IMSL®

IMSL Fortran Library User's Guide MATH/LIBRARY Volume 1 of 2

Mathematical Functions in Fortran

Trusted For Over 30 Years

IMSL®

IMSL Fortran Library User's Guide MATH/LIBRARY Volume 1 of 2

Mathematical Functions in Fortran

Visual Numerics

P/N 7693

[w w w . v n i . c o m]

Visual Numerics, Inc. – United States
Corporate Headquarters
2000 Crow Canyon Place, Suite 270
San Ramon, CA 94583
PHONE: 925-807-0138
FAX: 925-807-0145
e-mail: info@vni.com
Westminster, CO
PHONE: 303-379-3040

Houston, TX
PHONE: 713-784-3131

Visual Numerics International Ltd.
Sussex House
6 The Forbury
Reading, Berkshire RGI 3EJ
UNITED KINGDOM

PHONE: +44-1-189 25-3370
FAX: +44 –1-189-25-3371
e-mail: info@vniuk.co.uk
Support: support@vniuk.co.uk

Visual Numerics SARL
Immeuble le Wilson 1
70, avenue due General de Gaulle
F-92058 PARIS LA DEFENSE, Cedex
FRANCE

PHONE: +33-1-46-93-94-20
FAX: +33-1-46-93-94-39
e-mail: info@vni.paris.fr

Visual Numerics S. A. de C. V.
Florencia 57 Piso 10-01
Col. Juarez
Mexico D. F. C. P. 06600
MEXICO

PHONE: +52-55-514-9730 or 9628
FAX: +52-55-514-4873

Visual Numerics International GmbH
Zettachring 10
D-70567Stuttgart
GERMANY

PHONE: +49-711-13287-0
FAX: +49-711-13287-99
e-mail: vni@visual-numerics.de

Visual Numerics Japan, Inc.
GOBANCHO HIKARI BLDG. 4TH Floor
14 GOBAN-CHO CHIYODA-KU
TOKYO, JAPAN 102

PHONE: +81-3-5211-7760
FAX: +81-3-5211-7769
e-mail: vnijapan@vnij.co.jp

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Road
Taipei, Taiwan 110
ROC

PHONE: +(886) 2-2727-2255
FAX: +(886) 2-2727-6798
e-mail: info@vni.com.tw

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-1, MAPO-DONG, MAPO-GU
SEOUL, 121-050
KOREA SOUTH

PHONE: +82-2-3273-2632 or 2633
FAX: +82-2-3273--2634
e-mail: info@vni.co.kr

World Wide Web site: http://www.vni.com

IMSL Fortran, C, and Java Application Development Tools

Contents

Introduction

The IMSL Fortran Library

The IMSL Fortran 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 (Volumes 1 and 2)
  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 complex-double 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 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 Library provide a simpler, more reliable user interface than was possible with FORTRAN 77. Features of the IMSL Fortran 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 Library can take advantage of a standard (MPI) Message Passing Interface environment. Gray shading in the documentation cues the reader when this is an issue.

However, MPI is not required to use any of the routines in the Library. All documented routines can be called in a scalar environment.

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 to be used 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 all 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. Futhermore, 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 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.
- Example: at least one application of this routine showing input and required dimension and type statements.
- Output: results from the example(s).
- Comments: details pertaining to code usage.
- Description: a description of the algorithm and references to detailed information. In many cases, other IMSL routines with similar or complementary functions are noted.
- Programming notes: an optional section that contains programming details not covered elsewhere.
- References: periodicals and books with details of algorithm development.
- 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_`." 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 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 complex double-precision versions of the code.

The appropriate routine is identified by the Fortran 90 compiler. 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. For example, the module named "imsl_libraries" contains the interface modules for all routines in the library.

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

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

where the overstrike denotes complex conjugation. IMSL Fortran Library linear algebra software uses this convention to conserve the utility of generic documentation for that code subject. References to *orthogonal* matrices are 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 also are 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 which 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. (This would be caught by the compiler.)

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

Programming Tips

It is strongly suggested that users force all program variables to be explicitly typed. This is done by including the line "`IMPLICIT NONE`" as close to the first line as possible. Study some of the examples accompanying an IMSL Fortran Library routine early on. These examples are available online as part of the product.

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 typographical 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 "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 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 Solvers" 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 which existed in the Fortran 77 version of the Library. Note that it is not necessary to use "use" statements when calling these routines by themselves. Existing programs which 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 Solvers" 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, 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 sends a signal to 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 chosen to 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.

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 as well as the general nature of the error is designated by an "error type" with numbers 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 chapter of this manual), 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 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.

MPI REQUIRED

Users of the IMSL Fortran Library benefit by a standard (MPI) Message Passing Interface environment. This is needed to accomplish parallel computing within parts of the documentation. *Light shading in the documentation cues the reader when this is an issue*. If parallel computing is not required, then the MP Library suite of dummy MPI routines can be substituted for standard MPI routines. All requested MPI routines called by the MP 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 Library. The standard includes a library of MPI Fortran and C routines, MPI "include" files, usage documentation, and other run-time utilities.

In addition, high-level operators and functions are provided in the Library. They are described in Chapter 10, "Operators and Generic Functions - The Parallel Option." For information on writing a more compact and readable code, see Chapter 10, Linear Algebra Operators and Generic Functions. [1]

[1] ***Important Note: Please refer to the "Table of Contents" for locations of chapter references, example references, and function references.***

Using IMSL Fortran Library on Shared-Memory Multiprocessors

The IMSL Fortran 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 BLAS, and hardware-specific IMSL algorithms. Use of the IMSL Fortran 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 Library can be found in the online README file of the product distribution.

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.

Note that the leading examples in Chapters 1 and 2 have been written using operators and generic functions whenever appropriate. These examples are named as shown in Chapter 10, Table A - "Examples and Corresponding Operators." Less code is typically needed to compute equivalent results.

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 Library routines.

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^*

Defined Array Functions	**Matrix Operation**
`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)`	$\left[u_1, \ldots\right] = \left[a_1 / \lVert a_1 \rVert, \ldots\right]$
`F=DET(A)`	$det(A) = determinant$
`K=RANK(A)`	$rank(A) = rank$
`P=NORM(A[,[type=]i])`	$p = \lVert A \rVert_1 = \max_j (\sum_{i=1}^{m} \lvert a_{ij} \rvert)$ $p = \lVert A \rVert_2 = s_1 =$ largest singular va $p = \lVert A \rVert_{\infty \leftrightarrow huge(1)} = \max_i (\sum_{j=1}^{n} \lvert a_{ij} \rvert)$
`C=COND(A)`	$s_1 / s_{rank(A)}$
`Z=EYE(N)`	$Z = I_N$
`A=DIAG(X)`	$A = diag(x_1, \ldots)$
`X=DIAGONALS(A)`	$x = (a_{11}, \ldots)$
`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…*

Chapter 1: Linear Systems

Routines

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 staff, 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.

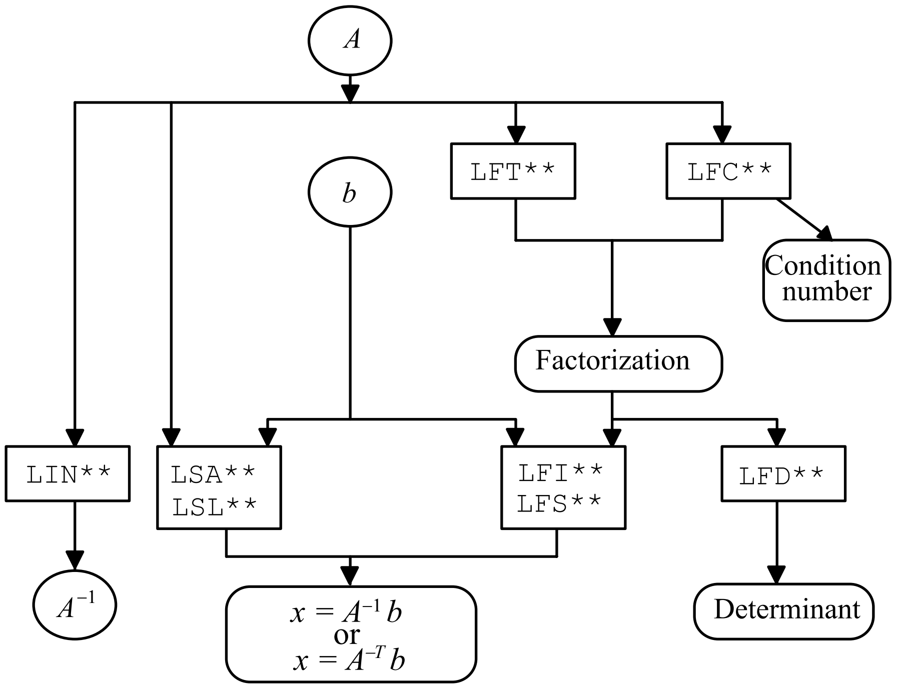

Figure 1-1 Solution and Factorization of Linear Systems

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 LSLRG (page 85) and LSARG (page 83) 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 $i = 1, 50$
    $r_i = Ax_{i-1} - b$ computed in higher precision
    $p_i = A^{-1} r_i$
    $x_i = x_{i-1} - p_i$
    if $(\| p_i \|_\infty \le \varepsilon \| x_i \|_\infty)$ 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 `LFC**`, 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`, page 352. Complex Toeplitz systems can be solved using `LSLTC`, page 354.

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`, page 356.

Iterative Solution of Linear Systems

The preconditioned conjugate gradient routines `PCGRC`, page 359, and `JCGRC`, page 365, 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`, page 368.

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`, page 392, or `LQRRV`, page 381. It returns R and the information needed to compute Q. To actually compute Q use `LQERR`, page 396. 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 $Rx = Q^TPb$. The routine `LQRSL`, page 398, can be used to find Q^TPb from the information computed by `LQRRR`. Then x can be computed by solving a triangular system using `LSLRT`, page 123. 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, $A \to A + \alpha xy^T$, the QR decomposition of A can be updated/down-dated using the routine `LUPQR`, page 402. 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.

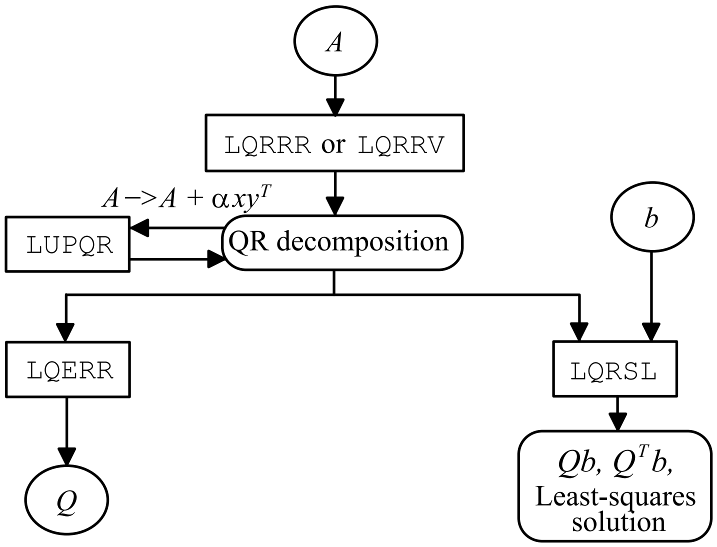

Figure 1-2 Least-Squares Routine

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])

B — Array of size $n \times nb$ containing the right-hand side matrix. (Input [/Output])

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 `lin_sol_gen`		
Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`lin_sol_gen_set_small`	1
`s_, d_, c_, z_`	`lin_sol_gen_save_LU`	2
`s_, d_, c_, z_`	`lin_sol_gen_solve_A`	3
`s_, d_, c_, z_`	`lin_sol_gen_solve_ADJ`	4
`s_, d_, c_, z_`	`lin_sol_gen_no_pivoting`	5
`s_, d_, c_, z_`	`lin_sol_gen_scan_for_NaN`	6
`s_, d_, c_, z_`	`lin_sol_gen_no_sing_mess`	7
`s_, d_, c_, z_`	`lin_sol_gen_A_is_sparse`	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_set_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 $i = 1, \ldots,$ `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 point 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`.

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`, Chapter 10, 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.
```

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, ..., 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).

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

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

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
!           <= sqrt(epsilon(one)))*abs(determinant(2))) then

      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`, Chapter 10, 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.
```

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.

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]

B — Array of size $n \times nb$ containing the right-hand side matrix. (Input [/Output]

X — Array of size $n \times nb$ containing the solution matrix. (Input [/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 the 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` + 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 `det`(2) is within exponent range, the value of the determinant is given by the expression 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 `lin_sol_self`		
Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`Lin_sol_self_set_small`	1
`s_, d_, c_, z_`	`Lin_sol_self_save_factors`	2
`s_, d_, c_, z_`	`Lin_sol_self_no_pivoting`	3
`s_, d_, c_, z_`	`Lin_sol_self_use_Cholesky`	4
`s_, d_, c_, z_`	`Lin_sol_self_solve_A`	5
`s_, d_, c_, z_`	`Lin_sol_self_scan_for_NaN`	6
`s_, d_, c_, z_`	`Lin_sol_self_no_sing_mess`	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 $i = 1, \ldots,$ `n` $- 1$ when using Aasen's method. When using the Cholesky method, `pivots`(i) = i for $i = 1, \ldots,$ `n`.

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

The Cholesky decomposition $PAP^T = R^TR$ 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 natrix 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`.

Example 1: Solving a Linear Least-squares System

This example solves a linear least-squares system $Cx \cong d$, where C_{mxn} 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.
```

Description

Rountine `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`".

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

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

the inverse matrix is returned as the "`ainv=`" optional argument. The scale factor σ^2 and Γ 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 "`iopt=`" 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, λ, 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`, Chapter 10.

```
      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.
```

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.

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

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

Least-squares solutions, where the unknowns are non-negative or have simple bounds, can be computed with `PARALLEL_NONEGATIVE_LSQ` on page 67 and `PARALLEL_BOUNDED_LSQ` on page 75. These codes can be restricted to execute without MPI.

Required Arguments

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

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]

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 `lin_sol_lsq`

Option Prefix = ?	Option Name	Option Value
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_set_small`	1
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_save_QR`	2
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_solve_A`	3
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_solve_ADJ`	4
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_no_row_pivoting`	5
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_no_col_pivoting`	6
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_scan_for_NaN`	7
`s_`, `d_`, `c_`, `z_`	`lin_sol_lsq_no_sing_mess`	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 $i = 1, \ldots,$ 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 $i = 1, \ldots,$ 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`.

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

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

using a discrete set of values in the interval $-1 \le x \le 1$. 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`, Chapter 10.

```
      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.
```

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.

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`, Chapter 10.

```
      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, ainv=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 δ^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`, Chapter 10.

```
      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_lsq(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_LSQ 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_LSQ 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 ε is the machine precision, but any larger value can be used. The fact that `lin_sol_lsq` 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`, Chapter 10.

```
      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.
```

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.

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]

B — Array of size $m \times nb$ containing the right-hand side matrix. (Input [/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` = 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.

`RANK = k` (Output)
Number of singular values that are at least as large as the value *Small*. It will satisfy `k` <= 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 `lin_sol_svd`

Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`lin_sol_svd_set_small`	1
`s_, d_, c_, z_`	`lin_sol_svd_overwrite_input`	2
`s_, d_, c_, z_`	`lin_sol_svd_safe_reciprocal`	3
`s_, d_, c_, z_`	`lin_sol_svd_scan_for_NaN`	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`.

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.
```

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.

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`, Chapter 10.

```
   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:

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

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`, Chapter 10.

```
      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(oldrms) <= 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.
```

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.

LIN_SOL_TRI

Solves multiple systems of linear equations

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

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])

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])

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])

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 = (size (`D`, 1))/2 and k = 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` = (size(`D`, 1))/2

`NPROB = k` (Input)
The number of systems solved.
Default: *k* = 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 `LIN_SOL_TRI`

Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`lin_sol_tri_set_small`	1
`s_, d_, c_, z_`	`lin_sol_tri_set_jolt`	2
`s_, d_, c_, z_`	`lin_sol_tri_scan_for_NaN`	3

Packaged Options for `LIN_SOL_TRI`		
`s_, d_, c_, z_`	`lin_sol_tri_factor_only`	4
`s_, d_, c_, z_`	`lin_sol_tri_solve_only`	5
`s_, d_, c_, z_`	`lin_sol_tri_use_Gauss_elim`	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 $2 \times jolt$.

Default: $0.25 \times epsilon()$

`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`.

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`, Chapter 10.

```
      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(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err
      real(kind(1d0)), 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.
```

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 $|a| > Small$. Alternately, t has the value $2 \times jolt$. (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.

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`, Chapter 10.

```
      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

$$\lambda_1, \ldots, \lambda_n$$

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`EVASB`. 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`, Chapter 10.

```
      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 $0 \le x \le \pi$ 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` (IMSL 2003, pp. 908–911) for a problem that uses reverse communication. Next see Example 4 of routine `IVPAG` (IMSL 2003, pp. 867-870) 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`, Chapter 10.

```
      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

        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.
```

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.

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])

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` <= 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 `LIN_SVD`

Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`lin_svd_set_small`	1
`s_, d_, c_, z_`	`lin_svd_overwrite_input`	2
`s_, d_, c_, z_`	`lin_svd_scan_for_NaN`	3
`s_, d_, c_, z_`	`lin_svd_use_qr`	4
`s_, d_, c_, z_`	`lin_svd_skip_orth`	5
`s_, d_, c_, z_`	`lin_svd_use_gauss_elim`	6
`s_, d_, c_, z_`	`lin_svd_set_perf_ratio`	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`.

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`, Chapter 10.

```
   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.
```

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.

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 α_i 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, Chapter 10.

```
      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

$$AX = Udiag\left(c_1, \ldots, c_n\right)$$

and

$$BX = Vdiag\left(s_1, \ldots, s_n\right)$$

The values s_i and c_i $_i$ 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

$$U^T A = diag\left(c_1, \ldots, c_n\right) W$$

and

$$V^T B = diag\left(s_1, \ldots, s_n\right) W$$

The matrix *W* has the same singular values as the matrix *D*. Also, see `operator_ex23`, Chapter 10.

```
      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 $(w_1, \ldots, w_m)$, 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

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

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 λ, $0.1 s_1 \le \lambda \le 10 s_1$. Array operations and intrinsics are used to evaluate the function and then to choose an approximate minimum. Following the computation of the optimum λ, the regularized solutions are computed. Also, see `operator_ex24`, Chapter 10.

```
      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(1d0)), parameter :: one=1d0
      real(kind(1d0)) log_lamda, log_lamda_t, delta_log_lamda
      real(kind(1d0)) 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.
```

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.

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: $A = \left[A_1 \mid A_2 \mid \ldots \mid A_k \right]$. An individual block of the partitioned matrix, say A_p, is located entirely on the processor with rank MP_RANK=$p-1$, where `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, Parallelism Using MPI.

PARALLEL_NONNEGATIVE_LSQ

MPI REQUIRED

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(1: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(1: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(1: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(1: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 give 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 PARALLEL_NONNEGATIVE_LSQ	
Option Name	**Option Value**
PNLSQ_SET_TOLERANCE	1
PNLSQ_SET_MAX_ITERATIONS	2
PNLSQ_SET_MIN_RESIDUAL	3

`IOPT(IO)=?_OPTIONS(PNLSQ_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(PNLSQ_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)= PNLSQ_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`.

Example 1: Distributed Linear Inequality Constraint Solver

The program `PNLSQ_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 $E = \left[G : h\right]$ 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 PNLSQ_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 PNLSQ_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.
```

Description

Subroutine `PARALLEL_NONNEGATIVE_LSQ` solves the linear least-squares system $Ax \cong b,\ \ x \ge 0$, 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.

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: $a_i^T x \cong b_i,\ i = 1,...,m,$ subject to $x \ge 0$. In this example we write the row vectors $\left[a_i^T : b_i \right]$ 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(1d0)), ALLOCATABLE, DIMENSION(:) :: &
     d_A(:,:), A(:,:), B, C, W, X, Y
   real(kind(1d0)) 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 e1pop("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

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(1: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(1: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(1:2,1:N) — (Input) Assumed-size array containing the bounds for x. The lower bound α_j is in `BND(1,J)`, and the upper bound β_j is in `BND(2,J)`.

X(1:N) — (Output) Assumed-size array of length `N` containing the solution, $\alpha \le x \le \beta$. 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(1: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 $j, 1 \le j \le n,$

- $\alpha_j = x_j = \beta_j$, and w_j arbitrary
- $\alpha_j = x_j$, and $w_j \le 0$
- $x_j = \beta_j$, and $w_j \ge 0$
- $\alpha_j < x_j < \beta_j$, 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 `PARALLEL_BOUNDED_LSQ`

Option Name	Option Value
`PBLSQ_SET_TOLERANCE`	1
`PBLSQ_SET_MAX_ITERATIONS`	2
`PBLSQ_SET_MIN_RESIDUAL`	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.

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
```

Output

```
 Example 1 for PARALLEL_BOUNDED_LSQ is correct.
```

Description

Subroutine `PARALLEL_BOUNDED_LSQ` solves the least-squares linear system $Ax \cong b,\ \alpha \le x \le \beta$, 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.

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:

$$\bullet f_i\left(x\right) = x_i + \sum_{j=1}^{n} x_j - \left(n+1\right), i = 1,...,n-1$$

$$\bullet f_n\left(x\right) = x_1 ... x_n - 1$$

The functions are zero at the point $x = \left(\delta,...,\delta,\delta^{1-n}\right)^T$, where $\delta > 1$ is a particular root of the polynomial equation $n\delta^n - \left(n+1\right)\delta^{n-1} + 1 = 0$. To avoid convergence to the local minimum $x = \left(0,...,0,n+1\right)^T$, we start at the standard point $x = \left(1/2,...,1/2,1/2\right)^T$ and develop the Newton method using the linear terms $f\left(x-y\right) \approx f\left(x\right) - J\left(x\right)y \cong 0$, where $J\left(x\right)$ 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 $x_n > y_n \geq \left(x_n - 1/2\right)$. These bounds avoid the local minimum and allow us to replace the last equation by $\sum_{j=1}^{n} \ln\left(x_j\right) = 0$, 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 $J\left(x\right)$ 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` = 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 the system $AX = B$ is solved.

`IPATH` = 2 means the system $A^TX = B$ 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`.

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
!                                 Declare variables
```

```
      PARAMETER  (LDA=3, N=3)
      REAL       A(LDA,LDA), B(N), X(N)
!
!                                 Set values for A and B
!
!                                 A = ( 33.0  16.0  72.0)
!                                     (-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 B/129.0, -96.0, 8.5/
!
      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
```

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

Description

Routine `LSARG` solves a system of linear algebraic equations having a real general coefficient matrix. It first uses the routine `LFCRG`, page 89, 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`, page 96.

`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 $1/\varepsilon$ (where ε 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.

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` = 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 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`.

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

!                                 Declare variables
      PARAMETER  (LDA=3, N=3)
      REAL       A(LDA,LDA), B(N), X(N)
!
!                                 Set values for A and B
!
!                                 A = ( 33.0  16.0  72.0)
!                                     (-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 B/129.0, -96.0, 8.5/
!
      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
```

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` × `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	
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.

Description

Routine `LSLRG` solves a system of linear algebraic equations having a real general coefficient matrix. It first uses the routine `LFCRG` (page 89) 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` (page 94).

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 $1/\varepsilon$ (where ε 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 `LSVRR`, page 415, or `LSARG`, page 83, be used.

Additional Example

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, $x_j = j$, $j = 1, ..., N$.

```
      USE L2LRG_INT
      USE IUMAG_INT
      USE WRRRN_INT
      USE SGEMV_INT
!                                 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
```

LFCRG

Computes the *LU* factorization of a real general matrix and estimate 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` = 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 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`.

Example

The inverse of a 3 × 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,LDA), AINV(LDA,LDA), FACT(LDFACT,LDFACT), RCOND, &
                 RES(N), RJ(N)
!                                 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 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
!                                 Print results
      CALL WRRRN ('AINV', AINV)
!
99998 FORMAT ('  RCOND = ',F5.3,/,'  L1 Condition number = ',F6.3)
      END
```

Output

```
RCOND = 0.015
L1 Condition number = 66.471

          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
```

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

Description

Routine `LFCRG` performs an *LU* factorization of a real general 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 ∞-norm.

The L_1 condition number of the matrix A is defined to be $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 $1/\varepsilon$ (where ε 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`, page 96, `LFSRG`, 94, and `LFDRG`, page 99. 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

LFDRG 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.

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 = 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 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 `DLFCRG`.

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 `LFCRG` (page 89) to perform the factorization, and `LFIRG` (page 96) 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
!                                 Print results
      CALL WRRRN ('X', X)
      END
```

Output

```
         X
        1       2
1  -2.000   1.000
2  -2.000  -1.000
3   3.000   4.000
```

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.

Description

Routine `LFTRG` performs an *LU* factorization of a real 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 samenorm.

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` (page 96), `LFSRG` (page 94) and `LFDRG` (page 99). 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 `LFDRG` 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

$$L^{-1} = L_{N-1}P_{N-1} \ldots L_1 P_1$$

where P_k is the identity matrix with rows *k* and p_k interchanged and L_k is the identity with F_{ik} for $i = k + 1, ..., N$ 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.

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` (page 89). (Input)

IPVT — Vector of length `N` containing the pivoting information for the *LU* factorization of `A` as output from subroutine `LFCRG` (page 89) or `LFTRG/DLFTRG` (page 92). (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` = 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).

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 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`.

Example

The inverse is computed for a real general 3 × 3 matrix. The input matrix is assumed to be well-conditioned, hence, `LFTRG` is used rather than `LFCRG`.

```
      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 = (  1.0   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 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
```

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 LFCRG, page 89, or LFTRG, page 92. 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, page 94, and LFIRG, page 96, 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 routine LFSRG is based on the LINPACK routine SGESL; see Dongarra et al. (1979).

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` = 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).

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`.

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 = 0.015
L1 Condition number = 66.471
           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
```

Comments

Informational error

Type	Code	
3	2	The input matrix is too ill-conditioned for iterative refinement to be effective.

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.

To compute the solution, the coefficient matrix must first undergo an *LU* factorization. This may be done by calling either `LFCRG`, page 89, or `LFTRG`, page 92.

Iterative refinement fails only if the matrix is very ill-conditioned.

Routines `LFIRG` (page 96) and `LFSRG` (page 94) 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`.

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 `LFCRG/DLFCRG` (page 89). (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 $1.0 \le |\mathrm{DET1}| < 10.0$ or `DET1` = 0.0.

DET2 — Scalar containing the exponent of the determinant. (Output)
The determinant is returned in the form $\det(A) = \mathrm{DET1} * 10^{\mathrm{DET2}}$.

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`.

Example

The determinant is computed for a real general 3 × 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.
```

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 LFCRG (page 89) or LFTRG (page 92). 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)

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 = 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).

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 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`.

Example

The inverse is computed for a real general 3 × 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
```

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.

Description

Routine `LINRG` computes the inverse of a real general matrix. It first uses the routine `LFCRG` (page 89) 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`, page 128, 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 $1/\varepsilon$ (where ε is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in A^{-1}.

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` = 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 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`.

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)
```

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^2containing 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.

Description

Routine LSACG solves a system of linear algebraic equations with a complex general coefficient matrix. It first uses the routine LFCCG, page 108, 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, page 116.

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 $1/\varepsilon$ (where ε 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.

LSLCG

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` = 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 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`.

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
```

Output

```
                           X
              1                  2                  3
( 1.000,-1.000)   ( 2.000, 1.000)   ( 0.000, 3.000)
```

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 × 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	
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.

Description

Routine LSLCG solves a system of linear algebraic equations with a complex general coefficient matrix. It first uses the routine LFCCG, page 108, 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, page 114.

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/ε (where ε 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, page 103, be used.

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 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` = 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 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`.

Example

The inverse of a 3 × 3 matrix is computed. `LFCCG` is called to factor the matrix and to check for singularity or ill-conditioning. `LFICG` (page 116) 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,LDA), AINV(LDA,LDA), RJ(N), FACT(LDFACT,LDFACT), &
                 RES(N)
!                                 Declare functions
      COMPLEX    CMPLX
!                                 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) = CMPLX(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) = CMPLX(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 = 0.016
L1 Condition number = 63.104

                         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)
```

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.

Description

Routine `LFCCG` performs an *LU* factorization of a complex general 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 ∞-norm.

The L_1 condition number of the matrix A is defined to be $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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`, page 116, `LFSCG`, page 114, and `LFDCG`, page 119. 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

$$L^{11} = L_{N-1}P_{N-1} \ldots L_1 P_1$$

where P_k is the identity matrix with rows k and p_k interchanged and L_k is the identity with F_{ik} for $i = k + 1, ..., N$ inserted below the diagonal. The strict lower half of F can also be thought of as containing the negative of the multipliers.

`LFCCG` is based on the LINPACK routine `CGECO`; see Dongarra et al. (1979). `CGECO` uses unscaled partial pivoting.

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` 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` = 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 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`.

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 WRCRN_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)
```

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	
4	2	The input matrix is singular.

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 ∞-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, page 116, LFSCG, page 114, and LFDCG, page 119. 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 LFDCG can be called to compute the determinant of the coefficient matrix after LFCCG (page 108) 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

$$L = L_{N\text{-}1}P_{N\text{-}1} \dots L_1 P_1$$

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.

LFTCG is based on the LINPACK routine CGEFA; see Dongarra et al. (1979). CGEFA uses unscaled partial pivoting.

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 = 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).

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 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.

Example

The inverse is computed for a complex general 3 × 3 matrix. The input matrix is assumed to be well-conditioned, hence LFTCG (page 111) 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    CMPLX
!                                 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) = CMPLX(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) = CMPLX(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)
```

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`, page 108, or `LFTCG`, page 111. 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` (page 114) and `LFICG` (page 116) 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`.

`LFSCG` is based on the LINPACK routine `CGESL`; see Dongarra et al. (1979).

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` = 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).

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`.

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 + 0.5*i* 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)
```

```
!                                  Declare functions
      COMPLEX    CMPLX
!                                  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 = 0.023
L1 Condition number = 42.799
                        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)
```

Comments

Informational error

Type	Code	
3	2	The input matrix is too ill-conditioned for iterative refinement to be effective

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, page 108, or LFTCG, page 111.

Iterative refinement fails only if the matrix is very ill-conditioned. Routines LFICG (page 116)and LFSCG (page 114) 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.

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/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)

DET1 — Complex scalar containing the mantissa of the determinant. (Output)
The value DET1 is normalized so that $1.0 \le |\text{DET1}| < 10.0$ or DET1 = 0.0.

DET2 — Scalar containing the exponent of the determinant. (Output)
The determinant is returned in the form det(*A*) = DET1 * 10^{DET}.

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 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`.

Example

The determinant is computed for a complex general 3 × 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.
```

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`, page 108, or `LFTCG`, page 111. 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.

`LFDCG` is based on the LINPACK routine `CGEDI`; see Dongarra et al. (1979).

LINCG

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` = 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).

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 LINCG (A, AINV [,…])`

Specific: The specific interface names are `S_LINCG` and `D_LINCG`.

FORTRAN 77 Interface

Single: `CALL LINCG (N, A, LDA, AINV, LDAINV)`

Double: The double precision name is `DLINCG`.

Example

The inverse is computed for a complex general 3 × 3 matrix.

```
      USE LINCG_INT
      USE WRCRN_INT
      USE CSSCAL_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)
```

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.

Description

Routine `LINCG` computes the inverse of a complex general matrix.

It first uses the routine `LFCCG`, page 108, 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`, page 136, is then used to compute `U`. Finally `A` is computed using `A=UL`.

`LINCG` 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 $1/\varepsilon$ (where ε is machine precision), a warning error is issued. This indicates that very small changes in *A* can cause very large changes in A^{-1}.

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` = 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 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.

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
```

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. `LSLRT` is based on the LINPACK routine `STRSL`; see Dongarra et al. (1979).

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` = 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.

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`.

Example

An estimate of the reciprocal condition number is computed for a 3 × 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 = 0.091
L1 Condition number = 10.968
```

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.

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 $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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/ε (where ε 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*.

LFCRT is based on the LINPACK routine STRCO; see Dongarra et al. (1979).

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 ≤ |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) = DET1 * 10^{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.

Example

The determinant is computed for a 3 × 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.
```

Comments

Informational error

Type	Code	
3	1	The input triangular matrix is singular.

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).

LINRT

Computes the determinant 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.

Example

The inverse is computed for a 3 × 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
```

Description

Routine `LINRT` computes the inverse of a real triangular matrix. It fails if `A` has a zero diagonal element.

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` = 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 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`.

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)
```

Comments

Informational error

Type	Code	
4	1	The input triangular matrix is singular. Some of its diagonal elements are near zero.

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. `LSLCT` is based on the LINPACK routine `CTRSL`; see Dongarra et al. (1979).

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` = 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.

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`.

Example

An estimate of the reciprocal condition number is computed for a 3 × 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 = 0.191
L1 Condition number = 5.223
```

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.

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 $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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. `LFCCT` is based on the LINPACK routine `CTRCO`; see Dongarra et al. (1979).

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 $1.0 \le |$`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)` = `DET1` * 10^{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 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`.

Example

The determinant is computed for a 3 × 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.
```

Comments

Informational error

Type	Code	
3	1	The input triangular matrix is singular.

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).

LINCT

Computes the inverse of a complex triangular matrixs.

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 = 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 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`.

Example

The inverse is computed for a 3 × 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)
```

Comments

Informational error

Type	Code	
4	1	The input triangular matrix is singular. Some of its diagonal elements are close to zero.

Description

Routine `LINCT` computes the inverse of a complex triangular matrix. It fails if `A` has a zero diagonal element.

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` = 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 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`.

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
```

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.

Description

Routine `LSADS` solves a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. It first uses the routine `LFCDS`, page 143, 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`, page 150. `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 $1/\varepsilon$ (where ε 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.

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` = 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`.

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
```

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` × `N` work array containing the R^TR 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.

Description

Routine `LSLDS` solves a system of linear algebraic equations having a real symmetric positive definite coefficient matrix. It first uses the routine `LFCDS`, page 143, to compute an R^TR

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`, page 148. `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 $1/\varepsilon$ (where ε 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`, page 138, be used.

LFCDS

Computes the R^TR 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` = 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 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`.

Example

The inverse of a 3 × 3 matrix is computed. `LFCDS` is called to factor the matrix and to check for nonpositive definiteness or ill-conditioning. `LFIDS` (page 150) 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 = 0.001
L1 Condition number = 674.727
```

```
          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
```

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.

Description

Routine `LSADS` computes an R^TR 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 $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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^TR factors are returned in a form that is compatible with routines `LFIDS`, page 150, `LFSDS`, page 148, and `LFDDS`, page 153. 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 `LFDDS` can be called to compute the determinant of the coefficient matrix after `LFCDS` has performed the factorization.

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. (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` = 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 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`.

Example

The inverse of a 3 × 3 matrix is computed. `LFTDS` is called to factor the matrix and to check for nonpositive definiteness. `LFSDS` (page 148) 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
```

Comments

Informational error

Type	Code	
4	2	The input matrix is not positive definite.

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`, page 150, `LFSDS`, page 148, and `LFDDS`, page 153. 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.

`LFTDS` is based on the LINPACK routine `SPOFA`; see Dongarra et al. (1979).

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` = 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 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`.

Example

A set of linear systems is solved successively. `LFTDS` (page 146) is called to factor the coefficient matrix. `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 `LFCDS` (page 143) to perform the factorization, and `LFIDS` (page 150) 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
```

Comments

Informational error

Type	Code	
4	1	The input matrix is singular.

Description

This routine 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^TR factorization. This may be done by calling either `LFCDS`, page 143, or `LFTDS`, page 146. R is an upper triangular matrix.

The solution to $Ax = b$ is found by solving the triangular systems $R^Ty = b$ and $Rx = y$.

`LFSDS`, (page 148) and `LFIDS`, page 150, both solve a linear system given its R^TR 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`.

`LFSDS` is based on the LINPACK routine `SPOSL`; see Dongarra et al. (1979).

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` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimesion 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 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`.

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
```

Comments

Informational error

Type	Code	
3	2	The input matrix is too ill-conditioned for iterative refinement to be effective.

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.

To compute the solution, the coefficient matrix must first undergo an $R^T R$ factorization. This may be done by calling either `LFCDS`, page 143, or `LFTDS`, page 146.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFIDS, page 150 and LFSDS, page 148, 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.

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 LFCDS/DLFCDS or LFTDS/DLFTDS. (Input)

DET1 — Scalar containing the mantissa of the determinant. (Output)
The value DET1 is normalized so that, $1.0 \le |\text{DET1}| < 10.0$ 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.

Example

The determinant is computed for a real positive definite 3 × 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.
```

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 `LFCDS`, page 143, or `LFTDS`, page 146. 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).

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 = size (A,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 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.

Example

The inverse is computed for a real positive definite 3 × 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
```

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.

Description

Routine `LINDS` computes the inverse of a real symmetric positive definite matrix. It first uses the routine `LFCDS`, page 143, to compute an R^TR factorization of the coefficient matrix and to estimate the condition number of the matrix. `LINRT`, page 128, is then used to compute R^{-1}. Finally A^{-1} is computed using $R^{-1} = R^{-1}\ R^{-T}$.

`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 $1/\varepsilon$ (where ε is machine precision), a warning error is issued. This indicates that very small changes in A can cause very large changes in A.

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` = 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 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`.

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
```

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 × 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

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.

Description

Routine `LSASF` solves systems of linear algebraic equations having a real symmetric indefinite coefficient matrix. It first uses the routine `LFCSF`, page 162, to compute a UDU^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`, page 169.

`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 $1/\varepsilon$ (where ε 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.

LSLSF

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` = 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 LSLSF (A, B, X [,…])`

Specific: The specific interface names are `S_LSLSF` and `D_LSLSF`.

FORTRAN 77 Interface

Single: `CALL LSLSF (N, A, LDA, B, X)`

Double: The double precision name is `DLSLSF`.

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 LSLSF_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 LSLSF (A, B, X)
!                                 Print results
      CALL WRRRN ('X', X, 1, N, 1)
      END
```

Output

```
          X
     1       2        3
-4.100  -3.500   1.200
```

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` × `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.

Integer Options with Chapter 11 Options Manager

16 This option uses four values to solve memory bank conflict (access inefficiency) problems. In routine `LSLSF` 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 `LSLSF`. Additional memory allocation for `FACT` and option value restoration are done automatically in `LSLSF`. Users directly calling `LSLSF` 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 `LSLSF` or `LSLSF`. 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 `LSLSF` temporarily replaces `IVAL`(2) by `IVAL`(1). The routine `L2CSF` computes the condition number if `IVAL`(2) = 2. Otherwise `L2CSF` skips this computation. `LSLSF` restores the option. Default values for the option are `IVAL(*)` = 1, 2.

Description

Routine `LSLSF` solves systems of linear algebraic equations having a real symmetric indefinite coefficient matrix. It first uses the routine `LFCSF`, page 162, 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`, page 167.

`LSLSF` fails if a block in *D* is singular. This occurs only if *A* either is singular or is very close to a singular matrix.

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` = 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 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`.

Example

The inverse of a 3 × 3 matrix is computed. LFCSF is called to factor the matrix and to check for singularity or ill-conditioning. LFISF (page 169) 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 LFCSF (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 = 0.034
L1 Condition number = 29.750

          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
```

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.

Description

Routine `LFCSF` performs a $U\,DU^T$ factorization of a real symmetric indefinite coefficient matrix. It also estimates the condition number of the matrix. The $U\,DU^T$ factorization is called the diagonal pivoting factorization.

The L_1 condition number of the matrix A is defined to be $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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 $U\,DU^T$ factors are returned in a form that is compatible with routines `LFISF`, page 169, `LFSSF`, page 167, and `LFDSF`, page 172. 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 `LFDSF` can be called to compute the determinant of the coefficient matrix after `LFCSF` has performed the factorization.

`LFCSF` is based on the LINPACK routine `SSICO`; see Dongarra et al. (1979).

LFTSF

Computes the $U\,DU^T$ 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` = 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 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`.

Example

The inverse of a 3 × 3 matrix is computed. `LFTSF` is called to factor the matrix and to check for singularity. `LFSSF` (page 167) 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
```

Comments

Informational error

Type	Code	
4	2	The input matrix is singular.

Description

Routine `LFTSF` performs a $U\,DU^T$ factorization of a real symmetric indefinite coefficient matrix. The $U\,DU^T$ factorization is called the diagonal pivoting factorization.

`LFTSF` fails if A is singular or very close to a singular matrix.

The $U\,DU^T$ factors are returned in a form that is compatible with routines `LFISF`, page 169, `LFSSF`, page 167, and `LFDSF`, page 172. 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.

`LFTSF` is based on the LINPACK routine `SSIFA`; see Dongarra et al. (1979).

LFSSF

Solves a real symmetric system of linear equations given the $U\,DU^T$ factorization of the coefficient matrix.

Required Arguments

FACT — `N` by `N` matrix containing the factorization of the coefficient matrix `A` as output from routine `LFCSF`/`DLFCSF` 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 `LFCSF`/`DLFCSF` 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` = size (`FACT`,2).

LDFACT — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDFACT` = size (`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`.

Example

A set of linear systems is solved successively. LFTSF (page 164) 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 (page 162) to perform the factorization, and LFISF (page 169) 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

```
                  X
        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
```

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 $U\,DU^T$ factorization. This may be done by calling either LFCSF, page 162, or LFTSF, page 164.

LFSSF, page 167, and LFISF, page 169, both solve a linear system given its $U\,DU^T$ 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.

LFSSF is based on the LINPACK routine SSISL; see Dongarra et al. (1979).

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 LFCSF/DLFCSF 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 LFCSF/DLFCSF 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.

RES — Vector of length N containing the residual vector at the improved solution. (Output)

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).

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 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`.

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 LFCSF_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 LFCSF (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 = 0.034
L1 Condition number = 29.750

           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
```

Comments

Informational error

Type	Code	
3	2	The input matrix is too ill-conditioned for iterative refinement to be effective.

Description

`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 $U\,DU^T$ factorization. This may be done by calling either `LFCSF`, page 162, or `LFTSF`, page 164.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFISF`, page 169 and `LFSSF`, page 167, both solve a linear system given its $U\,DU^T$ 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`.

LFDSF

Computes the determinant of a real symmetric matrix given the $U\,DU^T$ factorization of the matrix.

Required Arguments

FACT — `N` by `N` matrix containing the factored matrix A as output from subroutine `LFTSF`/`DLFTSF` or `LFCSF`/`DLFCSF`. (Input)

IPVT — Vector of length `N` containing the pivoting information for the $U\,DU^T$ factorization as output from routine `LFTSF`/`DLFTSF` or `LFCSF`/`DLFCSF`. (Input)

DET1 — Scalar containing the mantissa of the determinant. (Output)
The value `DET1` is normalized so that, 1.0 ≤ |`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) = `DET1 * 10`$^{\texttt{DET2}}$.

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 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 `DLFDSF`.

Example

The determinant is computed for a real symmetric 3 × 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.
```

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 `LFCSF`, page 162, or `LFTSF`, page 164. 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).

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` = 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 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`.

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 )
!              (                                   6.0+0.0i   1.0+1.0i )
!              (                                              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 /(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)
```

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` × `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	
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.

Description

Routine `LSADH` solves a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. It first uses the routine `LFCDH`, page 179, 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`, page 187.

`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 $1/\varepsilon$ (where ε 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.

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` = 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 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`.

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 )
!              (                                   6.0+0.0i   1.0+1.0i )
!              (                                              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 /(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)
```

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 × 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.

Description

Routine LSLDH solves a system of linear algebraic equations having a complex Hermitian positive definite coefficient matrix. It first uses the routine LFCDH, page 179, 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`, page 185.

`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 $1/\varepsilon$ (where ε 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`, page 173, be used.

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` = 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 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`.

Example

The inverse of a 5 × 5 Hermitian positive definite matrix is computed. `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.

```
      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 = 0.067
L1 Condition number = 14.961

                                     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)
```

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.

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 $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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`, page 187, `LFSDH`, page 185, and `LFDDH`, page 190. 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.

`LFCDH` is based on the LINPACK routine `CPOCO`; see Dongarra et al. (1979).

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` = 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 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`.

Example

The inverse of a 5 × 5 matrix is computed. `LFTDH` is called to factor the matrix and to check for nonpositive definiteness. `LFSDH` (page 185) is called to determine the columns of the inverse.

```
      USE LFTDH_INT
      USE LFSDH_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 LFSDH
!                                 reference places the J-th column of
!                                 the inverse of A in the J-th column
!                                 of AINV
         CALL LFSDH (FACT, RJ, AINV(:,J))
         RJ(J) = (0.0E0,0.0E0)
   10 CONTINUE
!                                 Print the results
```

```
      CALL WRCRN ('AINV', AINV, ITRING=1)
!
      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                   ( 0.4332, 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)
```

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.

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`, page 187, `LFSDH`, page 185, and `LFDDH`, page 190. 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.

`LFTDH` is based on the LINPACK routine `CPOFA`; see Dongarra et al. (1979).

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`/`DLFCDH` or `LFTDH`/`DLFTDH`. (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` = 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 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`.

Example

A set of linear systems is solved successively. `LFTDH` (page 182) is called to factor the coefficient matrix. `LFSDH` 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` (page 179) to perform the factorization, and `LFIDH` (page 187) to compute the solutions.

```
USE LFSDH_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 LFSDH (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)
```

Comments

Informational error

Type Code

4	1	The input matrix is singular.

Description

This routine 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, page 179, or LFTDH, page 182. R is an upper triangular matrix.

The solution to $Ax = b$ is found by solving the triangular systems $R^H y = b$ and $Rx = y$.

LFSDH and LFIDH, page 187, 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.

LFSDH is based on the LINPACK routine CPOSL; see Dongarra et al. (1979).

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 = 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 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`.

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 = 0.067
L1 Condition number = 14.961

                        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)
```

Comments

Informational error

Type	Code	
3	3	The input matrix is too ill-conditioned for iterative refinement to be effective.

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`, page 179, or `LFTDH`, page 182.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFIDH, page 187, and LFSDH, page 185, 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.

LFDDH

Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.

Required Arguments

FACT — Complex N by N matrix containing the $R^T R$ 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 $1.0 \le |\text{DET1}| < 10.0$ 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.

Example

The determinant is computed for a complex Hermitian positive definite 3 × 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.
```

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`, page 179, or `LFTDH`, page 182. 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).

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` = 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 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`.

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)
```

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 $U\,DU^H$ 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.

Description

Routine `LSAHF` solves systems of linear algebraic equations having a complex Hermitian indefinite coefficient matrix. It first uses the routine `LFCHF`, page 197 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 `LFIHF`, page 204.

`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 $1/\varepsilon$ (where ε 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.

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` = 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 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 `DLSLHF`.

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 )
!
!                               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 LSLHF (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)
```

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.

Description

Routine LSLHF solves systems of linear algebraic equations having a complex Hermitian indefinite coefficient matrix. It first uses the routine LFCHF, page 200, 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`, page 202. `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`, page 191 be used.

LFCHF

Computes the UDU^H factorization of a complex Hermitian matrix and estimate its L_1 condition number.

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)

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` = 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 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`.

Example

The inverse of a 3 × 3 complex Hermitian matrix is computed. `LFCHF` is called to factor the matrix and to check for singularity or ill-conditioning. `LFIHF` (page 204) is called to determine the columns of the inverse.

```
      USE LFCHF_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 LFCHF (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 = 0.240
L1 Condition number =  4.175

                              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)
```

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	
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.

Description

Routine LFCHF performs a $U\,DU^H$ factorization of a complex Hermitian indefinite coefficient matrix. It also estimates the condition number of the matrix. The $U\,DU^H$ factorization is called the diagonal pivoting factorization.

The L_1 condition number of the matrix A is defined to be $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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 $U D U^H$ factors are returned in a form that is compatible with routines LFIHF, page 204, LFSHF, page 202, and LFDHF, page 207. To solve systems of equations with multiple right-hand-side vectors, use LFCHF followed by either LFIHF or LFSHF called once for each right-hand side. The routine LFDHF can be called to compute the determinant of the coefficient matrix after LFCHF has performed the factorization.

LFCHF is based on the LINPACK routine CSICO; see Dongarra et al. (1979).

LFTHF

Computes the $U D U^H$ 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 = 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 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`.

Example

The inverse of a 3 × 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)
```

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.
4	4	The input matrix is not Hermitian. It has a diagonal entry with an imaginary part.

Description

Routine `LFTHF` performs a $U\,DU^H$ factorization of a complex Hermitian indefinite coefficient matrix. The $U\,DU^H$ factorization is called the diagonal pivoting factorization.

`LFTHF` fails if A is singular or very close to a singular matrix.

The $U\,DU^H$ factors are returned in a form that is compatible with routines `LFIHF`, page 204, `LFSHF`, page 202, and `LFDHF`, page 207. To solve systems of equations with multiple right-hand-side vectors, use `LFTHF` followed by either `LFIHF` 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.

`LFTHF` is based on the LINPACK routine `CSIFA`; see Dongarra et al. (1979).

LFSHF

Solves a complex Hermitian system of linear equations given the $U\,DU^H$ 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 = 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 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.

Example

A set of linear systems is solved successively. LFTHF (page 200) 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 (page 197) to perform the factorization, and LFIHF (page 204) 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)
```

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, page 197, or LFTHF, page 200.

LFSHF and LFIHF, page 204, 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.

LFSHF is based on the LINPACK routine CSISL; see Dongarra et al. (1979).

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` = 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 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`.

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 + 0.2*i* 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 LFCHF (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 = 0.240
L1 Condition number =  4.175
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)
```

Comments

Informational error

Type	Code	
3	3	The input matrix is too ill-conditioned for iterative refinement to be effective.

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 $U DU^H$ factorization. This may be done by calling either `LFCHF`, page 197, or `LFTHF`, page 200.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIHF` and `LFSHF`, page 202, both solve a linear system given its $U DU^H$ 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`.

LFDHF

Computes the determinant of a complex Hermitian matrix given the $U DU^H$ 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 $1.0 \le |\text{DET1}| < 10.0$ 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`.

Example

The determinant is computed for a complex Hermitian 3 × 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.
```

Description

Routine `LFDHF` computes the determinant of a complex Hermitian indefinite coefficient matrix. To compute the determinant, the coefficient matrix must first undergo a $U\,DU^H$ factorization. This may be done by calling either `LFCHF`, page 197, or `LFTHF`, page 200 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).

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` = 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`.

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
```

Comments

Informational error

Type	Code	
4	2	An element along the diagonal became exactly zero during execution.

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, page 211, should be considered as an alternative method to solve the system.

LSLCR

Computes the $L\,DU$ 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), …, 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), …, 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), …, 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), …, 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 = size (C,1).

IJOB — Flag to direct the desired factoring or solving step. (Input)
Default: IJOB = 1.

IJOB	**Action**
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`.

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
```

Description

Routine `LSLCR` factors and solves the real tridiagonal linear system $Ax = y$. The matrix is decomposed in the form $A = L\,DU$, 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`, page 209, or `LSLPB`, page 237, 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.

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 = 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 the system AX= B is solved.
IPATH = 2 means the system A^TX = 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`.

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
```

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) × `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.

Description

Routine LSARB solves a system of linear algebraic equations having a real banded coefficient matrix. It first uses the routine LFCRB, page 219, 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, page 227.

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/ε (where ε 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.

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` = 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 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`.

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
```

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 × `NLCA` + `NUCA` + 1) × `N` containing the *LU* factorization of `A` on output. If `A` is not needed, `A` can share the first (`NLCA` + `NUCA` + 1) `* 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.

Description

Routine `LSLRB` solves a system of linear algebraic equations having a real banded coefficient matrix. It first uses the routine `LFCRB`, page 219, 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`, page 225. `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/ε (where ε 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`, page 213, be used.

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` = 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 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`.

Example

The inverse of a 4 × 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` (page 227) 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
!                                Print results
      CALL WRRRN ('AINV', AINV)
!
99999 FORMAT ('  RCOND = ',F5.3,/,'  L1 Condition number = ',F6.3)
      END
```

Output

```
RCOND = 0.065
L1 Condition number = 15.351

               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
```

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.

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 ∞-norm.

The L_1 condition number of the matrix A is defined to be $\kappa(A) = \|A\|_1 \|A\|_1$. Since it is expensive to compute $\|A\|_1$, 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 ε 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.

`LSCRB` 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`, page 227, `LFSRB`, page 225, and `LFDRB`, page 230. 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} .

`LFCRB` is based on the LINPACK routine `SGBCO`; see Dongarra et al. (1979). `SGBCO` uses unscaled partial pivoting.

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` = 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, IPVT [,…])`

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`.

Example

A linear system with multiple right-hand sides is solved. `LFTRB` is called to factor the coefficient matrix. `LFSRB` (page 225,) 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 `LFCRB` (page 219) to perform the factorization, and `LFIRB` (page 227) 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
```

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.

Description

The 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, page 227, LFSRB, page 225, and LFDRB, page 230. To solve systems of equations with multiple right-hand-side vectors, use LFTRB 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 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.

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/DLFTRB. (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/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)
If B is not needed, B and X can share the same storage locations.

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).

IPATH — Path indicator. (Input)
IPATH = 1 means the system $AX = B$ is solved.
IPATH = 2 means the system $A^TX = B$ is solved.
Default: 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`.

Example

The inverse is computed for a real banded 4 × 4 matrix with one upper and one lower codiagonal. The input matrix is assumed to be well-conditioned, hence `LFTRB` (page 222) 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
```

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`, page 219, or `LFTRB`, page 222. 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`, page 225 and `LFIRB`, page 227, 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`.

`LFSRB` is based on the LINPACK routine `SGBSL`; see Dongarra et al. (1979).

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` = 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).

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`.

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 = 0.065
L1 Condition number = 15.351
                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
```

Comments

Informational error

Type	Code	
3	2	The input matrix is too ill-conditioned for iterative refinement to be effective

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`, page 219, or `LFTRB`, page 222.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIRB`, page 227, and `LFSRB`, page 225, 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`.

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 $1.0 \le |\texttt{DET1}| < 10.0$ or `DET1` = 0.0.

DET2 — Scalar containing the exponent of the determinant. (Output)
The determinant is returned in the form $\det(A) = \texttt{DET1} * 10^{\texttt{DET2}}$.

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 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`.

Example

The determinant is computed for a real banded 4 × 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.
```

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`, page 219, or `LFTRB`, page 222. 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).

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 = 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 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.

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
```

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.

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`, page 240, 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`, page 247.

`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 $1/\varepsilon$ (where ε 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.

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` = 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 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`.

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 LSLQS_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 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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of L2LQS/DL2LQS. The reference is:

   CALL L2LQS (N, A, LDA, NCODA, B, X, FACT, WK)

   The additional arguments are as follows:

   FACT — NCODA + 1 by N work array containing the R^TR factorization of A in band symmetric form on output. If A is not needed, A and FACT can share the same storage locations.

   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 L2LQS 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 LSLQS. Additional memory allocation for FACT and option value restoration are done automatically in LSLQS. Users directly calling L2LQS 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 LSLQS or L2LQS. 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 `LSLQS` temporarily replaces `IVAL`(2) by `IVAL`(1). The routine `L2CQS` computes the condition number if `IVAL`(2) = 2. Otherwise `L2CQS` skips this computation. `LSLQS` restores the option. Default values for the option are `IVAL`(*) = 1,2.

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`, page 240, to compute an R^TR 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`, page 245.

`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 $1/\varepsilon$ (where ε 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`, page 232, be used.

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` ≥ 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` = size (`A`,2).

LDA — Leading dimension of `A` exactly as specified in the dimension statement of the calling program. (Input)
Must satisfy `LDA` ≥ `N` + `NCODA`.
Default: `LDA` = 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 `A`$x = b$, where b is stored in column `NCODA` + 2 of array `A`. The vector x overwrites b in storage.
2	solve step only. Use b as column `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 `DLSLPB`.

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
```

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.

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 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, page 211, 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.

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^TR 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 = 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 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`.

Example

The inverse of a 4 × 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` (page 247) 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
```

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.

Description

Routine LFCQS computes an R^TR 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 $1/\varepsilon$ (where ε 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, page 247, LFSQS, page 245, and LFDQS, page 250. 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).

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 s not needed, A and FACT can share the same storage locations.

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).

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 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`.

Example

The inverse of a 3 × 3 matrix is computed. `LFTQS` is called to factor the matrix and to check for nonpositive definiteness. `LFSQS` (page 245) 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
```

Comments

Informational error

Type	Code	
4	2	The input matrix is not positive definite.

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, page 247, LFSQS, page 245, and LFDQS, page 250. 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).

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 an share the same storage locations.

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 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`.

Example

A set of linear systems is solved successively. `LFTQS` (page 243) 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` (page 240) to perform the factorization, and `LFIQS` (page 247) 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
```

Comments

Informational error

Type	Code	
4	1	The factored matrix is singular.

Description

This routine 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, page 240, or LFTQS, page 243. 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, page 247, 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).

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 = 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 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.

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
```

Comments

Informational error

Type	Code	
3	4	The input matrix is too ill-conditioned for iterative refinement to be effective.

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`, page 240, or `LFTQS`, page 243.

Iterative refinement fails only if the matrix is very ill-conditioned.

`LFIQS`, page 247 and `LFSQS`, page 245, 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`.

LFDQS

Computes the determinant of a real symmetric positive definite matrix given the R^TR 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 $1.0 \le |\texttt{DET1}| < 10.0$ or `DET1` = 0.0.

DET2 — Scalar containing the exponent of the determinant. (Output)
The determinant is returned in the form $\det(A) = \texttt{DET1} * 10^{\texttt{DET2}}$.

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`.

Example

The determinant is computed for a real positive definite 4 × 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.
```

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`, page 240, or `LFTQS`, page 243. 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).

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`.

Example

A system of $n = 4$ linear equations is solved.

```
USE LSLTQ_INT
USE WRCRL_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)
```

Comments

Informational error

Type	Code	
4	2	An element along the diagonal became exactly zero during execution.

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, page 254, should be considered as an alternative method to solve the system.

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), …, 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), …, A(N − 1). (Input/Output)

B — Complex array of size 2N containing the lower codiagonal of the N by N tridiagonal matrix in the entries B(1), …, 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),…,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 = size (C,1).

IJOB — Flag to direct the desired factoring or solving step. (Input)
Default: IJOB =1.

IJOB	**Action**
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 `DLSLCQ`.

Example

A real skew-symmetric tridiagonal matrix, A, of dimension $n = 1000$ is given by $c_k = -k$, $a_k = 0$, and $b_k = k$, $k = 1, \ldots, n - 1$, $a_n = 0$. 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 $\lambda = y^H Ay/y^H y$. (This example is contrived in the sense that the given tridiagonal skew-symmetric matrix eigenvalue problem is essentially equivalent to the tridiagonal symmetic 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
```

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`, page 252, or `LSLQB`, page 282, 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.

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` = 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 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`.

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

```
                                 X
            1                 2                 3                 4
( 3.000, 0.000)  (-1.000, 1.000)  ( 3.000, 0.000)  (-1.000, 1.000)
```

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.

Description

Routine `LSACB` solves a system of linear algebraic equations having a complex banded coefficient matrix. It first uses the routine `LFCCB`, page 262, 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`, page 271.

`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 $1/\varepsilon$ (where ε 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.

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` = 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 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`.

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
           1                 2                 3                 4
( 3.000, 0.000)  (-1.000, 1.000)  ( 3.000, 0.000)  (-1.000, 1.000)
```

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) × `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.

Description

Routine LSLCB solves a system of linear algebraic equations having a complex banded coefficient matrix. It first uses the routine LFCCB, page 262, 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, page 268.

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 $1/\varepsilon$ (where ε 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, page 257, be used.

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 = 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 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.

Example

The inverse of a 4 × 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)
```

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.

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 ∞-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 ε 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`, page 271, `LFSCB`, page 268, and `LFDCB`, page 274. 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 `LFDCB` 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.

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 = 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 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.

Example

A linear system with multiple right-hand sides is solved. LFTCB is called to factor the coefficient matrix. LFSCB (page 268), 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 (page 262) to perform the factorization, and LFICB (page 271) 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)
```

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.

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 ∞-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`, page 271, `LFSCB`, page 268, and `LFDCB`, page 274. 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 `LFDCB` 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.

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 = 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).

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 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`.

Example

The inverse is computed for a real banded 4 × 4 matrix with one upper and one lower codiagonal. The input matrix is assumed to be well-conditioned; hence LFTCB (page 265) 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)
```

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, page 262, or LFTCB, page 265. 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, page 271, 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).

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`/`DLFCCB` or `LFTCB`/`DLFTCB`. (Input)

IPVT — Vector of length `N` containing the pivoting information for the *LU* factorization of `A` as output from routine `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. (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` = 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).

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`.

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)
```

Comments

Informational error

Type   Code

3	3	The input matrix is too ill-conditioned for iterative refinement be effective.

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, page 262, or LFTCB, page 265.

Iterative refinement fails only if the matrix is very ill-conditioned.

LFICB and LFSCB, page 268, 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.

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 $1.0 \le |\text{DET1}| < 10.0$ or DET1 = 0.0.

DET2 — Scalar containing the exponent of the determinant. (Output)
The determinant is returned in the form $\text{det}(A) = \text{DET1} * 10^{\text{DET2}}$.

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.

Example

The determinant is computed for a complex banded 4 × 4 matrix with one upper and one lower codiagonal.

```
      USE LFDCB_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 LFDCB (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.
```

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`, page 262, or `LFTCB`, page 265. 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).

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 = 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 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`.

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)
```

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 (`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.

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`, page 290, 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`, page 292.

`LSAQH` fails if any submatrix of R is not positive definite, if R has a zero diagonal element, or if the iterative refinement agorithm 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/ε (where ε 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.

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` = 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 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`.

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)
```

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) × 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.

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`, page 290, 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`, page 290.

`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 $1/\varepsilon$ (where ε 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 sealed, it is recommended that `LSAQH`, page 276, be used.

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 ≥ 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 = size (A,2).

LDA — Leading dimension of A exactly as specified in the dimension statement of the calling program. (Input)
Must satisfy LDA ≥ N + NCODA.
Default: LDA = 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.

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
```

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.

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` (page 254) 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.

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` = 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 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`.

Example

The inverse of a 5 × 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` (page 292,) 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                   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)
```

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.
4	4	The input matrix is not Hermitian. It has a diagonal entry with an imaginary part

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 $1/\varepsilon$ (where ε 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`, page 292, `LFSQH`, page 290, and `LFDQH`, page 295. 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).

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` = 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 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`.

Example

The inverse of a 5 × 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)
```

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.

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`, page 292, `LFSQH`, page 290, and `LFDQH`, page 295. 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).

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` = 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 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`.

Example

A set of linear systems is solved successively. `LFTQH`, page 288, 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`, page 290, to perform the factorization, and `LFIQH`, page 292, 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)
```

Comments

Informational error

Type	Code	
4	1	The factored matrix has a diagonal element close to zero.

Description

This routine 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, page 290, or LFTQH, page 288. 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, page 292, 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).

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` = 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 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`.

Example

A set of linear systems is solved successively. The right-hand side vector is perturbed after solving the system each of the fisrt 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)
```

Comments

Informational error

Type	Code	
4	1	The factored matrix has a diagonal element close to zero.

Description

This routine 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`, page 290, or `LFTQH`, page 288. 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`, page 292, 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`.

LFDQH

Computes the determinant of a complex Hermitian positive definite matrix given the $R^T R$ 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 $1.0 \le |$`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) = `DET1 * 10`$^{\texttt{DET2}}$.

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 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`.

Example

The determinant is computed for a 5 × 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.
```

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, page 290, or LFTQH, page 288.

The formula det A = det R^H det R = (det R)□ is used to compute the determinant. Since the determinant of a triangular matrix is the product of the diagonal elements,

$$\det R = \prod\nolimits_{i=1}^{N} R_{ii}$$

LFDQH is based on the LINPACK routine CPBDI; see Dongarra et al. (1979).

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 = size (B,1).

NZ — The number of nonzero coefficients in the linear system. (Input)
Default: NZ = 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`.

Example

As an example consider the 6 × 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 $x^T = (1, 2, 3, 4, 5, 6)$ 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	5	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
```

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 + MAXNZ.

   IWK — Integer work vector of length LIWK.

   LIWK — The length of IWK, LIWK should be at least 17N + 4 * MAXNZ.

   The workspace limit is determined by MAXNZ, where

   MAXNZ = MIN0(LWK-2N, INT(0.25(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 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

IPARAM(2)	**Action**
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.

IPARAM(5)	**Action**
0	Default limit, see Comment 1.
integer	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.

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

$$A_{\texttt{irow}(i),\texttt{icol}(i)} = \texttt{a}(i), \qquad i = 1, \ldots, \texttt{nz}$$

with all other entries in A zero.

The routine `LSLXG` solves a system of linear algebraic equations having a real sparse coefficient matrix. It first uses the routine `LFTXG` (page 301) to perform an LU factorization of the coefficient matrix. The solution of the linear system is then found using `LFSXG` (page 306).

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

$$P\,AQ = LU$$

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:

$$1)\ Lz = Pb$$

$$2)\ Uy = z$$

$$3)\ x = Qy$$

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` = 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`.
Default: `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`.

Example

As an example, consider the 6 × 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), JPVT(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, JPVT)
!
      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 ', JPVT, 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
```

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 ∗ 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.

IPARAM(2)	**Action**
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.

IPARAM(5)	**Action**
0	Default limit, see Comment 1.
integer	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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

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

$$P\,AQ = LU$$

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` (page 306) by the following calculations:

$$1)\ Lz = Pb$$

$$2)\ Uy = z$$

$$3)\ x = Qy$$

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`.

Example

As an example, consider the 6 × 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), JPVT(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
```

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

$$A_{\texttt{irow}(i),\texttt{icol}(i)} = \texttt{a}(i), \qquad i = 1, \dots, \texttt{nz}$$

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` (page 301). The solution of the linear system is then found by the forward and backward substitution. The algorithm can be expressed as

$$P\,AQ = LU$$

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).

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` = size (`B`,1).

NZ — The number of nonzero coefficients in the linear system. (Input)
Default: `NZ` = 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`.

Example

As an example, consider the 6 × 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)
```

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 2`N`+ `MAXNZ`.

   IWK — Integer work vector of length `LIWK`.

   LIWK — The length of `IWK`, `LIWK` should be at least 17`N` + 4 `*` `MAXNZ`.

   The workspace limit is determined by `MAXNZ`, where

   `MAXNZ = MIN0(LWK-2N, INT(0.25(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.

IPARAM(2)	**Action**
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.

IPARAM(5)	**Action**
0	Default limit, see Comment 1.
integer	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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

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` (page 314) to perform an *LU* factorization of the coefficient matrix. The solution of the linear system is then found using `LFSZG` (page 309). 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

$$P\,AQ = LU$$

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:

$$1)\ Lz = Pb$$

$$2)\ Uy = z$$

$$3)\ x = Qy$$

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`.

Example

As an example, the following 6 × 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
```

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 * 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 `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.

IPARAM(2)	**Action**
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.

IPARAM(5)	**Action**
0	Default limit, see Comment 1.
integer	This integer value replaces the default workspace limit. When `L2TZG` is called, the values of `LWK` and `LIWK` are used instead of `IPARAM`(5).

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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

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

$$P\,AQ = LU$$

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 LFSZG (page 319) by the following calculations:

$$1)\ Lz = Pb$$

$$2)\ Uy = z$$

$$3)\ x = Qy$$

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` = 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`.

Example

As an example, consider the 6 × 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, IPVT(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, IPVT, JPVT)
!
      IPATH = 1
      DO 10 I = 1,2
!                                  Solve A * X(i) = B(i)
         CALL LFSZG (NFAC, NL, FACT, IRFAC, JCFAC, IPVT, JPVT,&
                    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)
```

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

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` (page 314). The solution of the linear system is then found by the forward and backward substitution. The algorithm can be expressed as

$$P\,AQ = LU$$

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).

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 nonzeroes 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` = 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 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`.

Example

As an example consider the 5 × 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
```

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 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.

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	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 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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\mathtt{irow}(i) \geq \mathtt{jcol}(i) \quad i = 1, \ldots, \mathtt{nz}$$

with all other entries in the lower triangle of A zero.

The subroutine `LSLXD` solves a system of linear algebraic equations having a real, sparse and positive definite coefficient matrix. It first uses the routine `LSCXD` (page 327) to compute a symbolic factorization of a permutation of the coefficient matrix. It then calls `LNFXD` (page 331) to perform the numerical factorization. The solution of the linear system is then found using `LFSXD` (page 336).

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

$$P\,AP^T = LL^T$$

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:

$$1)\ Ly_1 = Pb$$

$$2)\ L^T y_2 = y_1$$

$$3)\ x = P^T y_2$$

The routine `LFSXD` accepts *b* and the permutation vector which determines *P*. It then returns *x*.

LSCXD/DLSCXD

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`) ≥ `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` = 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` = 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.

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
```

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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\mathtt{irow}(i) \geq \mathtt{jcol}(i) \quad i = 1, \ldots, \mathtt{nz}$$

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` (page 331) produces the numerical entries in L so that we have

$$P\,AP^T = LL^T$$

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.

LNFXD

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` = size (`IPER`,1).

NZ — The number of nonzero coefficients in the linear system. (Input)
Default: `NZ` = 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 LNFXD (A, IROW, JCOL, MAXSUB, NZSUB, INZSUB, MAXNZ, ILNZ, IPER, INVPER, ISPACE, DIAGNL, RLNZ, RPARAM [,…])`

Specific: The specific interface names are `S_LNFXD` and `D_LNFXD`.

FORTRAN 77 Interface

Single: `CALL LNFXD (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 `DLNFXD`.

Example

As an example, consider the 5 × 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`, page 327, 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 LNFXD (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
```

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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\mathtt{irow}(i) \geq \mathtt{jcol}(i) \quad i = 1, \ldots, \mathtt{nz}$$

with all other entries in the lower triangle of A zero. The routine `LNFXD` produces the Cholesky factorization of $P\,AP^T$ given the symbolic factorization of A which is computed by `LSCXD` (page 327). That is, this routine computes L which satisfies

$$P\,AP^T = LL^T$$

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 $R(i), \ldots, R(i + k)$ 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.

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`.

Example

As an example, consider the 5 × 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 `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 LFSXD_INT
      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, 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 LNFXD (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
```

Comments

Informational error

Type	Code	
4	1	The input matrix is numerically singular.

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

$$A_{\mathtt{irow}i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\texttt{irow}(i) \geq \texttt{jcol}(i) \quad i = 1, \ldots, \texttt{nz}$$

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` (page 327) followed by `LNFXD` (page 331). 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

$$P\,AP^T = LL^T$$

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:

$$1)\ Ly_1 = Pb$$

$$2)\ L^T y_2 = y_1$$

$$3)\ x = P^T y_2$$

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 nonzeroes 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`.

Example

As an example, consider the 3 × 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)
```

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.

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

$$A_{\texttt{irow}(i),\texttt{icol}(i)} = \texttt{a}(i), \qquad i = 1, \dots, \texttt{nz}$$

$$\texttt{irow}(i) \ge \texttt{jcol}(i) \quad i = 1, \dots, \texttt{nz}$$

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` (page 327) to compute a symbolic factorization of a permutation of the coefficient matrix. It then calls `LNFZD` (page 344) to perform the numerical factorization. The solution of the linear system is then found using `LFSZD` (page 349).

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

$$P\,AP^T = LL^H$$

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.

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 `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 `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` = size (`IPER`,1).

NZ — The number of nonzero coefficients in the linear system. (Input)
Default: `NZ` = 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`.

Example

As an example, consider the 3 × 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)
```

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` + 3`NZ`.

IWK — Integer work vector of length `LIWK`.

LIWK — The length of `IWK`, `LIWK` should be at least 2`N`.

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.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\mathtt{irow}(i) \ge \mathtt{jcol}(i) \quad i = 1, \ldots, \mathtt{nz}$$

with all other entries in the lower triangle of A zero.

The routine `LNFZD` produces the Cholesky factorization of $P\,AP^T$ given the symbolic factorization of A which is computed by `LSCXD` (page 327). That is, this routine computes L which satisfies

$$P\,AP^T = LL^H$$

The diagonal of L is stored in `DIAGNL` and the strictly lower triangular part of L is stored in compressed subscript form in R = `RLNZ` as follows. The nonzeros in the jth column of L are stored in locations $R(i), \ldots, R(i + k)$ 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.

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 LNFZD/DLNFZD. (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 LNFZD/DLNFZD. (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, MAXZUB, 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`.

Example

As an example, consider the 3 × 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

$$x_2^T = (3+3i, 2+2i, 1+1i)$$

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)
```

Comments

Informational error

Type	Code	
4	1	The input matrix is numerically singular.

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

$$A_{\mathtt{irow}(i),\mathtt{icol}(i)} = \mathtt{a}(i), \qquad i = 1, \ldots, \mathtt{nz}$$

$$\mathtt{irow}(i) \geq \mathtt{jcol}(i) \quad i = 1, \ldots, \mathtt{nz}$$

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` (page 327) followed by `LNFZD` (page 344). 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

$$P\,AP^T = LL^H$$

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:

$$1)\ Ly_1 = Pb$$

$$2)\ L^H y_2 = y_1$$

$$3)\ x = P^T y_2$$

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 2`N` – 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` = (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`.

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
```

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), …, A(1, N), A(2, 1), …, A(N, 1) need to be stored.

Description

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

$$A = \begin{bmatrix} p_0 & p_1 & p_2 & p_4 \\ p_{-1} & p_0 & p_1 & p_2 \\ p_{-2} & p_{-1} & p_0 & p_1 \\ p_{-3} & p_{-2} & p_{-1} & p_0 \end{bmatrix}$$

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.

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` = 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`.

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)
```

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.

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), …, A(1, N), A(2, 1), …, A(N, 1) need to be stored.

Description

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

$$A = \begin{bmatrix} p_0 & p_1 & p_2 & p_4 \\ p_{-1} & p_0 & p_1 & p_2 \\ p_{-2} & p_{-1} & p_0 & p_1 \\ p_{-3} & p_{-2} & p_{-1} & p_0 \end{bmatrix}$$

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.

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` = 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`.

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)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of L2LCC/DL2LCC. The reference is:

   CALL L2LCC (N, A, B, IPATH, X, ACOPY, 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), …, A(1, N) need to be stored.

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

$$(Ax)_j = \sum_{i=1}^{N} p_{i-j+1} x_i = \sum_{i=1}^{N} q_{j-i+1} x_i = (q * x)_i$$

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,

$$\hat{x} = \hat{b} \varnothing \hat{q}$$

where $\varnothing$ 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.

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` = 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_LPCGRC.`

FORTRAN 77 Interface

Single: `CALL PCGRC (IDO, N, X, P, R, Z, RELERR, ITMAX)`

Double: The double precision name is `DPCGRC`.

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` on page 365.

```
      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
```

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 ITMAX iterations.

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` >> 1. See the references for further information.

Let *M* be the preconditioning matrix, let *b*, *p*, *r*, *x* and *z* be vectors and let τ be the desired relative error. Then the algorithm used is as follows.

```
$\lambda = -1$
$p_0 = x_0$
$r_1 = b - Ap$
For $k = 1, \ldots,$ 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 $(\|z_k\|_2 \le \tau(1 - \lambda)\|x_k\|_2)$ Then
        Recompute $\lambda$
        If $(\|z_k\|_2 \le \tau(1 - \lambda)\|x_k\|_2)$ Exit
  End if end loop
```

Here λ 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)

$$\frac{\|x_k - x\|_M}{\|x\|_M} \le \frac{1}{1 - \lambda_{\max}(G)} \frac{\|z_k\|_M}{\|x_k\|_M}$$

Where

$$\|x\|_M^2 = x^T M x$$

It is known that

$$\lambda_{\max}(T_1) \le \lambda_{\max}(T_2) \le \cdots \le \lambda_{\max}(G) < 1$$

where the T_n are the symmetric, tridiagonal matrices

$$T_n = \begin{bmatrix} \mu_1 & \omega_2 & & & \\ \omega_2 & \mu_2 & \omega_3 & & \\ & \omega_3 & \mu_3 & \omega_4 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

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.

Example 2

In this example, a more complicated preconditioner is used to find the solution of a linear system which occurs in a finite-difference solution of Laplace's equation on a 4 × 4 grid. The matrix is

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & & \\ 0 & -1 & 4 & -1 & 0 & -1 & & & \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ & & & -1 & 0 & -1 & 4 & -1 & 0 \\ & & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix}$$

The preconditioning matrix M is the symmetric tridiagonal part of A,

$$M = \begin{bmatrix} 4 & -1 & & & & & & & \\ -1 & 4 & -1 & & & & & & \\ & -1 & 4 & -1 & & & & & \\ & & -1 & 4 & -1 & & & & \\ & & & -1 & 4 & -1 & & & \\ & & & & -1 & 4 & -1 & & \\ & & & & & -1 & 4 & -1 & \\ & & & & & & -1 & 4 & -1 \\ & & & & & & & -1 & 4 \end{bmatrix}$$

Note that *M*, called `PRECND` in the program, is factored once.

```
      USE IMSL_LIBRARIES
      INTEGER    LDA, LDPRE, N, NCODA, NCOPRE
      PARAMETER  (N=9, NCODA=3, NCOPRE=1, LDA=2*NCODA+1,&
                 LDPRE=NCOPRE+1)
!
      INTEGER    IDO, ITMAX
      REAL       A(LDA,N), P(N), PRECND(LDPRE,N), PREFAC(LDPRE,N),&
                 R(N), RCOND, RELERR, X(N), Z(N)
!                                 Set A in band form
      DATA A/3*0.0, 4.0, -1.0, 0.0, -1.0, 2*0.0, -1.0, 4.0, -1.0, 0.0,&
          -1.0, 2*0.0, -1.0, 4.0, -1.0, 0.0, -1.0, -1.0, 0.0, -1.0,&
          4.0, -1.0, 0.0, -1.0, -1.0, 0.0, -1.0, 4.0, -1.0, 0.0,&
          -1.0, -1.0, 0.0, -1.0, 4.0, -1.0, 0.0, -1.0, -1.0, 0.0,&
          -1.0, 4.0, -1.0, 2*0.0, -1.0, 0.0, -1.0, 4.0, -1.0, 2*0.0,&
          -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`.

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)
!                                  (   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)
!                                 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
```

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	
4	1	The diagonal contains a zero.
4	2	The diagonal elements have different signs.
4	3	No convergence after `ITMAX` iterations.
4	4	The linear system is not definite.
4	5	The linear system is singular.

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, page 359.

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 » 1. See the references for further information.

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 $M^{-1}P$. When IDO = 3, it contains $AM^{-1}P$. When IDO = 0, it is ignored.

TOL — Stopping tolerance. (Input/Output)
The algorithm attempts to generate a solution x such that $|b - Ax| \le$ TOL*$|b|$. On output, TOL contains the final residual norm.

Optional Arguments

N — Order of the linear system. (Input)
Default: N = 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`.

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
```

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, USRNRM, 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 Krylor subspace dimension, i.e., the maximum allowable number of GMRES iterations before restarting. It must satisfy $1 \le \text{KDMAX} \le N$.
   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(INXY),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 *WORK***
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`

Description

The routine `GMRES` implements restarted `GMRES` with reverse communication to generate an approximate solution to $Ax = b$. It is based on GMRESD 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.

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`, page 297. Jacobi preconditioning is used, i.e. the preconditioning matrix M is the diagonal matrix with $M_{ii} = A_{ii}$, for $i = 1, \ldots, n$.

```
      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, 0.16666667/
      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)
!                                 SPECIFICATIONS FOR ARGUMENTS
      REAL       P(*), Z(*)
!                                 SPECIFICATIONS FOR PARAMETERS
      INTEGER    N
      PARAMETER  (N=6)
!                                 SPECIFICATIONS FOR LOCAL VARIABLES
      INTEGER    I
      INTEGER    IROW(NZ), JCOL(NZ)
      REAL       A(NZ)
      SAVE       A, IROW, JCOL
!                                 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 50×50 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, TPRE, 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
      TPRE    = CPSEC()
!                                 Compute the LDU factorization of M
!
      CALL LSLCR (C, A, B, Y, U, IR, IS, IJOB=6)
   20 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 20
      ELSE IF (IDO .EQ. 2) THEN
!
!                                 Set z = inv(M)*p
!
         CALL SCOPY (N, P, 1, Z, 1)
         CALL LSLCR (C, A, B, Z, U, IR, IS, IJOB=5)
         GO TO 20
      ELSE IF (IDO .EQ. 3) THEN
!
!                                 Set z = A*inv(M)*p
!
         CALL LSLCR (C, A, B, P, U, IR, IS, IJOB=5)
         CALL AMULTP (K, P, Z)
         GO TO 20
      END IF
      TPRE = CPSEC() - TPRE
      WRITE (NOUT,'(A35, I4)') 'Iterations, with preconditioning = ',&
                             INFO(8)
      WRITE (NOUT,'(A45, F10.5)') '(Precondition time)/(No '// &
                                'precondition time) = ', TPRE/TNOPRE
!
      END
!
      SUBROUTINE AMULTP (K, P, Z)
      USE IMSL_LIBRARIES
!                                 Specifications for arguments
      INTEGER    K
      REAL       P(*), Z(*)
!                                 Specifications for local variables
      INTEGER    I, N
!
      N = K*K
!                                 Multiply by diagonal blocks
!
      CALL SVCAL (N, 4.0, P, 1, Z, 1)
      CALL SAXPY (N-1, -1.0, P(2:(N-1)), 1, Z, 1)
      CALL SAXPY (N-1, -1.0, P, 1, Z(2:(N-1)), 1)
!
!                                 Correct for terms not properly in
!                                 block diagonal
```

```
      DO 10  I=K, N - K, K
         Z(I)   = Z(I) + P(I+1)
         Z(I+1) = Z(I+1) + P(I)
   10 CONTINUE
!                                Do the super and subdiagonal blocks,
!                                the -I's
!
      CALL SAXPY (N-K, -1.0, P((K+1):(N-K)), 1, Z, 1)
      CALL SAXPY (N-K, -1.0, P, 1, Z((K+1):(N-K)), 1)
!
      RETURN
      END
```

Output

```
Iterations, no preconditioner =  329
Iterations, with preconditioning =  192
(Precondition time)/(No precondition time) =    0.66278
```

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`.

Example

Consider the problem of finding the coefficients c_i in

$$f(x) = c_0 + c_1 x + c_2 x^2$$

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
```

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. 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

$$\left|r_{k+1,k+1}\right| \le \text{TOL} * \left|r_{11}\right|$$

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.

Description

Routine LSQRR solves the linear least-squares problem. The routine LQRRR, page 392, is first used to compute the *QR* decomposition of *A*. Pivoting, with all rows free, is used. Column *k* is in the basis if

$$\left|R_{kk}\right| \le \tau \left|R_{11}\right|$$

with τ = TOL. The truncated least-squares problem is then solved using IMSL routine LQRSL, page 398. 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.

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, …, 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 = 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.

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

$$\min_{x_j \in \mathbf{R}^n} \left\| Ax_j - b_j \right\|_2$$

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 exits 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
```

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` × (`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 (`NCA` + `NUMEXC` + 1) `*` (`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`.

Description

Routine `LSQRR` solves the linear least-squares problem. The routine `LQRRR`, page 392, is first used to compute the QR decomposition of A. Pivoting, with all rows free, is used. Column k is in the basis if

$$|R_{kk}| \le \tau |R_{11}|$$

with τ = TOL. The truncated least-squares problem is then solved using IMSL routine LQRSL, page 398. 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.

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 = 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 — 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.

Example

This example solves the linear least-squares problem with A, an 8×4 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 LSBRR (A, B, X, TOL=TOL, RES=RES, KBASIS=KBASIS)
!                                 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        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
```

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

   $$|r_{k+1,k+1}| \le \mathrm{TOL} * |r_{11}|$$

   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`. 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.

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).

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 to −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.

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
```

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	
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 sqrt(AMACH (4)).

   5 The value of this option is the absolute rank determination tolerance to be used. Default value is sqrt(AMACH (4)).

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

$$b_l \le C_x \le b_u$$

$$x_l \le x \le x_u$$

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 $Cx - \mathrm{y} = 0$ is a set of equality constraints. These constraints are realized by heavy weighting, i.e. a penalty method, Hanson, (1986, pages 826–834).

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` = 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).

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.

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

$$A^T A = (QRP^{-1})^T (QRP^{-1}) = P^{-T} R^T (Q^T Q)RP^{-1} = P R^T RP^T$$

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

$$x^T \left(A^T A\right)^{-1} x = x^T P R_1^{-1} R_1^{-T} P^{-1} x = \left\| R_1^{-T} P^{-1} x \right\|_2^2$$

In the following program, first the vector $t = P^{-1} x$ is computed. Then

$$t := R_1^{-T} t$$

Finally,

$$q = \|t\|^2$$

```
      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
```

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 2NCA – 1. Only 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, $k = 1, \ldots, \min(\text{NRA}, \text{NCA})$ 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 (page 396) 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 QR(1, 1)/QR(K, K), where K = MIN(NRA, NCA). This estimate can be used to select the number of columns, KBASIS, used in the solution step computed with routine LQRSL (page 398).

Description

The routine LQRRR computes the QR decomposition of a matrix using Householder transformations. It is based on the LINPACK routine SQRDC; see Dongarra et al. (1979).

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 $k = 1, 2, \ldots, \min(\text{NRA}, \text{NCA})$, 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, described on page 396, 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.

LQERR

Accumulates the orthogonal matrix Q from its factored form given the QR factorization of a rectangular matrix A.

Required Arguments

QR — Real NRQR by NCQR matrix containing 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)

QRAUX — Real vector of length NCQR containing information about the orthogonal part of the decomposition in the first min(NRQR, NCQR) position as output from routine LQRRR/DLQRRR. (Input)

Q — Real NRQR by NRQR 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: NRQR = size (QR,1).

NCQR — Number of columns in QR. (Input)
Default: NCQR = size (QR,2).

LDQR — Leading dimension of QR exactly as specified in the dimension statement of the calling program. (Input)
Default: LDQR = size (QR,1).

LDQ — Leading dimension of Q exactly as specified in the dimension statement of the calling program. (Input)
Default: LDQ = 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`.

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, NRA
         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
```

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.

Description

The routine LQERR accumulates the Householder transformations computed by IMSL routine LQRRR, page 392, to produce the orthogonal matrix Q.

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(`NRA`, `NCA`), where `NCA` is the number of columns in matrix A. The value `NCA` is an argument to routine `LQRRR` (page 392). 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 — `NRA` by `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^Tb;
`K` $\neq 0$ means compute $\mathrm{Q}^T\mathrm{b}$ and x;
`L` $\neq 0$ means compute $\mathrm{Q}^T\mathrm{b}$ and $b - Ax$;
`M` $\neq 0$ means compute Q^Tb 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^Tb if requested in the option `IPATH`. (Output)

X — Vector of length `KBASIS` containing the solution of the least-squares problem $A_kx = b$, 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`.

Example

Consider the problem of finding the coefficients c_i in

$$f(x) = c_0 + c_1 x + c_2 x^2$$

given data at $x_i = 2_i$, $\iota = 1, 2, 3, 4$, 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.

Comments

1. Informational error

   Type Code

   4 1 Computation of the least-squares solution of AK * 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)), …, *A*(IPVT (KBASIS))), where *A*(IPVT(I)) is the IPVT(I)-th column of the original matrix.

Description

Routine `LQRSL` is based on the LINPACK routine `SQRSL`, see Dongarra et al. (1979).

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` (page 392) 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

$$(PR^T)(Q^TQ)R(P^Tx) = (PR^T)Q^T b$$

If A^TA is nonsingular, then R is also nonsingular and the normal equations can be written as $R(P^Tx) = Q^T b$. `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^Tb and Ax, the orthogonal projection of x onto the column space of A.

LUPQR

Computes an updated QR factorization after the rank-one matrix αxy^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^Tx.

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^Tx.
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.

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

$$AP + \alpha xy^T = \left(A + \alpha x\left(Py\right)^T\right)P = \tilde{Q}\tilde{R}$$

The IMSL routine `PERMU` (See Chapter 11, 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
```

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`).

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 $e_1 = (1, 0, \ldots, 0)^T$. 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).

LCHRG

Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.

Required Arguments

A — `N` by `N` symmetric positive semidefinite 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` = 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).

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` = 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`.

Example

Routine `LCHRG` can be used together with the IMSL routines `PERMU` (see Chapter 11) and `LFSDS` (page 148) 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
```

Comments

1. Informational error

Type	Code	
4	1	The input matrix is not positive semidefinite.

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 semidefinite 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)
```

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.

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 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 = size (R,2).

LDR — Leading dimension of R exactly as specified in the dimension statement of the calling program. (Input)
Default: LDR = size (R,1).

LDRNEW — Leading dimension of `RNEW` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDRNEW` = 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 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`.

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
```

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 RNEW.

LUPCH determines an orthogonal matrix U as the product $G_N \ldots G_1$ 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

$$R^T R + xx^T = \tilde{R}^T \tilde{R}$$

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 $c_i = \cos\theta_i =$ `CS(I)`, $s_i = \sin\theta_i =$ `SN(I)` for some θ_i.

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 $R^T R - XX^T$. (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` = size (`R`,2).

LDR — Leading dimension of `R` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDR` = size (`R`,1).

LDRNEW — Leading dimension of `RNEW` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDRNEW` = 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`.

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
```

Comments

Informational error

Type	Code	
4	1	$R^TR - XX^T$ is not positive definite. R cannot be downdated.

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 $c_i = \cos\theta_i$ = `CS(I)`, $s_i = \sin\theta_i$ = `SN(I)` for some θ_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

$$x = \left(R^T 0\right) \begin{bmatrix} a \\ \alpha \end{bmatrix} = \left(R^T 0\right) U^T U \begin{bmatrix} a \\ \alpha \end{bmatrix} = \left(\tilde{R}^T \tilde{x}\right) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \tilde{x}$$

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 `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 — 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` = size (`A`,1).

NCA — Number of columns in the matrix `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 tolerance used to determine when a singular value is negligible. (Input)
If `TOL` is positive, then a singular value σ_i considered negligible if $\sigma_i \le$ `TOL` . If `TOL` is negative, then a singular value σ_i considered negligible if $\sigma_i \le |\mathtt{TOL}| * \|\mathtt{A}\|_\infty$. In this case, |`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(`NRA`, `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`.

Example

This example computes the singular value decomposition of a 6 × 4 matrix *A*. The matrices *U* and *V* containing the left and right singular vectors, respectively, and the diagonal of Σ, 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)
      CALL WRRRN ('S', S, 1, NCA, 1)
      CALL WRRRN ('V', V)
!
```

```
      END
```

Output

```
IRANK =   4
                              U
          1         2         3         4         5         6
1  -0.3805    0.1197    0.4391   -0.5654    0.0243   -0.5726
2  -0.4038    0.3451   -0.0566    0.2148    0.8089    0.1193
3  -0.5451    0.4293    0.0514    0.4321   -0.5723    0.0403
4  -0.2648   -0.0683   -0.8839   -0.2153   -0.0625   -0.3062
5  -0.4463   -0.8168    0.1419    0.3213    0.0621   -0.0799
6  -0.3546   -0.1021   -0.0043   -0.5458   -0.0988    0.7457

                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
```

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 — `NRA` × `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 `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 `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.

Description

The routine `LSVRR` is based on the LINPACK routine `SSVDC`; see Dongarra et al. (1979).

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

$$U^T A V = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \ge p \\ \begin{bmatrix} \Sigma \, 0 \end{bmatrix} & \text{if } n \le p \end{cases}$$

where $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m)$, and $m = \min(n, p)$. The scalars $\sigma_1 \ge \sigma_2 \ge \ldots \ge \sigma_m \ge 0$ 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 σ_k that is larger than a tolerance η. If τ is the parameter `TOL` in the program, then

$$\eta = \begin{cases} \tau & \text{if } \tau > 0 \\ |\tau| \, \|A\|_\infty & \text{if } \tau < 0 \end{cases}$$

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 = size (A,1).

NCA --- Number of columns in the matrix 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 — 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 ≤ TOL . If TOL is negative, then a singular value SI is considered negligible if
SI ≤ |TOL|*(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 = 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 = 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.

Example

This example computes the singular value decomposition of a 6 × 3 matrix *A*. The matrices *U* and *V* containing the left and right singular vectors, respectively, and the diagonal of Σ, 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)
      CALL WRCRN ('S', S, 1, NCA, 1)
      CALL WRCRN ('V', V)
!
      END
```

Output

```
IRANK =   3
                                           U
                   1                  2                  3                  4
1 ( 0.1968, 0.2186) ( 0.5011, 0.0217)  (-0.2007,-0.1003)  (-0.2036, 0.0405)
2 ( 0.3443,-0.3542) (-0.2933, 0.0248)  ( 0.1155,-0.2338)  (-0.2316, 0.0287)
3 ( 0.1457, 0.2307) (-0.5424, 0.1381)  (-0.4361,-0.4407)  ( 0.0281,-0.3088)
4 ( 0.3016,-0.0844) ( 0.2157, 0.2659)  (-0.0523,-0.0894)  ( 0.8617, 0.0223)
5 ( 0.2283,-0.6008) (-0.1325, 0.1433)  ( 0.3152,-0.0090)  (-0.0392,-0.0145)
6 ( 0.2876,-0.0350) ( 0.4377,-0.0400)  ( 0.0458,-0.6205)  (-0.2303, 0.0924)

                    5                  6
1  ( 0.4132,-0.0985)  (-0.6017, 0.1612)
2  (-0.5061, 0.0198)  (-0.5380,-0.0317)
3  ( 0.2043,-0.1853)  ( 0.1012, 0.2132)
4  (-0.1272,-0.0866)  (-0.0808,-0.0266)
5  ( 0.6482,-0.1033)  ( 0.0995,-0.0837)
6  (-0.1412, 0.1121)  ( 0.4897,-0.0436)

                            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)
```

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, ACOPY, 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 ACOPY 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 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 LSVCR. Additional memory allocation for ACOPY and option value restoration are done automatically in LSVCR. Users directly calling L2VCR 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 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.

Description

The IMSL routine LSVCR is based on the LINPACK routine CSVDC; see Dongarra et al. (1979).

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

$$U^T AV = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \ge p \\ \begin{bmatrix} \Sigma\, 0 \end{bmatrix} & \text{if } n \le p \end{cases}$$

where $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m)$, and $m = \min(n, p)$. The scalars $\sigma_1 \ge \sigma_2 \ge \ldots \ge 0$ 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 σ_k which are larger than a tolerance η. If τ is the parameter TOL in the program, then

$$\eta = \begin{cases} \tau & \text{if } \tau > 0 \\ |\tau| \, \|A\|_\infty & \text{if } \tau < 0 \end{cases}$$

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` = size (`A`,1).

NCA — Number of columns in the matrix `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 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 σ_i considered negligible if $\sigma_i \le$ `TOL` . If `TOL` is negative, then a singular value σ_i considered negligible if $\sigma_i \le$ |`TOL`| `*` $\|$`A`$\|_\infty$. In this case, |`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)

LDGINV — Leading dimension of `GINVA` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDGINV` = size (`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`.

Example

This example computes the generalized inverse of a 3 × 2 matrix A. The rank k = `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
```

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 $\texttt{NRA}^2 + \texttt{NCA}^2 + \min(\texttt{NRA} + 1, \texttt{NCA}) + \texttt{NRA} + \texttt{NCA} + \max(\texttt{NRA}, \texttt{NCA}) - 2$.

2. Informational error

Type	Code	
4	1	Convergence cannot be achieved for all the singular values and their corresponding singular vectors.

Description

Let k = `IRANK`, the rank of A; let n = `NRA`, the number of rows in A; let p = `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` (page 415) 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 $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m)$, $m = \min(n, p)$, such that $U^T AV = [\Sigma, 0]$ if $n \le p$ and $U^T AV = [\Sigma, 0]^T$ if $n \ge p$. Only the first p columns of U are computed. The rank k is estimated by counting the number of nonnegligible σ_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 $\Sigma_1 = \text{diag}(\sigma_1, \ldots, \sigma_k)$. The Moore-Penrose generalized inverse of A is

$$A^{\dagger} = V_1 \Sigma_1^{-1} U_1^T$$

Chapter 2: Eigensystem Analysis

Routines

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 λ 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 λ 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.

Symmetric and Hermitian Eigensystems			
	Symmetric Full	**Symmetric Band**	**Hermitian Full**
All eigenvalues	EVLSF p. 469	EVLSB p. 485	EVLHF p. 502
All eigenvalues and eigenvectors	EVCSF p. 471	EVCSB p. 487	EVCHF p. 505
Extreme eigenvalues	EVASF p. 473	EVASB p. 490	EVAHF p. 508
Extreme eigenvalues and eigenvectors	EVESF p. 475	EVESB p. 492	EVEHF p.510
Eigenvalues in an interval	EVBSF p. 478	EVBSB p. 495	EVBHF p. 513
Eigenvalues and eigevectors in an interval	EVFSF p. 480	EVFSB p. 498	EVFHF p 515
Performance index	EPISF p. 483	EPISB p. 501	EPIHF p. 518

General Eigensystems				
	Real General	**Complex General**	**Real Hessenberg**	**Complex Hessenberg**
All eigenvalues	EVLRG p. 455	EVLCG p. 462	EVLRH p. 455	EVLCH p. 525
All eigenvalues and eigenvectors	EVCRG p. 457	EVCCG p. 464	EVCRH p. 522	EVCCH p. 526
Performance index	EPIRG p. 460	EPICG p. 467	EPIRG p. 460	EPICG p. 467

Generalized Eigensystems A*x* = λB*x*			
	Real General	**Complex General**	**A Symmetric B Positive Definite**
All eigenvalues	`GVLRG` p. 529	`GVLCG` p. 537	`GVLSP` p. 544
All eigenvalues and eigenvectors	`GVCRG` p. 531	`GVCCG` p. 540	`GVCSP` p. 547
Performance index	`GPIRG` p. 535	`GPICG` p. 542	`GPISP` p. 549

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 $\| E\|_2 \le f(n) \|A\|_2 \varepsilon$. The value of $f(n)$ depends on the algorithm but is typically a small fractional power of n. The parameter ε is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan [1989, page 342],

$$\min\left|\tilde{\lambda} - \lambda\right| \le \kappa\left(X\right)\left\|E\right\|_2 \qquad \text{for all } \lambda \text{ in } \sigma\left(\mathrm{A}\right)$$

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 τ using `EPI**`. The performance index is defined by Smith et al. (1976, pages 124–126) to be

$$\tau = \max_{1 \le j \le n} \frac{\left\|A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j\right\|_1}{10n\varepsilon\left\| A\right\|_1 \left\|\tilde{x}_j\right\|_1}$$

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

$$\|x\|_1 = \sum_{k=1}^{N} \left(|\Re x_k| + |\Im x_k| \right)$$

The performance index τ is related to the error analysis because

$$\left\| E\tilde{x}_j \right\|_2 \doteq \left\| A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j \right\|_2$$

where E is the "nearby" matrix discussed above.

While the exact value of τ is machine and precision dependent, the performance of an eigensystem analysis routine is defined as excellent if $\tau < 1$, good if $1 \le \tau \le 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of τ can serve as a warning that there is a blunder in the calculation. There are also similar routines `GPI**` 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 τ 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 λ_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 λ_j correspond to those for μ_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`, page 544, and `GVCSP`, page 547, use $P = R^T$ and $Q = R$ where R is an upper triangular matrix obtained from a Cholesky decomposition, $B = R^TR$. The matrix $F = R^{-T} AR^{-1}$ is symmetric and real. Computation of the eigenvalue-eigenvector expansion for F is based on routine `EVCSF`, page 471.

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 `LIN_EIG_SELF`

Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`Lin_eig_self_set_small`	1
`s_, d_, c_, z_`	`Lin_eig_self_overwrite_input`	2
`s_, d_, c_, z_`	`Lin_eig_self_scan_for_NaN`	3
`s_, d_, c_, z_`	`Lin_eig_self_use_QR`	4
`s_, d_, c_, z_`	`Lin_eig_self_skip_Orth`	5
`s_, d_, c_, z_`	`Lin_eig_self_use_Gauss_elim`	6
`s_, d_, c_, z_`	`Lin_eig_self_set_perf_ratio`	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`.

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`, Chapter 10.

```
      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.
```

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.

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, $A = VDV^T$, 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
```

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:

$$\left(A - d_i I\right) v_i = b_i$$

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`, Chapter 10.

```
      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
```

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,

$$C = DV^T AVD$$

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.
```

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.

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 2×2 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

$$S = A^T U - U\overline{E}$$

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
`s_, d_, c_, z_`	`lin_eig_gen_set_small`	1
`s_, d_, c_, z_`	`lin_eig_gen_overwrite_input`	2
`s_, d_, c_, z_`	`lin_eig_gen_scan_for_NaN`	3
`s_, d_, c_, z_`	`lin_eig_gen_no_balance`	4
`s_, d_, c_, z_`	`lin_eig_gen_set_iterations`	5
`s_, d_, c_, z_`	`lin_eig_gen_in_Hess_form`	6
`s_, d_, c_, z_`	`lin_eig_gen_out_Hess_form`	7
`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 2×2 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 × 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`.

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`, Chapter 10.

```
      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.
```

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 Hessenberg matrix so that 2×2 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.

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 $a_{1,j} = -b_j$, $i = 1, \ldots, n$, 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 $|f(e_i)|$, $i = 1, \ldots, n$, at the eigenvalues of the matrix, which are the roots of $f(z)$. Also, see `operator_ex30`, Chapter 10.

```
      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

$$\left(T + hI\right) y = \overline{W}^T b$$

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`, Chapter 10.

```
     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 η is "small" but still significantly larger than machine precision. The matrix B satisfies $\|B\| \le \|A\|$. 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 η. With

$$e_i(\eta) \approx e_i + \eta \dot{e}_i \eta$$

and normalized eigenvectors, the bound

$$|\dot{e}_i| \le \frac{\|A\|}{|u_i^* v_i|}$$

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(1d0)), parameter :: one=1d0
      real(kind(1d0)) a(n,n), c(n,n), variation(n), y(n*n), temp(n), &
              norm_of_a, eta
      complex(kind(1d0)), 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.
```

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.

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 α and β, 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)

BETAV — 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 `lin_geig_gen`

Option Prefix = ?	Option Name	Option Value
`s_, d_, c_, z_`	`lin_geig_gen_set_small`	1
`s_, d_, c_, z_`	`lin_geig_gen_overwrite_input`	2
`s_, d_, c_, z_`	`lin_geig_gen_scan_for_NaN`	3
`s_, d_, c_, z_`	`lin_geig_gen_self_adj_pos`	4
`s_, d_, c_, z_`	`lin_geig_gen_for_lin_sol_self`	5
`s_, d_, c_, z_`	`lin_geig_gen_for_lin_eig_self`	6
`s_, d_, c_, z_`	`lin_geig_gen_for_lin_sol_lsq`	7
`s_, d_, c_, z_`	`lin_geig_gen_for_lin_eig_gen`	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`.

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 β 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.
```

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 ε is machine relative precision, each diagonal entry of R exceeds τ 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 α and β. The ordinary eigenvalue problem is $Cx = \lambda x$, where

$$C = H^T AP^T R^{-1}$$

and

$$\mathrm{RPv} = \mathrm{x}$$

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, $R^T R\, R = PBP^T$. 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

$$C = R^{-T} PAP^T R^{-1}$$

The self-adjoint eigenvalue problem is then solved.

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`, Chapter 10.

```
      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 μ. 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`, Chapter 10.

```
      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(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) 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 ε 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.
```

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.

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` = 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 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`.

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
!                                 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
             1                 2                 3                 4
( 4.000, 0.000)  ( 3.000, 0.000)  ( 2.000, 0.000)  ( 1.000, 0.000)
```

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.

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 balancing routine is based on the EISPACK routine BALANC. The reduction routine is based on the EISPACK routine ELMHES. See Smith et al. (1976) for the EISPACK routines. 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).

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`.

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`, page 460.

```
      USE EVCRG_INT
      USE EPIRG_INT
      USE UMACH_INT
      USE WRCRN_INT
!                                 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
```

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.

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 balancing routine is based on the EISPACK routine BALANC. The reduction routine is based on the EISPACK routines ORTHES and ORTRAN. The QR algorithm routine is based on the EISPACK routine HQR2. See Smith et al. (1976) for the EISPACK routines. Further details, some timing data, and credits are given in Hanson et al. (1990).

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.

Example

For an example of EPIRG, see IMSL routine EVCRG, page 457.

Comments

1. Workspace may be explicitly provided, if desired, by use of E2IRG/DE2IRG. The reference is:

   ```
   E2IRG(N, NEVAL, A, LDA, EVAL, EVEC, LDEVEC, 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.

Description

Let M = NEVAL, λ = EVAL, x_j = EVEC(*,J), the j-th column of EVEC. Also, let ε be the machine precision given by AMACH(4). The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j x_j \right\|_1}{10N\varepsilon \left\| A \right\|_1 \left\| x_j \right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\|v\|_1 = \sum_{i=1}^{N} \left\{ |\Re v_i| + |\Im v_i| \right\}$$

While the exact value of τ is highly machine dependent, the performance of `EVCSF` is considered excellent if $\tau < 1$, good if $1 \le \tau \le 100$, 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).

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, 1, N, 1)`

Double: The double precision name is `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)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of E3LCG/DE3LCG. The reference is:

   CALL E3LCG (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 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 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 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.

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 balancing routine is based on the EISPACK routine CBAL. The reduction routine is based on the EISPACK routine CORTH. The QR routine used is based on the EISPACK routine COMQR2. See Smith et al. (1976) for the EISPACK routines.

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`.

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`, page 467.

```
      USE EVCCG_INT
      USE EPICG_INT
      USE WRCRN_INT
      USE UMACH_INT
!                                 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
```

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`.

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 balancing routine is based on the EISPACK routine `CBAL`. The reduction routine is based on the EISPACK routine `CORTH`. The QR algorithm routine used is based on the EISPACK routine `COMQR2`. The back transformation routine is based on the EISPACK routine `CBABK2` . See Smith et al. (1976) for the EISPACK routines.

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` = 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: `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`.

Example

For an example of `EPICG`, see IMSL routine `EVCCG` on page 464.

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.

Description

Let M = `NEVAL`, λ = `EVAL`, x_j = `EVEC(*, J)`, the `j`-th column of `EVEC`. Also, let ε be the machine precision given by `AMACH`(4). The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j x_j \right\|_1}{10N\varepsilon \left\| A \right\|_1 \left\| x_j \right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\left\| v \right\|_1 = \sum_{i=1}^{N} \left\{ \left| \Re v_i \right| + \left| \Im v_i \right| \right\}$$

While the exact value of τ is highly machine dependent, the performance of `EVCSF` (page 471) is considered excellent if $\tau < 1$, good if $1 \le \tau \le 100$, 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).

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 `DEVLSF`.

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
!                                 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
```

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 2`N`.

   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.

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 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). Further details, some timing data, and credits are given in Hanson et al. (1990).

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.

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 on page 483.

```
USE EVCSF_INT
USE EPISF_INT
```

```
      USE UMACH_INT
      USE WRRRN_INT
!                                 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
```

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 3`N`.

   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.

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 reduction routine is based on the EISPACK routine TRED2. See Smith et al. (1976) for the EISPACK routines. Further details, some timing data, and credits are given in Hanson et al. (1990).

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`.

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
!                                  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
```

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 4`N`.

   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.

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).

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`.

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` on page 483.

```
      USE EVESF_INT
      USE EPISF_INT
      USE UMACH_INT
      USE WRRRN_INT
!                                 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 =  0.026
```

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 9`N`.

   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.

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`.

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`.

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
!                                 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
```

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 5N.

   IWK — Integer work array of length 1N.

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.

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).

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`.

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` on page 483.

```
      USE EVFSF_INT
      USE EPISF_INT
      USE WRRRN_INT
      USE UMACH_INT
!                                  Declare variables
      INTEGER    LDA, LDEVEC, MXEVAL, N
      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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `E3FSF/DE3FSF`. The reference is:

   ```
   ALL 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 `MXEVAL`. `NEVAL` 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.

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.

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`.

Example

For an example of `EPISF`, see routine `EVCSF`, on page 471.

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.

Description

Let M = `NEVAL`, λ = `EVAL`, x_j = `EVEC(*,J)`, the `j`-th column of `EVEC`. Also, let ε be the machine precision, given by `AMACH`(4) (see the Reference chapter). The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j x_j \right\|_1}{10N\varepsilon \left\| A \right\|_1 \left\| x_j \right\|_1}$$

While the exact value of τ is highly machine dependent, the performance of `EVCSF` (page 471) is considered excellent if $\tau < 1$, good if $1 \le \tau \le 100$, 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).

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`.

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 $k = 1, \ldots, N$ gives the the values in a different order. The eigenvalues of this real band symmetric matrix are computed and printed.

```
USE EVLSB_INT
USE WRRRN_INT
```

```
!                                 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
```

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.

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).

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`.

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, λ_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`, page 501.

```
      USE EVCSB_INT
      USE EPISB_INT
      USE UMACH_INT
      USE WRRRN_INT
!                                 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
```

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` (page 501).

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).

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`.

Example

The following example is given in Gregory and Karney (1969, page 63). The smallest four eigenvalues of the matrix

$$A=\begin{bmatrix}5&2&1&1&&&&&&&\\2&6&3&1&1&&&&&&\\1&3&6&3&1&1&&&&&\\1&1&3&6&3&1&1&&&&\\&1&1&3&6&3&1&1&&&\\&&1&1&3&6&3&1&1&&\\&&&1&1&3&6&3&1&1&\\&&&&1&1&3&6&3&1&1\\&&&&&1&1&3&6&3&1\\&&&&&&1&1&3&6&2\\&&&&&&&1&1&2&5\end{bmatrix}$$

are computed and printed.

```
      USE EVASB_INT
      USE WRRRN_INT
      USE SSET_INT
!                                 Declare variables
      INTEGER    LDA, N, NCODA, NEVAL
      PARAMETER  (N=11, NCODA=3, NEVAL=4, LDA=NCODA+1)
!
      REAL       A(LDA,N), EVAL(NEVAL)
      LOGICAL    SMALL
!                                 Set up matrix in band symmetric
!                                 storage mode
      CALL SSET (N, 6.0, A(4:,1), LDA)
      CALL SSET (N-1, 3.0, A(3:,2), LDA)
      CALL SSET (N-2, 1.0, A(2:,3), LDA)
      CALL SSET (N-3, 1.0, A(1:,4), LDA)
      CALL SSET (NCODA, 0.0, A(1:,1), 1)
      CALL SSET (NCODA-1, 0.0, A(1:,2), 1)
      CALL SSET (NCODA-2, 0.0, A(1:,3), 1)
      A(4,1) = 5.0
      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
```

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.

   WK — Work array of length 3`N`.

2. Informational error

Type	Code	
3	1	The iteration for an eigenvalue failed to converge. The best estimate will be returned.

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).

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.

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
!                                 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  -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
```

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	
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`.

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).

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`.

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
!                                 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 )
!                                 Representedin 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
```

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.

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).

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 (MXEVEL, 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`.

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` on page 501.

```
      USE EVFSB_INT
      USE EPISB_INT
      USE WRRRN_INT
      USE UMACH_INT
!                                  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
```

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 MXEVAL. NEVAL 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.

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).

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`.

Example

For an example of `EPISB`, see IMSL routine `EVCSB` on page 487.

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.

Description

Let M = `NEVAL`, λ = `EVAL`, x_j = `EVEC(*,J)`, the `j`-th column of `EVEC`. Also, let ε be the machine precision, given by `AMACH(4)`, see the Reference chapter of the manual. The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j x_j \right\|_1}{10 N \varepsilon \left\| A \right\|_1 \left\| x_j \right\|_1}$$

While the exact value of τ is highly machine dependent, the performance of `EVCSF` (page 471) is considered excellent if τ < 1, good if 1 ≤ τ ≤ 100, and poor if τ > 100. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

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 `DEVLHF`.

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
!                                 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
```

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.

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 reduction routine is based on the EISPACK routine `HTRIDI`. The QL routine is based on the EISPACK routine `IMTQL1`. See Smith et al. (1976) for the EISPACK routines.

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.

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 on page 518.

```
      USE IMSL_libraries

!                                 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
```

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` (page 518).

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`.

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 reduction routine is based on the EISPACK routine `HTRIDI`. The QL routine is based on the EISPACK routine `IMTQL2`. See Smith et al. (1976) for the EISPACK routines.

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 `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 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`.

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
!                                 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
```

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.

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.

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 `NEVEC` 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` = 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 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`.

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` on page 518.

```
      USE IMSL_LIBRARIES
!                                 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 WRCRN ('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
```

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` (page 518).

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 `TINVIT`. The back transformation routine is based on the EISPACK routine `HTRIBK`. See Smith et al. (1976) for the EISPACK routines.

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` = 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 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.

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
!                                 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
```

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.

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.

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`.

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 `MXEVAL` = 3. The routine `EVFHF` computes the number of eigenvalues in the given range. That value, `NEVAL`, is two. As a check on the computations, the performance index is also computed and printed. For more details, see routine `EPIHF` on page 518.

```
      USE IMSL_LIBRARIES

!                                 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 WRCRN ('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
```

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.

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.

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`.

Example

For an example of `EPIHF`, see IMSL routine `EVCHF`, page 505.

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.

Description

Let M = NEVAL, λ = EVAL, x_j = EVEC(*, J), the j-th column of EVEC. Also, let ε be the machine precision, given by AMACH(4), see the Reference chapter of this manual. The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j x_j \right\|_1}{10 N \varepsilon \left\| A \right\|_1 \left\| x_j \right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\left\| v \right\|_1 = \sum_{i=1}^{N} \left\{ \left| \Re v_i \right| + \left| \Im v_i \right| \right\}$$

While the exact value of τ is highly machine dependent, the performance of EVCSF (page 471) is considered excellent if τ < 1, good if 1 ≤ τ ≤ 100, and poor if τ > 100. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124–125).

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`.

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
!                                 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  )
!
      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)
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `E3LRH/DE3LRH`. The reference is:

   ```
   CALL E3LRH (N, A, LDA, EVAL, ACOPY, WK, IWK)
   ```

   The additional arguments are as follows:

   ACOPY — Real `N` by `N` work matrix.

   WK — Real vector of length 3*n*.

   IWK — Integer vector of length *n*.

2. Informational error

Type	Code	
4	1	The iteration for the eigenvalues failed to converge.

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).

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`.

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` on page 460 for details.

```
      USE EVCRH_INT
      USE EPIRG_INT
      USE UMACH_INT
      USE WRCRN_INT
!                                 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
```

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.

Description

Routine `EVCRH` 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).

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`.

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 WRCRN_INT
!                                 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 WRCRN ('EVAL', EVAL, 1, N, 1)
      END
```

Output

```
                              EVAL
             1                 2                 3                 4
(  8.22, 12.22)  (  3.40,  7.40)  (  1.60,  5.60)  ( -3.22,  0.78)
```

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.

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).

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 `DEVCCH`.

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`, page 467. The zeros in the lower part of the matrix are not referenced by `EVCCH`, but they are required by `EPICG` (page 467).

```
      USE EVCCH_INT
      USE EPICG_INT
      USE UMACH_INT
      USE WRCRN_INT
!                                 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)  ( 0.5429, 0.0000)
2 ( 0.6402, 0.0000) (-0.0046, 0.0000)  ( 0.6767, 0.0000)  ( 0.4298, 0.0000)
3 ( 0.2598, 0.0000) ( 0.7477, 0.0000)  (-0.3005, 0.0000)  ( 0.5277, 0.0000)
4 (-0.0948, 0.0000) (-0.6603, 0.0000)  ( 0.5625, 0.0000)  ( 0.4920, 0.0000)

Performance index =  0.020
```

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 2`N`.

   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.

3. The results of `EVCCH` can be checked using `EPICG` (page 467). This requires that the matrix `A` explicitly contains the zeros in `A`(`I`, `J`) for (`I` − 1) > `J` which are assumed by `EVCCH`.

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).

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 α_i, $i = 1, \ldots, n$. 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 β_i. (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 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).

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 GVLRG (N, A, LDA, B, LDB, ALPHA, BETAV)`

Double: The double precision name is DGVLRG.

Example

In this example, DATA statements are used to set *A* and *B*. The eigenvalues are computed and printed.

```
      USE IMSL_LIBRARIES
      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 GVLRG (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)
```

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, ACOPY, BCOPY,
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 `GVLRG` . Analogous comments hold for `BCOPY` and the values `IVAL`(5) – `IVAL`(8) . Additional memory allocation and option value restoration are automatically done in `GVLRG`. There is no requirement that users change existing applications that use `GVLRG` or `G3LRG`. Default values for the option are `IVAL(*)` = 1, 16, 0, 1, 1, 16, 0, 1.

Description

Routine `GVLRG` 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 λ, `GVLRG` returns α and β. If β 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 routine `GVLRG` uses the QZ algorithm due to Moler and Stewart (1973), as implemented by the EISPACK routines `QZHES`, `QZIT` and `QZVAL`; see Garbow et al. (1977).

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
$\beta_i \neq 0$, $\lambda_i = \alpha_i / \beta_{i,}$ $i = 1, \ldots, n$ are the eigenvalues of the system.

BETAV — Vector of size `N` containing scalars β_i. (Output)

EVEC — Complex matrix of order `N`. (Output)
The *J*-th eigenvector, corresponding to λ_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`.

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` (page 535).

```
 USE IMSL_LIBRARIES
```

```
      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
```

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.

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 λ, GVCRG returns complex numbers α and real numbers β. If β 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 routine GVCRG is based on the QZ algorithm due to Moler and Stewart (1973), as implemented by the EISPACK routines QZHES, QZIT and QZVAL; see Garbow et al. (1977).

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`.

Example

For an example of `GPIRG`, see routine `GVCRG` on page 531.

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	
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 `ALPHA`(`J`)/`BETAV`(`J`), its eigenvector should be in the `J`-th column of `EVEC`.

Description

Let M = NEVAL, x_j = EVEC(*,J) , the j-th column of EVEC. Also, let ε be the machine precision given by AMACH(4), see the Reference chapter of this manual. The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| \beta_j A x_j - \alpha_j B x_j \right\|_1}{\varepsilon \left(\left| \beta_j \right| \left\| A \right\|_1 + \left| \alpha_j \right| \left\| B \right\|_1 \right) \left\| x_j \right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\left\| v \right\|_1 = \sum_{i=1}^{N} \left\{ \left| \Re v_i \right| + \left| \Im v_i \right| \right\}$$

While the exact value of τ is highly machine dependent, the performance of EVCSF (page 471) is considered excellent if τ < 1, good if 1 ≤ τ ≤ 100, and poor if τ > 100. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

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)/betav(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 = 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).

FORTRAN 90 Interface

Generic: `CALL GVLCG (A, B, ALPHA, BETAV [,…])`

Specific: The specific interface names are `S_GVLCG` and `D_GVLCG`.

FORTRAN 77 Interface

Single: `CALL GVLCG (N, A, LDA, B, LDB, ALPHA, BETAV)`

Double: The double precision name is `DGVLCG`.

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
!                                 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 GVLCG (A, B, ALPHA, BETAV)
!                                 Compute eigenvalues
            EVAL = ALPHA/BETAV

!                                 Print results
      CALL WRCRN ('EVAL', EVAL, 1, N, 1)
      STOP
      END
```

Output

```
                                 EVAL
                1                   2                   3                   4
(-1.000,-1.333)   ( 0.765, 0.941)   (-0.353, 0.412)   (-0.353,-0.412)

                5
(-0.353,-0.412)
```

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.

Description

Routine `GVLCG` 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 λ, `GVLCG` returns α and β. If β is nonzero, then $\lambda = \alpha/\beta$. If the eigenvectors are needed, then use `GVCCG`. See page 540.

The routine `GVLCG` is based on routines for the generalized complex eigenvalue problem by Garbow (1978). The QZ algorithm is described by Moler and Stewart (1973). Some timing information is given in Hanson et al. (1990).

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, alpha(i)/betav(i) (for $i = 1, \ldots, n$), 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 `ALPHA`(`J`)/`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.

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 on page 542.

```
      USE IMSL_LIBRARIES

      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
```

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` (page 542).

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 λ, `GVCCG` returns α and β. If β 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).

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`.

Example

For an example of `GPICG`, see routine `GVCCG` on page 540.

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 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 `ALPHA`(`J`)/`BETAV` (`J`), its eigenvector should be in the `J`-th column of `EVEC`.

Algorithm

Let M = `NEVAL`, x_j = `EVEC(*, J)` , the `j`-th column of `EVEC`. Also, let ε be the machine precision given by `AMACH`(4). The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| \beta_j A x_j - \alpha_j B x_j \right\|_1}{\varepsilon \left(\left|\beta_j\right| \left\|A\right\|_1 + \left|\alpha_j\right| \left\|B\right\|_1 \right) \left\|x_j\right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\left\|v\right\|_1 = \sum_{i=1}^{N} \left\{ \left|\Re v_i\right| + \left|\Im v_i\right| \right\}$$

While the exact value of τ is highly machine dependent, the performance of `EVCSF` (page 471) is considered excellent if τ < 1, good if 1 ≤ τ ≤ 100, and poor if τ > 100.

The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

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` = 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).

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`.

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
!                                 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
```

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.

   WK1 — 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.

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

$$(R^{-T} AR^{-1})(Rx) = \lambda\ (Rx)$$

The eigenvalues of $C = R^{-T} AR^{-1}$ 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 `EVLSF`, page 469. Further discussion and some timing results are given Hanson et al. (1990).

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` = 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 of the calling program. (Input)
Default: `LDEVEC` = 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 CALL GVCSP (N, A, LDA, B, LDB, EVAL, EVEC, LDEVEC)`

Double: The double precision name is `DGVCSP`.

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` on page 549.

```
      USE GVCSP_INT
      USE GPISP_INT
      USE UMACH_INT
      USE WRRRN_INT
!                                 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        2        3
1.386  -0.058  -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
```

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`.Type Code

2. Informational errors

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` (page 549).

Description

Routine `GVLSP` (page 544) 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

$$(R^{-T} AR^{-1})(Rz) = \lambda\,(Rz)$$

The eigenvalues and eigenvectors of $C = R^{-T} AR^{-1}$ 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 `EVCSF`, page 471. Further discussion and some timing results are given Hanson et al. (1990).

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`.

Example

For an example of `GPISP`, see routine `GVCSP` on page 547.

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 ALPHA(J)/BETAV(J), its eigenvector should be in the J-th column of EVEC.

Description

Let M = NEVAL, λ = EVAL, x_j = EVEC(*, J) , the j-th column of EVEC. Also, let ε be the machine precision given by AMACH(4). The performance index, τ, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\left\| Ax_j - \lambda_j Bx_j \right\|_1}{\varepsilon \left(\left\| A \right\|_1 + \left| \lambda_j \right| \left\| B \right\|_1 \right) \left\| x_j \right\|_1}$$

The norms used are a modified form of the 1-norm. The norm of the complex vector v is

$$\left\| v \right\|_1 = \sum_{i=1}^{N} \left\{ \left| \Re v_i \right| + \left| \Im v_i \right| \right\}$$

While the exact value of τ is highly machine dependent, the performance of EVCSF (page 471) is considered excellent if τ < 1, good if 1 ≤ τ ≤ 100, and poor if τ > 100. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Garbow et al. (1977, pages 77–79).

Chapter 3: Interpolation and Approximation

Routines

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

$$p(x) = \sum_{j=1}^{k} c_{ji} \frac{(x-\xi_i)^{j-1}}{(j-1)!} \qquad \text{for } \xi_i \le x < \xi_{i+1}$$

The breakpoint sequence ξ 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

$$c_{1,i+1} = p(\xi_{i+1}) = c_{1i} + c_{2i}\Delta\xi_i + \ldots + c_{ki}\frac{(\Delta\xi_i)^{k-1}}{(k-1)!}$$

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 ξ_i, then the number ξ_i should occur $k - j - 1$ times in the knot sequence. Assuming that ξ_1, and ξ_n are the endpoints of the interval of interest, one chooses the first k knots equal to ξ_1 and the last k knots equal to ξ_n. This can be done since the B-splines are defined to be right continuous near ξ_1 and left continuous near ξ_n.

When the above construction is completed, we will have generated a knot sequence $\boldsymbol{t}$ of length M; and there will be $m := M - k$ B-splines of order k, say B_1 ,…, 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

$$p = a_1 B_1 + a_2 B_2 + \ldots + a_m B_m$$

as a linear combination of B-splines. The B-spline routines will consistently make use of the following identifiers for FORTRAN variables:

a = BSCOEF

$\mathbf{t}$ = XKNOT

m = NCOEF

M = 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

$$B_{i,k,\boldsymbol{t}}$$

to denote the i-th B-spline of order k for the knot sequence $\mathbf{t}$.

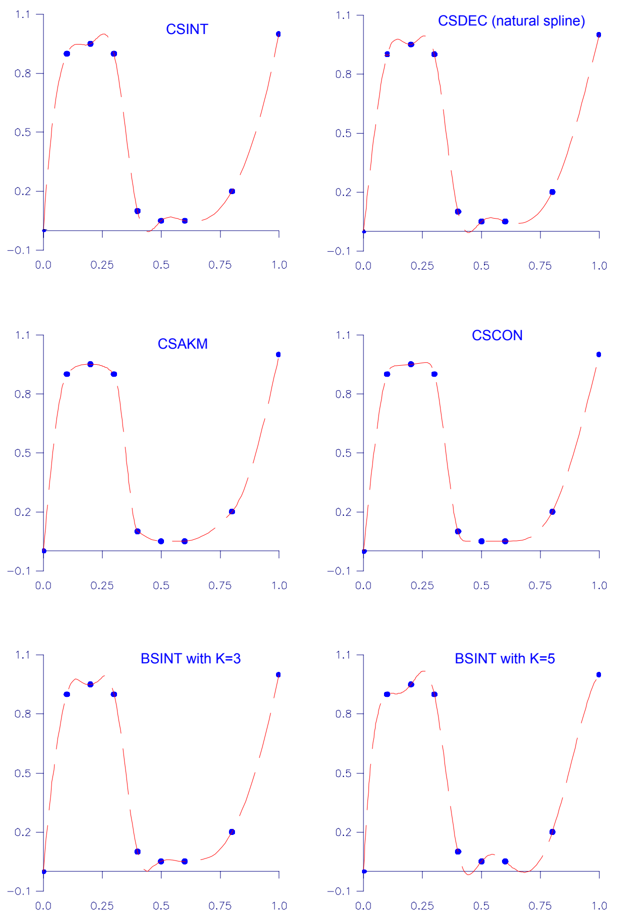

Figure 3-1 Spline Interpolants of the Same Data

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` (page 587), `CSINT` (page 590), `CSDEC` (page 593), `CSHER` (page 597), `CSAKM` (page 600), `CSCON` (page 603), and `CSPER` (page 606). 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 CONFT (page 734) 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 (page 603) Notice the different characteristics of the interpolants. The interpolation routines CSAKM (page 600) 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, page 631), compute the three-dimensional tensor-product spline coefficients given three-dimensional interpolation data (BS3IN, page 635) compute the two-dimensional tensor-product spline coefficients for a tensor-product least squares problem (BSLS2, page 743), and compute the three-dimensional tensor-product spline coefficients for a tensor-product least squares problem (BSLS3, page 748). In addition, we provide evaluation, differentiation, and integration routines for the twoand three-dimensional tensor-product spline functions. The relevant routines are BS2VL (page 651), BS3VL (page 664), BS2DR (page 653), BS3DR (page 666), BS2GD (page 656), BS3GD (page 670), BS2IG (page 661), and BS3IG (page 676).

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.

Scattered Data 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.

Least Squares

Routines are provided to smooth noisy data: regression using linear polynomials (RLINE), regression using arbitrary polynomials (RCURV, page 716), and regression using user-supplied functions (FNLSQ, page 720). Additional routines compute the least-squares fit using splines with fixed knots (BSLSQ, page 725) or free knots (BSVLS, page 729). These routines can produce cubic-spline least-squares fit simply by setting the order to 4. The routine CONFT (page 734) 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, page 743) and (BSLS3, page 748).

Smoothing by Cubic Splines

Two “smoothing spline” routines are provided. The routine CSSMH (page 758) returns the cubic spline that smooths the data, given a smoothing parameter chosen by the user. Whereas, CSSCV (page 761) 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 CSSED (page 754) 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 (page 764) 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 (page 587) 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 (page 618) 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.

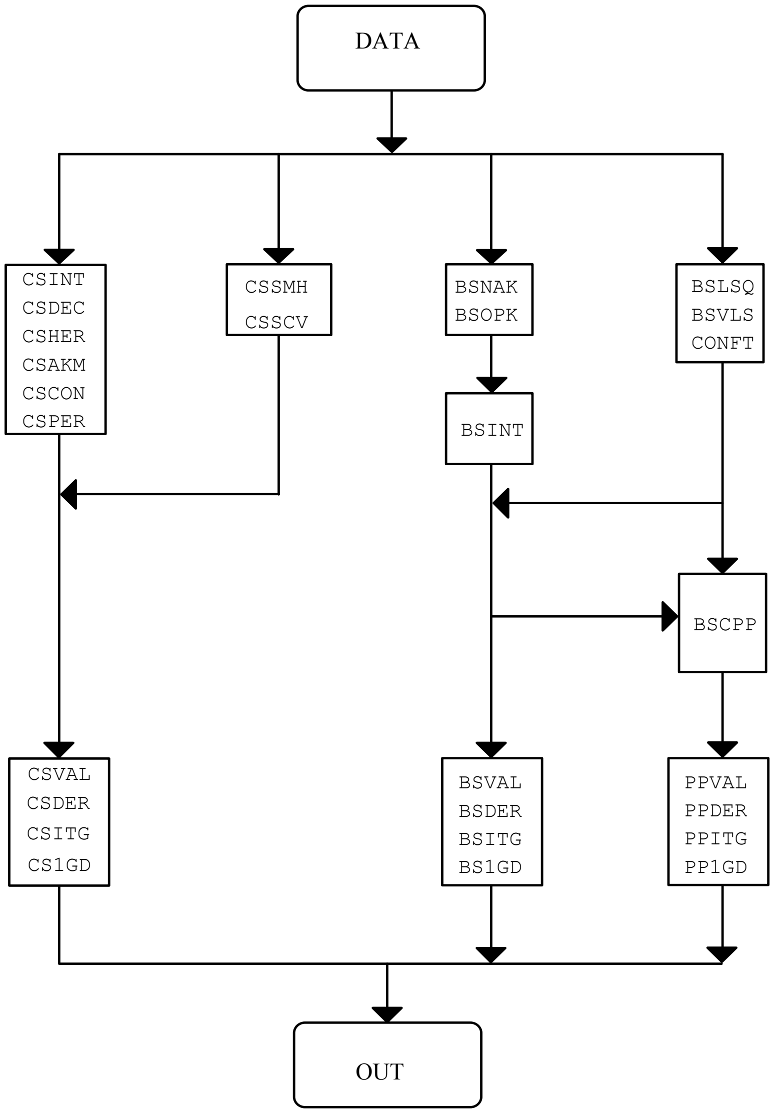

Figure 3-2 Interrelation of the Spline Routines

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 18 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`. Two- and three-dimensional tensor-product interpolation routines are also provided. For two-dimensional scattered data, the appropriate routine is `SURF` .

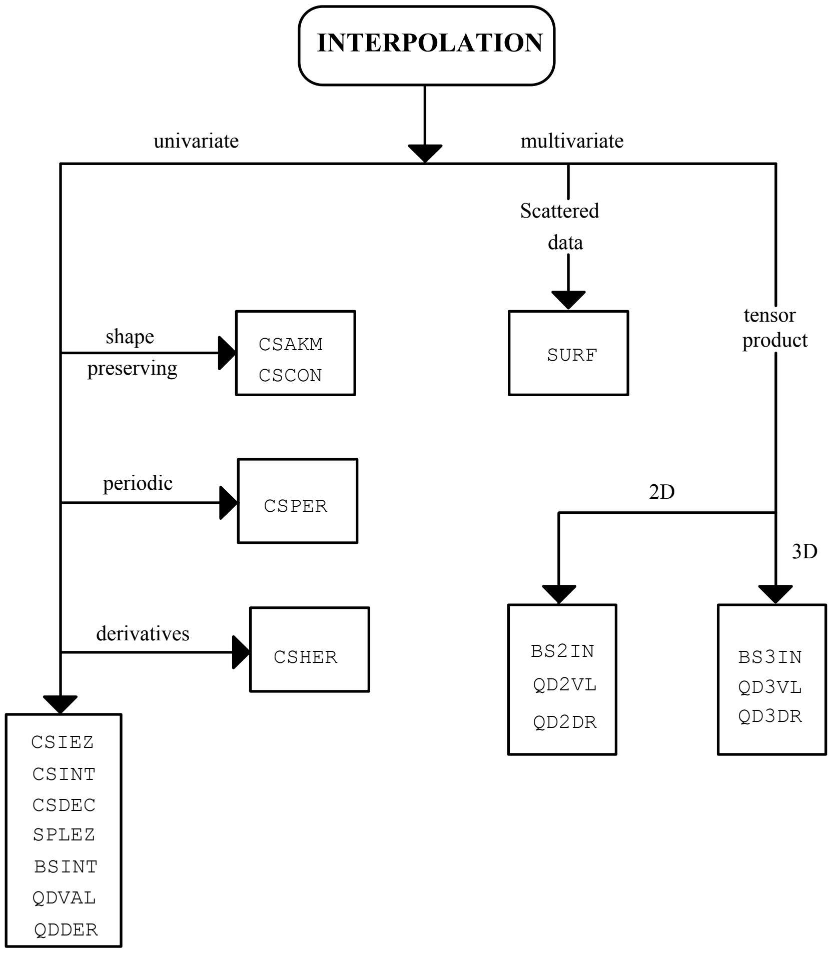

Figure 3-3 Choosing an Interplation Routine

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 $N = size(C)$ satisfies the identity *N - 1 + spline_degree = size (?_knots)*, 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

$$e(x) = \left(b(x)^T Gb(x)\right)^{1/2}$$

where

$$b(x) = \left[B_1(x), \ldots, B_N(x)\right]^T$$

and G is the covariance matrix associated with the coefficients of the spline

$$c = \left[c_1, \ldots, c_N\right]^T$$

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, VARAIBLES, 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) =` x_i, `data(2,i) =` y_i, and `data(3,i) =` σ_i, $i = 1,...,ndata$. If the variances are not known but are proportional to an unknown value, users may set `data(3,i) =` 1, $i = 1,...,ndata$.

`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 `spline_fitting`		
Prefix = None	Option Name	Option Value
	`Spline_fitting_tol_equal`	1
	`Spline_fitting_tol_least`	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`.

Example 1: Natural Cubic Spline Interpolation to Data

The function

$$g(x) = \exp\left(-x^2/2\right)$$

is interpolated by cubic splines on the grid of points

$$x_i = (i-1)\,\Delta x,\ i = 1, \ldots, ndata$$

Those natural conditions are

$$f(x_i) = g(x_i),\ i = 0, \ldots, ndata;\ \frac{d^2 f}{dx^2}(x_i) = \frac{d^2 g}{dx^2}(x_i),\ i = 0 \text{ and } ndata$$

Our program checks the term *const.* appearing in the maximum truncation error term

$$error \approx const. \times \Delta x^4$$

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.
```

Description

This routine has similar scope to `CONFT/DCONFT` 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 `CONFT/DCONFT` routine uses `QPROG` (*loc cit*., p. 959), which requires that the least-squares equations be of full rank.

Additional Examples

Example 2: Shaping a Curve and its Derivatives

The function

$$g(x) = \exp\left(-x^2/2\right)\left(1 + noise\right)$$

is fit by cubic splines on the grid of equally spaced points

$$x_i = (i-1)\Delta x,\ i = 1,\ldots,ndata$$

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 for $0 \le x \le 1$ convex upward for $1 < x \le 4$, 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 and 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

$$\begin{aligned} g(x) &= \exp\left(-x^2/2\right), -1 < x < 1 \\ &= 0, |x| \ge 1 \end{aligned}$$

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

$$\frac{d}{dx}\left(q(x) - u\right) = f(x), -1 < x < 1$$

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), &
            xvalues(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.
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `spline_fitting`. These error messages are numbered 1340–1367.

SURFACE_CONSTRAINTS

To further shape a surface defined by a tensor product of B-splines, the routine **`suface_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 $M = size\ (C,1)$ and $N = size\ (C,2)$ satisfies the respective identities N -1 + *spline_degree* = *size* (*?_knotsx*), and M -1 + *spline_degree* = *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

$$e(x,y) = \left(b(x,y)^T\ Gb(x,y)\right)^{1/2}$$

where

$$b(x,y) = \left[B_1(x)B_1(y),\ldots,B_N(x)B_1(y),\ldots\right]^T$$

and G is the covariance matrix associated with the coefficients of the spline

$$c = \left[c_{11},\ldots,c_{N1},\ldots\right]^T$$

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:

`data(1,i) =` x_i,

`data(2,i) =` y_i,

`data(3,i) =` z_i,

`data(4,i) =` σ_i, $i = 1,...,ndata$.

If the variances are not known, but are proportional to an unknown value, use

`data(4,i) =` 1, $i = 1,...,ndata$.

`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 = knotsy` (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
	`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

$$size = \sum | \, data(3,:) / data(4,:) \, | / \left(ndata + 1 \right).$$

`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`$^{-4}$.

`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`$^{-4}$.

`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 =` $10^{-3} \times size$,, where

$$size = \sum | data(3,:) / data(4,:) | / \left(ndata + 1 \right).$$

FORTRAN 90 Interface

Generic: `CALL SURFACE_FITTING (DATA, KNOTSX, KNOTSX, KNOTSY[,…])`

Specific: The specific interface names are `S_SURFACE_FITTING` and `D_SURFACE_FITTING`.

Example 1: Tensor Product Spline Fitting of Data

The function

$$g\left(x, y\right) = \exp\left(-x^2 - y^2\right)$$

is least-squares fit by a tensor product of cubic splines on the square

$$\left[0,2\right] \otimes \left[0,2\right]$$

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.
```

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).

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*,

$$x(u,v) = a\cos(u)\cos(v),\ -\pi \le 2u \le \pi$$
$$y(u,v) = a\cos(u)\sin(v),\ -\pi \le v \le \pi$$
$$z(u,v) = a\sin(u)$$

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

$$-\pi \le u \le \pi$$

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,

$$-\pi \le 2u \le \pi$$

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
```

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(1d0)), 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 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(1d0)), parameter :: zero=0d0, one=1d0
      real(kind(1d0)), parameter :: TOLERANCE=1d-3
      real(kind(1d0)), 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(1d0)), 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.
```

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.

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`.

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
      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
```

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 MAX0(N, NDATA) + N.

WK1 — Real work array of length 5 * NDATA.

WK2 — Real work array of length 2 * N.

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 $i = 1, \ldots,$ `NDATA`. The output for this routine consists of a vector of values of the computed cubic spline. Specifically, let $n = N$, $v =$ `XVEC`, and $y =$ `VALUE`, then if s is the computed spline we set

$$y_j = s(v_j) \qquad\qquad j = 1, \ldots, n$$

Additional documentation can be found by referring to the IMSL routines `CSINT` (page 590) or `SPLEZ` (page 618).

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`.

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
!                                 Specifications
      INTEGER    NDATA
      PARAMETER  (NDATA=11)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CSCOEF(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, 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 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,CSCOEF),&
                                   F(X) - CSVAL(X,BREAK,&
                                   CSCOEF)
   20 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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `C2INT/DC2INT`. The reference is:

   `CALL C2INT (NDATA, XDATA, FDATA, BREAK, CSCOEF, IWK)`

   The additional argument is

   IWK — Work array of length `NDATA`.

2. The cubic spline can be evaluated using `CSVAL` (page 609); its derivative can be evaluated using `CSDER` (page 610).

3. Note that column `NDATA` of `CSCOEF` is used as workspace.

Description

The routine `CSINT` computes a C^2 cubic spline interpolant to a set of data points (x_i, f_i) for $i = 1, \ldots,$ `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 ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{[\xi_1, \xi_N]} \le C \left\| f^{(4)} \right\|_{[\xi_1, \xi_N]} \left| \xi \right|^4$$

where

$$\left| \xi \right| := \max_{i=2,\ldots,N} \left| \xi_i - \xi_{i-1} \right|$$

For more details, see de Boor (1978, pages 55–56).

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)

ILEFT	**Condition**
0	"Not-a-knot" condition
1	First derivative specified by `DLEFT`
2	Second derivative specified by `DLEFT`

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)

IRIGHT	**Condition**
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`.

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

      INTEGER    ILEFT, IRIGHT, NDATA
      PARAMETER  (ILEFT=1, IRIGHT=2, NDATA=11)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), COS, CSCOEF(4,NDATA), DLEFT,&
                 DRIGHT, F, FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA)
      INTRINSIC  COS, FLOAT, SIN
!                                  Define function
      F(X) = SIN(15.0*X)
!                                  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, 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          0.675       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
```

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, CSCOEF, IWK)
   ```

   The additional argument is:

   IWK — Work array of length `NDATA`.

2. The cubic spline can be evaluated using `CSVAL` (page 609); its derivative can be evaluated using `CSDER` (page 610).

3. Note that column `NDATA` of `CSCOEF` is used as workspace.

Description

The routine `CSDEC` computes a C^2 cubic spline interpolant to a set of data points (x_i, f_i) for $i = 1, \ldots,$ `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 ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{[\xi_1, \xi_N]} \le C \left\| f^{(4)} \right\|_{[\xi_1, \xi_N]} \left| \xi \right|^4$$

where

$$\left| \xi \right| := {}_{i=2,\ldots,N} \left| \xi_i - \xi_{i-1} \right|$$

For more details, see de Boor (1978, Chapter 4 and 5).

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
      INTEGER    ILEFT, IRIGHT, NDATA
      PARAMETER  (ILEFT=2, IRIGHT=2, NDATA=11)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CSCOEF(4,NDATA), DLEFT, DRIGHT,&
                 F, FDATA(NDATA), FLOAT, SIN, X, XDATA(NDATA)
      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, 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           0.667       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`.

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

      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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `C2HER/DC2HER`. The reference is:

   ```
   CALL C2HER (NDATA, XDATA, FDATA, DFDATA, BREAK,
   CSCOEF, 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` (page 609); its derivative can be evaluated using `CSDER` (page 610).

4. Note that column `NDATA` of `CSCOEF` is used as workspace.

Description

The routine `CSHER` computes a C^1 cubic spline interpolant to the set of data points

$$\left(x_i, f_i\right) \text{and} \left(x_i, f_i'\right)$$

for i = 1, …, 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 ξ be the

breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{[\xi_1, \xi_N]} \le C \left\| f^{(4)} \right\|_{[\xi_1, \xi_N]} \left|\xi\right|^4$$

where

$$\left|\xi\right| := _{i=2,\ldots,N} \left|\xi_i - \xi_{i-1}\right|$$

For more details, see de Boor (1978, page 51).

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)

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 CSAKM (XDATA, FDATA, BREAK, CSCOEF [,…])`

Specific: The specific interface names are `S_CSAKM` and `D_CSAKM`.

FORTRAN 77 Interface

Single: `CALL CSAKM(NDATA, XDATA, FDATA, BREAK, CSCOEF)`

Double: The double precision name is `DCSAKM`.

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

      INTEGER    NDATA
      PARAMETER  (NDATA=11)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CSCOEF(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, 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          0.818      -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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `C2AKMD/C2AKM`. The reference is:

   `CALL C2AKM (NDATA, XDATA, FDATA, BREAK, CSCOEF, IWK)`

   The additional argument is:

   IWK — Work array of length `NDATA`.

2. The cubic spline can be evaluated using `CSVAL` (page 609); its derivative can be evaluated using `CSDER` (page 610).

3. Note that column `NDATA` of `CSCOEF` is used as workspace.

Description

The routine `CSAKM` computes a C^1 cubic spline interpolant to a set of data points (x_i, f_i) for $i = 1, \ldots,$ `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 ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1, \xi_N]} \le C \left\| f^{(2)} \right\|_{[\xi_1, \xi_N]} |\xi|^2$$

where

$$|\xi| := \max_{i=2,\ldots,N} |\xi_i - \xi_{i-1}|$$

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.)

CSCON

Computes a cubic spline interpolant that is consistent with the concavity of the data.

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)

IBREAK — The number of breakpoints. (Output)
It will be less than 2 * NDATA.

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 * NDATA.

CSCOEF — Matrix of size 4 by N where N is the dimension of BREAK. (Output)
The first IBREAK – 1 columns of CSCOEF contain the local coefficients of the cubic pieces.

Optional Arguments

NDATA — Number of data points. (Input)
NDATA must be at least 3.
Default: NDATA = size (XDATA,1).

FORTRAN 90 Interface

Generic: `CALL CSCON (XDATA, FDATA, IBREAK, BREAK, CSCOEF [,…])`

Specific: The specific interface names are S_CSCON and D_CSCON.

FORTRAN 77 Interface

Single: `CALL CSCON (NDATA, XDATA, FDATA, IBREAK, BREAK, CSCOEF)`

Double: The double precision name is DCSCON.

Example

We first compute the shape-preserving interpolant using CSCON, and display the coefficients and breakpoints. Second, we interpolate the same data using CSINT (page 590) 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
!                                 Specifications
      INTEGER    NDATA
      PARAMETER  (NDATA=9)
!
      INTEGER    IBREAK, NOUT
      REAL       BREAK(2*NDATA), CSCOEF(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, CSCOEF)
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 Print the BREAK points and the
!                                 coefficients (CSCOEF) for
!                                 checking purposes.
      WRITE (NOUT,'(1X,A,I2)') 'IBREAK = ', IBREAK
      CALL WRRRL ('BREAK', BREAK, RLABEL, CLABEL, 1, IBREAK, 1, &
                  FMT='(F9.3)')
      CALL WRRRL ('CSCOEF', CSCOEF, 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

                              CSCOEF
           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
```

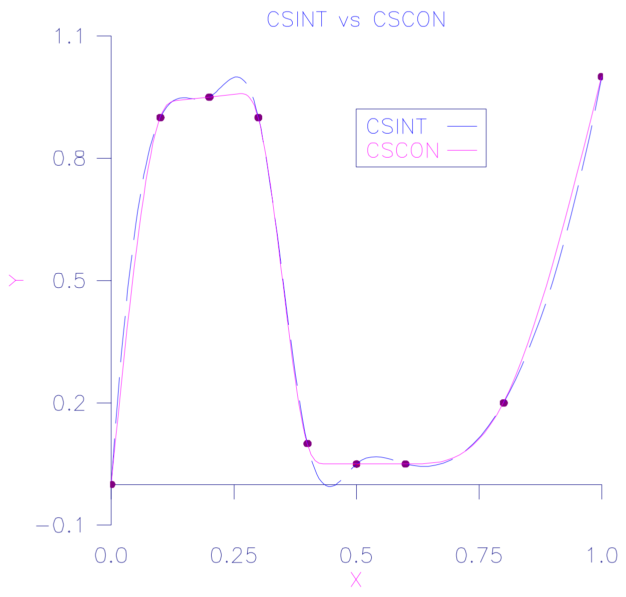

Figure 3-4   CSCON vs. CSINT

Comments

1. Workspace may be explicitly provided, if desired, by use of `C2CON/DC2CON`. The reference is:

   ```
   CALL C2CON (NDATA, XDATA, FDATA, IBREAK, BREAK, CSCOEF, 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 `C2CON`/`DC2CON` to set a larger number for `ITMAX`.
4	3	The `XDATA` values must be distinct.

3. The cubic spline can be evaluated using `CSVAL` (page 609); its derivative can be evaluated using `CSDER` (page 610).

4. The default value for `ITMAX` is 25. This can be reset by calling `C2CON`/`DC2CON` directly.

Descritpion

The routine `CSCON` computes a cubic spline interpolant to n = `NDATA` data points $\{x_i, f_i\}$ for $i = 1, \ldots, n$. 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.

CSPER

Computes the cubic spline interpolant with periodic boundary 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)

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 4.
Default: `NDATA` = size (`XDATA`,1).

FORTRAN 90 Interface

Generic: `CALL CSPER (XDATA, FDATA, BREAK, CSCOEF [,…])`

Specific: The specific interface names are `S_CSPER` and `D_CSPER`.

FORTRAN 77 Interface

Single: `CALL CSPER (NDATA, XDATA, FDATA, BREAK, CSCOEF)`

Double: The double precision name is `DCSPER`.

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
      INTEGER    NDATA
      PARAMETER  (NDATA=11)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CSCOEF(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, 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
      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,CSCOEF),&
                                    F(X) - CSVAL(X,BREAK,&
                                    CSCOEF)
   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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C2PER/DC2PER. The reference is:

CALL C2PER (NDATA, XDATA, FDATA, BREAK, CSCOEF, WK, IWK)

The additional arguments are as follows:

WK — Work array of length 6 * 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 XDATA points are not equal. The value at the smallest XDATA point is used.

3. The cubic spline can be evaluated using CSVAL (page 609) and its derivative can be evaluated using CSDER (page 610).

Description

The routine CSPER computes a C^2 cubic spline interpolant to a set of data points (x_i, f_i) for $i = 1, \ldots,$ 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 ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_1,\xi_N]} \le C \left\| f^{(4)} \right\|_{[\xi_1,\xi_N]} |\xi|^4$$

where

$$|\xi| := \max_{i=2,\ldots,N} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, pages 320–322).

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.

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)

FORTRAN 90 Interface

Generic: `CSVAL (X, BREAK, CSCOEF[,…])`

Specific: The specific interface names are `S_CSVAL` and `D_CSVAL`.

FORTRAN 77 Interface

Single: `CSVAL(X, NINTV, BREAK, CSCOEF)`

Double: The double precision function name is `DCSVAL`.

Example

For an example of the use of `CSVAL`, see IMSL routine `CSINT` (page 590).

Description

The routine `CSVAL` evaluates a cubic spline at a given point. It is a special case of the routine `PPDER` (page 684), 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).

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` = size (`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`.

Example

In this example, we compute a cubic spline interpolant to a function *f* using IMSL routine `CSINT` (page 590). 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

      INTEGER    NDATA
      PARAMETER  (NDATA=10)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CDDF, CDF, CF, COS, CSCOEF(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, CSCOEF)
!                                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,CSCOEF)
         CDF  = CSDER(1,X,BREAK,CSCOEF)
         CDDF = CSDER(2,X,BREAK,CSCOEF)
         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
```

Description

The function `CSDER` evaluates the derivative of a cubic spline at a given point. It is a special case of the routine `PPDER` (page 684), 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).

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.

Example

To illustrate the use of CS1GD, we modify the example program for CSINT (page 590). 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
!                                 Specifications
      INTEGER    NDATA, N
      PARAMETER  (NDATA=11, N=2*NDATA-1)
!
      INTEGER    I, NINTV, NOUT
      REAL       BREAK(NDATA), CSCOEF(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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C21GD/DC21GD. The reference is:

   ```
   CALL C21GD (IDERIV, N, XVEC, NINTV, BREAK, CSCOEF,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.

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 $i = 1, \ldots, n - 1$, a derivative value j, and a cubic spline s that is represented by a breakpoint sequence and coefficient matrix this routine returns the values

$$s^{(j)}(x_i) \qquad\qquad i = 1, \ldots, n$$

in the array VALUE. The functionality of this routine is the same as that of CSDER (page 610) called in a loop, however CS1GD should be much more efficient.

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` = size (`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`.

Example

This example computes a cubic spline interpolant to the function x^2 using `CSINT` (page 590) 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
```

```
      INTEGER    NDATA
      PARAMETER  (NDATA=10)
!
      INTEGER    I, NINTV, NOUT
      REAL       A, B, BREAK(NDATA), CSCOEF(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, CSCOEF)
!                                  Compute the integral of F over
!                                  [0.0,0.5]
      A     = 0.0
      B     = 0.5
      NINTV = NDATA - 1
      VALUE = CSITG(A,B,BREAK,CSCOEF)
      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,CSCOEF)
      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
```

Description

The function CSITG evaluates the integral of a cubic spline over an interval. It is a special case of the routine PPITG (page 690), which evaluates the integral of a piecewise polynomial. (A cubic spline is a piecewise polynomial of order 4.)

SPLEZ

Computes the values of a spline that either interpolates or fits user-supplied data.

Required Arguments

XDATA — Array of length NDATA containing the data point abscissae. (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 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

NDATA — Number of data points. (Input)
Default: NDATA = size (XDATA,1).

All choices of ITYPE are valid if NDATA is larger than 6. More specifically,

NDATA > ITYPE	or ITYPE = 1.
NDATA > 3	for ITYPE = 2, 3.
NDATA > (ITYPE – 3)	for ITYPE = 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.

Example

In this example, all the ITYPE parameters are exercised. The values of the spline are then compared with the exact function values and derivatives.

```
      USE IMSL_LIBRARIES
      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)
!                                 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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `S2LEZ/DS2LEZ`. The reference is:

   ```
   CALL S2LEZ (NDATA, XDATA, FDATA, ITYPE, IDER, N, XVEC, VALUE,
   WRK, IWK)
   ```

   The additional arguments are as follows:

   WRK — Work array of length `32 * NDATA + 4 * N + 22`.

   IWK — Work array of length `MAX0(NDATA . N) + N`.

2. Informational errors

Type	Code	
4	1	`XDATA` entries are not unique.
4	2	`XVEC` entries are not unique.

3. The workspace listed above is the maximum that is needed. Depending on the choice of `ITYPE`, the actual amount used may be less. If workspace is a critical resource, consult the explicit routines listed under `ITYPE` to see if less workspace can be used.

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 $i = 1,\square\ldots,$ `NDATA` if `ITYPE` = 1, …, 8. If `ITYPE` ≥ 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 = `IDER`, n = `N`, v = `XVEC`, and y = `VALUE`, then if s is the computed spline we set

$$y_j = s^{(i)}(v_j) \qquad j = 1, \ldots, n$$

The routines called are listed above under the `ITYPE` heading. Additional documentation can be found by referring to these routines.

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`.

Example

In this example, a spline interpolant s, to

$$f(x) = \sqrt{x}$$

is computed. The interpolated values are then compared with the exact function values using the IMSL routine `BSVAL` (page 641).

```
      USE BSINT_INT
      USE BSNAK_INT
      USE UMACH_INT
      USE BSVAL_INT
      INTEGER    KORDER, NDATA, NKNOT
      PARAMETER  (KORDER=3, NDATA=5, NKNOT=NDATA+KORDER)
!
      INTEGER    I, NCOEF, NOUT
      REAL       BSCOEF(NDATA), BT, F, FDATA(NDATA), FLOAT,&
                 SQRT, X, XDATA(NDATA), XKNOT(NKNOT), XT
      INTRINSIC  FLOAT, SQRT
!                                  Define function
      F(X) = SQRT(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)
!                                  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
      BT    = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOEF)
      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,BSCOEF)
         WRITE (NOUT,99998) XT, BT, F(XT) - BT
         XT = XDATA(I)
!                                  Evaluate spline
         BT = BSVAL(XT,KORDER,XKNOT,NCOEF,BSCOEF)
         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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of `B2INT/DB2INT`. The reference is:

```
CALL B2INT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF, 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`.

2. Informational errors

Type	Code	
3	1	The interpolation matrix is ill-conditioned.
4	3	The `XDATA` values must be distinct.
4	4	Multiplicity of the knots cannot exceed the order of the spline.
4	5	The knots must be nondecreasing.
4	15	The `I`-th smallest element of the data point array must be greater than the Ith knot and less than the (`I` + `KORDER`)-th knot.
4	16	The largest element of the data point array must be greater than the (`NDATA`)-th knot and less than or equal to the (`NDATA` + `KORDER`)-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 (`KORDER` + 1)st knot.

3. The spline can be evaluated using `BSVAL` (page 641), and its derivative can be evaluated using `BSDER` (page 643).

Description

Following the notation in de Boor (1978, page 108), let $B_j = B_{j,k,\mathbf{t}}$ denote the j-th B-spline of order k with respect to the knot sequence **t**. Then, `BSINT` computes the vector a satisfying

$$\sum_{j=1}^{N} a_j B_j\left(x_i\right) = f_i$$

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 = `XDATA` and f = `FDATA`. Let $\mathbf{t}$ = `XKNOT`, k = `KORDER`, and N = `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

$$\begin{aligned} x_i &< x_{i+1} && i = 1, \ldots, N-1 \\ \mathbf{t}_i &< \mathbf{t}_{i+1} && i = 1, \ldots, N \\ \mathbf{t}_i &\le \mathbf{t}_{i+k} && i = 1, \ldots, N+k-1 \end{aligned}$$

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 $\mathbf{t}_k \le x_i \le \mathbf{t}_{N+1}$ 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,

$$B_{j-k+1}, \ldots, B_j \qquad \text{where } j \text{ satisfies } \mathbf{t}_j \le x_i < \mathbf{t}_{j+1}$$

be well-defined (that is, $j - k + 1 \ge 1$).

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

$$\left\| f - s \right\|_{[\mathbf{t}_k, \mathbf{t}_{N+1}]} \le C \left\| f^{(k)} \right\|_{[\mathbf{t}_k, \mathbf{t}_{N+1}]} \left| \mathbf{t} \right|^k$$

where

$$\left| \mathbf{t} \right| := \max_{i=k,\ldots,N} \left| \mathbf{t}_{i+1} - \mathbf{t}_i \right|$$

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` (page 590) by calling `BSNAK` (page 625) to obtain the proper knots, then calling `BSINT` yielding the B-spline coefficients, and finally calling IMSL routine `BSCPP` (page 680) to convert to piecewise polynomial form.

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`.

Example

In this example, we compute (for $k = 3, \ldots, 8$) 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` (page 622) 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
      INTEGER    KMAX, KMIN, NDATA
      PARAMETER  (KMAX=8, KMIN=3, NDATA=20)
!
      INTEGER    I, K, KORDER, NOUT
      REAL       ABS, AMAX1, BSCOEF(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 BSNAK (NDATA, XDATA, KORDER, XKNOT)
!                                 Interpolate
         CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
         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
```

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.

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

$$\mathbf{t}_i = x_1 \quad \text{for } i = 1, \ldots, k$$

$$\mathbf{t}_i = x_{i-k/2} \quad \text{for } i = k + 1, \ldots, N$$

$$\mathbf{t}_i = x_N + \varepsilon \quad \text{for } i = N + 1, \ldots, N + k$$

where ε 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

$$\mathbf{t}_i = x_1 \quad \text{for } i = 1, \ldots, k$$

$$\mathbf{t}_i = (x_{i-\frac{k-1}{2}} + x_{i-1-\frac{k-1}{2}})/2 \quad \text{for } i = k + 1, \ldots, N$$

$$\mathbf{t}_i = x_N + \varepsilon \quad \text{for } i = N + 1, \ldots, N + k$$

It is not necessary to sort the values in x since this is done in the routine BSNAK.

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`.

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` (page 622) 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
      INTEGER    KMAX, KMIN, NDATA
      PARAMETER  (KMAX=8, KMIN=3, NDATA=20)
!
      INTEGER    I, K, KORDER, NOUT
      REAL       ABS, AMAX1, BSCOEF(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, BSCOEF)
         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        0.0096
 4        0.0018
 5        0.0005
 6        0.0004
 7        0.0007
 8        0.0035
```

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 (NDATA – KORDER) * (3 * KORDER – 2) + 6 * NDATA + 2 * 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.

Description

Given the abscissas x = XDATA for an interpolation problem and the order of the spline interpolant k = KORDER, BSOPK returns the knot sequence **t** = XKNOT that minimizes the constant in the error estimate

$$\| f - s \| \le c \, \| f^{(k)} \|$$

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 **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).

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` = size (`XDATA`,1).

NYDATA — Number of data points in the `Y`-direction. (Input)
Default: `NYDATA` = size (`YDATA`,1).

LDF — The leading dimension of `FDATA` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDF` = 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`.

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 × 4 grid are displayed.

```
      USE BS2IN_INT
      USE BSNAK_INT
      USE BS2VL_INT
      USE UMACH_INT
!                                  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       BSCOEF(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-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 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, BSCOEF)

             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
```

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, BSCOEF, 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 `MAX(NXDATA . NYDATA).`

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	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.

Description

The routine `BS2IN` computes a tensor product spline interpolant. The tensor product spline interpolant to data $\{(x_i, y_j, f_{ij})\}$, where $1 \le i \le N_x$ and $1 \le j \le N_y$, 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

$$\mathbf{t}_x(k_x) \le x_i \le \mathbf{t}_x(N_x + 1) \quad 1 \le i \le N_x$$

$$\mathbf{t}_y(k_y) \le y_j \le \mathbf{t}_y(N_y + 1) \quad 1 \le j \le N_y$$

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

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum\nolimits_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}x}(x_i)$$

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:

$$\sum_{m=1}^{N_y} h_{mi} B_{m,k_y,\mathbf{t}_y}(y_j) = f_{ij}$$

The same matrix appears in all of the equations above:

$$\left[B_{m,k_y,\mathbf{t}_y}(y_j)\right] \qquad 1 \le m, j \le N_y$$

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

$$\sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) = h_{mi}$$

for m from 1 to N_y, which again involves one factorization and N_y solutions to the different right-hand sides. The routine `BS2IN` is based on the routine `SPLI2D` by de Boor (1978, page 347).

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 = size (XDATA,1).

NYDATA — Number of data points in the y-direction. (Input)
Default: NYDATA = size (YDATA,1).

NZDATA — Number of data points in the *z*-direction. (Input)
Default: `NZDATA` = size (`ZDATA`,1).

LDF — Leading dimension of `FDATA` exactly as specified in the dimension statement of the calling program. (Input)
Default: `LDF` = size (`FDATA`,1).

MDF — Middle dimension of `FDATA` exactly as specified in the dimension statement of the calling program. (Input)
Default: `MDF` = 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`.

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 × 4 × 2 grid are displayed.

```
      USE BS3IN_INT
      USE BSNAK_INT
      USE UMACH_INT
      USE BS3GD_INT
!                                  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-11)/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, BSCOEF, 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
```

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, BSCOEF, WK,
   IWK)
   ```

   The additional arguments are as follows:

WK — Work array of `length MAX((2 * KXORD – 1) * NXDATA, (2 * KYORD – 1) * NYDATA, (2 * KZORD – 1) * NZDATA) + MAX((3 * KXORD – 2) * NXDATA, (3 * KYORD – 2) * NYDATA + (3 * KZORD – 2) * NZDATA) + NXDATA * NYDATA *NZDATA + 2 * MAX(NXDATA, NYDATA, NZDATA).`

IWK — Work array of length `MAX(NXDATA, NYDATA, 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 Ith 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.

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 $1 \le i \le N_x$, $1 \le j \le N_y$, and $1 \le k \le N_z$ has the form

$$\sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y) B_{l,k_z,\mathbf{t}_z}(z)$$

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

$$\begin{aligned} \mathbf{t}_x(k_x) &\le x_i \le \mathbf{t}_x(N_x+1) \quad 1 \le i \le N_x \\ \mathbf{t}_y(k_y) &\le y_j \le \mathbf{t}_y(N_y+1) \quad 1 \le j \le N_y \\ \mathbf{t}_z(k_z) &\le z_k \le \mathbf{t}_z(N_z+1) \quad 1 \le k \le N_z \end{aligned}$$

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

$$\sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) B_{l,k_z,\mathbf{t}_z}(z_k) = f_{ijk}$$

Setting

$$h_{lij} = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j)$$

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:

$$\sum_{l=1}^{N_z} h_{lij} B_{l,k_z,\mathbf{t}_z}(z_k) = f_{ijk}$$

The same interpolation matrix appears in all of the equations above:

$$\left[B_{l,k_z,\mathbf{t}_z}(z_k)\right] \qquad 1 \le l, k \le N_z$$

Thus, we need only factor this matrix once and then apply it to the N_xN_y right-hand sides. Once this is done and we have computed h_{lij}, then we must solve for the coefficients c_{nml} using the relation

$$\sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) = h_{lij}$$

that is the *bivariate* tensor-product problem addressed by the IMSL routine `BS2IN` (page 631). 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).

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`.

Example

For an example of the use of `BSVAL`, see IMSL routine `BSINT` (page 622).

Comments

1. Workspace may be explicitly provided, if desired, by use of `B2VAL/DB2VAL`. The reference is:

   ```
   CALL B2VAL(X, KORDER, XKNOT, NCOEF, BSCOEF, 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.

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` (page 643), 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 **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 `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.

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`.

Example

A spline interpolant to the function

$$f(x) = \sqrt{x}$$

is constructed using BSINT (page 622). 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 BSNAK_INT
      USE UMACH_INT

      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 BSNAK (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
```

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.

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 **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 **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.

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 = size (XVEC,1).

NCOEF — Number of B-spline coefficients. (Input)
Default: NCOEF = 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.

Example

To illustrate the use of BS1GD, we modify the example program for BSDER (page 643). 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
      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), BSCOEF(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 BSNAK (NDATA, XDATA, KORDER, XKNOT)
!                                  Interpolate
      CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
      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, BSCOEF, ANS0)
      CALL BS1GD (1, XVEC, KORDER, XKNOT, BSCOEF, 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
```

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

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 $i = 1, \ldots, n - 1$, a derivative value j, and a B-spline s that is represented by a knot sequence and coefficient sequence, this routine returns the values

$$s^{(j)}(x_i) \quad i = 1, \ldots, n$$

in the array VALUE. The functionality of this routine is the same as that of BSDER (page 643) 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 (page 680), and then uses the IMSL routine PPVAL (page 681) for evaluation.

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.

Example

We integrate the quartic ($k = 5$) spline that interpolates x^3 at the points $\{i/10 : i = -10, \ldots, 10\}$ over the interval [0, 1]. The exact answer is 1/4 since the interpolant reproduces cubic polynomials.

```
USE BSITG_INT
USE BSNAK_INT
```

```
      USE BSINT_INT
      USE UMACH_INT
      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 BSNAK (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
```

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.

Description

The function `BSITG` computes the integral of a spline given its B-spline representation. Specifically, given the knot sequence $\mathbf{t}$ = `XKNOT`, the order k = `KORDER`, the coefficients a = `BSCOEF` , n = `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 $\mathbf{t}_1 = \ldots = \mathbf{t}_k$ and $\mathbf{t}_{n+1} = \ldots = \mathbf{t}_{n+k}$.

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)

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: `BS2VL(X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOEF)`

Specific: The specific interface names are `S_BS2VL` and `D_BS2VL`.

FORTRAN 77 Interface

Single: `BS2VL(X, Y, KXORD, KYORD, XKNOT, YKNOT, NXCOEF, NYCOEF, BSCOEF)`

Double: The double precision function name is `DBS2VL`.

Example

For an example of the use of `BS2VL`, see IMSL routine `BS2IN` (page 631).

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 * MAX(KXORD, KYORD) + KYORD`.

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` (page 653), 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

$$s(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y)$$

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`).

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`.

Example

In this example, a spline interpolant *s* to a function *f* is constructed. We use the IMSL routine `BS2IN` (page 631) 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 × 4 grid and then displayed.

```
      USE BS2DR_INT
      USE BSNAK_INT
      USE UMACH_INT
      USE BS2IN_INT
!                                 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

```
                                 (2,1)
    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       0.000001
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
```

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 * MAX(KXORD, KYORD) + KYORD`.

2. Informational errors

Type	Code	
3	1	The point `X` does not satisfy `XKNOT(KXORD) .LE. X .LE. XKNOT(NXCOEF + 1)`.
3	2	The point `Y` does not satisfy `YKNOT(KYORD) .LE. Y .LE. YKNOT(NYCOEF + 1)`.

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

$$s^{(p,q)}(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}^{(p)}(x) B_{m,k_y,\mathbf{t}_y}^{(q)}(y)$$

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`).

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 = size (XVEC,1).

NY — Number of grid points in the Y-direction. (Input)
Default: NY = size (YVEC,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.

LDVALU — Leading dimension of VALUE exactly as specified in the dimension statement of the calling program. (Input)
Default: LDVALU = size (VALUE,1).

FORTRAN 90 Interface

Generic: `CALL BS2GD (IXDER, IDER, 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`.

Example

In this example, a spline interpolant *s* to a function *f* is constructed. We use the IMSL routine `BS2IN` (page 631) 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 × 4 grid and then displayed.

```
      USE BS2GD_INT
      USE BS2IN_INT
      USE BSNAK_INT
      USE UMACH_INT
!                                  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
!                yj                  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-11)/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

```
                                       (2,1)
             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       0.000001
        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
```

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 `YKNOT (KYORD) .LE. YVEC(I) .LE. YKNOT(NYCOEF` + 1)
4	3	`XVEC` is not strictly increasing.
4	4	`YVEC` is not strictly increasing.

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 $i = 1, \ldots, nx$ and $j = 1, \ldots, ny$ given the coefficients c by computing (for all (x, y) in the grid)

$$s^{(p,q)}(x, y) = \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nm} B_{n,k_x,\mathbf{t}_x}^{(p)}(x) B_{m,k_y,\mathbf{t}_y}^{(q)}(y)$$

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 $y_j < y_{j+1}$.

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 = size (XKNOT,1) – KXORD.

NYCOEF — Number of B-spline coefficients in the Y-direction. (Input)
Default: NYCOEF = 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`.

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 $\{(i/10, j/5) : i = -10, \ldots, 10 \text{ and } j = 0, \ldots, 5\}$ 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
!                                  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, BSCOEF(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, BSCOEF)
!                                  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)
!                                  Get output unit number
      CALL UMACH (2, NOUT)
!                                  Print results
      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
```

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 * `(MAX(KXORD, KYORD) + 1) + NYCOEF`.

2. Informational errors

Type	Code	
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).
4	13	Multiplicity of the knots cannot exceed the order of the spline.
4	14	The knots must be nondecreasing.

Description

The function `BS2IG` computes the integral of a tensor-product two-dimensional spline given its B-spline representation. Specifically, given the knot sequence $\mathbf{t}_x$ = `XKNOT`, $\mathbf{t}_y$ = `YKNOT`, the order k_x = `KXORD`, k_y = `KYORD`, the coefficients β = `BSCOEF`, the number of coefficients n_x = `NXCOEF`, n_y = `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

$$B_{i,j}(x, y) = B_{i,k_x,\mathbf{t}_x}(x) B_{j,k_y,\mathbf{t}_y}(y)$$

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,$t_1 = \ldots = t_k$ and $t_{n+1} = \ldots = t_{n+k}$, where k is the order of the spline in the x or y direction.

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.

Example

For an example of the use of BS3VL, see IMSL routine BS3IN (page 635).

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 * MAX(KXORD, KYORD, KZORD) + KYORD * KZORD + KZORD.

Description

The function BS2IG 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` (page 666), 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

$$s(x, y, z) = \sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y) B_{l,k_z,\mathbf{t}_z}(z)$$

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`).

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 (`IXDER`, `IYDER`, `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 `NXCOEF` + `KXORD` containing the knot sequence in the `X`-direction. (Input)
`KNOT` 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: 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.

Example

In this example, a spline interpolant s to a function $f(x, y, z) = x^4 + y(xz)^3$ is constructed using BS3IN (page 635). 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 BSNAK_INT
      USE UMACH_INT
!                                 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       BSCOEF(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-11)/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 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,BSCOEF)
              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

```
                                      (2,0,1)
    X           Y           Z    S      (X,Y,Z)   Error
-1.0000      0.0000      0.0000   -0.000107    0.000107
-1.0000      0.0000      1.0000    0.000053   -0.000053
-1.0000      0.3333      0.0000    0.064051   -0.064051
-1.0000      0.3333      1.0000   -5.935941   -0.064059
-1.0000      0.6667      0.0000    0.127542   -0.127542
-1.0000      0.6667      1.0000  -11.873034   -0.126966
-1.0000      1.0000      0.0000    0.191166   -0.191166
-1.0000      1.0000      1.0000  -17.808527   -0.191473
-0.3333      0.0000      0.0000   -0.000002    0.000002
-0.3333      0.0000      1.0000    0.000000    0.000000
-0.3333      0.3333      0.0000    0.021228   -0.021228
-0.3333      0.3333      1.0000   -1.978768   -0.021232
-0.3333      0.6667      0.0000    0.042464   -0.042464
-0.3333      0.6667      1.0000   -3.957536   -0.042464
-0.3333      1.0000      0.0000    0.063700   -0.063700
-0.3333      1.0000      1.0000   -5.936305   -0.063694
 0.3333      0.0000      0.0000   -0.000003    0.000003
 0.3333      0.0000      1.0000    0.000000    0.000000
 0.3333      0.3333      0.0000   -0.021229    0.021229
 0.3333      0.3333      1.0000    1.978763    0.021238
 0.3333      0.6667      0.0000   -0.042465    0.042465
 0.3333      0.6667      1.0000    3.957539    0.042462
 0.3333      1.0000      0.0000   -0.063700    0.063700
 0.3333      1.0000      1.0000    5.936304    0.063697
 1.0000      0.0000      0.0000   -0.000098    0.000098
 1.0000      0.0000      1.0000    0.000053   -0.000053
 1.0000      0.3333      0.0000   -0.063855    0.063855
 1.0000      0.3333      1.0000    5.936146    0.063854
 1.0000      0.6667      0.0000   -0.127631    0.127631
 1.0000      0.6667      1.0000   11.873067    0.126933
 1.0000      1.0000      0.0000   -0.191442    0.191442
 1.0000      1.0000      1.0000   17.807940    0.192060
```

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, BSCOEF, WK)
   ```

The additional argument is:

WK — Work array of length 3 * `MAX0(KXORD, KYORD, KZORD) + KYORD * KZORD + KZORD`.

2. Informational errors

Type	Code	
3	1	The point `X` does not satisfy `XKNOT(KXORD) .LE. X .LE. XKNOT(NXCOEF` + 1).
3	2	The point `Y` does not satisfy `YKNOT(KYORD) .LE. Y .LE. YKNOT(NYCOEF` + 1).
3	3	The point `Z` does not satisfy `ZKNOT (KZORD) .LE. Z .LE. ZKNOT(NZCOEF` + 1).

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`).

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`.

Example

In this example, a spline interpolant s to a function $f(x, y, z) = x^4 + y(xz)^3$ is constructed using `BS3IN` (page 635). 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 × 4 × 2 grid and then displayed.

```
      USE BS3GD_INT
      USE BS3IN_INT
      USE BSNAK_INT
      USE UMACH_INT
      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       BSCOEF(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, BSCOEF)
!
      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, BSCOEF, 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

```
                                            (2,0,1)
          X            Y          Z          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
```

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, BSCOEF, 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 XKNOT(KXORD) ≤ XVEC(I) ≤ XKNOT(NXCOEF + 1).
3	2	YVEC(I) does not satisfy YKNOT(KYORD) ≤ YVEC(I) ≤ YKNOT(NYCOEF + 1).
3	3	ZVEC(I) does not satisfy ZKNOT(KZORD) ≤ ZVEC(I) ≤ ZKNOT(NZCOEF + 1).
4	4	XVEC is not strictly increasing.
4	5	YVEC is not strictly increasing.
4	6	ZVEC is not strictly increasing.

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 $i = 1, \ldots, nx$, $j = 1, \ldots, ny$, and $k = 1, \ldots, nz$ given the coefficients c by computing (for all (x, y, z) on the grid)

$$s^{(p,q,r)}(x, y, z) = \sum_{l=1}^{N_z} \sum_{m=1}^{N_y} \sum_{n=1}^{N_x} c_{nml} B_{n,k_x,\mathbf{t}_x}^{(p)}(x) B_{m,k_y,\mathbf{t}_y}^{(q)}(y) B_{l,k_z,\mathbf{t}_z}^{(r)}(z)$$

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). The grid must be ordered in the sense that $x_i < x_{i+1}$, $y_j < y_{j+1}$, and $z_k < z_{k+1}$.

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.

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

$$\left\{\left(i/10,\ j/5,\ m/7\right) : i = -10, \ldots, 10,\ j = 0, \ldots, 5, \text{ and } m = 0, \ldots, 7\right\}$$

over the rectangle [0, 1] × [.5, 1] × [0, .5]. The exact answer is 11/128.

```
USE BS3IG_INT
USE BS3IN_INT
```

```
      USE BSNAK_INT
      USE UMACH_INT
!                                  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, BSCOEF(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-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)/FLOAT(NYDATA-1)
   20 CONTINUE
!                                  Generate knot sequence
      CALL BSNAK (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 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
      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
```

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, BSCOEF, WK)
   ```

   The additional argument is:

   WK — Work array of length `4 * (MAX(KXORD, KYORD, KZORD) + 1) + NYCOEF + NZCOEF`.

2. Informational errors

Type	Code	
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.

Description

The routine `BS3IG` computes the integral of a tensor-product three-dimensional spline, given its B-spline representation. Specifically, given the knot sequence $\mathbf{t}_x$ = `XKNOT`, $\mathbf{t}_y$ = `YKNOT`, $\mathbf{t}_z$ = `ZKNOT`, the order k_x = `KXORD`, k_y = `KYORD`, k_z = `KZORD`, the coefficients β = `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

$$B_{ijm}(x, y, z) = B_{i,k_x,\mathbf{t}_x}(x) B_{j,k_y,\mathbf{t}_y}(y) B_{m,k_z,\mathbf{t}_z}(z)$$

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, $\mathbf{t}_1 = \ldots = \mathbf{t}_k$ and $\mathbf{t}_{n+1} = \ldots = \mathbf{t}_{n+k}$, where *k* is the order of the spline in the *x*, *y*, or *z* direction.

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, BSCOEF, NPPCF, BREAK, PPCOEF)`

Specific: The specific interface names are `S_BSCPP` and `D_BSCPP`.

FORTRAN 77 Interface

Single: `CALL BSCPP (KORDER, XKNOT, NCOEF, BSCOEF, NPPCF, BREAK, PPCOEF)`

Double: The double precision name is `DBSCPP`.

Example

For an example of the use of `BSCPP`, see `PPDER` (page 684).

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.

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.

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 = size (PPCOEF,1).

NINTV — Number of polynomial pieces. (Input)
Default: NINTV = 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.

Example

In this example, a spline interpolant to a function *f* is computed using the IMSL routine BSINT (page 622). 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 (page 680). The piecewise polynomial is evaluated using PPVAL. These values are compared to the corresponding values of *f*.

```
USE PPVAL_INT
USE BSNAK_INT
USE BSCPP_INT
USE BSINT_INT
```

```
      USE UMACH_INT
      INTEGER    KORDER, NCOEF, NDATA, NKNOT
      PARAMETER  (KORDER=4, NCOEF=20, NDATA=20, NKNOT=NDATA+KORDER)
!
      INTEGER    I, NOUT, NPPCF
      REAL       BREAK(NCOEF), BSCOEF(NCOEF), EXP, F, FDATA(NDATA),&
                 FLOAT, PPCOEF(KORDER,NCOEF), S, X, XDATA(NDATA),&
                 XKNOT(NKNOT)
      INTRINSIC  EXP, FLOAT
!                                  Define function
      F(X) = X*EXP(X)
!                                  Set up interpolation points
      DO 30  I=1, NDATA
         XDATA(I) = FLOAT(I-1)/FLOAT(NDATA-1)
         FDATA(I) = F(XDATA(I))
   30 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 40  I=1, NDATA
         X = FLOAT(I-1)/FLOAT(NDATA-1)
!                                  Compute value of the piecewise
!                                  polynomial
         S = PPVAL(X,BREAK,PPCOEF)
         WRITE (NOUT,'(2F12.3, E14.3)') X, S, F(X) - S


   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
```

Description

The routine `PPVAL` evaluates a piecewise polynomial at a given point. This routine is a special case of the routine `PPDER` (page 684), 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).

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` = size (`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`.

Example

In this example, a spline interpolant to a function *f* is computed using the IMSL routine `BSINT` (page 622). 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` (page 680). 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
      INTEGER    KORDER, NCOEF, NDATA, NKNOT
      PARAMETER  (KORDER=4, NCOEF=20, NDATA=20, NKNOT=NDATA+KORDER)
!
      INTEGER    I, NOUT, NPPCF
      REAL       BREAK(NCOEF), BSCOEF(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
```

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 ξ (a vector of length N = `NINTV` + 1), and if the coefficients of the piecewise polynomial representation are stored in **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 ξ_1, then *i* is set to 1 in the above formula; if *x* is greater than or equal to ξ_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.

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 `NINTV` + 1 containing the breakpoints for the piecewise polynomial representation. (Input)
`BREAK` must be strictly increasing.

PPCOEF — Matrix of size `KORDER` by `NINTV` 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` = size (`XVEC`,1).

KORDER — Order of the polynomial. (Input)
Default: `KORDER` = size (`PPCOEF`,1).

NINTV — Number of polynomial pieces. (Input)
Default: `NINTV` = size (`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`.

Example

To illustrate the use of `PP1GD`, we modify the example program for `PPDER` (page 684). 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

      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 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)
!                                  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
```

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.

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 $i = 1, \ldots, n - 1$, 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

$$f^{(j)}(x_i) \quad i = 1, \ldots, n$$

in the array VALUE. The functionality of this routine is the same as that of PPDER (page 684) called in a loop, however PP1GD is much more efficient.

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: `PP1TG (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.

Example

In this example, we compute a quadratic spline interpolant to the function x^2 using the IMSL routine BSINT (page 622). 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
      INTEGER    KORDER, NDATA, NKNOT
      PARAMETER  (KORDER=3, NDATA=10, NKNOT=NDATA+KORDER)
!
      INTEGER    I, NOUT, NPPCF
      REAL       A, B, BREAK(NDATA), BSCOEF(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 BSNAK (NDATA, XDATA, KORDER, XKNOT)
!                                  Interpolate
      CALL BSINT (NDATA, XDATA, FDATA, KORDER, XKNOT, BSCOEF)
!                                  Convert to piecewise polynomial
      CALL BSCPP (KORDER, XKNOT, NDATA, BSCOEF, NPPCF, BREAK, PPCOEF)
!                                  Compute the integral of F over
!                                  [0.0,0.5]
      A     = 0.0
      B     = 0.5
      VALUE = PPITG(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
```

Description

The routine `PPITG` evaluates the integral of a piecewise polynomial over an interval.

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`.

Example

In this example, the value of sin *x* is approximated at π/4 by using `QDVAL` on a table of 33 equally spaced values.

```
      USE IMSL_LIBRARIES
      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
```

Comments

Informational error

Type	Code	
4	3	The `XDATA` values must be strictly increasing.

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 $i = 1, \ldots, n$ 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 $q(x)$.

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`.

Example

In this example, the value of sin *x* and its derivatives are approximated at π/4 by using `QDDER` on a table of 33 equally spaced values.

```
      USE IMSL_LIBRARIES
      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

```
                        IDER
   X       IDER       F     (X)      QDDER       ERROR

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
```

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.

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 $i = 1, \ldots, n$ 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 = `IDERIV`.

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`.

Example

In this example, the value of sin(*x* + *y*) at *x* = *y* = π/4 is approximated by using `QDVAL` on a table of size 21 × 42 equally spaced values on the unit square.

```
      USE IMSL_LIBRARIES
      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
!                                  Define function
      F(X,Y) = SIN(X+Y)
!                                  Set up X-grid
      DO 10  I=1, NXDATA
         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
```

Comments

Informational errors

Type   Code

4	6	The `XDATA` values must be strictly increasing.
4	7	The `YDATA` values must be strictly increasing.

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 $i = 1, \ldots, n_x$ and $j = 1, \ldots, n_y$ 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), $(x_{i\pm1}, y_j)$, and $(x_i, y_{j\pm1})$. The sixth point is the nearest point to (x, y) of the grid points $(x_{i\pm1}, y_{j\pm1})$. The value $q(x, y)$ is returned by `QD2VL`.

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 fucntion name is DQD2DR.

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 × 42 equally spaced values on the rectangle [0, 2] × [0, 2].

```
      USE IMSL_LIBRARIES

      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

```
                                       (IDX,IDY)
     X          Y     IDX   IDY    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
```

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.

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 $i = 1, \ldots, n_x$ and $j = 1, \ldots, n_y$ 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), $(x_{i\pm1}, y_j)$, and $(x_i, y_{j\pm1})$. The sixth point is the nearest point to (x, y) of the grid points $(x_{i\pm1}, y_{j\pm1})$. The value $q^{(p,\ r)}(x, y)$ is returned by QD2DR, where p = IXDER and r = IYDER.

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` = 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: `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`.

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
      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
```

Comments

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.

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 $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$, and $k = 1, \ldots, n_z$ 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

$$\left(x_i, y_j, z_k\right), \left(x_{i\pm1}, y_j, z_k\right), \left(x_i, y_{j\pm1}, z_k\right) \text{ and } \left(x_i, y_j, z_{k\pm1}\right)$$

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.

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`.

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 × 42 × 18 equally spaced values on the cube $[0, 2]^3$.

```
      USE IMSL_LIBRARIES
      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
```

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.

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 $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$, and $k = 1, \ldots, n_z$ 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

$$\left(x_i, y_j, z_k\right), \left(x_{i\pm1}, y_j, z_k\right), \left(x_i, y_{j\pm1}, z_k\right) \text{ and } \left(x_i, y_j, z_{k\pm1}\right)$$

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 = `IXDER`, r = `IYDER`, and t = `IZDER`.

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`.

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 3×3 grid.

```
      USE IMSL_LIBRARIES
      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, NXOUT
         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
```

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 * `NDATA + NXOUT * NYOUT`.

   WK — Work array of length 6 * `NDATA`.

2. Informational errors

Type	Code	
4	5	The data point values must be distinct.
4	6	The `XOUT` values must be strictly increasing.
4	7	The `YOUT` values must be strictly increasing.

3. This method of interpolation reproduces linear functions.

Description

This routine is designed to compute a C^1 interpolant to scattered data in the plane. Given the data points

$$\left\{\left(x_i, y_i, f_i\right)\right\}_{i=1}^{N} \; in \; \mathbf{R}^3$$

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

$$\left\{\left(x_i, y_i\right)\right\}_{i=1}^{N}$$

is computed. On each triangle *T* in this triangulation, *s* has the form

$$s\left(x, y\right) = \sum_{m+n\le 5} c_{mn}^{T} x^m y^n \qquad \forall x, y \in T$$

Thus, *s* is a bivariate quintic polynomial on each triangle of the triangulation. In addition, we have

$$s(x_i, y_i) = f_i \qquad\qquad \text{for } i = 1, \ldots, N$$

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.

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)

I	ISTAT(I)
1	Mean of XDATA
2	Mean of YDATA
3	Sample variance of XDATA
4	Sample variance of YDATA
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`.

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
      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
```

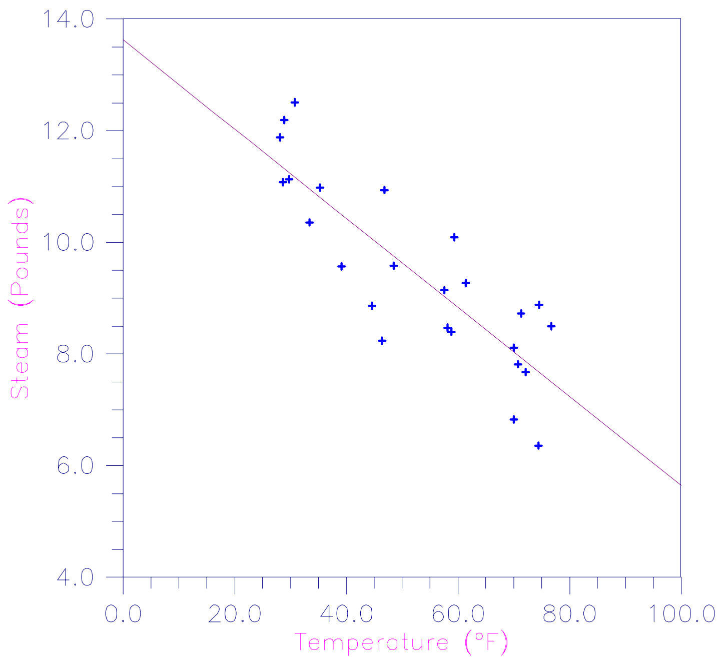

Figure 3-5   Plot of the Data and the Least Squares Line

Comments

Informational error

Type	Code	
4	1	Each (x, y) point contains NaN (not a number). There are no valid data.

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.

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

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \cdots + \hat{\beta}_k x^k$$

Optional Arguments

NOBS — Number of observations. (Input)
Default: `NOBS` = size (`XDATA`,1).

NDEG — Degree of polynomial. (Input)
Default: `NDEG` = 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 $i = 1, 2, \ldots,$ `NDEG`, `SSPOLY`(`i` + 1) contains the sum of squares due to x^i adjusted for the mean, $x, x^2, \ldots,$ 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`.

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
      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
```

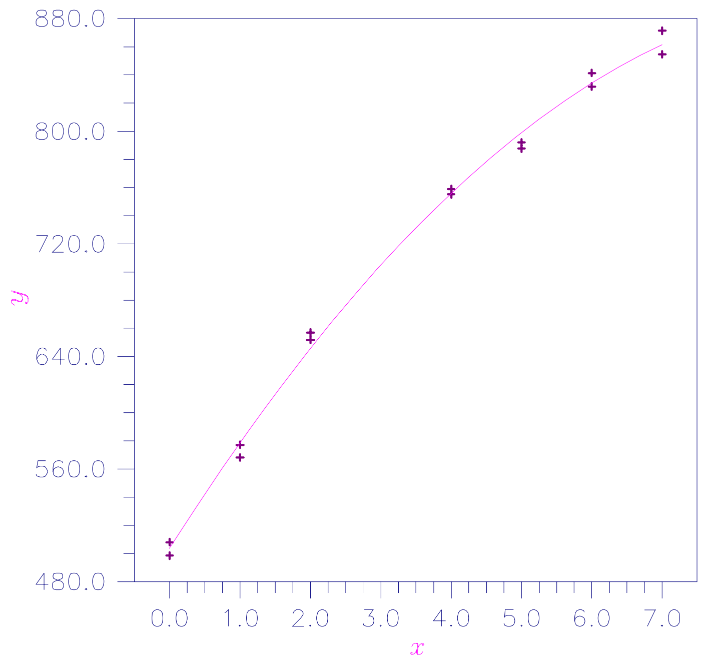

Figure 3-6   Plot of Data and Second Degree Polynomial Fit

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 * NOBS + 11 * NDEG + 5 + (NDEG + 1) * (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.

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,

$$R^2 = \frac{\sum_{i=1}^{n}\left(\hat{y}_i - \overline{y}\right)^2}{\sum_{i=1}^{n}\left(y_i - \overline{y}\right)^2}100\%$$

where

$$\hat{y}_i$$

is the fitted y value at x_i and

$$\overline{y}$$

(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).

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`) +…+ `A`(`NBASIS`) * `F`(`NBASIS`, `X`) if `INTCEP` = 0 and is approximated by `A`(1) + `A`(2) * `F`(1, `X`) +…+ `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` + I) 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`.

Example

In this example, we fit the following two functions (indexed by δ)

$$1 + \sin x + 7 \sin 3x + \delta\varepsilon$$

where ε is random uniform deviate over the range [−1, 1], and δ 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, …, 4, with and without the intercept.

```
      USE FNLSQ_INT
      USE RNSET_INT
      USE UMACH_INT
      USE RNUNF_INT
      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)
!                                  Set random number seed
      CALL RNSET (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
```

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 (INTCEP + NBASIS)**2 + 4 * (INTCEP + NBASIS) + IWT + 1. On output, the first (INTCEP + 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 A are set to zero.
3	2	Linear dependence of the constant function and basis functions exists. One or more components of A are set to zero.
4	1	Negative weight encountered.

Description

The routine FNLSQ computes a best least-squares approximation to given univariate data of the form

$$\left\{\left(x_i, f_i\right)\right\}_{i=1}^{N}$$

by *M* basis functions

$$\{F_j\}_{j=1}^{M}$$

(where M = NBASIS). In particular, if INTCEP = 0, this routine returns the error sum of squares SSE and the coefficients *a* which minimize

$$\sum_{i=1}^{N} w_i \left(f_i - \sum_{j=1}^{M} a_j F_j\left(x_i\right) \right)^2$$

where w = WEIGHT, N = NDATA, x = XDATA, and, f = 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.

$$\sum_{i=1}^{N} w_i \left(f_i - a_1 - \sum_{j=1}^{M} a_{j+1} F_j\left(x_i\right) \right)^2$$

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.

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, KORDER, XKNOT, NCOEF, BSCOEF [,…])`

Specific: The specific interface names are `S_BSLSQ` and `D_BSLSQ`.

FORTRAN 77 Interface

Single: `CALL BSLSQ (NDATA, XDATA, FDATA, WEIGHT, KORDER, XKNOT, NCOEF, BSCOEF)`

Double: The double precision name is `DBSLSQ`.

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
      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/
!                                  Define function
      F(X) = 8.0*X*(1.0-X)
!                                  Set random number seed
      CALL RNSET (12345679)
      NDATA = 50
!                                  Set up interior knots
```

```
      DO 10  I=1, NCOEF - KORDER + 2
         XKNOT(I+KORDER-1) = FLOAT(I-1)/FLOAT(NCOEF-KORDER+1)
   10 CONTINUE
!                                 Stack knots
      DO 20  I=1, KORDER - 1
         XKNOT(I) = XKNOT(KORDER)
         XKNOT(I+NCOEF+1) = XKNOT(NCOEF+1)
   20 CONTINUE
!                                 Set up data points excluding
!                                 the endpoints 0 and 1.
!                                 The function values have noise
!                                 introduced.
      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
!                                 Compute least squares B-spline
!                                 representation.
      CALL BSLSQ (XDATA1, FDATA1, KORDER, XKNOT, NCOEF, BSCOF1)
!                                 Now use same XDATA values but with
!                                 the endpoints included.  These
!                                 points will have large weights.
      NDATA = 52
      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))
!                                 Compute least squares B-spline
!                                 representation.
      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
```

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 + NCOEF) * 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 `KORD`th knot.
4	9	The largest element of the data point array must be less than or equal to the (`NCOEF` + 1)st knot.

3. The B-spline representation can be evaluated using `BSVAL` (page 641), and its derivative can be evaluated using `BSDER` (page 643).

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 $i = 1, \ldots, N$ (where N = `NDATA`). In other words, it finds B-spline coefficients, a = `BSCOEF`, such that

$$\sum_{i=1}^{N} \left| f_i - \sum_{j=1}^{m} a_j B_j\left(x_i\right) \right|^2 w_i$$

is a minimum, where m = `NCOEF` and B_j denotes the j-th B-spline for the given order, `KORDER`, 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

$$\int \left| F - s \right|^2 dx$$

The Gauss quadrature weights and points can be obtained using the IMSL routine `GQRUL` (see Chapter 4, Integration and Differentiation).

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 `NCOEF` + `KORDER` containing the initial guess of knots. (Input)
`XGUESS` must be nondecreasing.

XKNOT — Array of length `NCOEF` + `KORDER` containing the (nondecreasing) knot sequence. (Output)

BSCOEF — Array of length `NCOEF` containing the B-spline representation. (Output)

SSQ — The square root of the sum of the squares of the error. (Output)

Optonal Arguments

NDATA — Number of data points. (Input)
`NDATA` must be at least 2.
Default: `NDATA` = 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`.

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 (page 725) 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
      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
```

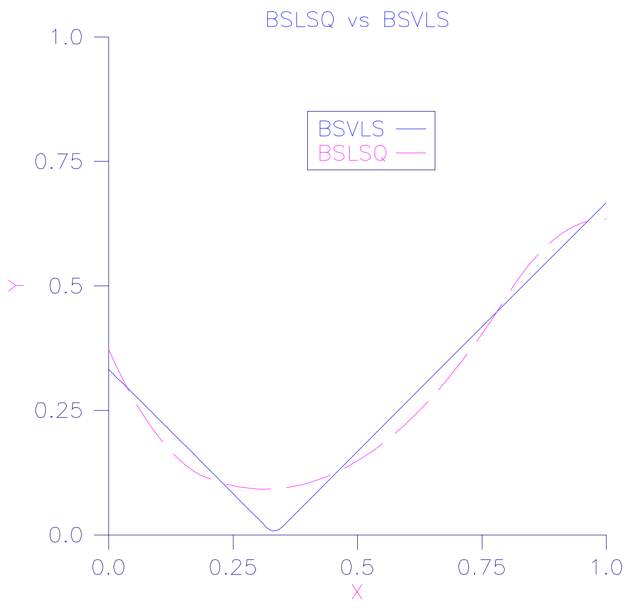

Figure 3-7 `BSVLS vs. BSLSQ`

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, BSCOEF, 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	
3	12	The knots found to be optimal are stacked more than KORDER. 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.

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

$$F(a,\mathbf{t}) = \sum_{i=1}^{M} w_i \left(f_i - \sum_{j=1}^{N} a_j B_{j,k,\mathbf{t}}(x_j) \right)^2$$

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 **t**.

The technique employed in BSVLS uses the fact that for a fixed knot sequence **t** the minimization in a is a linear least-squares problem that can be solved by calling the IMSL routine BSLSQ (page 725). Thus, we can think of our objective function F as a function of just **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, $\mathbf{t}^g$ = `XGUESS`, for the knot sequence. This guess must be a *valid* knot sequence for the splines of order k with

$$\mathbf{t}_1^g \le \ldots \le \mathbf{t}_k^g \le x_i \le \mathbf{t}_{N+1}^g \le \ldots \le \mathbf{t}_{N+k}^g, \qquad i = 1, \ldots, M$$

with $\mathbf{t}^g$ nondecreasing, and

$$\mathbf{t}_i^g < \mathbf{t}_{i+k}^g \quad i = 1, \ldots, N$$

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`.

CONFT

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 ≤ `NHARD` ≤ `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 preceed 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	$\mathrm{BL(I)} = f^{(d_i)}(x_i)$
2	$f^{(d_i)}(x_i) \le \mathrm{BU(I)}$
3	$f^{(d_i)}(x_i) \ge \mathrm{BL(I)}$
4	$\mathrm{BL(I)} =\le f^{(d_i)}(x_i) \le \mathrm{BU(I)}$
$(d_i = -1)1$	$\mathrm{BL(I)} = \int_c^d f(t)\,dt$
$(d_i = -1)2$	$\int_c^d f(t)\,dt \le \mathrm{BU(I)}$
$(d_i = -1)3$	$\int_c^d f(t)\,dt \ge \mathrm{BL(I)}$
$(d_i = -1)4$	$\mathrm{BL(I)} \le \int_c^d f(t)\,dt \le \mathrm{BU(I)}$
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.

$\mathrm{ITYPE(I)}$	$\mathrm{I-th\ Contraint}$
-1	$\mathrm{BL(I)} = f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1})$
-2	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \le \mathrm{BU(I)}$
-3	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \ge \mathrm{BL(I)}$
-4	$\mathrm{BL(I)} \le f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \le \mathrm{BU(I)}$

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`.

Example 1

This is a simple application of `CONFT`. 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` (page 725). We then force the derivative to be greater than 0 at `NXVAL` = 15 equally spaced points and call `CONFT`. The resulting curve is monotone. We print the error for the two fits averaged over 100 equally spaced points.

```
      USE IMSL_LIBRARIES
      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, ERRNFT, 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.
      CALL BSLSQ (XDATA, FDATA, KORDER, XKNOT, NCOEF, BSCLSQ)
!                                  Construct the constraints for
!                                  CONFT.
      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 CONFT
      NHARD = 0
      CALL CONFT (XDATA, FDATA, XVAL, NHARD, 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
         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 CONFT fit:  ', F8.5)
      END
```

Output

```
Average error with BSLSQ fit:   0.20250
Average error with CONFT fit:   0.14334
```

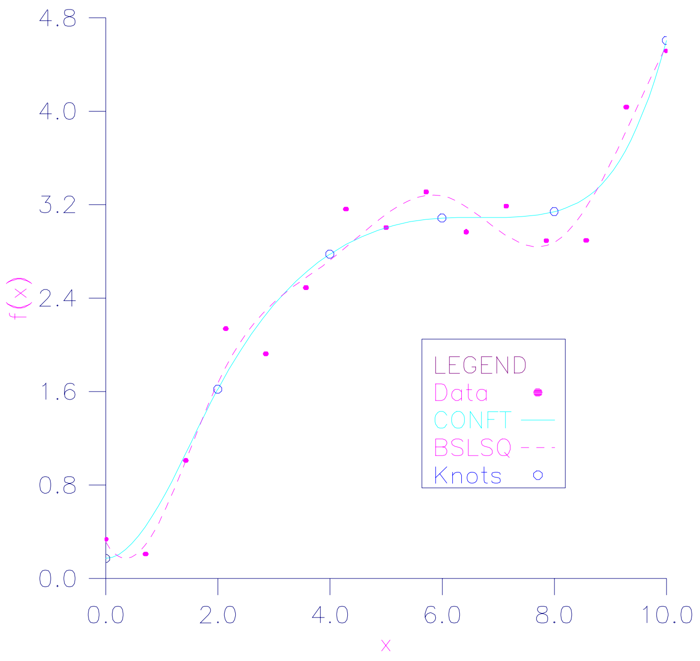

Figure 3-8 CONFT vs. BSLSQ Forcing Monotonicity

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 permutaion 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	
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 KORD-th knot.
4	15	The largest element of the data point array must be less than or equal to the (NCOEF + 1)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 constaint.
4	20	The upper limit of integration must be greater than the lower limit of integration for constraints involving the integral of the approximation.

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.

$$\begin{aligned} E_p[f] &= f^{(j_p)}(y_p) \\ \text{or} \quad &= f^{(j_p)}(y_p) - f^{(j_{p+1})}(y_{p+1}) \\ \text{or} \quad &= \int_{y_p}^{y_{p+1}} f(t)\,dt \\ \text{or} \quad &= \text{periodic end conditions} \end{aligned}$$

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 $i = 1, \ldots, N$ (where N = `NDATA`), that is the data which is to be fit. Second, we have the weights to be used in the least squares fit (w = `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.

$$\sum_{i=1}^{N} \left| f_i - \sum_{j=1}^{m} a_j B_j\left(x_i\right) \right|^2 w_i$$

$$\text{subject to} \qquad E_p \left[\sum_{j=1}^{m} a_j B_j \right] \in I_p \quad p = 1, \ldots, n_f$$

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.

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` (page 725). 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:

$$s(-7) \quad \geq 0$$
$$\int_{-7}^{7} s(x)dx \quad \leq 2.3$$
$$s(-7) \quad = s(7)$$

Notice that the last constraint was generated using the periodic option (requiring only the zeroeth derivative to be periodic). We print the error for the two fits averaged over 100 equally spaced points.

```
      USE IMSL_LIBRARIES
      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 deafult 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
!                                 CONFT
      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 CONFT
      CALL CONFT (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 CONFT fit:  ', F8.5)
      END
```

Output

```
Average error with BSLSQ fit:   0.01783
Average error with CONFT fit:   0.01339
```

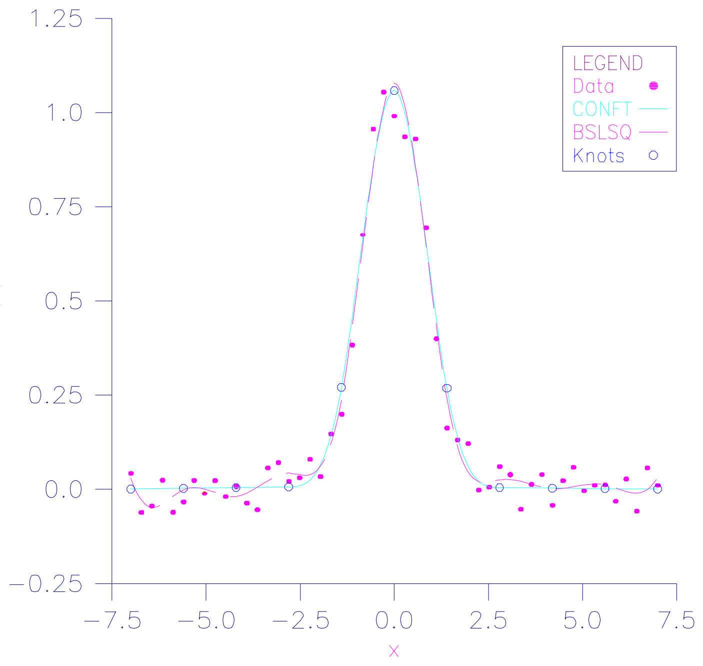

Figure 3-9   CONFT vs. BSLSQ Approximating $1/(1 + x^4)$

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 = size (XDATA,1).

NYDATA — Number of data points in the Y-direction. (Input)
Default: NYDATA = size (YDATA,1).

LDF — Leading dimension of FDATA exactly as specified in the dimension statement of calling program. (Input)
Default: LDF = size (FDATA,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.

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.

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, ε is a uniform random variable with range $[-1, 1]$. We sample this function on a 100×50 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 3×5 grid and compare these values with the values of the tensor-product spline that was computed using the IMSL routine `BSLS2`.

```
      USE IMSL_LIBRARIES
      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
!                                 Define function
      F(X,Y) = EXP(X)*SIN(X+Y)
!                                 Set random number seed
      CALL RNSET (1234579)
!                                 Set up X knot sequence.
      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
!                                 Stack knots.
      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 BSLS2 (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
```

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 (`NXCOEF` + 1) `*` `NYDATA` + `KXORD` `*` `NXCOEF` + `KYORD` `*` `NYCOEF` + 3 `*` `MAX`(`KXORD`, `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.

Description

The routine `BSLS2` 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 = `XDATA` and y = `YDATA` of length n = `NXDATA` and m = `NYDATA`, respectively. A two-dimensional array f = `FDATA` contains the data values that are to be fit. The two vectors w_x = `XWEIGH` and w_y = `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 = `KXORD`, $\mathbf{t}_x$ = `XKNOT`, and N = `NXCOEF` for the spline in the first variable, and in k_y = `KYORD` , $\mathbf{t}_y$ = `YKNOT` and M = `NYCOEF` for the spline in the second variable. The coefficients of the resulting tensor-product spline are returned in c = `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

$$\sum_{i=1}^{n}\sum_{j=1}^{m} w_x(i)\, w_y(j)\left[\sum_{k=1}^{N}\sum_{l=1}^{M} c_{kl} B_{kl}\left(x_i, y_j\right) - f_{ij}\right]^2$$

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` (page 651) and its partial derivatives can be evaluated using `BS2DR` (page 653).

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 = size (XDATA,1).

NYDATA — Number of data points in the *y*-direction. (Input)
NYDATA must be greater than or equal to NYCOEF.
Default: NYDATA = size (YDATA,1).

NZDATA — Number of data points in the *z*-direction. (Input)
NZDATA must be greater than or equal to NZCOEF.
Default: NZDATA = size (ZDATA,1).

LDFDAT — Leading dimension of FDATA exactly as specified in the dimension statement of the calling program. (Input)
Default: LDFDAT = size (FDATA,1).

MDFDAT — Second dimension of FDATA exactly as specified in the dimension statement of the calling program. (Input)
Default: MDFDAT = size (FDATA,2).

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.

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`.

Example

The data for this example arise from the function $e^{(y-z)}\sin(x+y)+\varepsilon$ on the rectangle $[0, 3] \times [0, 2] \times [0, 1]$. Here, ε 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
      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       BSCOEF(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
```

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 `NYCOEF` * (`NZDATA` + `KYORD` + `NZCOEF`) + `NZDATA` * (1 + `NYDATA`) + `NXCOEF` * (`KXORD` + `NYDATA` * `NZDATA`) + `KZORD` * `NZCOEF` + 3 * `MAX0`(`KXORD`, `KYORD`, `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.

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`, $\mathbf{t}_x$ = `XKNOT`, and K = `NXCOEF` for the spline in the first variable, in k_y = `KYORD`, $\mathbf{t}_y$ = `YKNOT` and L = `NYCOEF` for the spline in the second variable, and in k_z = `KZORD`, $\mathbf{t}_z$ = `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

$$\sum_{i=l}^{k}\sum_{j=1}^{l}\sum_{p=1}^{m} w_x(i)\,w_y(j)\,w_z(p)\left[\sum_{s=1}^{K}\sum_{t=1}^{L}\sum_{u=1}^{M} c_{stu}B_{stu}\left(x_i, y_j, z_p\right) - f_{ijp}\right]^2$$

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

$$B_{stu}(x, y, z) = B_{s,k_x,\mathbf{t}_x}(x)\,B_{t,k_y,\mathbf{t}_y}(y)\,B_{u,k_z,\mathbf{t}_z}(z)$$

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` (page 664) and its partial derivatives can be evaluated using `BS3DR` (page 666). If the values on a grid are desired then we recommend `BS3GD` (page 670).

CSSED

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

NDATA — Number of data points. (Input)
Default: `NDATA` = 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`.

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
      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
```

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 * NDATA + 30.

   IWK — Work array of length 2 * NDATA.

2. Informational error

Type	Code	
3	1	The maximum number of iterations allowed has been reached.

3. The arrays FDATA and SDATA may the the same.

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 NDATA = 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 $s = f$)

$$(x_j, s_j) \qquad j = i-3, \ldots, i+3 \; j \neq i,$$

where $i = 4, \ldots, n-3$ using CSAKM (page 600). 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

$$\left|pe_i\right| \le sc\left(x_{i+3} - x_{i-3}\right)/6 \qquad i = 4, \ldots, n-3$$

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` $\le 2n$, 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.

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)

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).

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, CSCOEF [,…])`

Specific: The specific interface names are `S_CSSMH` and `D_CSSMH`.

FORTRAN 77 Interface

Single: `CALL CSSMH (NDATA, XDATA, FDATA, WEIGHT, SMPAR, BREAK, CSCOEF)`

Double: The double precision name is `DCSSMH`.

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
      INTEGER    NDATA
      PARAMETER  (NDATA=300)
!
      INTEGER    I, NOUT
      REAL       BREAK(NDATA), CSCOEF(4,NDATA), ERROR, F,&
                 FDATA(NDATA), FLOAT, FVAL, SDEV, SMPAR, SQRT,&
                 SVAL, WEIGHT(NDATA), X, XDATA(NDATA), XT
      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
```

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, CSCOEF, WK, IWK)
   ```

   The additional arguments are as follows:

   WK — Work array of length 8 * 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 (page 609); its derivative can be evaluated using CSDER (page 610).

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

$$\sum_{i=1}^{N} \left| \frac{S(x_i) - f_i}{w_i} \right|^2 \le \sigma$$

where w = `WEIGHT`, σ = `SMPAR` is the smoothing parameter, and N = `NDATA`.

Recommended values for σ 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 σ should be chosen in the confidence interval corresponding to the left side of the above inequality. That is,

$$N - \sqrt{2N} \le \sigma \le N + \sqrt{2N}$$

The routine `CSSMH` is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pages 235–243).

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)

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 3.
Default: NDATA = size (XDATA,1).

FORTRAN 90 Interface

Generic: `CALL CSSCV (XDATA, FDATA, IEQUAL, BREAK, CSCOEF [,…])`

Specific: The specific interface names are S_CSSCV and D_CSSCV.

FORTRAN 77 Interface

Single: `CALL CSSCV (NDATA, XDATA, FDATA, IEQUAL, BREAK, CSCOEF)`

Double: The double precision name is DCSSCV.

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
      INTEGER    NDATA
      PARAMETER  (NDATA=300)
!
      INTEGER    I, IEQUAL, NOUT
      REAL       BREAK(NDATA), CSCOEF(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, CSCOEF)
!                                  Print results
      WRITE (NOUT,99999)
      DO 30  I=1, 10
         XT    = 90.0*(FLOAT(I-1)/FLOAT(NDATA-1))
         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         0.2528         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
```

Comments

1. Workspace may be explicitly provided, if desired, by use of C2SCV/DC2SCV. The reference is:

   CALL C2SCV (NDATA, XDATA, FDATA, IEQUAL, BREAK, CSCOEF, 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, XDATA, must be distinct.

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_σ is the unique C^2 function that minimizes

$$\int_a^b S''_\sigma(x)^2 \, dx$$

subject to the constraint

$$\sum_{i=1}^{N} \left| S_\sigma(x_i) - f_i \right|^2 \le \sigma$$

where σ is the smoothing parameter and N = `NDATA`. The reader should consult Reinsch (1967) for more information concerning smoothing splines. The IMSL subroutine `CSSMH` (see page 758) solves the above problem when the user provides the smoothing parameter σ. 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 σ so that the smoothing spline (S_σ) 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 $i = 1, \ldots, N$. For more information on this topic, we refer the reader to Craven and Wahba (1979).

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` = size (`P`,1) – 1.

M — The degree of the denominator. (Input)
Default: `M` = 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`.

Example

In this example, we compute the best rational approximation to the gamma function, Γ, 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
      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
```

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`).

Description

The routine `RATCH` is designed to compute the best weighted L_∞ (Chebyshev) approximant to a given function. Specifically, given a weight function w = `WEIGHT`, a monotone function ϕ = `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

$$\left\| \frac{f - R_M^N}{w} \right\| := \max_{x \in [a,b]} \frac{\left| f(x) - \frac{\sum_{i=1}^{N+1} P_i \phi^{i-1}(x)}{\sum_{i=1}^{M+1} Q_i \phi^{i-1}(x)} \right|}{w(x)}$$

Notice that setting $\phi(x) = x$ yields ordinary rational approximation. A typical use of the function ϕ 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 $\mathbf{t}_1 < \ldots < \mathbf{t}_{N+M+2}$ 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).

Chapter 4: Integration and Differentiation

Routines

Usage Notes

Univariate Quadrature

The first nine 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` (page 772) (even if no endpoint singularities are present). If more efficiency is desired, then the use of `QDAG` (page 775) (or `QDAG*`) should be considered. These routines are organized as follows:

- w = 1

  – `QDAGS`

  – `QDAG`

  – `QDAGP`

  – `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 ε is `ERRABS`, while the requested relative error ρ 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:

$$\left|\int_a^b f(x)\,w(x)\,dx - R\right| \le E \le \max\left\{\varepsilon,\ \rho\left|\int_a^b f(x)\,w(x)\,dx\right|\right\}$$

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 Apprximation”, with one of the integration routines `CSINT`, `BSINT`, or `PPITG`.

Multivariate Quadrature

Two routines are described in this chapter that are of use in approximating certain multivariate integrals. In particular, the routine `TWODQ` returns an approximation to an iterated two-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y)\, dy\, dx$$

The second routine, `QAND`, returns an approximation to the integral of a function of *n* variables over a hyper-rectangle

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f(x_1, \ldots, x_n)\, dx_n \ldots dx_1$$

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` (page 811) produces the points $\{x_i\}$ and weights $\{w_i\}$ for $i = 1, \ldots, N$ 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 2*N*. The weight functions *w* may be selected from the following table:

$w(x)$	**Interval**	**Name**
1	$(-1, 1)$	Legendre
$1/\sqrt{1\text{-}x^2}$	$(-1, 1)$	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` (page 818) produces the three-term recurrence relation for the monic orthogonal polynomials with respect to the above weight functions.

Another routine, GQRCF (page 815), 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 (page 821) 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 (page 824), 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 (page 827), 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`.

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
      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
```

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` ≤ (`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 probably divergent or slowly convergent.

3. If `EXACT` is the exact value, `QDAGS` attempts to find `RESULT` such that |`EXACT` – `RESULT`| ≤ 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.

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` (page 775), this routine uses an extrapolation procedure known as the ε-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.

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
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`.

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
      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
```

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.

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.

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`.

Example

The value of

$$\int_0^3 x^3 \ln\left|\left(x^2-1\right)\left(x^2-2\right)\right| dx = 61\ln 2 + \frac{77}{4}\ln 7 - 27$$

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
      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
```

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)), …, `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 `ABS`(`EXACT` – `RESULT`)`.LE.MAX`(`ERRABS, ERREL * ABS`(`EXACT`)). To specify only a relative error, set `ERRABS` to zero. Similarly, to specify only an absolute error, set `ERRREL` to zero.

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` (page 775), this routine employs an extrapolation procedure known as the ε-algorithm. The routine `QDAGP` is an implementation of the subroutine `QAGP`, which is fully documented by Piessens et al. (1983).

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)

INTERV	**Interval**

−1	(−∞, BOUND)
1	(BOUND, + ∞)
2	(−∞, + ∞)

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`.

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
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
```

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 .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)), …, 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).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.

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` (page 775), this subroutine employs an extrapolation procedure known as the ε-algorithm. The routine `QDAGI` is an implementation of the subroutine `QAGI`, which is fully documented by Piessens et al. (1983).

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)

IWEIGH	**Weight**
1	`COS(OMEGA * X)`
2	`SIN(OMEGA * X)`

OMEGA — Parameter in the weight function. (Input)

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 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 ω*x* or sin ω*x*. Depending on the length of the subinterval in relation to the size of ω, 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 (page 775), this subroutine uses an extrapolation procedure known as the ε-algorithm. The routine QDAWO is an implementation of the subroutine QAWO, which is fully documented by Piessens et al. (1983).

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

      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
```

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, …, 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))`, …, `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) .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.

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)

IWEIGH	**Weight**
1	`COS(OMEGA * X)`
2	`SIN(OMEGA * X)`

OMEGA — Parameter in the weight function. (Input)

RESULT — Estimate of the integral from `A` to infinity 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.

ERREST — Estimate of the absolute value of the error. (Output)
Default: `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`.

Example

The value of

$$\int_0^\infty x^{-1/2} \cos\left(\pi x / 2\right) dx = 1$$

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

      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
```

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, …, `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**
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 * MAXSUB + 25 * MAXCBY.

IWK — Work array of length 2 * 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 ABS(EXACT – RESULT) .LE. ERRABS.

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 (page 785) followed by extrapolation. The routine QDAWF is an implementation of the subroutine QAWF, which is fully documented by Piessens et al. (1983).

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	`(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`.

Example

The value of

$$\int_0^1 \left[(1+x)(1-x)\right]^{1/2} x \ln(x)\, dx = \frac{3\ln(2)-4}{9}$$

is estimated. The values of the actual and estimated error are machine dependent.

```
      USE QDAWS_INT
      USE UMACH_INT
      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
```

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. (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. (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 `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.

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` (page 775), this routine uses an extrapolation procedure known as the ε-algorithm. The routine `QDAWS` is an implementation of the routine `QAWS`, which is fully documented by Piessens et al. (1983).

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: `ERREL` =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`.

Example

The Cauchy principal value of

$$\int_{-1}^{5} \frac{1}{x\left(5x^3+6\right)}\,dx = \frac{\ln\left(125/631\right)}{18}$$

is estimated. The values of the actual and estimated error are machine dependent.

```
      USE QDAWC_INT
      USE UMACH_INT
      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
```

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 `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)), …, `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, `QDAWC` 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.

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` (page 775), this routine uses an extrapolation procedure known as the ε-algorithm. The routine `QDAWC` is an implementation of the subroutine `QAWC`, which is fully documented by Piessens et al. (1983).

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`.

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
      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
```

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 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.

3. This routine is designed for efficiency, not robustness. If the above error is encountered, try QDAGS.

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 (page 772).

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:

IRULE	**Points**
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`.

Example 1

In this example, we approximate the integral

$$\int_0^1 \int_1^3 y \cos\left(x + y^2\right) dy\, dx$$

The value of the error estimate is machine dependent.

```
      USE TWODQ_INT

      USE UMACH_INT
      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
```

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 `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.

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 (page 775). 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.

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

$$\int_0^1 \int_{-2x}^{5x} y \cos\left(x + y^2\right) dy\, dx$$

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
```

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.

Example 1

In this example, we approximate the integral of

$$e^{-\left(x_1^2+x_2^2+x_3^2\right)}$$

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
      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
```

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 `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.

Description

The routine `QAND` approximates the n-dimensional iterated integral

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f\left(x_1, \ldots, x_n\right) dx_n \ldots dx_1$$

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 $i = 1, \ldots, n$ 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`.

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

$$\int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} f\left(x_1, \ldots, x_n\right)\, dx_n \ldots dx_1$$

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: $\left\lfloor \texttt{base}^{m/2-1} \right\rfloor$, where $m = \left\lfloor \log B / \log \texttt{base} \right\rfloor$ 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`.

Example

This example evaluates the n-dimensional integral

$$\int_0^1 \cdots \int_0^1 \sum_{i=1}^{w} \prod_{j=1}^{i} (-1)^i x_j \, dx_1 \ldots dx_n = -\frac{1}{3}\left[1-\left(-\frac{1}{2}\right)^n\right]$$

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
```

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*.

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.

IWEIGH	**WT(X)**	**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.

BETAW — Parameter used in the weight function with some values of `IWEIGH`, otherwise it is ignored. (Input)
Default: `BETAW` = 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`.

Example 1

In this example, we obtain the classical Gauss-Legendre quadrature formula, which is accurate for polynomials of degree less than 2*N*, 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
```

```
      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.
```

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.

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 $2N$ − `NFIX`.

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
      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`.

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` (page 818).

```
      USE GQRCF_INT
      USE UMACH_INT
      USE RECCF_INT
      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
```

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.

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

$$p_i(x) = (x - b_i)\, p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i=1, \ldots, N$$

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 $2N$ – `NFIX`.

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.

IWEIGH	**WT(X)**	**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`=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`.

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
      PARAMETER (N=6)
      INTEGER    I, IWEIGH, NOUT
      REAL       ALPHA, B(N), C(N)
!                                  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
```

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).

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

$$p_i(x) = (x - b_i)\, p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i{=}1, \ldots, N$$

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 $c_i > 0$.

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` = size (`QX`,1).

NTERM — Number of recurrence coefficients. (Input)
`NTERM` must be less than or equal to `N`.
Default: `NTERM` = 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`.

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
      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
```

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:

   WKWK — 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)`.

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

$$p_i(x) = (x - b_i)\,p_{i-1}(x) - c_i p_{i-2}(x) \quad \text{for } i=1, \ldots, N$$

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 `GQRCF` (page 815). Given the recurrence coefficients, the routine `GQRCF` produces the corresponding Gauss quadrature formula, whereas the routine `RECQR` produces the recurrence coefficients given the quadrature formula.

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.

IWEIGH	**WT(X)**
1	1
2	1/(`X` – `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`.

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

$$\int_0^1 x \sin\left(41\pi x^2\right) dx = \frac{1}{41\pi}$$

```
      USE FQRUL_INT
      USE UMACH_INT
      USE CONST_INT
      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.
```

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

$$\int_A^B F(X) * WT(X)\,dX = \sum_{I=1}^{N} F\big(QX(I)\big) * QW(I)$$

3. The routine `FQRUL` uses an `FFT`, so it is most efficient when `N` is the product of small primes.

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 Milovanofic (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

$$Q(f) := \sum_{j=1}^{N} w_j f\left(x_j\right)$$

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 $l = 0, \ldots, N - 1$. This is a system of linear equations for the unknown weights w_j that can be simplified by noting that

$$x_j = \cos\frac{(2j-1)\pi}{2N} \qquad j = 1, \ldots, N$$

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 `QCOSB`. 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 `QCOSB`. For more information on this topic, consult Davis and Rabinowitz (1984, pages 84–86) and Gautschi (1968, page 259).

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`.

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
      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
```

Comments

1. Informational errors

Type	Code	
3	2	Roundoff error became dominant before estimates converged. Increase precision and/or increase BGSTEP.
4	1	Unable to achieve desired tolerance in derivative estimation. Increase precision, increase TOL and/or change BGSTEP. If this error continues, the function may not have a derivative at X.

2. Convergence is assumed when

$$\frac{2}{3}\left|D2 - D1\right| < \text{TOL}$$

for two successive derivative estimates D1 and D2.

3. The initial step size, BGSTEP, must be chosen small enough that FCN is defined and reasonably smooth in the interval (X – 4 * BGSTEP, X + 4 * BGSTEP), yet large enough to avoid roundoff problems.

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.

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
      INTEGER    KORDER, NOUT
      REAL       BGSTEP, DERV, X
      DOUBLE PRECISION DBGSTE, DDERV, DFCN, DTOL, DX
      EXTERNAL   DFCN, FCN
!                                 Get output unit number
      CALL UMACH (2, NOUT)
!                                 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
```

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_∞ (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 singleprecision 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:
$A \leftarrow A + \alpha x \overline{y}^T$.

`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:
$A \leftarrow A + \alpha x \overline{x}^T$ with x complex and α real.

`CHER2` Computes a rank-two update of an Hermitian matrix:
$A \leftarrow A + \alpha x \overline{y}^T + \overline{\alpha} y \overline{x}^T$.

`CHER2K` Computes one of the Hermitian rank $2k$ operations:
$C \leftarrow \alpha A \overline{B}^T + \overline{\alpha} B \overline{A}^T + \beta C$ or $C \leftarrow \alpha \overline{A}^T B + \overline{\alpha} \overline{B}^T A + \beta C$,
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:
$C \leftarrow \alpha A \overline{A}^T + \beta C$ or $C \leftarrow \alpha \overline{A}^T A + \beta C$,
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:
$C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$ 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:
$C \leftarrow \alpha A A^T + \beta C$ 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:
$x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\overline{A}^T\right)^{-1} x,$
where A is a triangular matrix in band storage mode.

`CTRSM` Solves one of the complex matrix equations:
$B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha B A^{-1}, B \leftarrow \alpha\left(A^{-1}\right)^T B, B \leftarrow \alpha B\left(A^{-1}\right)^T,$

$B \leftarrow \alpha\left(\overline{A}^T\right)^{-1} B,$ or $B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$
where A is a triangular matrix.

`CTRSV` Solves one of the complex triangular systems:
$x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x,$ or $x \leftarrow \left(\overline{A}^T\right)^{-1} x,$
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 2*k* operations: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ 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: $C \leftarrow \alpha AA^T + \beta C$ 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: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$, where *A* is a triangular matrix in band storage mode.
`STRSM`	Solves one of the matrix equations: $B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha\left(A^{-1}\right)^T B$, or $B \leftarrow \alpha B\left(A^{-1}\right)^T$ where *B* is an *m* by *n* matrix and *A* is a triangular matrix.
`STRSV`	Solves one of the triangular linear systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$, 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

`TRNRR`	Transposes a rectangular matrix.

D1b4 Multiplication by vector

`BLINF`	Computes the bilinear form x^TAy.
`CGBMV`	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, or $y \leftarrow \alpha \overline{A}^T + \beta y$, where *A* is a matrix stored in band storage mode.

CGEMV	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$,
CHBMV	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is an Hermitian band matrix in band Hermitian storage.
CHEMV	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is an Hermitian matrix.
CTBMV	Computes one of the matrix-vector operations: $x \leftarrow Ax,\ x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$, where A is a triangular matrix in band storage mode.
CTRMV	Computes one of the matrix-vector operations: $x \leftarrow Ax,\ x \leftarrow A^T x,$ or $x \leftarrow \overline{A}^T x$, 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: $y \leftarrow \alpha Ax + \beta y,$ 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: $y \leftarrow \alpha Ax + \beta y,$ 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: $x \leftarrow Ax$ or $x \leftarrow A^T x$, where A is a triangular matrix in band storage mode.
STRMV	Computes one of the matrix-vector operations: $x \leftarrow Ax$ 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: $C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T$ $+\beta C, C \leftarrow \alpha A^T B^T + \beta C, C \leftarrow \alpha A\overline{B}^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T B + \beta C, C \leftarrow \alpha A^T \overline{B}^T + \beta C,$ $C \leftarrow \alpha \overline{A}^T B^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T \overline{B}^T + \beta C$
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: $B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, B \leftarrow \alpha BA, B \leftarrow \alpha BA^T$, $B \leftarrow \alpha \overline{A}^T B,$ or $B \leftarrow \alpha B\overline{A}^T$ 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 $A^Tx = b$ 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 $x \leftarrow \left(A^{-1}\right)^T x$, 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 $A_jx_j = y_j$, $j = 1, \ldots, k$. 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: $B \leftarrow \alpha A^{-1}B$, $B \leftarrow \alpha BA^{-1}$, $B \leftarrow \alpha\left(A^{-1}\right)^T B$, or $B \leftarrow \alpha B\left(A^{-1}\right)^T$ where B is an m by n matrix and A is a triangular matrix.

`STRSV`	Solves one of the triangular linear systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$ 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 UDU^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 UDU^T factorization of the coefficient matrix.
`LFTSF`	Computes the UDU^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 $R^T R$ Choleksy 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: $x \leftarrow A^{-1}x$, $x \leftarrow \left(A^{-1}\right)^T x$, or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$, 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 $A_j x_j = y_j$, $j = 1, \ldots, k$. 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: $B \leftarrow \alpha A^{-1}B$, $B \leftarrow \alpha BA^{-1}$, $B \leftarrow \alpha\left(A^{-1}\right)^T B$, $B \leftarrow \alpha B\left(A^{-1}\right)^T$, $B \leftarrow \alpha\left(\overline{A}^T\right)^{-1} B$, or $B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$ where A is a traiangular matrix.
CTRSV	Solves one of the complex triangular systems: $x \leftarrow A^{-1}x$, $x \leftarrow \left(A^{-1}\right)^T x$, or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$ 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 UDU^H factorization of a complex Hermitian matrix and estimate its L_1 condition number.
`LFDHF`	Computes the determinant of a complex Hermitian matrix given the UDU^H factorization of the matrix.
`LFIHF`	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 UDU^H factorization of the coefficient matrix.
`LFTHF`	Computes the UDU^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.
`LFIDH`	Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.
`LFSDH`	Solves a complex Hermitian positive definite system of linear equations given the $R^H R$ factorization of the coefficient matrix.

`LFTDH`	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.
`LNFZD`	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

`LFDCB`	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

`LFDDH`	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H\ R$ Cholesky factorization of the matrix.

D3d2.....Positive definite banded

`LFDQH`	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

D4a.......Ordinary eigenvalue problems ($Ax = \lambda x$)

D4a1.....Real symmetric

`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 $A = VDV^T$, where V is an $n \times n$ orthogonal matrix and D is a real diagonal matrix.

D4a2.....Real nonsymmetric

`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 2×2 or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

D4a3.....Complex Hermitian

`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 $A = VDV^T$, where V is an $n \times n$ orthogonal matrix and D is a real diagonal matrix.

D4a4 ..... Complex non-Hermitian

`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 2×2 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.
`EVESB`	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.
`EVLSB`	Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode.

D4b .... Generalized eigenvalue problems (e.g., $Ax = \lambda Bx$)

D4b1 ..... Real symmetric

`GVCSP`	Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.
`GVLSP`	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 α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.

D4b2 ..... Real general

`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 α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.

D4b4.....Complex general

`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 α and β 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

`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^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.
`LUPCH`	Updates the R^TR 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 αxy^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 $A^Ty \cong d$ given the factorization of A. An optional argument is provided for computing the following unscaled covariance matrix: $C = (A^TA)^{-1}$.
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.

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.
`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.

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.
	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.
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.
`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)

`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)

`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.

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

I2a. .... Initial boundary value problems

I2a1 ...... Parabolic

`PDE_1D_MG` Integrates an initial-value PDE problem with one space variable.

I2a1a..... One spatial dimension

`MOLCH` 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 threedimensional 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.
CONFT	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_& NONONEGATIVE_LSQ`	Solves multiple systems of linear equations $A_j x_j = y_j$, $j = 1, \ldots, k$. 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_∞) 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.
`CSSED`	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.
`CSVAL`	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 .... Mulitivariate

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., $y = \beta_0 + \beta_1 x + \varepsilon$) (*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 = $\beta_0 + \beta_1 x + \beta_2 x2 + \varepsilon$) (*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., $y = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k + \varepsilon$)

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.

`CVTSI` 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)

`IWKCIN`	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

`IIDEX`	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, $y_1 \le y_2 \le \ldots y_n$.

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.
`IERCD`	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

IMSL MATH/LIBRARY

ACBCB	1441	Adds two complex band matrices, both in band storage mode.
ACHAR	1624	Returns a character given its ASCII value.
AMACH	1685	Retrieves single-precision machine constants.
ARBRB	1438	Adds two band matrices, both in band storage mode.
BCLSF	1274	Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.
BCLSJ	1281	Solves a nonlinear least squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.
BCNLS	1288	Solves a nonlinear least-squares problem subject to bounds on the variables and general linear constraints.
BCOAH	1263	Minimizes a function of `N` variables subject to bounds the variables using a modified Newton method and a user-supplied Hessian.
BCODH	1257	Minimizes a function of `N` variables subject to bounds the variables using a modified Newton method and a finite-difference Hessian.
BCONF	1243	Minimizes a function of `N` variables subject to bounds the variables using a quasi-Newton method and a finite-difference gradient.
BCONG	1249	Minimizes a function of `N` variables subject to bounds the variables using a quasi-Newton method and a user-supplied gradient.
BCPOL	1271	Minimizes a function of `N` variables subject to bounds the variables using a direct search complex algorithm.

BLINF	1427	Computes the bilinear form x^TAy.
BS1GD	656	Evaluates the derivative of a spline on a grid, given its B-spline representation.
BS2DR	653	Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS2GD	656	Evaluates the derivative of a two-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.
BS2IG	661	Evaluates the integral of a tensor-product spline on a rectangular domain, given its tensor-product B-spline representation.
BS2IN	631	Computes a two-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.
BS2VL	651	Evaluates a two-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS3DR	666	Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation.
BS3GD	670	Evaluates the derivative of a three-dimensional tensor-product spline, given its tensor-product B-spline representation on a grid.
BS3IG	676	Evaluates the integral of a tensor-product spline in three dimensions over a three-dimensional rectangle, given its tensorproduct B-spline representation.
BS3IN	635	Computes a three-dimensional tensor-product spline interpolant, returning the tensor-product B-spline coefficients.
BS3VL	664	Evaluates a three-dimensional tensor-product spline, given its tensor-product B-spline representation.
BSCPP	680	Converts a spline in B-spline representation to piecewise polynomial representation.
BSDER	643	Evaluates the derivative of a spline, given its B-spline representation.
BSINT	622	Computes the spline interpolant, returning the B-spline coefficients.
BSITG	649	Evaluates the integral of a spline, given its B-spline representation.

BSLS2	743	Computes a two-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.
BSLS3	748	Computes a three-dimensional tensor-product spline approximant using least squares, returning the tensor-product B-spline coefficients.
BSLSQ	725	Computes the least-squares spline approximation, and return the B-spline coefficients.
BSNAK	625	Computes the 'not-a-knot' spline knot sequence.
BSOPK	628	Computes the 'optimal' spline knot sequence.
BSVAL	641	Evaluates a spline, given its B-spline representation.
BSVLS	729	Computes the variable knot B-spline least squares approximation to given data.
BVPFD	870	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	882	Solves a (parameterized) system of differential equations with boundary conditions at two points, using a multiple-shooting method.
CADD	1319	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all complex.
CAXPY	1320	Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all complex.
CCBCB	1393	Copies a complex band matrix stored in complex band storage mode.
CCBCG	1400	Converts a complex matrix in band storage mode to a complex matrix in full storage mode.
CCGCB	1398	Converts a complex general matrix to a matrix in complex band storage mode.
CCGCG	1390	Copies a complex general matrix.
CCONV	1064	Computes the convolution of two complex vectors.
CCOPY	1319	Copies a vector x to a vector y, both complex.
CCORL	1073	Computes the correlation of two complex vectors.
CDGRD	1336	Approximates the gradient using central differences.
CDOTC	1320	Computes the complex conjugate dot product, $\bar{x}^T y$.
CDOTU	1320	Computes the complex dot product $x^T y$.

CGBMV	1330	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$, where A is a matrix stored in band storage mode.
CGEMM	1333	Computes one of the matrix-matrix operations: $C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C,\ C \leftarrow \alpha AB^T$ $+\beta C,\ C \leftarrow \alpha A^T B^T + \beta C,\ C \leftarrow \alpha A\overline{B}^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T B + \beta C,\ C \leftarrow \alpha A^T \overline{B}^T + \beta C,$ $C \leftarrow \alpha \overline{A}^T B^T + \beta C,$ or $C \leftarrow \alpha \overline{A}^T \overline{B}^T + \beta C$
CGEMV	1329	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y,$ or $y \leftarrow \alpha \overline{A}^T + \beta y$,
CGERC	1384	Computes the rank-one update of a complex general matrix: $A \leftarrow A + \alpha x\overline{y}^T$.
CGERU	1384	Computes the rank-one update of a complex general matrix: $A \leftarrow A + \alpha xy^T$.
CHBCB	1411	Copies a complex Hermitian band matrix stored in band Hermitian storage mode to a complex band matrix stored in band storage mode.
CHBMV	1381	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is an Hermitian band matrix in band Hermitian storage.
CHEMM	1385	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.
CHEMV	1381	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is an Hermitian matrix.
CHER	1384	Computes the rank-one update of an Hermitian matrix: $A \leftarrow A + \alpha x\overline{x}^T$ with x complex and α real.
CHER2	1384	Computes a rank-two update of an Hermitian matrix: $A \leftarrow A + \alpha x\overline{y}^T + \overline{\alpha} y\overline{x}^T$.
CHER2K	1387	Computes one of the Hermitian rank $2k$ operations: $C \leftarrow \alpha A\overline{B}^T + \overline{\alpha} B\overline{A}^T + \beta C$ or $C \leftarrow \alpha \overline{A}^T B + \overline{\alpha}\overline{B}^T A + \beta C$, 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	1386	Computes one of the Hermitian rank k operations: $C \leftarrow \alpha A\overline{A}^T + \beta C$ or $C \leftarrow \alpha \overline{A}^T A + \beta C$, 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.
CHFCG	1408	Extends a complex Hermitian matrix defined in its upper triangle to its lower triangle.
CHGRD	1349	Checks a user-supplied gradient of a function.
CHHES	1352	Checks a user-supplied Hessian of an analytic function.
CHJAC	1355	Checks a user-supplied Jacobian of a system of equations with `M` functions in `N` unknowns.
CHOL	1475	Computes the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix, A.
COND	1476	Computes the condition number of a rectangular matrix, A.
CONFT	734	Computes the least-squares constrained spline approximation, returning the B-spline coefficients.
CONST	1669	Returns the value of various mathematical and physical constants.
CPSEC	1631	Returns CPU time used in seconds.
CRBCB	1405	Converts a real matrix in band storage mode to a complex matrix in band storage mode.
CRBRB	1392	Copies a real band matrix stored in band storage mode.
CRBRG	1397	Converts a real matrix in band storage mode to a real general matrix.
CRGCG	1402	Copies a real general matrix to a complex general matrix.
CRGRB	1395	Converts a real general matrix to a matrix in band storage mode.
CRGRG	1389	Copies a real general matrix.
CRRCR	1403	Copies a real rectangular matrix to a complex rectangular matrix.
CS1GD	602	Evaluates the derivative of a cubic spline on a grid.
CSAKM	500	Computes the Akima cubic spline interpolant.
CSBRB	1409	Copies a real symmetric band matrix stored in band symmetric storage mode to a real band matrix stored in band storage mode.

CSCAL	1319	Multiplies a vector by a scalar, $y \leftarrow ay$, both complex.
CSCON	603	Computes a cubic spline interpolant that is consistent with the concavity of the data.
CSDEC	593	Computes the cubic spline interpolant with specified derivative endpoint conditions.
CSDER	610	Evaluates the derivative of a cubic spline.
CSET	1318	Sets the components of a vector to a scalar, all complex.
CSFRG	1406	Extends a real symmetric matrix defined in its upper triangle to its lower triangle.
CSHER	597	Computes the Hermite cubic spline interpolant.
CSIEZ	587	Computes the cubic spline interpolant with the 'not-a-knot' condition and return values of the interpolant at specified points.
CSINT	590	Computes the cubic spline interpolant with the 'not-a-knot' condition.
CSITG	616	Evaluates the integral of a cubic spline.
CSPER	506	Computes the cubic spline interpolant with periodic boundary conditions.
CSROT	1325	Applies a complex Givens plane rotation.
CSROTM	1326	Applies a complex modified Givens plane rotation.
CSSCAL	1319	Multiplies a complex vector by a single-precision scalar, $y \leftarrow ay$.
CSSCV	761	Computes a smooth cubic spline approximation to noisy data using cross-validation to estimate the smoothing parameter.
CSSED	754	Smooths one-dimensional data by error detection.
CSSMH	758	Computes a smooth cubic spline approximation to noisy data.
CSUB	1319	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all complex.
CSVAL	609	Evaluates a cubic spline.
CSVCAL	1319	Multiplies a complex vector by a single-precision scalar and store the result in another complex vector, $y \leftarrow ax$.
CSWAP	1320	Interchanges vectors x and y, both complex.
CSYMM	1334	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.

CSYR2K	1335	Computes one of the symmetric rank 2*k* operations: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ 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	1334	Computes one of the symmetric rank *k* operations: $C \leftarrow \alpha AA^T + \beta C$ 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.
CTBMV	1331	Computes one of the matrix-vector operations: $x \leftarrow Ax, x \leftarrow A^T x$, or $x \leftarrow \overline{A}^T x$, where *A* is a triangular matrix in band storage mode.
CTBSV	1332	Solves one of the complex triangular systems: $x \leftarrow A^{-1}x$, $x \leftarrow \left(A^{-1}\right)^T x$, or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$, where *A* is a triangular matrix in band storage mode.
CTRMM	1335	Computes one of the matrix-matrix operations: $B \leftarrow \alpha AB, B \leftarrow \alpha A^T B, B \leftarrow \alpha BA, B \leftarrow \alpha BA^T$, $B \leftarrow \alpha \overline{A}^T B$, or $B \leftarrow \alpha B\overline{A}^T$ where *B* is an *m* by *n* matrix and *A* is a triangular matrix.
CTRMV	1331	Computes one of the matrix-vector operations: $x \leftarrow Ax, x \leftarrow A^T x$, or $x \leftarrow \overline{A}^T x$, where *A* is a triangular matrix.
CTRSM	1336	Solves one of the complex matrix equations: $B \leftarrow \alpha A^{-1}B, B \leftarrow \alpha BA^{-1}, B \leftarrow \alpha\left(A^{-1}\right)^T B, B \leftarrow \alpha B\left(A^{-1}\right)^T$, $B \leftarrow \alpha\left(\overline{A}^T\right)^{-1} B$, or $B \leftarrow \alpha B\left(\overline{A}^T\right)^{-1}$ where *A* is a traiangular matrix.
CTRSV	1331	Solves one of the complex triangular systems: $x \leftarrow A^{-1}x, x \leftarrow \left(A^{-1}\right)^T x$, or $x \leftarrow \left(\overline{A}^T\right)^{-1} x$, where *A* is a triangular matrix.
CUNIT	1672	Converts `X` in units `XUNITS` to `Y` in units `YUNITS`.
CVCAL	1319	Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all complex.
CVTSI	1630	Converts a character string containing an integer number into the corresponding integer form.

CZCDOT	1321	Computes the sum of a complex scalar plus a complex conjugate dot product, $a + \bar{x}^T y$, using a double-precision accumulator.
CZDOTA	1321	Computes the sum of a complex scalar, a complex dot product and the double-complex accumulator, which is set to the result ACC ← ACC + $a + x^T y$.
CZDOTC	1320	Computes the complex conjugate dot product, $\bar{x}^T y$, using a double-precision accumulator.
CZDOTI	1321	Computes the sum of a complex scalar plus a complex dot product using a double-complex accumulator, which is set to the result ACC ← $a + x^T y$.
CZDOTU	1320	Computes the complex dot product $x^T y$ using a double-precision accumulator.
CZUDOT	1321	Computes the sum of a complex scalar plus a complex dot product, $a + x^T y$, using a double-precision accumulator.
DASPG	889	Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, using Petzold–Gear BDF method.
DERIV	827	Computes the first, second or third derivative of a user-supplied function.
DET	1477	Computes the determinant of a rectangular matrix, A.
DIAG	1479	Constructs a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array.
DIAGONALS	1479	Extracts a rank-1 array whose values are the diagonal terms of a rank-2 array argument.
DISL1	1452	Computes the 1-norm distance between two points.
DISL2	1450	Computes the Euclidean (2-norm) distance between two points.
DISLI	1454	Computes the infinity norm distance between two points.
DLPRS	1297	Solves a linear programming problem via the revised simplex algorithm.
DMACH	1686	See AMACH.
DQADD	1460	Adds a double-precision scalar to the accumulator in extended precision.
DQINI	1460	Initializes an extended-precision accumulator with a double-precision scalar.

DQMUL	1460	Multiplies double-precision scalars in extended precision.
DQSTO	1460	Stores a double-precision approximation to an extended-precision scalar.
DSDOT	1371	Computes the single-precision dot product $x^T y$ using a double precision accumulator.
DUMAG	1664	This routine handles MATH/LIBRARY and STAT/LIBRARY type `DOUBLE PRECISION` options.
EIG	1480	Computes the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.
EPICG	467	Computes the performance index for a complex eigensystem.
EPIHF	518	Computes the performance index for a complex Hermitian eigensystem.
EPIRG	460	Computes the performance index for a real eigensystem.
EPISB	501	Computes the performance index for a real symmetric eigensystem in band symmetric storage mode.
EPISF	483	Computes the performance index for a real symmetric eigensystem.
ERROR_POST	1568	Prints error messages that are generated by IMSL routines using EPACK
ERSET	1679	Sets error handler default print and stop actions.
EVAHF	508	Computes the largest or smallest eigenvalues of a complex Hermitian matrix.
EVASB	490	Computes the largest or smallest eigenvalues of a real symmetric matrix in band symmetric storage mode.
EVASF	473	Computes the largest or smallest eigenvalues of a real symmetric matrix.
EVBHF	513	Computes the eigenvalues in a given range of a complex Hermitian matrix.
EVBSB	495	Computes the eigenvalues in a given interval of a real symmetric matrix stored in band symmetric storage mode.
EVBSF	478	Computes selected eigenvalues of a real symmetric matrix.
EVCCG	464	Computes all of the eigenvalues and eigenvectors of a complex matrix.
EVCCH	526	Computes all of the eigenvalues and eigenvectors of a complex upper Hessenberg matrix.

EVCHF	505	Computes all of the eigenvalues and eigenvectors of a complex Hermitian matrix.
EVCRG	457	Computes all of the eigenvalues and eigenvectors of a real matrix.
EVCRH	522	Computes all of the eigenvalues and eigenvectors of a real upper Hessenberg matrix.
EVCSB	487	Computes all of the eigenvalues and eigenvectors of a real symmetric matrix in band symmetric storage mode.
EVCSF	471	Computes all of the eigenvalues and eigenvectors of a real symmetric matrix.
EVEHF	510	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a complex Hermitian matrix.
EVESB	492	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix in band symmetric storage mode.
EVESF	475	Computes the largest or smallest eigenvalues and the corresponding eigenvectors of a real symmetric matrix.
EVFHF	515	Computes the eigenvalues in a given range and the corresponding eigenvectors of a complex Hermitian matrix.
EVFSB	498	Computes the eigenvalues in a given interval and the corresponding eigenvectors of a real symmetric matrix stored in band symmetric storage mode.
EVFSF	480	Computes selected eigenvalues and eigenvectors of a real symmetric matrix.
EVLCG	462	Computes all of the eigenvalues of a complex matrix.
EVLCH	525	Computes all of the eigenvalues of a complex upper Hessenberg matrix.
EVLHF	502	Computes all of the eigenvalues of a complex Hermitian matrix.
EVLRG	455	Computes all of the eigenvalues of a real matrix.
EVLRH	520	Computes all of the eigenvalues of a real upper Hessenberg matrix.
EVLSB	485	Computes all of the eigenvalues of a real symmetric matrix in band symmetric storage mode.
EVLSF	469	Computes all of the eigenvalues of a real symmetric matrix.
EYE	1481	Creates a rank-2 square array whose diagonals are all the value one.

FAURE_FREE	1655	Frees the structure containing information about the Faure sequence.
FAURE_INIT	1655	Shuffled Faure sequence initialization.
FAURE_NEXT	1656	Computes a shuffled Faure sequence.
FAST_DFT	992	Computes the Discrete Fourier Transform of a rank-1 complex array, *x*.
FAST_2DFT	1000	Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, *x*.
FAST_3DFT	1006	Computes the Discrete Fourier Transform (2DFT) of a rank-3 complex array, *x*.
FCOSI	1030	Computes parameters needed by `FCOST`.
FCOST	1028	Computes the discrete Fourier cosine transformation of an even sequence.
FDGRD	1338	Approximates the gradient using forward differences.
FDHES	1340	Approximates the Hessian using forward differences and function values.
FDJAC	1346	Approximates the Jacobian of `M` functions in `N` unknowns using forward differences.
FFT	1482	The Discrete Fourier Transform of a complex sequence and its inverse transform.
FFT_BOX	1482	The Discrete Fourier Transform of several complex or real sequences.
FFT2B	1048	Computes the inverse Fourier transform of a complex periodic two-dimensional array.
FFT2D	1045	Computes Fourier coefficients of a complex periodic two-dimensional array.
FFT3B	1055	Computes the inverse Fourier transform of a complex periodic three-dimensional array.
FFT3F	1051	Computes Fourier coefficients of a complex periodic threedimensional array.
FFTCB	1019	Computes the complex periodic sequence from its Fourier coefficients.
FFTCF	1017	Computes the Fourier coefficients of a complex periodic sequence.
FFTCI	1022	Computes parameters needed by `FFTCF` and `FFTCB`.
FFTRB	1012	Computes the real periodic sequence from its Fourier coefficients.

FFTRF	1009	Computes the Fourier coefficients of a real periodic sequence.
FFTRI	1015	Computes parameters needed by `FFTRF` and `FFTRB`.
FNLSQ	720	Computes a least-squares approximation with user-supplied basis functions.
FPS2H	961	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	967	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.
FQRUL	824	Computes a Fejér quadrature rule with various classical weight functions.
FSINI	1026	Computes parameters needed by `FSINT`.
FSINT	1024	Computes the discrete Fourier sine transformation of an odd sequence.
GDHES	1343	Approximates the Hessian using forward differences and a user-supplied gradient.
GGUES	1359	Generates points in an `N`-dimensional space.
GMRES	368	Uses restarted `GMRES` with reverse communication to generate an approximate solution of $Ax = b$.
GPICG	542	Computes the performance index for a generalized complex eigensystem $Az = \lambda Bz$.
GPIRG	535	Computes the performance index for a generalized real eigensystem $Az = \lambda Bz$.
GPISP	549	Computes the performance index for a generalized real symmetric eigensystem problem.
GQRCF	815	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	811	Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.
GVCCG	540	Computes all of the eigenvalues and eigenvectors of a generalized complex eigensystem $Az = \lambda Bz$.
GVCRG	531	Computes all of the eigenvalues and eigenvectors of a generalized real eigensystem $Az = \lambda Bz$.

GVCSP	547	Computes all of the eigenvalues and eigenvectors of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.
GVLCG	537	Computes all of the eigenvalues of a generalized complex eigensystem $Az = \lambda Bz$.
GVLRG	529	Computes all of the eigenvalues of a generalized real eigensystem $Az = \lambda Bz$.
GVLSP	544	Computes all of the eigenvalues of the generalized real symmetric eigenvalue problem $Az = \lambda Bz$, with B symmetric positive definite.
HRRRR	1425	Computes the Hadamard product of two real rectangular matrices.
HYPOT	1675	Computes $\sqrt{a^2 + b^2}$ without underflow or overflow.
IACHAR	1625	Returns the integer ASCII value of a character argument.
IADD	1319	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all integer.
ICAMAX	1324	Finds the smallest index of the component of a complex vector having maximum magnitude.
ICAMIN	1323	Finds the smallest index of the component of a complex vector having minimum magnitude.
ICASE	1626	Returns the ASCII value of a character converted to uppercase.
ICOPY	1319	Copies a vector x to a vector y, both integer.
IDYWK	1637	Computes the day of the week for a given date.
IERCD	1680	Retrieves the code for an informational error.
IFFT	1483	The inverse of the Discrete Fourier Transform of a complex sequence.
IFFT_BOX	1484	The inverse Discrete Fourier Transform of several complex or real sequences.
IFNAN(X)	1686	Checks if a value is NaN (not a number).
IICSR	1627	Compares two character strings using the ASCII collating sequence but without regard to case.
IIDEX	1629	Determines the position in a string at which a given character sequence begins without regard to case.
IIMAX	1323	Finds the smallest index of the maximum component of a integer vector.
IIMIN	1323	Finds the smallest index of the minimum of an integer vector.

IMACH	1683	Retrieves integer machine constants.
INLAP	1078	Computes the inverse Laplace transform of a complex function.
ISAMAX	1374	Finds the smallest index of the component of a single-precision vector having maximum absolute value.
ISAMIN	1374	Finds the smallest index of the component of a single-precision vector having minimum absolute value.
ISET	1318	Sets the components of a vector to a scalar, all integer.
ISMAX	1374	Finds the smallest index of the component of a single-precision vector having maximum value.
ISMIN	1374	Finds the smallest index of the component of a single-precision vector having minimum value.
ISNAN	1485	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.
ISRCH	1620	Searches a sorted integer vector for a given integer and return its index.
ISUB	1319	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all integer.
ISUM	1322	Sums the values of an integer vector.
ISWAP	1320	Interchanges vectors x and y, both integer.
IUMAG	1658	Sets or retrieves MATH/LIBRARY integer options.
IVMRK	844	Solves an initial-value problem $y' = f(t, y)$ for ordinary differential equations using Runge-Kutta pairs of various orders.
IVPAG	854	Solves an initial-value problem for ordinary differential equations using either Adams-Moulton's or Gear's BDF method.
IVPRK	837	Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.
IWKCIN	1701	Initializes bookkeeping locations describing the character workspace stack.
IWKIN	1700	Initializes bookkeeping locations describing the workspace stack.
JCGRC	365	Solves a real symmetric definite linear system using the Jacobi preconditioned conjugate gradient method with reverse communication.

LCHRG	406	Computes the Cholesky decomposition of a symmetric positive semidefinite matrix with optional column pivoting.
LCLSQ	388	Solves a linear least-squares problem with linear constraints.
LCONF	1310	Minimizes a general objective function subject to linear equality/inequality constraints.
LCONG	1316	Minimizes a general objective function subject to linear equality/inequality constraints.
LDNCH	412	Downdates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is removed.
LFCCB	262	Computes the *LU* factorization of a complex matrix in band storage mode and estimate its L_1 condition number.
LFCCG	108	Computes the *LU* factorization of a complex general matrix and estimate its L_1 condition number.
LFCCT	132	Estimates the condition number of a complex triangular matrix.
LFCDH	179	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix and estimate its L_1 condition number.
LFCDS	143	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix and estimate its L_1condition number.
LFCHF	197	Computes the $U DU^H$ factorization of a complex Hermitian matrix and estimate its L_1 condition number.
LFCQH	284	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.
LFCQS	240	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.
LFCRB	219	Computes the *LU* factorization of a real matrix in band storage mode and estimate its L_1 condition number.
LFCRG	89	Computes the *LU* factorization of a real general matrix and estimate its L_1 condition number.
LFCRT	125	Estimates the condition number of a real triangular matrix.

LFCSF	162	Computes the $U\,DU^T$ factorization of a real symmetric matrix and estimate its L_1 condition number.
LFDCB	274	Computes the determinant of a complex matrix given the *LU* factorization of the matrix in band storage mode.
LFDCG	119	Computes the determinant of a complex general matrix given the *LU* factorization of the matrix.
LFDCT	134	Computes the determinant of a complex triangular matrix.
LFDDH	190	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H\,R$ Cholesky factorization of the matrix.
LFDDS	153	Computes the determinant of a real symmetric positive definite matrix given the $R^H\,R$ Cholesky factorization of the matrix.
LFDHF	207	Computes the determinant of a complex Hermitian matrix given the $U\,DU^H$ factorization of the matrix.
LFDQH	295	Computes the determinant of a complex Hermitian positive definite matrix given the $R^H\,R$ Cholesky factorization in band Hermitian storage mode.
LFDQS	250	Computes the determinant of a real symmetric positive definite matrix given the $R^T\,R$ Cholesky factorization of the band symmetric storage mode.
LFDRB	230	Computes the determinant of a real matrix in band storage mode given the *LU* factorization of the matrix.
LFDRG	99	Computes the determinant of a real general matrix given the *LU* factorization of the matrix.
LFDRT	127	Computes the determinant of a real triangular matrix.
LFDSF	172	Computes the determinant of a real symmetric matrix given the $U\,DU^T$ factorization of the matrix.
LFICB	270	Uses iterative refinement to improve the solution of a complex system of linear equations in band storage mode.
LFICG	116	Uses iterative refinement to improve the solution of a complex general system of linear equations.
LFIDH	187	Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations.
LFIDS	150	Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations.

LFIHF	204	Uses iterative refinement to improve the solution of a complex Hermitian system of linear equations.
LFIQH	292	Uses iterative refinement to improve the solution of a complex Hermitian positive definite system of linear equations in band Hermitian storage mode.
LFIQS	247	Uses iterative refinement to improve the solution of a real symmetric positive definite system of linear equations in band symmetric storage mode.
LFIRB	227	Uses iterative refinement to improve the solution of a real system of linear equations in band storage mode.
LFIRG	96	Uses iterative refinement to improve the solution of a real general system of linear equations.
LFISF	169	Uses iterative refinement to improve the solution of a real symmetric system of linear equations.
LFSCB	268	Solves a complex system of linear equations given the *LU* factorization of the coefficient matrix in band storage mode.
LFSCG	114	Solves a complex general system of linear equations given the *LU* factorization of the coefficient matrix.
LFSDH	184	Solves a complex Hermitian positive definite system of linear equations given the $R^H R$ factorization of the coefficient matrix.
LFSDS	148	Solves a real symmetric positive definite system of linear equations given the $R^T R$ Choleksy factorization of the coefficient matrix.
LFSHF	202	Solves a complex Hermitian system of linear equations given the $U D U^H$ factorization of the coefficient matrix.
LFSQH	290	Solves a complex Hermitian positive definite system of linear equations given the factorization of the coefficient matrix in band Hermitian storage mode.
LFSQS	245	Solves a real symmetric positive definite system of linear equations given the factorization of the coefficient matrix in band symmetric storage mode.
LFSRB	225	Solves a real system of linear equations given the *LU* factorization of the coefficient matrix in band storage mode.
LFSRG	94	Solves a real general system of linear equations given the *LU* factorization of the coefficient matrix.
LFSSF	167	Solves a real symmetric system of linear equations given the $U D U^T$ factorization of the coefficient matrix.

LFSXD	336	Solves a real sparse symmetric positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.
LFSXG	306	Solves a sparse system of linear equations given the *LU* factorization of the coefficient matrix.
LFSZD	349	Solves a complex sparse Hermitian positive definite system of linear equations, given the Cholesky factorization of the coefficient matrix.
LFSZG	319	Solves a complex sparse system of linear equations given the *LU* factorization of the coefficient matrix.
LFTCB	265	Computes the *LU* factorization of a complex matrix in band storage mode.
LFTCG	111	Computes the *LU* factorization of a complex general matrix.
LFTDH	182	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix.
LFTDS	146	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix.
LFTHF	200	Computes the $U DU^H$ factorization of a complex Hermitian matrix.
LFTQH	288	Computes the $R^H R$ factorization of a complex Hermitian positive definite matrix in band Hermitian storage mode.
LFTQS	243	Computes the $R^T R$ Cholesky factorization of a real symmetric positive definite matrix in band symmetric storage mode.
LFTRB	222	Computes the *LU* factorization of a real matrix in band storage mode.
LFTRG	92	Computes the *LU* factorization of a real general matrix.
LFTSF	164	Computes the $U DU^T$ factorization of a real symmetric matrix.
LFTXG	301	Computes the *LU* factorization of a real general sparse matrix.
LFTZG	314	Computes the *LU* factorization of a complex general sparse matrix.
LINCG	121	Computes the inverse of a complex general matrix.
LINCT	136	Computes the inverse of a complex triangular matrix.
LINDS	154	Computes the inverse of a real symmetric positive definite matrix.

`LINRG`	101	Computes the inverse of a real general matrix.
`LINRT`	128	Computes the inverse of a real triangular matrix.
`LIN_EIG_GEN`	439	Computes the eigenvalues of a self-adjoint matrix, A.
`LIN_EIG_SELF`	432	Computes the eigenvalues of a self-adjoint matrix, A.
`LIN_GEIG_SELF`	448	Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av = \lambda Bv$.
`LIN_SOL_GEN`	9	Solves a general system of linear equations $Ax = b$.
`LIN_SOL_LSQ`	27	Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense.
`LIN_SOL_SELF`	17	Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix.
`LIN_SOL_SVD`	36	Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition.
`LIN_SOL_TRI`	44	Solves multiple systems of linear equations.
`LIN_SVD`	57	Computes the singular value decomposition (SVD) of a rectangular matrix, A.
`LNFXD`	331	Computes the numerical Cholesky factorization of a sparse symmetrical matrix A.
`LNFZD`	344	Computes the numerical Cholesky factorization of a sparse Hermitian matrix A.
`LQERR`	396	Accumulates the orthogonal matrix Q from its factored form given the QR factorization of a rectangular matrix A.
`LQRRR`	392	Computes the QR decomposition, $AP = QR$, using Householder transformations.
`LQRRV`	381	Computes the least-squares solution using Householder transformations applied in blocked form.
`LQRSL`	398	Computes the coordinate transformation, projection, and complete the solution of the least-squares problem $Ax = b$.
`LSACB`	257	Solves a complex system of linear equations in band storage mode with iterative refinement.
`LSACG`	103	Solves a complex general system of linear equations with iterative refinement.
`LSADH`	173	Solves a Hermitian positive definite system of linear equations with iterative refinement.
`LSADS`	138	Solves a real symmetric positive definite system of linear equations with iterative refinement.

LSAHF	191	Solves a complex Hermitian system of linear equations with iterative refinement.
LSAQH	276	Solves a complex Hermitian positive definite system of linear equations in band Hermitian storage mode with iterative refinement.
LSAQS	232	Solves a real symmetric positive definite system of linear equations in band symmetric storage mode with iterative refinement.
LSARB	213	Solves a real system of linear equations in band storage mode with iterative refinement.
LSARG	83	Solves a real general system of linear equations with iterative refinement.
LSASF	156	Solves a real symmetric system of linear equations with iterative refinement.
LSBRR	385	Solves a linear least-squares problem with iterative refinement.
LSCXD	327	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.
LSGRR	424	Computes the generalized inverse of a real matrix.
LSLCB	259	Solves a complex system of linear equations in band storage mode without iterative refinement.
LSLCC	356	Solves a complex circulant linear system.
LSLCG	106	Solves a complex general system of linear equations without iterative refinement.
LSLCQ	253	Computes the *LDU* factorization of a complex tridiagonal matrix *A* using a cyclic reduction algorithm.
LSLCR	211	Computes the *LDU* factorization of a real tridiagonal matrix *A* using a cyclic reduction algorithm.
LSLCT	130	Solves a complex triangular system of linear equations.
LSLDH	176	Solves a complex Hermitian positive definite system of linear equations without iterative refinement.
LSLDS	140	Solves a real symmetric positive definite system of linear equations without iterative refinement.
LSLHF	194	Solves a complex Hermitian system of linear equations without iterative refinement.

LSLPB	237	Computes the $R^T DR$ Cholesky factorization of a real symmetric positive definite matrix A in codiagonal band symmetric storage mode. Solve a system $Ax = b$.
LSLQB	281	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	279	Solves a complex Hermitian positive definite system of linearequations in band Hermitian storage mode without iterative refinement.
LSLQS	234	Solves a real symmetric positive definite system of linear equations in band symmetric storage mode without iterative refinement.
LSLRB	216	Solves a real system of linear equations in band storage mode without iterative refinement.
LSLRG	85	Solves a real general system of linear equations without iterative refinement.
LSLRT	123	Solves a real triangular system of linear equations.
LSLSF	159	Solves a real symmetric system of linear equations without iterative refinement.
LSLTC	354	Solves a complex Toeplitz linear system.
LSLTO	352	Solves a real Toeplitz linear system.
LSLTQ	252	Solves a complex tridiagonal system of linear equations.
LSLTR	209	Solves a real tridiagonal system of linear equations.
LSLXD	323	Solves a sparse system of symmetric positive definite linear algebraic equations by Gaussian elimination.
LSLXG	297	Solves a sparse system of linear algebraic equations by Gaussian elimination.
LSLZD	340	Solves a complex sparse Hermitian positive definite system of linear equations by Gaussian elimination.
LSLZG	309	Solves a complex sparse system of linear equations by Gaussian elimination.
LSQRR	378	Solves a linear least-squares problem without iterative refinement.
LSVCR	419	Computes the singular value decomposition of a complex matrix.
LSVRR	415	Computes the singular value decomposition of a real matrix.

LUPCH	409	Updates the R^TR Cholesky factorization of a real symmetric positive definite matrix after a rank-one matrix is added.
LUPQR	402	Computes an updated *QR* factorization after the rank-one matrix αxy^T is added.
MCRCR	1423	Multiplies two complex rectangular matrices, *AB*.
MOLCH	946	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.
MRRRR	1421	Multiplies two real rectangular matrices, *AB*.
MUCBV	1436	Multiplies a complex band matrix in band storage mode by a complex vector.
MUCRV	1435	Multiplies a complex rectangular matrix by a complex vector.
MURBV	1433	Multiplies a real band matrix in band storage mode by a real vector.
MURRV	1431	Multiplies a real rectangular matrix by a vector.
MXTXF	1415	Computes the transpose product of a matrix, A^TA.
MXTYF	1416	Multiplies the transpose of matrix *A* by matrix *B*, A^TB.
MXYTF	1418	Multiplies a matrx *A* by the transpose of a matrix *B*, AB^T.
NAN	1486	Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN. .
N1RTY	1680	Retrieves an error type for the most recently called IMSL routine.
NDAYS	1634	Computes the number of days from January 1, 1900, to the given date.
NDYIN	1636	Gives the date corresponding to the number of days since January 1, 1900.
NEQBF	1169	Solves a system of nonlinear equations using factored secant update with a finite-difference approximation to the Jacobian.
NEQBJ	1174	Solves a system of nonlinear equations using factored secant update with a user-supplied Jacobian.
NEQNF	1162	Solves a system of nonlinear equations using a modified Powell hybrid algorithm and a finite-difference approximation to the Jacobian.

NEQNJ	1165	Solves a system of nonlinear equations using a modified Powell hybrid algorithm with a user-supplied Jacobian.
NNLPF	1323	Uses a sequential equality constrained QP method.
NNLPG	1329	Uses a sequential equality constrained QP method.
NORM	1487	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.
NR1CB	1449	Computes the 1-norm of a complex band matrix in band storage mode.
NR1RB	1447	Computes the 1-norm of a real band matrix in band storage mode.
NR1RR	1444	Computes the 1-norm of a real matrix.
NR2RR	1446	Computes the Frobenius norm of a real rectangular matrix.
NRIRR	1443	Computes the infinity norm of a real matrix.
OPERATOR: .h.	1472	Computes transpose and conjugate transpose of a matrix.
OPERATOR: .hx.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:.i.	1473	Computes the inverse matrix, for square non-singular matrices.
OPERATOR:.ix.	1474	Computes the inverse matrix times a vector or matrix for square non-singular matrices.
OPERATOR:..t.	1472	Computes transpose and conjugate transpose of a matrix.
OPERATOR:.tx.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:.x.	1471	Computes matrix-vector and matrix-matrix products..
OPERATOR:..xh.	1471	Computes matrix-vector and matrix-matrix products.
OPERATOR:..xi.	1474	Computes the inverse matrix times a vector or matrix for square non-singular matrices.
OPERATORS:.xt.	1471	Computes matrix-vector and matrix-matrix products.
ORTH	1488	Orthogonalizes the columns of a rank-2 or rank-3 array.
PCGRC	359	Solves a real symmetric definite linear system using a preconditioned conjugate gradient method with reverse communication.
PARALLEL_NONNEGATIVE_LSQ	67	Solves a linear, non-negative constrained least-squares system.
PARALLEL_BOUNDED_LSQ	75	Solves a linear least-squares system with bounds on the unknowns.
PDE_1D_MG	913	Method of lines with Variable Griddings.

PERMA	1602	Permutes the rows or columns of a matrix.
PERMU	1600	Rearranges the elements of an array as specified by a permutation.
PGOPT	1599	Sets or retrieves page width and length for printing.
PLOTP	1664	Prints a plot of up to 10 sets of points.
POLRG	1429	Evaluates a real general matrix polynomial.
PP1GD	687	Evaluates the derivative of a piecewise polynomial on a grid.
PPDER	684	Evaluates the derivative of a piecewise polynomial.
PPITG	690	Evaluates the integral of a piecewise polynomial.
PPVAL	681	Evaluates a piecewise polynomial.
PRIME	1668	Decomposes an integer into its prime factors.
QAND	806	Integrates a function on a hyper-rectangle.
QCOSB	1041	Computes a sequence from its cosine Fourier coefficients with only odd wave numbers.
QCOSF	1039	Computes the coefficients of the cosine Fourier transform with only odd wave numbers.
QCOSI	1043	Computes parameters needed by QCOSF and QCOSB.
QD2DR	699	Evaluates the derivative of a function defined on a rectangular grid using quadratic interpolation.
QD2VL	696	Evaluates a function defined on a rectangular grid using quadratic interpolation.
QD3DR	705	Evaluates the derivative of a function defined on a rectangular three-dimensional grid using quadratic interpolation.
QD3VL	702	Evaluates a function defined on a rectangular three-dimensional grid using quadratic interpolation.
QDAG	775	Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.
QDAGI	782	Integrates a function over an infinite or semi-infinite interval.
QDAGP	779	Integrates a function with singularity points given.
QDAGS	772	Integrates a function (which may have endpoint singularities).
QDAWC	796	Integrates a function F(X)/(X – C) in the Cauchy principal value sense.
QDAWF	789	Computes a Fourier integral.

QDAWO	785	Integrates a function containing a sine or a cosine.
QDAWS	793	Integrates a function with algebraic-logarithmic singularities.
QDDER	694	Evaluates the derivative of a function defined on a set of points using quadratic interpolation.
QDNG	799	Integrates a smooth function using a nonadaptive rule.
QDVAL	692	Evaluates a function defined on a set of points using quadratic interpolation.
QMC	809	Integrates a function over a hyperrectangle using a quasi-Monte Carlo method.
QPROG	1307	Solves a quadratic programming problem subject to linear equality/inequality constraints.
QSINB	1034	Computes a sequence from its sine Fourier coefficients with only odd wave numbers.
QSINF	1032	Computes the coefficients of the sine Fourier transform with only odd wave numbers.
QSINI	1037	Computes parameters needed by `QSINF` and `QSINB`.
RAND	1489	Computes a scalar, rank-1, rank-2 or rank-3 array of random numbers.
RAND_GEN	1639	Generates a rank-1 array of random numbers.
RANK	1490	Computes the mathematical rank of a rank-2 or rank-3 array.
RATCH	764	Computes a rational weighted Chebyshev approximation to a continuous function on an interval.
RCONV	1059	Computes the convolution of two real vectors.
RCORL	1068	Computes the correlation of two real vectors.
RCURV	716	Fits a polynomial curve using least squares.
RECCF	818	Computes recurrence coefficients for various monic polynomials.
RECQR	821	Computes recurrence coefficients for monic polynomials given a quadrature rule.
RLINE	713	Fits a line to a set of data points using least squares.
RNGET	1648	Retrieves the current value of the seed used in the IMSL random number generators.
RNOPT	1650	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.
RNSET	1649	Initializes a random seed for use in the IMSL random number generators.

RNUN	1653	Generates pseudorandom numbers from a uniform (0, 1) distribution.
RNUNF	1651	Generates a pseudorandom number from a uniform (0, 1) distribution.
SADD	1370	Adds a scalar to each component of a vector, $x \leftarrow x + a$, all single precision.
SASUM	1373	Sums the absolute values of the components of a single-precision vector.
SAXPY	1370	Computes the scalar times a vector plus a vector, $y \leftarrow ax + y$, all single precision.
ScaLaPACK_READ	1545	Reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by *ScaLAPACK* routines.
ScaLaPACK_WRITE	1547	Writes the matrix data to a file.
SCASUM	1322	Sums the absolute values of the real part together with the absolute values of the imaginary part of the components of a complex vector.
SCNRM2	1322	Computes the Euclidean norm of a complex vector.
SCOPY	1369	Copies a vector x to a vector y, both single precision.
SDDOTA	1321	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 ← ACC + $a + x^T y$.
SDDOTI	1372	Computes the sum of a single-precision scalar plus a singleprecision dot product using a double-precision accumulator, which is set to the result ACC ← $a + x^T y$.
SDOT	1370	Computes the single-precision dot product $x^T y$.
SDSDOT	1371	Computes the sum of a single-precision scalar and a single precision dot product, $a + x^T y$, using a double-precision accumulator.
SGBMV	1381	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y,$ or $y \leftarrow \alpha A^T x + \beta y$, where A is a matrix stored in band storage mode.
SGEMM	1385	Computes one of the matrix-matrix operations: $C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha A^T B + \beta C, C \leftarrow \alpha AB^T + \beta C$, or $C \leftarrow \alpha A^T B^T + \beta C$

SGEMV	1381	Computes one of the matrix-vector operations: $y \leftarrow \alpha Ax + \beta y$, or $y \leftarrow \alpha A^T x + \beta y$,
SGER	1383	Computes the rank-one update of a real general matrix: $A \leftarrow A + \alpha xy^T$.
SHOW	1571	Prints rank-1 or rank-2 arrays of numbers in a readable format.
SHPROD	1372	Computes the Hadamard product of two single-precision vectors.
SINLP	1081	Computes the inverse Laplace transform of a complex function.
SLCNT	986	Calculates the indices of eigenvalues of a Sturm-Liouville problem with boundary conditions (at regular points) in a specified subinterval of the real line, [α, β].
SLEIG	973	Determines eigenvalues, eigenfunctions and/or spectral density functions for Sturm-Liouville problems in the form with boundary conditions (at regular points).
SLPRS	1301	Solves a sparse linear programming problem via the revised simplex algorithm.
SNRM2	1373	Computes the Euclidean length or L_2 norm of a single-precision vector.
SORT_REAL	1604	Sorts a rank-1 array of real numbers x so the y results are algebraically nondecreasing, $y_1 \le y_2 \le \ldots y_n$.
SPLEZ	618	Computes the values of a spline that either interpolates or fits user-supplied data.
SPLINE_CONSTRAINTS	562	Returns the derived type array result.
SPLINE_FITTING	564	Weighted least-squares fitting by B-splines to discrete One-Dimensional data is performed.
SPLINE_VALUES	563	Returns an array result, given an array of input
SPRDCT	1373	Multiplies the components of a single-precision vector.
SRCH	1618	Searches a sorted vector for a given scalar and return its index.
SROT	1375	Applies a Givens plane rotation in single precision.
SROTG	1374	Constructs a Givens plane rotation in single precision.
SROTM	1377	Applies a modified Givens plane rotation in single precision.
SROTMG	1376	Constructs a modified Givens plane rotation in single precision.

SSBMV	1382	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is a symmetric matrix in band symmetric storage mode.
SSCAL	1369	Multiplies a vector by a scalar, $y \leftarrow ay$, both single precision.
SSET	1369	Sets the components of a vector to a scalar, all single precision.
SSRCH	1622	Searches a character vector, sorted in ascending ASCII order, for a given string and return its index.
SSUB	1370	Subtracts each component of a vector from a scalar, $x \leftarrow a - x$, all single precision.
SSUM	1372	Sums the values of a single-precision vector.
SSWAP	1370	Interchanges vectors x and y, both single precision.
SSYMM	1385	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.
SSYMV	1382	Computes the matrix-vector operation $y \leftarrow \alpha Ax + \beta y$, where A is a symmetric matrix.
SSYR	1384	Computes the rank-one update of a real symmetric matrix: $A \leftarrow A + \alpha xx^T$.
SSYR2	1384	Computes the rank-two update of a real symmetric matrix: $A \leftarrow A + \alpha xy^T + \alpha yx^T$.
SSYR2K	1386	Computes one of the symmetric rank $2k$ operations: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ 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	1386	Computes one of the symmetric rank k operations: $C \leftarrow \alpha AA^T + \beta C$ 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.
STBMV	1382	Computes one of the matrix-vector operations: $x \leftarrow Ax$ or $x \leftarrow A^T x$, where A is a triangular matrix in band storage mode.

STBSV	1383	Solves one of the triangular systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$, where A is a triangular matrix in band storage mode.
STRMM	1387	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.
STRMV	1382	Computes one of the matrix-vector operations: $x \leftarrow Ax$ or $x \leftarrow A^T x$, where A is a triangular matrix.
STRSM	1387	Solves one of the matrix equations: $B \leftarrow \alpha A^{-1}B$, $B \leftarrow \alpha BA^{-1}$, $B \leftarrow \alpha\left(A^{-1}\right)^T B$, or $B \leftarrow \alpha B\left(A^{-1}\right)^T$ where B is an m by n matrix and A is a triangular matrix.
STRSV	1383	Solves one of the triangular linear systems: $x \leftarrow A^{-1}x$ or $x \leftarrow \left(A^{-1}\right)^T x$ where A is a triangular matrix.
SUMAG	1664	Sets or retrieves MATH/LIBRARY single-precision options.
SURF	710	Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.
SURFACE_CONSTRAINTS	574	Returns the derived type array result given optional input.
SURFACE_FITTING	577	Weighted least-squares fitting by tensor product B-splines to discrete two-dimensional data is performed.
SURFACE_VALUES	575	Returns a tensor product array result, given two arrays of independent variable values.
SVCAL	1369	Multiplies a vector by a scalar and store the result in another vector, $y \leftarrow ax$, all single precision.
SVD	1491	Computes the singular value decomposition of a rank-2 or rank-3 array, $A = USV^T$.
SVIBN	1615	Sorts an integer array by nondecreasing absolute value.
SVIBP	1617	Sorts an integer array by nondecreasing absolute value and returns the permutation that rearranges the array.
SVIGN	1610	Sorts an integer array by algebraically increasing value.

SVIGP	1611	Sorts an integer array by algebraically increasing value and returns the permutation that rearranges the array.
SVRBN	1612	Sorts a real array by nondecreasing absolute value.
SVRBP	1614	Sorts a real array by nondecreasing absolute value and returns the permutation that rearranges the array.
SVRGN	1607	Sorts a real array by algebraically increasing value.
SVRGP	1608	Sorts a real array by algebraically increasing value and returns the permutation that rearranges the array.
SXYZ	1372	Computes a single-precision *xyz* product.
TDATE	1633	Gets today's date.
TIMDY	1632	Gets time of day.
TRNRR	1413	Transposes a rectangular matrix.
TWODQ	801	Computes a two-dimensional iterated integral.
UMACH	1688	Sets or retrieves input or output device unit numbers.
UMAG	1661	Handles MATH/LIBRARY and STAT/LIBRARY type `REAL` and double precision options.
UMCGF	1219	Minimizes a function of `N` variables using a conjugate gradient algorithm and a finite-difference gradient.
UMCGG	1223	Minimizes a function of `N` variables using a conjugate gradient algorithm and a user-supplied gradient.
UMIAH	1213	Minimizes a function of `N` variables using a modified Newton method and a user-supplied Hessian.
UMIDH	1208	Minimizes a function of `N` variables using a modified Newton method and a finite-difference Hessian.
UMINF	1196	Minimizes a function of `N` variables using a quasi-New method and a finite-difference gradient.
UMING	1202	Minimizes a function of `N` variables using a quasi-New method and a user-supplied gradient.
UMPOL	1227	Minimizes a function of `N` variables using a direct search polytope algorithm.
UNIT	1492	Normalizes the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.
UNLSF	1231	Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a finite-difference Jacobian.
UNLSJ	1237	Solves a nonlinear least squares problem using a modified Levenberg-Marquardt algorithm and a user-supplied Jacobian.

UVMGS	1193	Finds the minimum point of a nonsmooth function of a single variable.
UVMID	1189	Finds the minimum point of a smooth function of a single variable using both function evaluations and first derivative evaluations.
UVMIF	1186	Finds the minimum point of a smooth function of a single variable using only function evaluations.
VCONC	1457	Computes the convolution of two complex vectors.
VCONR	1455	Computes the convolution of two real vectors.
VERML	1638	Obtains IMSL MATH/LIBRARY-related version, system and license numbers.
WRCRL	1588	Prints a complex rectangular matrix with a given format and labels.
WRCRN	1586	Prints a complex rectangular matrix with integer row and column labels.
WRIRL	1583	Prints an integer rectangular matrix with a given format and labels.
WRIRN	1581	Prints an integer rectangular matrix with integer row and column labels.
WROPT	1591	Sets or retrieves an option for printing a matrix.
WRRRL	1577	Prints a real rectangular matrix with a given format and labels.
WRRRN	1575	Prints a real rectangular matrix with integer row and column labels.
ZANLY	1153	Finds the zeros of a univariate complex function using Müller’s method.
ZBREN	1156	Finds a zero of a real function that changes sign in a given interval.
ZPLRC	1148	Finds the zeros of a polynomial with real coefficients using Laguerre’s method.
ZPOCC	1152	Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.
ZPORC	1150	Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.
ZQADD	1460	Adds a double complex scalar to the accumulator in extended precision.
ZQINI	1460	Initializes an extended-precision complex accumulator to a double complex scalar.

ZQMUL	1460	Multiplies double complex scalars using extended precision.
ZQSTO	1460	Stores a double complex approximation to an extended-precision complex scalar.
ZREAL	1159	Finds the real zeros of a real function using Müller’s method.

Appendix C: References

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.

Arushanian et al.

Arushanian, O.B., M.K. Samarin, V.V. Voevodin, E.E. Tyrtyshikov, 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.

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*, Elseview 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 Journa*l, **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.

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.

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.

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 Milovanofic

Gautschi, Walter, and Gradimir V. Milovanofic (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 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.

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.

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 Journa*l, **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, *Mathmetics 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.

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 Hillstrom (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 Journa*l **7**, 308–313.

Neter and Wasserman

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

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.

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.

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 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*, SpringerVerlag, 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.

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.

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, J*ournal of the Society for Industrial and Applied Mathematics*, **12**, 515–522.

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. (1965),*The Algebraic Eigenvalue Problem*, Oxford University Press, London, 635.

Product Support

Contacting Visual Numerics Support

Users within support warranty may contact Visual Numerics regarding the use of the IMSL Libraries. Visual Numerics can consult on the following topics:

- Clarity of documentation
- Possible Visual Numerics-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem
- Evolution of the IMSL Libraries

Not included in these consultation topics are mathematical/statistical consulting and debugging of your program.

Consultation

Contact Visual Numerics Product Support by faxing 713/781-9260 or by emailing:

`support@houston.vni.com`.

The following describes the procedure for consultation with Visual Numerics.

1. Include your serial (or license) number
2. Include the product name and version number: IMSL Fortran Library Version 5.0
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