
 REAL types v  
 real vectors 1289, 1299  
 record keys, sorting 1848  
 rectangular domain 801  
 rectangular grid 838, 840, 843, 847  
 recurrence coefficients 983, 986, 988  
 Reducing the Index 1063  
 reduction  
     array of black and white 41  
 References  
     Parabolic PDE  
         Banded Linear System 1063  
 regularizing term 47  
 Reid i  
 required arguments viii, ix  
 reserved names 1954  
 reverse communication 54  
 revised simplex method 1359, 1505  
 ridge regression 64  
     cross-validation  
     example 64  
 Rodrigue 47  
 row and column pivoting 27, 30  
 row vector, heavily weighted 35  
 Runge-Kutta-order method 1011  
 Runge-Kutta-Verner fifth-order  
     method 1003  
 Runge-Kutta-Verner sixth-order  
     method 1003

## S

*ScaLAPACK*  
     contents 1790  
     data types 1790  
     definition of library 1789  
     interface modules 1792  
     reading utility  
         block-cyclic distributions 1795,  
         1805, 1807  
 scattered data 851  
 scattered data interpolation 699  
 Schur form 533, 539  
 search 1860, 1862, 1864  
 second derivative 995  
 self-adjoint  
     eigenvalue problem 532  
     linear system 25  
     matrix 18, 21, 528, 529, 532, xvii

- eigenvalues 24, 526, 533, xvii
- tridiagonal 20
- semi-infinite interval 935
- sequence 1263, 1270
- serial number 1881
- simplex algorithm 1494, 1497
- sine 938
- sine Fourier coefficients 1263
- sine Fourier transform 1261
- single precision i
- SINGLE PRECISION options 1911
- Single Program, Multiple Data
  - SPMD 1789
- singular value decomposition 510
- Singular Value Decomposition
  - ARPACK 451, 526, 651, 668
- singular value decomposition (SVD)
  - 36, 1780, 1781, xvii
- singularity 8
- singularity points 925, 929, 960, 966
- smooth bivariate interpolant 851
- smoothing 900
- smoothing formulas 32
- smoothing spline routines 699
- solvable 547
- solving
  - general system 10
  - linear equations 18
  - rectangular
- least squares 36
- system 27
  - solving linear equations 5
  - sorting 1849, 1850, 1852, 1853, 1855, 1856, 1857, 1859, 1860, 1862, 1864
  - sorting an array, example 1847
  - Sparse <atrix, Complex
    - Harwell-Boeing column-oriented sparse form 1703
  - sparse linear programming 1497
  - Sparse Matrix Computations, Examples
    - Plane Poisson Problem with Dirichlet Boundary Conditions 1733
  - sparse matrix storage mode 1952
  - Sparse Matrix, Complex 1700, 1702
    - Accumulate entries of sparse matrix 1705
    - Collection of Triplets 1702, 1705
    - Compressed Sparse Column Format 1703
    - Conversion of Triplets to Harwell-Boeing form 1705
    - Derived types for sparse matrices 1702
    - Triplets types for sparse matrices 1702
  - Sparse Matrix, Real 1700
  - sparse system 366, 377
  - spline approximation 870, 879
  - spline interpolant 761, 771
  - spline knot sequence 765, 768
  - splines 699, 758, 782, 783, 786, 789
    - cubic 697
    - tensor product 698
  - square matrices
    - eigenvalue-eigenvector expansion 529
    - polar decomposition 39, 49
  - square root 1925
  - Stiff Solver 1057
  - Sturm-Liouville problem 1201, 1213
  - subprograms
    - library v
      - optional arguments ix
  - SVD 57, 62, xvii
  - SVD Example
    - Expanded Matrix 672
    - Normal Matrix 672
    - Partial Expansion 672
    - Type Extension 672
  - SVRGN 1847
  - Switching from Fortran to NVIDIA BLAS
    - set value 1602
  - Switchover Size 1601
  - symmetric Markowitz strategy* 373
  - Symmetric Matrix
    - eigenvectors 655
  - system of ordinary differential equations
    - order one or two 1072

## T

- tensor product splines 698
- tensor-product B-spline coefficients 771, 776, 889, 894
- tensor-product B-spline representation 792, 794, 797, 801, 805, 807, 811, 817
- tensor-product spline 792, 794, 797, 801, 805, 807, 811, 817
- tensor-product spline approximant 889, 894
- tensor-product spline interpolant 776
- terminal errors 1931
- third derivative 995
- time 1875
- Timing
  - Benchmarking
    - list, parallel codes* ix

list, scalar version iv  
parallel version i, v  
Toeplitz linear system 426  
Toeplitz matrices 8  
traceback 1936  
transfer 1771  
transportation problem 1504  
transpose 1723, 1724, 1726  
tridiagonal 45  
  matrix 47  
  matrix solving, example 53  
triple inner product 1583  
two-dimensional data fitting 33

## U

unconstrained minimization 1358  
underflow vi  
uniform (0, 1) distribution 1900,  
  1902  
uniform mesh 1194  
unitary matrix vi  
univariate functions 1358  
univariate quadrature 916  
upper Hessenberg matrix 538  
user errors 1931  
user interface i  
User-Defined Linear Solver 1061,  
  1069  
user-supplied function 995  
user-supplied gradient 1399, 1427,  
  1528  
Using LAPACK, LINPACK, and  
  EISPACK xv  
using library subprograms v

## V

Van Loan 13, 21, 30, 35, 60, 62, 65,  
  528, 532, 537  
variable knot B-spline 874  
variable order 1037  
variable order Adams method 1072  
variances 1885  
variational equation 54  
vectors 1579, 1580, 1581, 1583,  
  1584, 1592, 1661, 1663, 1683,  
  1685  
  complex 1685  
  real 1683  
version 1881

## W

workspace allocation 1955  
*World Wide Web*

*URL for ScaLAPACK User's  
Guide 1790*

## Z

zero of a real function 1331  
zero of a real univariate function  
  1328  
zeros of a polynomial 1320, 1322,  
  1324  
zeros of a univariate complex  
  function 1325  
zeros of the polynomial 1319