Visual Numerics

VERSION 5.5

IMSL

C Numerical Library™

User's Guide

VOLUME 1 of 4: C Math Library™ [CHAPTERS 1-7]

Visual Numerics, Inc.
Corporate Headquarters
2500 Wilcrest Drive, Ste 200
Houston, Texas 77042-2759
USA

PHONE: 713-784-3131
FAX: 713-781-9260
e-mail: info@vni.com

Visual Numerics International Ltd.
Centennial Court
Suite 1, North Wing
Easthampstead Road
BRACKNELL BERSHIRE
RG12 1YQ
United Kingdom

PHONE: +44-1-344-45-8700
FAX: +44-1-344-45-8748
e-mail: info@vniuk.co.uk

Visual Numerics SARL
Tour Europe
33 Place des Corolles Cedex
F-92049 Paris La Defense
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. 06000
Mexico
PHONE: +52-5514-9730 or 9628
FAX: +52-5514-5880

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
Germany

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

Visual Numerics Japan, Inc
GOBANCHO HIKARI Building 4th Floor
14 Goban-cho Chliyoda-KU
Tokyo, 113
JAPAN

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

Visual Numerics, Inc.
7/F, #510, Chung Hsiao E. Road
Section 5
Taipei, TAIWAN 110
Republic of China

PHONE: (886) 2-727-2255
FAX: (886) 2-727-6798
e-mail: info@vni.com.tw

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-Mapo-Dong, Mapo-gu
Seoul 121-050
Korea

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 and C and Java
Application Development Tools

CMath Library /V1- Table of Contents

Introduction ix

Chapter 1: Linear Systems 1

Chapter 2: Eigensystem Analysis 115

Chapter 3: Interpolation and Approximation 139

Chapter 4: Quadrature 235

Chapter 5: Differential Equations 289

Chapter 6: Transforms 339

Chapter 7: Nonlinear Equations 383

Appendix A: References A-1

Appendix B: Alphabetical Summary of Routines B-1

Index i

Introduction

IMSL C/Math/Library

The IMSL C/Math/Library is a library of C functions useful in scientific programming. Each function is designed and documented to be used in research activities as well as by technical specialists. A number of the example programs also show graphs of resulting output.

Getting Started

To use any of the IMSL C/Math/Library functions, you first must write a program in C to call the function. Each function 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 functions makes it easy to use more than one function in a given application. Also, you will find that the design consistency enables you to apply your experience with one IMSL C/Math/Library function to all other IMSL functions that you use.

ANSI C vs. Non-ANSI C

All of the examples in this user's manual conform to ANSI C. If you are not using ANSI C, you will need to modify your examples in which functions are declared or in which arrays are initialized as the type *float*.

The following is an ANSI C program in which a function is declared. The program estimates the value of the following:

$$\int_0^1 \ln(x)\, x^{-1/2}\, dx = -4$$

```
1 #include <math.h>
2 #include <imsl.h>
3
4 float           fcn(float x);
5
6 main()
7 {
```

```
8     float       q, exact;
9                       /* evaluate the integral */
10    q = imsl_f_int_fcn_sing (fcn, 0.0, 1.0, 0);
11                      /* print the result and the exact answer */
12    exact = -4.0;
13    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
14 }
15
16 float fcn(float x)
17 {
18    return log(x)/sqrt(x);
19 }
```

If using non-ANSI C, you would need to modify lines 4 and 16 as follows:

```
4   float          fcn(); /* function is not prototyped */
     .
     .
     .
16  float fcn(x)          /*Only variable of function defined here */
16a float x;             /* Type of variable declared here */
```

Non-ANSI C does not allow for automatic aggregate initialization, and thus, all *auto* arrays that are initialized as type *float* in ANSI C must be initialized as type *static float* in non-ANSI C. The next program contains arrays that are initialized as type *float*.

```
1 #include <imsl.h>
2
3 main()
4 {
5     int          n = 3;
6     float        *x;
7     float        a[] = {1.0, 3.0, 3.0,
8                         1.0, 3.0, 4.0,
9                         1.0, 4.0, 3.0};
10
11    float        b[] = {1.0, 4.0, -1.0};
12                          /* Solve Ax = b for x */
13    x = imsl_f_lin_sol_gen (n, a, b, 0);
14                          /* Print x */
15    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, 3, x, 0);
16 }
```

If using non-ANSI C, you would need to modify lines 7 and 11 as follows:

```
7     static float       a[] = {1.0, 3.0, 3.0,
         .
         .
         .
11    static float       b[] = {1.0, 4.0, -1.0};
```

The imsl.h File

The include file `<imsl.h>` is used in all of the examples in this manual. This file contains prototypes for all IMSL-defined functions; the spline structures, *Imsl_f_ppoly*, *Imsl_d_ppoly*, *Imsl_f_spline*, and *Imsl_d_spline*; enumerated data

types, *Imsl_quad*, *Imsl_write_options*, *Imsl_page_options*, *Imsl_ode*, and *Imsl_error*; and the
IMSL-defined data types *f_complex* (which is the type *float* complex) and *d_complex* (which is the type *double* complex).

Thread Safe Usage

On systems that support either POSIX threads or WIN32 threads, IMSL C/Math/Library can be safely called from a multithreaded application. When IMSL C/Math/Library is used in a multithreaded application, the calling program must adhere to a few important guidelines. In particular, IMSL C/Math/Library's implementation of signal handling, error handling, and I/O must be understood.

Signal Handling

When calling C/Math/Library from a multithreaded application it is necessary to turn C/Math/Library 's signal-handling capability off. This is accomplished by making a single call to `imsl_error_options` *before* any calls are made to C/Math/Library. For an example of turning off C/Math/Library's internal signal handling , see "Utilities" chapter, Example 3 of `imsl_error_options`.

C/Math/Library 's error handling in a multithreaded application behaves similarly to how it behaves in a single-threaded application. The major difference is that an error stack exists for each thread calling C/Math/Library functions. The result of separate error stacks for each thread is greater control of the error handler options for each thread. Each thread can set its own options for the C/Math/Library error handler using `imsl_error_options`. For an example of setting error handler options for separate threads, see the "Utilities chapter, Example 3 of `imsl_error_options`.

Routines that Produce Output

A number of routines in C/Math/Library can be used to produce output. The function `imsl_output_file` can be used to control which file the output is directed. In an application with a single thread of execution, a single call to `imsl_output_file` can be used to set the file to which the output will be directed. In a multithreaded application each thread must call `imsl_output_file` to change the default setting of where output will be directed. See the "Utilities" chapter, Example 2 of `imsl_output_file` for more details.

Input Arguments

In a multithreaded application attention must be given to the data sent to C/Math/Library. Some arguments that may appear to be input-only are

temporarily modified during the call and restored before returning to the caller. Care must be used to avoid usage of the same data space in separate threads calling functions in C/Math/Library.

Matrix Storage Modes

In this section, the word *matrix* is used to refer to a mathematical object and the word *array* is used to refer to its representation as a C data structure. In the following list of array types, the IMSL C/Math/Library functions require input consisting of matrix dimension values and all values for the matrix entries. These values are stored in
row-major order in the arrays.

Each function processes the input array and typically returns a pointer to a "result."
For example, in solving linear algebraic systems, the pointer is to the solution. For general, real eigenvalue problems, the pointer is to the eigenvalues. Normally, the input array values are not changed by the functions.

In the IMSL C/Math/Library, an array is a pointer to a contiguous block of data. They are *not* pointers to pointers to the rows of the matrix. Typical declarations are:

```
float *a = {1, 2, 3, 4};
float b[2][2] = {1, 2, 3, 4};
float c[] = {1, 2, 3, 4};
```

Note: If you are using non-ANSI C and the variables are of type auto*, then the above declarations would need to be declared as type* static float*.*

General Mode

A *general* matrix is a square $n \times n$ matrix. The data type of a general array can be *float*, *double*, *f_complex,* or *d_complex*.

Rectangular Mode

A *rectangular* matrix is an $m \times n$ matrix. The data type of a rectangular array can be *float, double*, *f_complex,* or *d_complex*.

Symmetric Mode

A *symmetric* matrix is a square $n \times n$ matrix A, such that $A^T = A$. (The matrix A^T is the transpose of A.) The data type of a symmetric array can be *float* or *double*.

Hermitian Mode

A *Hermitian* matrix is a square $n \times n$ matrix A, such that

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

The matrix $\overline{A}$ is the complex conjugate of A, and

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

is the conjugate transpose of A. For Hermitian matrices $A^H = A$. The data type of a Hermitian array can be *f_complex* or *d_complex*.

Sparse Coordinate Storage Format

Only the nonzero elements of a sparse matrix need to be communicated to a function. Sparse coordinate storage format stores the value of each matrix entry along with that entry's row and column index. The following four non-homogeneous data structures are defined to support this concept:

```
typedef struct {
        int row;
        int col;
        float val;
} Imsl_f_sparse_elem;

typedef struct {
        int row;
        int col;
        double val;
} Imsl_d_sparse_elem;

typedef struct {
        int row;
        int col;
        f_complex val;
} Imsl_c_sparse_elem;

typedef struct {
        int row;
        int col;
        d_complex val;
} Imsl_z_sparse_elem;
```

See the "User Errors" section in the "Reference Material" for further detailsSee the Reference Material at the end of this manual for a discussion of the complex data types *f_complex* and *d_complex*. Note that the only difference in these structures involves changes in underlying data types. A sparse matrix is passed to functions that accept sparse coordinate format by forming an array of one of these data types. The number of elements in that array will be equal to the number of nonzeros in the sparse matrix.

As an example consider the 6×6 matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & -3 & -1 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \\ -2 & 0 & 0 & -7 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

The matrix *A* has 15 nonzero elements, and the sparse coordinate representation would be

row	0	1	1	1	2	3	3	3	4	4	4	4	5	5	5
col	0	1	2	3	2	0	3	4	0	3	4	5	0	1	5
val	2	9	–3	–1	5	–2	–7	–1	–1	–5	1	–3	–1	–2	6

Since this representation does not rely on order, an equivalent form would be

row	5	4	3	0	5	1	2	1	4	3	1	4	3	5	4
col	0	0	0	0	1	1	2	2	3	3	3	4	4	5	5
val	–1	–1	–2	2	–2	9	5	–3	–5	–7	–1	1	–1	6	–3

There are different ways this data could be used to initialize an array of type, for example, *Imsl_f_sparse_elem*. Consider the following program fragment:

```
#include <imsl.h>
main()
{
Imsl_f_sparse_elem a[] = {
       {0, 0, 2.0},
       {1, 1, 9.0},
       {1, 2, -3.0},
       {1, 3, -1.0},
       {2, 2, 5.0},
       {3, 0, -2.0},
       {3, 3, -7.0},
       {3, 4, -1.0},
       {4, 0, -1.0},
       {4, 3, -5.0},
       {4, 4, 1.0},
       {4, 5, -3.0},
       {5, 0, -1.0},
       {5, 1, -2.0},
       {5, 5, 6.0} };
Imsl_f_sparse_elem b[15];

       b[0].row = b[0].col = 0;          b[0].val = 2.0;
       b[1].row = b[1].col = 1;          b[1].val = 9.0;
       b[2].row = 1; b[2].col = 2;       b[2].val = -3.0;
       b[3].row = 1; b[3].col = 3;       b[3].val = -1.0;
       b[4].row = b[4].col = 2;          b[4].val = 5.0;
       b[5].row = 3; b[5].col = 0;       b[5].val = -2.0;
       b[6].row = b[6].col = 3;          b[6].val = -7.0;
       b[7].row = 3; b[7].col = 4;       b[7].val = -1;
```

```
        b[8].row = 4; b[8].col = 0;       b[8].val = -1.0;
        b[9].row = 4; b[9].col = 3;       b[9].val = -5.0;
        b[10].row = b[10].col = 4;        b[10].val = 1.0;
        b[11].row = 4; b[11].col = 5;     b[11].val = -3.0;
        b[12].row = 5; b[12].col = 0;     b[12].val = -1.0;
        b[13].row = 5; b[13] = 1;         b[13].val = -2.0;
        b[14].row = b[14].col = 5;        b[14].val = 6.0;
}
```

Both `a` and `b` represent the sparse matrix A, and the functions in this module would produce identical results regardless of which identifier was sent through the argument list.

A sparse symmetric or Hermitian matrix is a special case, since it is only necessary to store the diagonal and either the upper or lower triangle. As an example, consider the
5×5 linear system:

$$H = \begin{bmatrix} (4,0) & (1,-1) & 0 & 0 \\ (1,1) & (4,0) & (1,-1) & 0 \\ 0 & (1,1) & (4,0) & (1,-1) \\ 0 & 0 & (1,1) & (4,0) \end{bmatrix}$$

The Hermitian and symmetric positive definite system solvers in this library expect the diagonal and lower triangle to be specified. The sparse coordinate form for the lower triangle is given by

row	0	1	2	3	1	2	3
col	0	1	2	3	0	1	2
val	(4,0)	(4,0)	(4,0)	(4,0)	(1,1)	(1,1)	(1,1)

As before, an equivalent form would be

row	0	1	1	2	2	3	3
col	0	0	1	1	2	2	3
val	(4,0)	(1,1)	(4,0)	(1,1)	(4,0)	(1,1)	(4,0)

The following program fragment will initialize both `a` and `b` to H.

```
#include <imsl.h>
main()
{
      Imsl_c_sparse_elem a[] = {
              {0, 0, {4.0, 0.0}},
              {1, 1, {4.0, 0.0}},
              {2, 2, {4.0, 0.0}},
              {3, 3, {4.0, 0.0}},
              {1, 0, {1.0, 1.0}},
              {2, 1, {1.0, 1.0}},
              {3, 2, {1.0, 1.0}}
              }
      Imsl_c_sparse_elem b[7];
```

```
    b[0].row = b[0].col = 0;
          b[0].val = imsl_cf_convert (4.0, 0.0);
    b[1].row = 1; b[1].col = 0;
          b[1].val = imsl_cf_convert (1.0, 1.0);
    b[2].row = b[2].col = 1;
          b[2].val = imsl_cf_convert (4.0, 0.0);
    b[3].row = 2; b[3].col = 1;
          b[3].val = imsl_cf_convert (1.0, 1.0);
    b[4].row = b[4].col = 2;
          b[4].val = imsl_cf_convert (4.0, 0.0);
    b[5].row = 3; b[5].col = 2;
          b[5].val = imsl_cf_convert (1.0, 1.0);
    b[6].row = b[6].col = 3;
          b[6].val = imsl_cf_convert (4.0, 0.0);
}
```

There are some important points to note here. H is not symmetric, but rather Hermitian. The functions that accept Hermitian data understand this and operate assuming that

$$h_{ij} = \overline{h}_{ij}$$

The IMSL C/Math/Library cannot take advantage of the symmetry in matrices that are not positive definite. The implication here is that a symmetric matrix that happens to be indefinite cannot be stored in this compact symmetric form. Rather, both upper and lower triangles must be specified and the sparse general solver called.

Band Storage Format

A band matrix is an $M \times N$ matrix with all of its nonzero elements "close" to the main diagonal. Specifically, values $A_{ij} = 0$ if $i - j >$ `nlca` or $j - i >$ `nuca`. The integer
$m =$ `nlca` + `nuca` + 1 is the total band width. The diagonals, other than the main diagonal, are called codiagonals. While any $M \times N$ matrix is a band matrix, band storage format is only useful when the number of nonzero codiagonals is much less than N.

In band storage format, the `nlca` lower codiagonals and the `nuca` upper codiagonals are stored in the rows of an array of size $m \times N$. The elements are stored in the same column of the array as they are in the matrix. The values A_{ij} inside the band width are stored in the linear array in positions [$(i - j$ + `nuca` + 1) * $n + j$]. This results in a row-major, one-dimensional mapping from the two-dimensional notion of the matrix.

For example, consider the 5 × 5 matrix A with 1 lower and 2 upper codiagonals:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & 0 & 0 \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & 0 \\ 0 & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ 0 & 0 & A_{3,2} & A_{3,3} & A_{3,4} \\ 0 & 0 & 0 & A_{4,3} & A_{4,4} \end{bmatrix}$$

In band storage format, the data would be arranged as

$$\begin{bmatrix} 0 & 0 & A_{0,2} & A_{1,3} & A_{2,4} \\ 0 & A_{0,1} & A_{1,2} & A_{2,3} & A_{3,4} \\ A_{0,0} & A_{1,1} & A_{2,2} & A_{3,3} & A_{4,4} \\ A_{1,0} & A_{2,1} & A_{3,2} & A_{4,3} & 0 \end{bmatrix}$$

This data would then be stored contiguously, row-major order, in an array of length 20.

As an example, consider the following tridiagonal matrix:

$$A = \begin{bmatrix} 10 & 1 & 0 & 0 & 0 \\ 5 & 20 & 2 & 0 & 0 \\ 0 & 6 & 30 & 3 & 0 \\ 0 & 0 & 7 & 40 & 4 \\ 0 & 0 & 0 & 8 & 50 \end{bmatrix}$$

The following declaration will store this matrix in band storage format:

```
float a[] = {
        0.0, 1.0, 2.0, 3.0, 4.0,
        10.0, 20.0, 30.0, 40.0, 50.0,
        5.0, 6.0, 7.0, 8.0, 0.0};
```

As in the sparse coordinate representation, there is a space saving symmetric version of band storage. As an example, look at the following 5×5 symmetric problem:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & 0 & 0 \\ A_{0,1} & A_{1,1} & A_{1,2} & A_{1,3} & 0 \\ A_{0,2} & A_{1,2} & A_{2,2} & A_{2,3} & A_{2,4} \\ 0 & A_{1,3} & A_{2,3} & A_{3,3} & A_{3,4} \\ 0 & 0 & A_{2,4} & A_{3,4} & A_{4,4} \end{bmatrix}$$

In band symmetric storage format, the data would be arranged as

$$\begin{bmatrix} 0 & 0 & A_{0,2} & A_{1,3} & A_{2,4} \\ 0 & A_{0,1} & A_{1,2} & A_{2,3} & A_{3,4} \\ A_{0,0} & A_{1,1} & A_{2,2} & A_{3,3} & A_{4,4} \end{bmatrix}$$

The following Hermitian example illustrates the procedure:

$$H = \begin{bmatrix} (8,0) & (1,1) & (1,1) & 0 & 0 \\ (1,-1) & (8,0) & (1,1) & (1,1) & 0 \\ (1,-1) & (1,-1) & (8,0) & (1,1) & (1,1) \\ 0 & (1,-1) & (1,-1) & (8,0) & (1,1) \\ 0 & 0 & (1,-1) & (1,-1) & (8,0) \end{bmatrix}$$

The following program fragments would store H in `h`, using band symmetric storage format.

```
f_complex h[] = {
              {0.0, 0.0}, {0.0, 0.0}, {1.0, 1.0}, {1.0, 1.0}, {1.0, 1.0},
              {0.0, 0.0}, {1.0, 1.0}, {1.0, 1.0}, {1.0, 1.0}, {1.0, 1.0},
              {8.0, 0.0}, {8.0, 0.0}, {8.0, 0.0}, {8.0, 0.0}, {8.0, 0.0}};
```

or equivalently

```
f_complex h[15];
       h[0] = h[1] = h[5] = imsl_cf_convert (0.0, 0.0);
       h[2] = h[3] = h[4] = h[6] = h[7] = h[8] = h[9] =
              imsl_cf_convert (1.0, 1.0);
       h[10] = h[11] = h[12] = h[13] = h[14] =
              imsl_cf_convert (8.0, 0.0);
```

Choosing Between Banded and Coordinate Forms

It is clear that any matrix can be stored in either sparse coordinate or band format. The choice depends on the sparsity pattern of the matrix. A matrix with all nonzero data stored in bands close to the main diagonal would probably be a good candidate for band format. If nonzero information is scattered more or less uniformly through the matrix, sparse coordinate format is the best choice. As extreme examples, consider the following two cases: (1) an $n \times n$ matrix with all elements on the main diagonal and the $(0, n-1)$ and $(n-1, 0)$ entries nonzero. The sparse coordinate vector would be $n + 2$ units long. An array of length $n(2n-1)$ would be required to store the band representation, nearly twice as much storage as a dense solver might require. Secondly, a tridiagonal matrix with all diagonal, superdiagonal and subdiagonal entries nonzero. In band format, an array of length $3n$ is needed. In sparse coordinate, format a vector of length $3n-2$ is required. But the problem is that, for example, float precision on a 32-bit machine, each of those $3n-2$ units in coordinate format requires three times as much storage as any of the $3n$ units needed for band representation. This is due to carrying the row and column indices in coordinate form. Band storage evades this requirement by being essentially an ordered list, and defining location in the original matrix by position in the list.

Compressed Sparse Column (CSC) Format

Functions that accept data in coordinate format can also accept data stored in the format described in the *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*. The scheme is column oriented, with each column held as a sparse vector, represented by a list of the row indices of the entries in an integer array and a list of the corresponding values in a separate *float* (*double*, *f_complex*, *d_complex*) array. Data for each column are stored consecutively and in order. A separate integer array holds the location of the first entry of each column and the first free location. Only entries in the lower triangle and diagonal are stored for symmetric and Hermitian matrices. All arrays are based at zero, which is in contrast to the Harwell-Boeing test suite's one-based arrays.

As in the *Harwell-Boeing Users' Guide*, the storage scheme is illustrated with the following example: The 5 × 5 matrix

$$\begin{bmatrix} 1 & -3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & -4 & 0 \\ 5 & 0 & -5 & 0 & 6 \end{bmatrix}$$

would be stored in the arrays `colptr` (location of first entry), `rowind` (row indices), and `values` (nonzero entries) as follows.

Subscripts	0	1	2	3	4	5	6	7	8	9	10
`colptr`	0	3	5	7	9	11					
`rowind`	0	4	2	3	0	1	4	0	3	4	1
`values`	1	5	2	4	–3	–2	–5	–1	–4	6	3

The following program fragment shows the relation between CSC storage format and coordinate representation:

```
k = 0;
for (i=0; i<n; i++) {
       start = colptr[i];
       stop = colptr[i+1];
       for (j=start; j<stop; j++) {
            a[k].row = rowind[j];
            a[k].col = i;
            a[k++].val = values[j];
       }
}
nz =k;
```

Memory Allocation for Output Arrays

Many functions return a pointer to an array containing the computed answers. If the function invocation uses the optional arguments

`IMSL_RETURN_USER`, *float* `a[]`

then the computed answers are stored in the user-provided array `a`, and the pointer returned by the function is set to point to the user-provided array `a`. If an invocation does not use `IMSL_RETURN_USER`, then the function initializes the pointer (through a memory allocation request to `malloc`) and stores the answers there. (To release this space, `free` can be used. Both `malloc` and `free` are standard C library functions declared in the header <`stdlib.h`>.) In this way, the allocation of space for the computed answers can be made either by the user or internally by the function.

Similarly, other optional arguments specify whether additional computed output arrays are allocated by the user or are to be allocated internally by the function. For example, in many functions in "Linear Systems," the optional arguments

`IMSL_INVERSE_USER`, *float* `inva[]` (Output)
`IMSL_INVERSE`, *float* `**p_inva` (Output)

specify two mutually exclusive optional arguments. If the first option is chosen, the inverse of the matrix is stored in the user-provided array `inva`. In the second option, *float* `**p_inva` refers to the address of a pointer to the inverse. If the second option is chosen, on return, the pointer is initialized (through a memory allocation request to `malloc`), and the inverse of the matrix is stored there. Typically, *float* `*p_inva` is declared, `&p_inva` is used as an argument to this function, and `free(p_inva)` is used to release the space.

Finding the Right Routine

The IMSL C/Math/Library is organized into chapters; each chapter contains functions with similar computational or analytical capabilities. To locate the right function for a given problem, you may use either the table of contents located in each chapter introduction, or the alphabetical "Summary of Functions" at the end of this manual.

Often the quickest way to use the IMSL C/Math/Library is to find an example similar to your problem and then mimic the example. Each function in the document has at least one example demonstrating its application.

Organization of the Documentation

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

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

- **Section Name:** Usually, the common root for the type *float* and type *double* versions of the function is given.
- **Purpose:** A statement of the purpose of the function.
- **Synopsis:** The form for referencing the subprogram with required arguments listed.
- **Required Arguments:** A description of the required arguments in the order of their occurrence, as follows:

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

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

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

- **Return Value:** The value returned by the function.
- **Synopsis with Optional Arguments:** The form for referencing the function with both required and optional arguments listed.
- **Optional Arguments:** A description of the optional arguments in the order of their occurrence.
- **Description:** A description of the algorithm and references to detailed information. In many cases, other IMSL functions with similar or complementary functions are noted.
- **Examples:** At least one application of this function showing input and optional arguments.

- **Errors:** Listing of any errors that may occur with a particular function. A discussion on error types is given in the "User Errors" section of the Reference Material. The errors are listed by their type as follows:

Informational Errors: List of informational errors that may occur with the function.

Alert Errors: List of alert errors that may occur with the function.

Warning Errors: List of warning errors that may occur with the function.

Fatal Errors: List of fatal errors that may occur with the function.

Naming Conventions

Most functions are available in both a type *float* and a type *double* version, with names of the two versions sharing a common root. Some functions also are available in type *int*, or the IMSL-defined types *f_complex* or *d_complex* versions. A list of each type and the corresponding prefix of the function name in which multiple type versions exist follows:

Type	Prefix
float	`imsl_f_`
double	`imsl_d_`
int	`imsl_i_`
f_complex	`imsl_c_`
d_complex	`imsl_z_`

The section names for the functions only contain the common root to make finding the functions easier. For example, the functions `imsl_f_lin_sol_gen` and `imsl_d_lin_sol_gen` can be found in section `lin_sol_gen` in Chapter 1, "Linear Systems".

Where appropriate, the same variable name is used consistently throughout a chapter in the IMSL C/Math/Library. For example, in the functions for eigensystem analysis, `eval` denotes the vector of eigenvalues and `n_eval` denotes the number of eigenvalues computed or to be computed.

When writing programs accessing the IMSL C/Math/Library, the user should choose C names that do not conflict with IMSL external names. The careful user can avoid any conflicts with IMSL names if, in choosing names, the following rule is observed:

- Do not choose a name beginning with "`imsl_`" in any combination of uppercase or lowercase characters.

Error Handling, Underflow, Overflow, and Document Examples

The functions in the IMSL C/Math/Library attempt to detect and report errors and invalid input. 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. Errors are classified according to severity and are assigned a code number. By default, errors of moderate or higher severity result in messages being automatically printed by the function. Moreover, errors of highest 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 symbolic names `IMSL_FATAL`, `IMSL_WARNING`, etc.
See the "User Errors" section in the "Reference Material" for further details.

In general, the IMSL C/Math/Library codes are written so that computations are not affected by underflow, provided the system (hardware or software) replaces an underflow with the value zero. Normally, system error messages indicating underflow can be ignored.

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

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

Output from document examples can be system dependent and the user's results may vary depending upon the system used.

Printing Results

Most functions in the IMSL C/Math/Library do not print any of the results; the output is returned in C variables. You can print the results yourself.

The IMSL C/Math/Library contains some special functions just for printing arrays. For example, `imsl_f_write_matrix` is a convenient function for printing matrices of type *float*. See Chapter 11, "Printing Functions," for detailed descriptions of these functions.

Complex Arithmetic

Users can perform computations with complex arithmetic by using IMSL predefined data types. These types are available in two floating-point precisions:

- `f_complex` for single-precision complex values
- `d_complex` for double-precision complex values

A description of complex data types and functions is given in the Reference Material.

Missing Values

Some of the functions in the IMSL C/Math/Library allow the data to contain missing values. These functions recognize as a missing value the special value referred to as "not a number," or NaN. The actual value is different on different computers, but it can be obtained by reference to the IMSL function `imsl_f_machine`, described in Chapter 12, "Utilities."

The way that missing values are treated depends on the individual function and is described in the documentation for the function.

Passing Data to User-Supplied Functions

In some cases it may be advantageous to pass problem-specific data to a user-supplied function through the IMSL C/Math/Library interface. This ability can be useful if a user-supplied function requires data that is local to the user's calling function, and the user wants to avoid using global data to allow the user-supplied function to access the data. Functions in IMSL C/Math/Library that accept user-supplied functions have an optional argument(s) that will accept an alternative user-supplied function, along with a pointer to the data, that allows user-specified data to be passed to the function. The example below demonstrates this feature using the IMSL C/Math/Library function `imsl_f_min_uncon` and optional argument `IMSL_FCN_W_DATA`.

```
#include "imsl.h"
#include <math.h>

static float fcn_w_data(float x, void *data_ptr);
static float fcn(float);

void main()
{
  float a = -100.0;
```

```
  float b = 100.0;
  float fx, x;
  float usr_data[] = {5.0, 10.0};
  x = imsl_f_min_uncon (fcn, a, b,
                        IMSL_FCN_W_DATA, fcn_w_data, usr_data,
                        0);
  fx = fcn_w_data(x, (void*)usr_data);

  printf ("The solution is: %8.4f\n", x);
  printf ("The function evaluated at the solution is: %8.4f\n", fx);
}

/*
 * User function that accepts additional data in a (void*) pointer.
 * This (void*) pointer can be cast to any type and dereferenced to
 * get at any sort of data-type or structure that is needed.
 * For example, to get at the data in this example
 *  *((float*)data_ptr)   contains the value  5.0
 *  *((float*)data_ptr+1) contains the value 10.0
 */
static float fcn_w_data(float x, void *data_ptr)
{
  return exp(x) - (*((float*)data_ptr))*x + *((float*)data_ptr+1);
}

/*  Dummy function to satisfy C prototypes. */
static float fcn(float x)
{
  return;
}
```

Chapter 1: Linear Systems

Routines

1.1 Linear Equations with Full Matrices

Factor, Solve, and Inverse for General Matrices
Real matrices ..... lin_sol_gen 4
Complex matrices..... lin_sol_gen (complex) 11

Factor, Solve, and Inverse for Positive Definite Matrices
Real matrices ..... lin_sol_posdef 17
Complex matrices..... lin_sol_posdef (complex) 22

1.2 Linear Equations with Band Matrices

Factor and Solve for Band Matrices
Real matrices ..... lin_sol_gen_band 26
Complex matrices..... lin_sol_gen_band (complex) 31

Factor and Solve for Positive Definite Matrices Symmetric
Real matrices ..... lin_sol_posdef_band 35
Complex matrices..... lin_sol_posdef_band (complex) 39

1.3 Linear Equations with General Sparse Matrices

Factor and Solve for Sparse Matrices
Real matrices ..... lin_sol_gen_coordinate 44
Complex matrices..... lin_sol_gen_coordinate (complex) 54

Factor and Solve for Positive Definite Matrices
Real matrices ..... lin_sol_posdef_coordinate 62
Complex matrices..... lin_sol_posdef_coordinate (complex) 68

1.4 Iterative Methods

Restarted generalized minimum
residual (GMRES) method ..... lin_sol_gen_min_residual 73
Conjugate gradient method ..... lin_sol_def_cg 78

1.5 Linear Least-squares with Full Matrices

Least-squares and QR decomposition
Least-squares solve, QR decomposition ..... lin_least_squares_gen 84
Linear constraints ..... lin_lsq_lin_constraints 92

Singular Value Decompositions (SVD) and Generalized Inverse
Real matrix ..... lin_svd_gen 96
Complex matrix ..... lin_svd_gen (complex) 102

Factor, Solve, and Generalized Inverse for Positive Semidefinite Matrices
Real matrices ..... lin_sol_nonnegdef 107

Usage Notes

Solving Systems of Linear Equations

A square system of linear equations has the form Ax = b, where A is a user-specified $n \times n$ matrix, b is a given right-hand side n vector, and x is the solution n vector. Each entry of A and b must be specified by the user. The entire vector x is returned as output.

When A is invertible, a unique solution to $Ax = b$ exists. The most commonly used direct method for solving $Ax = b$ factors the matrix A into a product of triangular matrices and solves the resulting triangular systems of linear equations. Functions that use direct methods for solving systems of linear equations all compute the solution to $Ax = b$. Thus, if a function with the prefix "`imsl_f_lin_sol`" is called with the required arguments, a pointer to x is returned by default. Additional tasks, such as only factoring the matrix A into a product of triangular matrices, can be done using keywords.

Matrix Factorizations

In some applications, it is desirable to just factor the $n \times n$ matrix A into a product of two triangular matrices. This can be done by calling the appropriate function for solving the system of linear equations $Ax = b$. Suppose that in addition to the solution x of a linear system of equations $Ax = b$, the LU factorization of A is desired. Use the keyword `IMSL_FACTOR` in the function `imsl_f_lin_sol_gen` (page 4) to obtain access to the factorization. If only the factorization is desired, use the keywords `IMSL_FACTOR_ONLY` and `IMSL_FACTOR`.

Besides the basic matrix factorizations, such as LU and LL^T, additional matrix factorizations also are provided. For a real matrix A, its QR factorization can be computed by the function `imsl_f_lin_least_squares_gen` (page 84). Functions for computing the singular value decomposition (SVD) of a matrix are discussed in a later section.

Matrix Inversions

The inverse of an $n \times n$ nonsingular matrix can be obtained by using the keyword `IMSL_INVERSE` in functions for solving systems of linear equations. The inverse of a

matrix need not be computed if the purpose is to *solve* one or more systems of linear equations. Even with multiple right-hand sides, solving a system of linear equations by computing the inverse and performing matrix multiplication is usually more expensive than the method discussed in the next section.

Multiple Right-Hand Sides

Consider the case where a system of linear equations has more than one right-hand side vector. It is most economical to find the solution vectors by first factoring the coefficient matrix A into products of triangular matrices. Then, the resulting triangular systems of linear equations are solved for each right-hand side. When A is a real general matrix, access to the LU factorization of A is computed by using the keywords `IMSL_FACTOR` and `IMSL_FACTOR_ONLY` in function `imsl_f_lin_sol_gen` (page 4). The solution x_k for the k-th right-hand side vector b_k is then found by two triangular solves, $Ly_k = b_k$ and $Ux_k = y_k$. The keyword `IMSL_SOLVE_ONLY` in the function `imsl_f_lin_sol_gen` is used to solve each right-hand side. These arguments are found in other functions for solving systems of linear equations.

Least-Squares Solutions and *QR* Factorizations

Least-squares solutions are usually computed for an over-determined system of linear equations $A_{m\times n}\, x = b$, where $m > n$. A least-squares solution x minimizes the Euclidean length of the residual vector $r = Ax - b$. The function `imsl_f_lin_least_squares_gen` (page 84) computes a unique least-squares solution for x when A has full column rank. If A is rank-deficient, then the *base* solution for some variables is computed. These variables consist of the resulting columns after the interchanges. The QR decomposition, with column interchanges or pivoting, is computed such that $AP = QR$. Here, Q is orthogonal, R is upper-trapezoidal with its diagonal elements nonincreasing in magnitude, and P is the permutation matrix determined by the pivoting. The base solution x_B is obtained by solving $R(P^T)x = Q^T b$ for the base variables. For details, see "Description" in `imsl_f_lin_least_squares_gen` (page 84). The QR factorization of a matrix A such that $AP = QR$ with P specified by the user can be computed using keywords.

Singular Value Decompositions and Generalized Inverses

The SVD of an $m \times n$ matrix A is a matrix decomposition $A = USV^T$. With $q = \min(m, n)$, the factors $U_{m\times q}$ and $V_{n\times q}$ are orthogonal matrices, and $S_{q\times q}$ is a nonnegative diagonal matrix with nonincreasing diagonal terms. The function `imsl_f_lin_svd_gen` (page 96) computes the singular values of A by default. Using keywords, part or all of the U and V matrices, an estimate of the rank of A, and the generalized inverse of A, also can be obtained.

Ill-Conditioning and Singularity

An $m \times n$ matrix A is mathematically singular if there is an $x \neq 0$ such that $Ax = 0$. In this case, the system of linear equations $Ax = b$ does not have a unique solution. On the other hand, a matrix A is *numerically* singular if it is "close" to a mathematically

singular matrix. Such problems are called *ill-conditioned*. If the numerical results with an ill-conditioned problem are unacceptable, users can either use more accuracy if it is available (for type *float* accuracy switch to *double*) or they can obtain an *approximate* solution to the system. One form of approximation can be obtained using the SVD of A: If $q = \min(m, n)$ and

$$A = \sum_{i=1}^{q} s_{i,i} u_i v_i^T$$

then the approximate solution is given by the following:

$$x_k = \sum_{i=1}^{k} t_{i,i} \left(b^T u_i \right) v_i$$

The scalars $t_{i,i}$ are defined below.

$$t_{i,i} = \begin{cases} s_{i,i}^{-1} & \text{if } s_{i,i} \geq tol > 0 \\ 0 & \text{otherwise} \end{cases}$$

The user specifies the value of *tol*. This value determines how "close" the given matrix is to a singular matrix. Further restrictions may apply to the number of terms in the sum, $k \leq q$. For example, there may be a value of $k \leq q$ such that the scalars $|(b^T u_i)|$, $i > k$ are smaller than the average uncertainty in the right-hand side b. This means that these scalars can be replaced by zero; and hence, b is replaced by a vector that is within the stated uncertainty of the problem.

lin_sol_gen

Solves a real 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, computing the inverse matrix A^{-1}, solving $A^T x = b$, or computing the solution of $Ax = b$ given the LU factorization of A.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_gen` (*int* `n`, *float* `a[]`, *float* `b[]`, ..., 0)

The type *double* procedure is `imsl_d_lin_sol_gen`.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

float `a[]` (Input)
Array of size $n \times n$ containing the matrix.

float `b[]` (Input)
Array of size *n* containing the right-hand side.

Return Value

A pointer to the solution *x* of the linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_lin_sol_gen (int n, float a[], float b[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_TRANSPOSE,
        IMSL_RETURN_USER, float x[],
        IMSL_FACTOR, int **p_pvt, float **p_factor,
        IMSL_FACTOR_USER, int pvt[], float factor[],
        IMSL_FAC_COL_DIM, int fac_col_dim,
        IMSL_INVERSE, float **p_inva,
        IMSL_INVERSE_USER, float inva[],
        IMSL_INV_COL_DIM, int inva_col_dim,
        IMSL_CONDITION, float *cond,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_INVERSE_ONLY,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = *n*

`IMSL_TRANSPOSE`
Solve $A^T x = b$.
Default: Solve $Ax = b$

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *int* `**p_pvt`, *float* `**p_factor` (Output)

`p_pvt`: The address of a pointer to an array of length *n* containing the pivot sequence for the factorization. On return, the necessary space is allocated by `imsl_f_lin_sol_gen`. Typically, *int* `*p_pvt` is declared, and `&p_pvt` is used as an argument.

`p_factor`: The address of a pointer to an array of size $n \times n$ containing the *LU* factorization of *A* with column pivoting. On return, the necessary space is allocated by `imsl_f_lin_sol_gen`. The lower-triangular part of this array contains information necessary to construct *L*, and the upper-triangular part

contains *U*. Typically, *float* `*p_factor` is declared, and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *int* `pvt[]`, *float* `factor[]` (Input/Output)

`pvt[]`: A user-allocated array of size *n* containing the pivot sequence for the factorization.

`factor[]`: A user-allocated array of size $n \times n$ containing the *LU* factorization of *A*. The strictly lower-triangular part of this array contains information necessary to construct *L*, and the upper-triangular part contains *U*. If *A* is not needed, `factor` and `a` can share the same storage.

These parameters are *input* if `IMSL_SOLVE` is specified. They are *output* otherwise.

`IMSL_FAC_COL_DIM`, *int* `fac_col_dim` (Input)
The column dimension of the array containing the *LU* factorization of *A*. Default: `fac_col_dim` = *n*

`IMSL_INVERSE`, *float* `**p_inva` (Output)
The address of a pointer to an array of size $n \times n$ containing the inverse of the matrix *A*. On return, the necessary space is allocated by `imsl_f_lin_sol_gen`. Typically, *float* `*p_inva` is declared, and `&p_inva` is used as an argument.

`IMSL_INVERSE_USER`, *float* `inva[]` (Output)
A user-allocated array of size $n \times n$ containing the inverse of *A*.

`IMSL_INV_COL_DIM`, *int* `inva_col_dim` (Input)
The column dimension of the array containing the inverse of *A*. Default: `inva_col_dim` = *n*

`IMSL_CONDITION`, *float* `*cond` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix *A*. This option cannot be used with the option `IMSL_SOLVE_ONLY`.

`IMSL_FACTOR_ONLY`
Compute the *LU* factorization of *A* with partial pivoting. If `IMSL_FACTOR_ONLY` is used, either `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required. The argument `b` is then ignored, and the returned value of `imsl_f_lin_sol_gen` is `NULL`.

`IMSL_SOLVE_ONLY`
Solve $Ax = b$ given the *LU* factorization previously computed by `imsl_f_lin_sol_gen`. By default, the solution to $Ax = b$ is pointed to by `imsl_f_lin_sol_gen`. If `IMSL_SOLVE_ONLY` is used, argument `IMSL_FACTOR_USER` is required, and the argument `a` is ignored.

`IMSL_INVERSE_ONLY`
Compute the inverse of the matrix *A*. If `IMSL_INVERSE_ONLY` is used, either

IMSL_INVERSE or IMSL_INVERSE_USER is required. The argument b is then ignored, and the returned value of imsl_f_lin_sol_gen is NULL.

Description

The function imsl_f_lin_sol_gen solves a system of linear algebraic equations with a real coefficient matrix A. It first computes the LU factorization of A with partial pivoting such that $L^{-1}A = U$. The matrix U is upper triangular, while $L^{-1}A \equiv P_n L_{n-1} P_{n-1} \ldots L_1 P_1 A \equiv U$. 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 factorization efficiency is based on a technique of "loop unrolling and jamming" by Dr. Leonard J. Harding of the University of Michigan, Ann Arbor, Michigan. The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x. The function imsl_f_lin_sol_gen fails if U, the upper triangular part of the factorization, has a zero diagonal element.

Examples

Example 1

This example solves a system of three linear equations. This is the simplest use of the function. The equations follow below:

$$x_1 + 3x_2 + 3x_3 = 1$$
$$x_1 + 3x_2 + 4x_3 = 4$$
$$x_1 + 4x_2 + 3x_3 = -1$$

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       *x;
    float       a[] =  {1.0, 3.0, 3.0,
                        1.0, 3.0, 4.0,
                        1.0, 4.0, 3.0};
    float       b[] =  {1.0, 4.0, -1.0};
                              /* Solve Ax = b  for  x  */
```

```
    x = imsl_f_lin_sol_gen (n, a, b, 0);
                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, 3, x, 0);
}
```

Output

```
Solution, x, of Ax = b
 1           2           3
-2          -2           3
```

Example 2

This example solves the transpose problem $A^T x = b$ and returns the LU factorization of A with partial pivoting. The same data as the initial example is used, except the solution $x = A^{-T} b$ is returned in an array allocated in the main program. The L matrix is returned in implicit form.

```
#include <imsl.h>

main()
{
    int         n = 3, pvt[3];
    float       factor[9];
    float       x[3];
    float       a[] =  {1.0, 3.0, 3.0,
                        1.0, 3.0, 4.0,
                        1.0, 4.0, 3.0};

    float       b[] =  {1.0, 4.0, -1.0};
                                /* Solve trans(A)*x = b  for  x  */
    imsl_f_lin_sol_gen (n, a, b,
                        IMSL_TRANSPOSE,
                        IMSL_RETURN_USER, x,
                        IMSL_FACTOR_USER, pvt, factor,
                        0);

                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of trans(A)x = b", 1, n, x, 0);

                                /* Print factors and pivot sequence */
    imsl_f_write_matrix ("LU factors of A", n, n, factor, 0);
    imsl_i_write_matrix ("Pivot sequence", 1, n, pvt, 0);
}
```

Output

```
Solution, x, of trans(A)x = b
     1           2           3
     4          -4           1

           LU factors of A
            1           2           3
1           1           3           3
2          -1           1           0
3          -1           0           1
```

```
Pivot sequence
   1   2   3
   1   3   3
```

Example 3

This example computes the inverse of the 3×3 matrix A of the initial example and solves the same linear system. The matrix product $C = A^{-1}A$ is computed and printed. The function `imsl_f_mat_mul_rect` is used to compute C. The approximate result $C = I$ is obtained.

```
#include <imsl.h>

float    a[] =  {1.0, 3.0, 3.0,
                 1.0, 3.0, 4.0,
                 1.0, 4.0, 3.0};

float    b[] =  {1.0, 4.0, -1.0};

main()
{
    int          n = 3;
    float        *x;
    float        *p_inva;
    float        *C;
                                    /* Solve Ax = b  */
    x = imsl_f_lin_sol_gen (n, a, b,
        IMSL_INVERSE, &p_inva,
        0);

                                    /* Print solution */

    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                                    /* Print input and inverse matrices */
    imsl_f_write_matrix ("Input A", n, n, a, 0);
    imsl_f_write_matrix ("Inverse of A", n, n, p_inva, 0);
                                    /* Check result and print */
    C = imsl_f_mat_mul_rect("A*B",
        IMSL_A_MATRIX, n, n, p_inva,
        IMSL_B_MATRIX, n, n, a,
        0);
    imsl_f_write_matrix ("Product matrix, inv(A)*A",n,n,C,0);
}
```

Output

```
Solution, x, of Ax = b
   1           2           3
  -2          -2           3

              Input A
             1           2           3
1            1           3           3
2            1           3           4
```

```
3           1           4           3

            Inverse of A
            1           2           3
1           7          -3          -3
2          -1           0           1
3          -1           1           0

      Product matrix, inv(A)*A
            1           2           3
1           1           0           0
2           0           1           0
3           0           0           1
```

Example 4

This example computes the solution of two systems. Only the right-hand sides differ. The matrix and first right-hand side are given in the initial example. The second right-hand side is the vector $c = [0.5, 0.3, 0.4]^T$. The factorization information is computed with the first solution and is used to compute the second solution. The factorization work done in the first step is avoided in computing the second solution.

```
#include <imsl.h>

main()
{
    int         n = 3, pvt[3];
    float       factor[9];
    float       *x,*y;

    float       a[] =  {1.0, 3.0, 3.0,
                        1.0, 3.0, 4.0,
                        1.0, 4.0, 3.0};

    float       b[] =  {1.0, 4.0, -1.0};
    float       c[]  = {0.5, 0.3,  0.4};

                                /* Solve A*x = b  for  x  */
    x = imsl_f_lin_sol_gen (n, a, b,
                        IMSL_FACTOR_USER, pvt, factor,
                        0);

                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                               /* Solve for A*y = c for y */
    y = imsl_f_lin_sol_gen (n, a, c,
                        IMSL_SOLVE_ONLY,
                        IMSL_FACTOR_USER, pvt, factor,
                        0);
    imsl_f_write_matrix ("Solution, y, of Ay = c", 1, n, y, 0);

}
```

Output

```
Solution, x, of Ax = b
   1           2           3
```

```
  -2          -2           3

Solution, y, of Ay = c
   1           2           3
 1.4        -0.1        -0.2
```

Warning Errors

`IMSL_ILL_CONDITIONED`	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

`IMSL_SINGULAR_MATRIX`	The input matrix is singular.

lin_sol_gen (complex)

Solves a complex 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, computing the inverse matrix A^{-1}, solving $A^H x = b$, or computing the solution of $Ax = b$ given the LU factorization of A.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_gen` (*int* `n`, *f_complex* `a[]`, *f_complex* `b[]`, …, 0)

The type *d_complex* procedure is `imsl_z_lin_sol_gen`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

f_complex `a[]` (Input)
Array of size $n \times n$ containing the matrix.

f_complex `b[]` (Input)
Array of length n containing the right-hand side.

Return Value

A pointer to the solution x of the linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
f_complex *imsl_c_lin_sol_gen (int n, f_complex a[], f_complex b[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_TRANSPOSE,
        IMSL_RETURN_USER, f_complex x[],
        IMSL_FACTOR, int **p_pvt, f_complex **p_factor,
        IMSL_FACTOR_USER, int pvt[], f_complex factor[],
        IMSL_FAC_COL_DIM, int fac_col_dim,
        IMSL_INVERSE, f_complex **p_inva,
        IMSL_INVERSE_USER, f_complex inva[],
        IMSL_INV_COL_DIM, int inva_col_dim,
        IMSL_CONDITION, float *cond,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_INVERSE_ONLY,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array *a*.
Default: `a_col_dim` = *n*

`IMSL_TRANSPOSE`
Solve $A^H x = b$
Default: Solve $Ax = b$

`IMSL_RETURN_USER`, *f_complex* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *int* `**p_pvt`, *f_complex* `**p_factor` (Output)

`p_pvt`: The address of a pointer to an array of length *n* containing the pivot sequence for the factorization. On return, the necessary space is allocated by `imsl_c_lin_sol_gen`. Typically, *int* `*p_pvt` is declared, and `&p_pvt` is used as an argument.

`p_factor`: The address of a pointer to an array of size $n \times n$ containing the *LU* factorization of *A* with column pivoting. On return, the necessary space is allocated by `imsl_c_lin_sol_gen`. The lower-triangular part of this array contains information necessary to construct *L*, and the upper-triangular part contains *U*. Typically, *f_complex* `*p_factor` is declared, and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *int* `pvt[]`, *f_complex* `factor[]` (Input/Output)

`pvt[]`: A user-allocated array of size *n* containing the pivot sequence for the factorization.

`factor[]`: A user-allocated array of size $n \times n$ containing the *LU* factorization of *A*. The lower-triangular part of this array contains information necessary to construct *L*, and the upper-triangular part contains *U*.

These parameters are *input* if `IMSL_SOLVE` is specified. They are *output* otherwise. If A is not needed, `factor` and `a` can share the same storage.

`IMSL_FAC_COL_DIM`, *int* `fac_col_dim` (Input)
The column dimension of the array containing the LU factorization of A. Default: `fac_col_dim` = n

`IMSL_INVERSE`, *f_complex* `**p_inva` (Output)
The address of a pointer to an array of size $n \times n$ containing the inverse of the matrix A. On return, the necessary space is allocated by `imsl_c_lin_sol_gen`. Typically, *f_complex* `*p_inva` is declared, and `&p_inva` is used as an argument.

`IMSL_INVERSE_USER`, *f_complex* `inva[]` (Output)
A user-allocated array of size $n \times n$ containing the inverse of A.

`IMSL_INV_COL_DIM`, *int* `inva_col_dim` (Input)
The column dimension of the array containing the inverse of A. Default: `inva_col_dim` = n

`IMSL_CONDITION`, *float* `*cond` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix A. Do not use this option with `IMSL_SOLVE_ONLY`.

`IMSL_FACTOR_ONLY`
Compute the LU factorization of A with partial pivoting. If `IMSL_FACTOR_ONLY` is used, either `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required. The argument `b` is then ignored, and the returned value of `imsl_c_lin_sol_gen` is `NULL`.

`IMSL_SOLVE_ONLY`
Solve $Ax = b$ given the LU factorization previously computed by `imsl_c_lin_sol_gen`. By default, the solution to $Ax = b$ is pointed to by `imsl_c_lin_sol_gen`. If `IMSL_SOLVE_ONLY` is used, argument `IMSL_FACTOR_USER` is required and argument `a` is ignored.

`IMSL_INVERSE_ONLY`
Compute the inverse of the matrix A. If `IMSL_INVERSE_ONLY` is used, either `IMSL_INVERSE` or `IMSL_INVERSE_USER` is required. Argument `b` is then ignored, and the returned value of `imsl_c_lin_sol_gen` is `NULL`.

Description

The function `imsl_c_lin_sol_gen` solves a system of linear algebraic equations with a complex coefficient matrix A. It first computes the LU factorization of A with partial pivoting such that $L^{-1}A = U$. The matrix U is upper-triangular, while $L^{-1}A \equiv P_n L_{n-1} P_{n-1} \ldots L_1 P_1 A \equiv U$. 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 in the strictly lower-triangular part of column i of the working array containing the decomposition information.

The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is computed, an estimate of the L_1 condition number of A is computed using the algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x. The function `imsl_c_lin_sol_gen` fails if U, the upper-triangular part of the factorization, has a zero diagonal element.

Examples

Example 1

This example solves a system of three linear equations. The equations are:

$$(1 + i)\,x_1 + (2 + 3i)\,x_2 + (3 - 3i)\,x_3 = 3 + 5i$$

$$(2 + i)\,x_1 + (5 + 3i)\,x_2 + (7 - 5i)\,x_3 = 22 + 10i$$

$$(-2 + i)\,x_1 + (-4 + 4i)\,x_2 + (5 + 3i)\,x_3 = -10 + 4i$$

```
#include <imsl.h>

f_complex    a[] =  {{1.0, 1.0}, {2.0, 3.0}, {3.0, -3.0},
                     {2.0, 1.0}, {5.0, 3.0}, {7.0, -5.0},
                     {-2.0, 1.0}, {-4.0, 4.0}, {5.0, 3.0}};

f_complex    b[] = {{3.0, 5.0}, {22.0, 10.0}, {-10.0, 4.0}};

main()
{
    int           n = 3;
    f_complex       *x;
                                    /* Solve Ax = b  for  x  */
    x = imsl_c_lin_sol_gen (n, a, b, 0);

                                    /* Print x */
    imsl_c_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
                        Solution, x, of Ax = b
                      1                         2                         3
(         1,        -1)  (         2,         4)  (         3,        -0)
```

Example 2

This example solves the conjugate transpose problem $A^H x = b$ and returns the LU factorization of A using partial pivoting. This example differs from the first example in that the solution array is allocated in the main program.

```
#include <imsl.h>

f_complex    a[] =  {{1.0, 1.0},  {2.0, 3.0},   {3.0, -3.0},
                     {2.0, 1.0},  {5.0, 3.0},   {7.0, -5.0},
                     {-2.0, 1.0}, {-4.0, 4.0},  {5.0,  3.0}};

f_complex    b[] =  {{3.0, 5.0},  {22.0, 10.0}, {-10.0, 4.0}};

main()
{
    int           n = 3, pvt[3];
    f_complex     factor[9];
    f_complex     x[3];
                                  /* Solve ctrans(A)*x = b  for  x  */
    imsl_c_lin_sol_gen (n, a, b,
                IMSL_TRANSPOSE,
                IMSL_RETURN_USER, x,
                IMSL_FACTOR_USER, pvt, factor,
                0);
                                  /* Print x */
    imsl_c_write_matrix ("Solution, x, of ctrans(A)x = b", 1, n, x, 0);

                                  /* Print factors and pivot sequence */
    imsl_c_write_matrix ("LU factors of A", n, n, factor, 0);
    imsl_i_write_matrix ("Pivot sequence", 1, n, pvt, 0);
}
```

Output

```
                    Solution, x, of ctrans(A)x = b
                     1                      2                      3
(     -9.79,     11.23)  (      2.96,     -3.13)  (      1.85,      2.47)

                              LU factors of A
                        1                      2                      3
1  (    -2.000,    1.000)  (    -4.000,     4.000)  (     5.000,     3.000)
2  (     0.600,    0.800)  (    -1.200,     1.400)  (     2.200,     0.600)
3  (     0.200,    0.600)  (    -1.118,     0.529)  (     4.824,     1.294)
Pivot sequence
   1   2   3
   3   3   3
```

Example 3

This example computes the inverse of the 3×3 matrix A in the first example and also solves the linear system. The product matrix $C = A^{-1}A$ is computed as a check. The approximate result is $C = I$.

```
#include <imsl.h>

f_complex    a[] =  {{1.0, 1.0},  {2.0, 3.0},   {3.0, -3.0},
                     {2.0, 1.0},  {5.0, 3.0},   {7.0, -5.0},
```

```
                    {-2.0, 1.0}, {-4.0, 4.0},  {5.0, 3.0}};

f_complex    b[] =  {{3.0, 5.0},  {22.0, 10.0}, {-10.0, 4.0}};

main()
{
    int           n = 3;
    f_complex     *x;
    f_complex     *p_inva;
    f_complex     *C;

                                  /* Solve Ax = b  for  x  */
    x = imsl_c_lin_sol_gen (n, a, b,
                    IMSL_INVERSE, &p_inva,
                    0);

                                  /* Print solution */
    imsl_c_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                                  /* Print input and inverse matrices */
    imsl_c_write_matrix ("Input A", n, n, a, 0);
    imsl_c_write_matrix ("Inverse of A", n, n, p_inva, 0);

                                  /* Check and print result */
    C = imsl_c_mat_mul_rect ("A*B",
                    IMSL_A_MATRIX, n,n, p_inva,
                    IMSL_B_MATRIX, n,n, a,
                    0);
    imsl_c_write_matrix ("Product, inv(A)*A", n, n, C, 0);
}
```

Output

```
                        Solution, x, of Ax = b
                      1                         2                         3
(         1,        -1)  (         2,         4)  (         3,        -0)

                                  Input A
                        1                         2                         3
1 (         1,         1)  (         2,         3)  (         3,        -3)
2 (         2,         1)  (         5,         3)  (         7,        -5)
3 (        -2,         1)  (        -4,         4)  (         5,         3)
                               Inverse of A
                        1                         2                         3
1 (     1.330,     0.594)  (    -0.151,     0.028)  (    -0.604,     0.613)
2 (    -0.632,    -0.538)  (     0.160,     0.189)  (     0.142,    -0.245)
3 (    -0.189,     0.160)  (     0.193,    -0.052)  (     0.024,     0.042)

                              Product, inv(A)*A
                        1                         2                         3
1 (         1,        -0)  (        -0,        -0)  (        -0,         0)
2 (         0,         0)  (         1,         0)  (         0,        -0)
3 (        -0,        -0)  (        -0,         0)  (         1,         0)
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of the L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_SINGULAR_MATRIX	The input matrix is singular.

lin_sol_posdef

Solves a real symmetric positive definite system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the Cholesky factor, L, of A such that $A = LL^T$, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the Cholesky factor, L.

Synopsis

#*include* `<imsl.h>`

float `*imsl_f_lin_sol_posdef` (*int* `n`, *float* `a[]`, *float* `b[]`, ..., 0)

The type *double* procedure is `imsl_d_lin_sol_posdef`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

float `a[]` (Input)
Array of size $n \times n$ containing the matrix.

float `b[]` (Input)
Array of size n containing the right-hand side.

Return Value

A pointer to the solution x of the symmetric positive definite linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#*include* `<imsl.h>`

```
float *imsl_f_lin_sol_posdef (int n, float a[], float b[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_RETURN_USER, float x[],
        IMSL_FACTOR, float **p_factor,
        IMSL_FACTOR_USER, float factor[],
        IMSL_FAC_COL_DIM, int fac_col_dim,
        IMSL_INVERSE, float **p_inva,
```

```
IMSL_INVERSE_USER, float inva[],
IMSL_INV_COL_DIM, int inv_col_dim,
IMSL_CONDITION, float *cond,
IMSL_FACTOR_ONLY,
IMSL_SOLVE_ONLY,
IMSL_INVERSE_ONLY,
0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length n containing the solution x.

`IMSL_FACTOR`, *float* `**p_factor` (Output)
The address of a pointer to an array of size $n \times n$ containing the LL^T factorization of A. On return, the necessary space is allocated by `imsl_f_lin_sol_posdef`. The lower-triangular part of this array contains L and the upper-triangular part contains L^T. Typically, *float* `*p_factor` is declared, and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *float* `factor[]` (Input/Output)
A user-allocated array of size $n \times n$ containing the LL^T factorization of A. The lower-triangular part of this array contains L, and the upper-triangular part contains L^T. If A is not needed, `a` and `factor` can share the same storage. If `IMSL_SOLVE` is specified, it is *input*; otherwise, it is *output*.

`IMSL_FAC_COL_DIM`, *int* `fac_col_dim` (Input)
The column dimension of the array containing the LL^T factorization of A.
Default: `fac_col_dim` = n

`IMSL_INVERSE`, *float* `**p_inva` (Output)
The address of a pointer to an array of size $n \times n$ containing the inverse of the matrix A. On return, the necessary space is allocated by `imsl_f_lin_sol_posdef`. Typically, *float* `*p_inva` is declared, and `&p_inva` is used as an argument.

`IMSL_INVERSE_USER`, *float* `inva[]` (Output)
A user-allocated array of size $n \times n$ containing the inverse of A.

`IMSL_INV_COL_DIM`, *int* `inva_col_dim` (Input)
The column dimension of the array containing the inverse of A.
Default: `inva_col_dim` = n

`IMSL_CONDITION`, *float* `*cond` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix A. Do not use this option with `IMSL_SOLVE_ONLY`.

IMSL_FACTOR_ONLY
: Compute the Cholesky factorization LL^T of A. If IMSL_FACTOR_ONLY is used, either IMSL_FACTOR or IMSL_FACTOR_USER is required. The argument b is then ignored, and the returned value of imsl_f_lin_sol_posdef is NULL.

IMSL_SOLVE_ONLY
: Solve $Ax = b$ given the LL^T factorization previously computed by imsl_f_lin_sol_posdef. By default, the solution to $Ax = b$ is pointed to by imsl_f_lin_sol_posdef. If IMSL_SOLVE_ONLY is used, argument IMSL_FACTOR_USER is required and the argument a is ignored.

IMSL_INVERSE_ONLY
: Compute the inverse of the matrix A. If IMSL_INVERSE_ONLY is used, either IMSL_INVERSE or IMSL_INVERSE_USER is required. The argument b is then ignored, and the returned value of imsl_f_lin_sol_posdef is NULL.

Description

The function imsl_f_lin_sol_posdef solves a system of linear algebraic equations having a symmetric positive definite coefficient matrix A. The function first computes the Cholesky factorization LL^T of A. The solution of the linear system is then found by solving the two simpler systems, $y = L^{-1}b$ and $x = L^{-T}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The function imsl_f_lin_sol_posdef fails if L, the lower-triangular matrix in the factorization, has a zero diagonal element.

Examples

Example 1

A system of three linear equations with a symmetric positive definite coefficient matrix is solved in this example. The equations are listed below:

$$x_1 - 3x_2 + 2x_3 = 27$$

$$-3x_1 + 10x_2 - 5x_3 = -78$$

$$2x_1 - 5x_2 + 6x_3 = 64$$

```
#include <imsl.h>

main()
{
```

```
    int         n = 3;
    float       *x;
    float       a[] =  {1.0, -3.0,  2.0,
                        -3.0, 10.0, -5.0,
                         2.0, -5.0,  6.0};
    float       b[] =  {27.0, -78.0, 64.0};

                                /* Solve Ax = b  for  x  */
    x = imsl_f_lin_sol_posdef (n, a, b, 0);

                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
Solution, x, of Ax = b
   1           2           3
   1          -4           7
```

Example 2

This example solves the same system of three linear equations as in the initial example, but this time returns the LL^T factorization of A. The solution x is returned in an array allocated in the main program.

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       x[3], *p_factor;
    float       a[] =  {1.0, -3.0,  2.0,
                        -3.0, 10.0, -5.0,
                         2.0, -5.0,  6.0};
    float       b[] =  {27.0, -78.0, 64.0};

                                /* Solve Ax = b  for  x  */
    imsl_f_lin_sol_posdef (n, a, b,
                IMSL_RETURN_USER, x,
                IMSL_FACTOR, &p_factor,
                0);

                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                                /* Print Cholesky factor of A */
    imsl_f_write_matrix ("Cholesky factor L, and trans(L), of A",
                         n, n, p_factor, 0);
}
```

Output

```
Solution, x, of Ax = b
1           2           3
1          -4           7

Cholesky factor L, and trans(L), of A
```

```
             1           2           3
1            1          -3           2
2           -3           1           1
3            2           1           1
```

Example 3

This example solves the same system as in the initial example, but given the Cholesky factors of A.

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       *x, *a;
    float       factor[ ] =  {1.0, -3.0, 2.0,
                              -3.0, 1.0, 1.0,
                              2.0,  1.0, 1.0};
    float       b[ ] =  {27.0, -78.0, 64.0};

                                /* Solve Ax = b  for  x  */
    x = imsl_f_lin_sol_posdef (n, a, b,
                    IMSL_FACTOR_USER, factor,
                    IMSL_SOLVE_ONLY,
                    0);

                                /* Print x */
    imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
Solution, x, of Ax = b
1           2           3
1          -4           7
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_NONPOSITIVE_MATRIX	The leading # by # submatrix of the input matrix is not positive definite.
IMSL_SINGULAR_MATRIX	The input matrix is singular.
IMSL_SINGULAR_TRI_MATRIX	The input triangular matrix is singular. The index of the first zero diagonal element is #.

lin_sol_posdef (complex)

Solves a complex Hermitian positive definite system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the Cholesky factor, L, of A such that $A = LL^H$ or computing the solution to $Ax = b$ given the Cholesky factor, L.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef` (*int* `n`, *f_complex* `a[]`, *f_complex* `b[]`, `..., 0)`

The type *d_complex* procedure is `imsl_z_lin_sol_posdef`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

f_complex `a[]` (Input)
Array of size $n \times n$ containing the matrix.

f_complex `b[]` (Input)
Array of size n containing the right-hand side.

Return Value

A pointer to the solution x of the Hermitian positive definite linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef` (*int* `n`, *f_complex* `a[]`, *f_complex* `b[]`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_RETURN_USER`, *f_complex* `x[]`,
`IMSL_FACTOR`, *f_complex* `**p_factor`,
`IMSL_FACTOR_USER`, *f_complex* `factor[]`,
`IMSL_FAC_COL_DIM`, *int* `fac_col_dim`,
`IMSL_CONDITION`, *float* `*cond`,
`IMSL_FACTOR_ONLY`,
`IMSL_SOLVE_ONLY`,
0)

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

IMSL_RETURN_USER, *f_complex* x[] (Output)
A user-allocated array of size n containing the solution x.

IMSL_FACTOR, *f_complex* **p_factor (Output)
The address of a pointer to an array of size $n \times n$ containing the LL^H factorization of A. On return, the necessary space is allocated by imsl_c_lin_sol_posdef. The lower- triangular part of this array contains L, and the upper-triangular part contains L^H. Typically, *f_complex* *p_factor is declared, and &p_factor is used as an argument.

IMSL_FACTOR_USER, *f_complex* factor[] (Input/Output)
A user-allocated array of size $n \times n$ containing the LL^H factorization of A. The lower- triangular part of this array contains L, and the upper-triangular part contains L^H. If A is not needed, a and factor can share the same storage. If IMSL_SOLVE is specified, Factor is *input*. Otherwise, it is *output*.

IMSL_FAC_COL_DIM, *int* fac_col_dim (Input)
The column dimension of the array containing the LL^H factorization of A.
Default: fac_col_dim = n

IMSL_CONDITION, *float* *cond (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix A. Do not use this option with IMSL_SOLVE_ONLY.

IMSL_FACTOR_ONLY
Compute the Cholesky factorization LL^H of A. If IMSL_FACTOR_ONLY is used, either IMSL_FACTOR or IMSL_FACTOR_USER is required. The argument b is then ignored, and the returned value of imsl_c_lin_sol_posdef is NULL.

IMSL_SOLVE_ONLY
Solve $Ax = b$ given the LL^H factorization previously computed by imsl_c_lin_sol_posdef. By default, the solution to $Ax = b$ is pointed to by imsl_c_lin_sol_posdef. If IMSL_SOLVE_ONLY is used, argument IMSL_FACTOR_USER is required and argument a is ignored.

Description

The function imsl_c_lin_sol_posdef solves a system of linear algebraic equations having a Hermitian positive definite coefficient matrix A. The function first computes the LL^H factorization of A. The solution of the linear system is then found by solving the two simpler systems, $y = L^{-1}b$ and $x = L^{-H}y$. When the solution to the linear system is required, an estimate of the L_1 condition number of A is computed using the algorithm in Dongarra et al. (1979). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x. The function imsl_c_lin_sol_posdef fails if L, the lower-triangular matrix in the factorization, has a zero diagonal element.

Examples

Example 1

A system of five linear equations with a Hermitian positive definite coefficient matrix is solved in this example. The equations are as follows:

$$2x_1 + (-1 + i)x_2 = 1 + 5i$$

$$(-1 - i)x_1 + 4x_2 + (1 + 2i)x_3 = 12 - 6i$$

$$(1 - 2i)x_2 + 10x_3 + 4ix_4 = 1 - 16i$$

$$-4ix_3 + 6x_4 + (1 + i)x_5 = -3 - 3i$$

$$(1 - i)x_4 + 9x_5 = 25 + 16i$$

```
#include <imsl.h>

main()
{
    int         n = 5;
    f_complex   *x;
    f_complex   a[] = {
         {2.0,0.0},  {-1.0,1.0},{0.0,0.0}, {0.0,0.0}, {0.0,0.0},
         {-1.0,-1.0},{4.0,0.0}, {1.0,2.0}, {0.0,0.0}, {0.0,0.0},
         {0.0,0.0},  {1.0,-2.0},{10.0,0.0},{0.0,4.0}, {0.0,0.0},
         {0.0,0.0},  {0.0,0.0}, {0.0,-4.0},{6.0,0.0}, {1.0,1.0},
         {0.0,0.0},  {0.0,0.0}, {0.0,0.0}, {1.0,-1.0},{9.0,0.0}
                     };

    f_complex   b[] = {
         {1.0,5.0}, {12.0,-6.0}, {1.0,-16.0}, {-3.0,-3.0}, {25.0,16.0}
                     };
                               /* Solve Ax = b  for  x  */
    x = imsl_c_lin_sol_posdef(n, a, b, 0);

                               /* Print x */
    imsl_c_write_matrix("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
                    Solution, x, of Ax = b
                  1                        2                        3
(         2,         1)  (         3,        -0)  (        -1,        -1)

                  4                        5
(         0,        -2)  (         3,         2)
```

Example 2

This example solves the same system of five linear equations as in the first example. This time, the LL^H factorization of A and the solution x is returned in an array allocated in the main program.

```
#include <imsl.h>

main()
```

```
{
    int          n = 5;
    f_complex    x[5], *p_factor;
    f_complex    a[] = {
        {2.0,0.0},  {-1.0,1.0},{0.0,0.0}, {0.0,0.0}, {0.0,0.0},
        {-1.0,-1.0},{4.0,0.0}, {1.0,2.0}, {0.0,0.0}, {0.0,0.0},
        {0.0,0.0},  {1.0,-2.0},{10.0,0.0},{0.0,4.0}, {0.0,0.0},
        {0.0,0.0},  {0.0,0.0}, {0.0,-4.0},{6.0,0.0}, {1.0,1.0},
        {0.0,0.0},  {0.0,0.0}, {0.0,0.0}, {1.0,-1.0},{9.0,0.0}
                      };
    f_complex    b[] = {
        {1.0,5.0}, {12.0,-6.0}, {1.0,-16.0}, {-3.0,-3.0}, {25.0,16.0}
                      };
                                /* Solve Ax = b  for  x  */
    imsl_c_lin_sol_posdef(n, a, b,
                    IMSL_RETURN_USER, x,
                    IMSL_FACTOR, &p_factor,
                    0);

                                /* Print x */
    imsl_c_write_matrix("Solution, x, of Ax = b", 1, n, x, 0);

                                /* Print Cholesky factor of A */
    imsl_c_write_matrix("Cholesky factor L, and ctrans(L), of A",
                        n, n, p_factor, 0);
}
```

Output

```
                          Solution, x, of Ax = b
                        1                       2                       3
(         2,         1)  (         3,        -0)  (        -1,        -1)

                        4                       5
(         0,        -2)  (         3,         2)
```

```
                    Cholesky factor L, and ctrans(L), of A
                         1                       2                       3
1 (     1.414,     0.000)  (    -0.707,     0.707)  (     0.000,    -0.000)
2 (    -0.707,    -0.707)  (     1.732,     0.000)  (     0.577,     1.155)
3 (     0.000,     0.000)  (     0.577,    -1.155)  (     2.887,     0.000)
4 (     0.000,     0.000)  (     0.000,     0.000)  (     0.000,    -1.386)
5 (     0.000,     0.000)  (     0.000,     0.000)  (     0.000,     0.000)

                         4                       5
1 (     0.000,    -0.000)  (     0.000,    -0.000)
2 (     0.000,    -0.000)  (     0.000,    -0.000)
3 (     0.000,     1.386)  (     0.000,    -0.000)
4 (     2.020,     0.000)  (     0.495,     0.495)
5 (     0.495,    -0.495)  (     2.917,     0.000)
```

Warning Errors

IMSL_HERMITIAN_DIAG_REAL_1	The diagonal of a Hermitian matrix must be real. Its imaginary part is set to zero.
IMSL_HERMITIAN_DIAG_REAL_2	The diagonal of a Hermitian matrix must be real. The imaginary part will be used as zero in the algorithm.
IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_NONPOSITIVE_MATRIX	The leading # by # minor matrix of the input matrix is not positive definite.
IMSL_HERMITIAN_DIAG_REAL	During the factorization the matrix has a large imaginary component on the diagonal. Thus, it cannot be positive definite.
IMSL_SINGULAR_TRI_MATRIX	The triangular matrix is singular. The index of the first zero diagonal term is #.

lin_sol_gen_band

Solves a real general band 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, solving $A^T x = b$, or computing the solution of $Ax = b$ given the LU factorization of A.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_gen_band` (*int* `n`, *float* `a[]`, *int* `nlca`, *int* `nuca`, *float* `b[]`, …, 0)

The type *double* procedure is `imsl_d_lin_sol_gen_band`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

float `a[]` (Input)
Array of size (*nlca* + *nuca* + 1) containing the $n \times n$ banded coefficient matrix in band storage mode.

int `nlca` (Input)
Number of lower codiagonals in `a`.

int `nuca` (Input)
Number of upper codiagonals in `a`.

float `b[]` (Input)
Array of size *n* containing the right-hand side.

Return Value

A pointer to the solution *x* of the linear system $Ax = b$. To release this space use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_lin_sol_gen_band (int n, float a[], int nlca,
        int nuca, float b[],
        IMSL_TRANSPOSE,
        IMSL_RETURN_USER, float x[],
        IMSL_FACTOR, int **p_pvt, float **p_factor,
        IMSL_FACTOR_USER, int pvt[], float factor[],
        IMSL_CONDITION, float *condition,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_BLOCKING_FACTOR, int block_factor,
        0)
```

Optional Arguments

`IMSL_TRANSPOSE`
Solve $A^T x = b$.
Default: Solve $Ax = b$.

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *int* `**p_pvt`, *float* `**p_factor` (Output)

`p_pvt`: The address of a pointer to an array of length *n* containing the pivot sequence for the factorization. On return, the necessary space is allocated by `imsl_f_lin_sol_gen_band`. Typically, *int* `*p_pvt` is declared and `&p_pvt` is used as an argument.

`p_factor`: The address of a pointer to an array of size $(2nlca + nuca + 1) \times n$ containing the *LU* factorization of *A* with column pivoting. On return, the necessary space is allocated by `imsl_f_lin_sol_gen_band`. Typically, *float* `*p_factor` is declared and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *int* `pvt[]`, *float* `factor[]` (Input/Output)

`pvt[]`: A user-allocated array of size *n* containing the pivot sequence for the factorization.

`factor[]`: A user-allocated array of size $(2nlca + nuca + 1) \times n$ containing the *LU* factorization of *A*. The strictly lower triangular part of this array contains information necessary to construct *L*, and the upper triangular part contains *U*. If *A* is not needed, `factor` and `a` can share the first $(nlca + nuca + 1) \times n$ locations.

These parameters are "Input" if `IMSL_SOLVE_ONLY` is specified. They are "Output" otherwise.

`IMSL_CONDITION`, *float* `*condition` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix *A*. This option cannot be used with the option `IMSL_SOLVE_ONLY`.

`IMSL_FACTOR_ONLY`
Compute the *LU* factorization of *A* with partial pivoting. If `IMSL_FACTOR_ONLY` is used, either `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required. The argument `b` is then ignored, and the returned value of `imsl_f_lin_sol_gen_band` is `NULL`.

`IMSL_SOLVE_ONLY`
Solve $Ax = b$ given the *LU* factorization previously computed by `imsl_f_lin_sol_gen_band`. By default, the solution to $Ax = b$ is pointed to by `imsl_f_lin_sol_gen_band`. If `IMSL_SOLVE_ONLY` is used, argument `IMSL_FACTOR_USER` is required and the argument `a` is ignored.

`IMSL_BLOCKING_FACTOR`, *int* `block_factor` (Input)
The blocking factor. `block_factor` must be set no larger than 32.
Default: `block_factor` = 1

Description

The function `imsl_f_lin_sol_gen_band` solves a system of linear algebraic equations with a real band matrix *A*. It first computes the *LU* factorization of *A* based on the blocked *LU* factorization algorithm given in Du Croz et al. (1990). Level-3 BLAS invocations are replaced with inline loops. The blocking factor `block_factor` has the default value of 1, but can be reset to any positive value not exceeding 32.

The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of *A* is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in *A* may produce large changes in the solution *x*. The function `imsl_f_lin_sol_gen_band` fails if *U*, the upper triangular part of the factorization, has a zero diagonal element.

Examples

Example 1

This example demonstrates the simplest use of this function by solving a system of four linear equations. This is the simplest usage of the function. The equations are as follows:

$$2x_1 - x_2 = 3$$

$$-3x_1 + x_2 - 2x_3 = 1$$

$$-x_3 + 2x_4 = 11$$

$$2x_3 + x_4 = -2$$

```
#include <imsl.h>

void main ()
{
        int         n = 4;
        int         nuca = 1;
        int         nlca = 1;
        float      *x;

                        /*  Note that a is in band storage mode */

        float 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};
        float b[] = {3.0, 1.0, 11.0, -2.0};

        x = imsl_f_lin_sol_gen_band (n, a, nlca, nuca, b, 0);

        imsl_f_write_matrix ("Solution x, of Ax = b", 1, n, x, 0);
}
```

Output

```
  Solution x, of Ax = b
 1           2           3           4
 2           1          -3           4
```

Example 2

In this example, the problem $Ax = b$ is solved using the data from the first example. This time, the factorizations are returned and the problem $A^T x = b$ is solved without recomputing LU.

```
#include <imsl.h>

void main ()
{
```

```
        int         n = 4;
        int         nuca = 1;
        int         nlca = 1;
        int        *pivot;
        float       x[4];
        float      *factor;

                        /*  Note that a is in band storage mode */

        float 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};
        float b[] = {3.0, 1.0, 11.0, -2.0};

                        /*  Solve Ax = b and return LU */

        imsl_f_lin_sol_gen_band (n, a, nlca, nuca, b,
                IMSL_FACTOR, &pivot, &factor,
                IMSL_RETURN_USER, x,
                0);

        imsl_f_write_matrix ("Solution of Ax = b", 1, n, x, 0);

                        /*  Use precomputed LU to solve trans(A)x = b */
                        /*  The original matrix A is not needed */

        imsl_f_lin_sol_gen_band (n, (float*) 0, nlca, nuca, b,
                IMSL_FACTOR_USER, pivot, factor,
                IMSL_SOLVE_ONLY,
                IMSL_TRANSPOSE,
                IMSL_RETURN_USER, x,
                0);

        imsl_f_write_matrix ("Solution of trans(A)x = b", 1, n, x, 0);
}
```

Output

```
      Solution of Ax = b
 1           2           3           4
 2           1          -3           4

   Solution of trans(A)x = b
 1           2           3           4
-6          -5          -1          -0
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_SINGULAR_MATRIX	The input matrix is singular.

lin_sol_gen_band (complex)

Solves a complex general band 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, solving $A^H x = b$, or computing the solution of $Ax = b$ given the LU factorization of A.

Synopsis

```
#include <imsl.h>

f_complex *imsl_c_lin_sol_gen_band (int n, f_complex a[], int nlca,
        int nuca, f_complex b[], ..., 0)
```

The type *double* procedure is `imsl_z_lin_sol_gen_band`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

f_complex `a[]` (Input)
Array of size $(nlca + nuca + 1) \times n$ containing the $n \times n$ banded coefficient matrix in band storage mode.

int `nlca` (Input)
Number of lower codiagonals in `a`.

int `nuca` (Input)
Number of upper codiagonals in `a`.

f_complex `b[]` (Input)
Array of size n containing the right-hand side.

Return Value

A pointer to the solution x of the linear system $Ax = b$. To release this space use `free`. If no solution was computed, `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

f_complex *imsl_c_lin_sol_gen_band (int n, f_complex a[],
        int nlca, int nuca, f_complex b[],
        IMSL_TRANSPOSE,
        IMSL_RETURN_USER, f_complex x[],
        IMSL_FACTOR, int **p_pvt, f_complex **p_factor,
        IMSL_FACTOR_USER, int pvt[], f_complex factor[],
        IMSL_CONDITION, float *condition,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        0)
```

Optional Arguments

`IMSL_TRANSPOSE`
Solve $A^H x = b$
Default: Solve $Ax = b$.

`IMSL_RETURN_USER`, *f_complex* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *int* `**p_pvt`, *f_complex* `**p_factor` (Output)

`p_pvt`: The address of a pointer to an array of length *n* containing the pivot sequence for the factorization. On return, the necessary space is allocated by `imsl_c_lin_sol_gen_band`. Typically, *int* `*p_pvt` is declared and `&p_pvt` is used as an argument.

`p_factor`: The address of a pointer to an array of size $(2nlca + nuca + 1) \times n$ containing the *LU* factorization of *A* with column pivoting. On return, the necessary space is allocated by `imsl_c_lin_sol_gen_band`. Typically, *f_complex* `*p_factor` is declared and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *int* `pvt[]`, *f_complex* `factor[]` (Input/Output)

`pvt[]`: A user-allocated array of size *n* containing the pivot sequence for the factorization.

`factor[]`: A user-allocated array of size $(2nlca + nuca + 1) \times n$ containing the *LU* factorization of *A*. If *A* is not needed, `factor` and `a` can share the first $(nlca + nuca + 1) \times n$ locations.

These parameters are "Input" if `IMSL_SOLVE_ONLY` is specified. They are "Output" otherwise.

`IMSL_CONDITION`, *float* `*condition` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix *A*. This option cannot be used with the option `IMSL_SOLVE_ONLY`.

`IMSL_FACTOR_ONLY`
Compute the *LU* factorization of *A* with partial pivoting. If `IMSL_FACTOR_ONLY` is used, either `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required. The argument `b` is then ignored, and the returned value of `imsl_c_lin_sol_gen_band` is `NULL`.

`IMSL_SOLVE_ONLY`
Solve $Ax = b$ given the *LU* factorization previously computed by `imsl_c_lin_sol_gen_band`. By default, the solution to $Ax = b$ is pointed to by `imsl_c_lin_sol_gen_band`. If `IMSL_SOLVE_ONLY` is used, argument `IMSL_FACTOR_USER` is required and argument `a` is ignored.

Description

The function `imsl_c_lin_sol_gen_band` solves a system of linear algebraic equations with a complex band matrix *A*. It first computes the *LU* factorization of *A*

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 L_∞ norm. The factorization fails if U has a zero diagonal element. This can occur only if A is singular or very close to a singular matrix.

The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x. The function `imsl_c_lin_sol_gen_band` fails if U, the upper triangular part of the factorization, has a zero diagonal element. The function `imsl_c_lin_sol_gen_band` is based on the LINPACK subroutine CGBFA; see Dongarra et al. (1979). CGBFA uses unscaled partial pivoting.

Examples

Example 1

The following linear system is solved:

$$\begin{bmatrix} -2-3i & 4 & 0 & 0 \\ 6+i & -0.5+3i & -2+2i & 0 \\ 0 & 1+i & 3-3i & -4-1 \\ 0 & 0 & 2i & 1-i \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -10-5i \\ 9.5+5.5i \\ 12-12i \\ 8i \end{bmatrix}$$

```
#include <imsl.h>

void main()
{
        int          n = 4;
        int          nlca = 1;
        int          nuca = 1;
        f_complex    *x;

                     /* Note that a is in band storage mode */

        f_complex    a[] =
                {{0.0, 0.0}, {4.0, 0.0}, {-2.0, 2.0}, {-4.0, -1.0},
                {-2.0, -3.0}, {-0.5, 3.0}, {3.0, -3.0}, {1.0, -1.0},
                {6.0, 1.0}, {1.0, 1.0}, {0.0, 2.0}, {0.0, 0.0}};

        f_complex    b[] =
                {{-10.0, -5.0}, {9.5, 5.5}, {12.0, -12.0}, {0.0, 8.0}};

        x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b, 0);

        imsl_c_write_matrix ("Solution, x, of Ax = b", n, 1, x, 0);
```

Output

```
  Solution, x, of Ax = b
1  (         3,        -0)
2  (        -1,         1)
3  (         3,         0)
4  (        -1,         1)
```

Example 2

This example solves the problem $Ax = b$ using the data from the first example. This time, the factorizations are returned and then the problem $A^H x = b$ is solved without recomputing LU.

```
#include <imsl.h>


#include <stdlib.h>
void main()
{
        int             n = 4;
        int             nlca = 1;
        int             nuca = 1;
        int            *pivot;
        f_complex      *x;
        f_complex      *factor;

                        /* Note that a is in band storage mode */

        f_complex       a[] =
                {{0.0, 0.0}, {4.0, 0.0}, {-2.0, 2.0}, {-4.0, -1.0},
                {-2.0, -3.0}, {-0.5, 3.0}, {3.0, -3.0}, {1.0, -1.0},
                {6.0, 1.0}, {1.0, 1.0}, {0.0, 2.0}, {0.0, 0.0}};
        f_complex       b[] =
                {{-10.0, -5.0}, {9.5, 5.5}, {12.0, -12.0}, {0.0, 8.0}};

                        /* Solve Ax = b and return LU */

        x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
                IMSL_FACTOR, &pivot, &factor,
                0);

        imsl_c_write_matrix ("solution of Ax = b", n, 1, x, 0);
        free (x);

                        /* Use precomputed LU to solve ctrans(A)x = b */

        x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
                IMSL_FACTOR_USER, pivot, factor,
                IMSL_TRANSPOSE,
                0);

        imsl_c_write_matrix ("solution of ctrans(A)x = b", n, 1, x, 0);
}
```

Output

```
    solution of Ax = b
1   (          3,         -0)
2   (         -1,          1)
3   (          3,          0)
4   (         -1,          1)

solution of ctrans(A)x = b
1   (       5.58,      -2.91)
2   (      -0.48,      -4.67)
3   (      -6.19,       7.15)
4   (      12.60,      30.20)
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_SINGULAR_MATRIX	The input matrix is singular.

lin_sol_posdef_band

Solves a real symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the R^TR Cholesky factorization of A, computing the solution of $Ax = b$ given the Cholesky factorization of A, or estimating the L_1 condition number of A.

Synopsis

#*include* <imsl.h>

float *imsl_f_lin_sol_posdef_band (*int* n, *float* a[], *int* ncoda, *float* b[], …, 0)

The type *double* procedure is imsl_d_lin_sol_posdef_band.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

float a[] (Input)
Array of size (*ncoda* + 1) × *n* containing the *n* × *n* positive definite band coefficient matrix in band symmetric storage mode.

int ncoda (Input)
Number of upper codiagonals of the matrix.

float `b[]` (Input)
Array of size *n* containing the right-hand side.

Return Value

A pointer to the solution *x* of the linear system $Ax = b$. To release this space use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_lin_sol_posdef_band` (*int* `n`, *float* `a[]`, *int* `ncoda`, *float* `b[]`,
`IMSL_RETURN_USER`, *float* `x[]`,
`IMSL_FACTOR`, *float* `**p_factor`,
`IMSL_FACTOR_USER`, *float* `factor[]`,
`IMSL_CONDITION`, *float* `*cond`,
`IMSL_FACTOR_ONLY`,
`IMSL_SOLVE_ONLY`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *float* `**p_factor` (Output)
The address of a pointer to an array of size $(ncoda + 1) \times n$ containing the LL^T factorization of *A*. On return, the necessary space is allocated by `imsl_f_lin_sol_posdef_band`. Typically, *float* `*p_factor` is declared and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *float* `factor[]` (Input/Output)
A user-allocated array of size $(ncoda + 1) \times n$ containing the LL^T factorization of *A* in band symmetric form. If *A* is not needed, `factor` and `a` can share the same storage.

These parameters are "Input" if `IMSL_SOLVE` is specified. They are "Output" otherwise.

`IMSL_CONDITION`, *float* `*cond` (Output)
A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix *A*. This option cannot be used with the option `IMSL_SOLVE_ONLY`.

`IMSL_FACTOR_ONLY`
Compute the LL^T factorization of *A*. If `IMSL_FACTOR_ONLY` is used, either `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required. The argument `b` is then ignored, and the returned value of `imsl_f_lin_sol_posdef_band` is `NULL`.

`IMSL_SOLVE_ONLY`
Solve $Ax = b$ given the LL^T factorization previously computed by

imsl_f_lin_sol_posdef_band. By default, the solution to $Ax = b$ is pointed to by imsl_f_lin_sol_posdef_band. If IMSL_SOLVE_ONLY is used, argument IMSL_FACTOR_USER is required and the argument a is ignored.

Description

The function imsl_f_lin_sol_posdef_band solves a system of linear algebraic equations with a real symmetric positive definite band coefficient matrix A. It computes the R^TR Cholesky factorization of A. R is an upper triangular band matrix.

When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The function imsl_f_lin_sol_posdef_band 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 function imsl_f_lin_sol_posdef_band is partially based on the LINPACK subroutines CPBFA and SPBSL; see Dongarra et al. (1979).

Example 1

Solves a system of linear equations $Ax = b$, where

$$A = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 4 & 2 & 1 \\ -1 & 2 & 7 & -1 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 6 \\ -11 \\ -11 \\ 19 \end{bmatrix}$$

```
#include <imsl.h>

void main()
{
        int         n = 4;
        int         ncoda = 2;
        float      *x;

                        /* Note that a is in band storage mode */

        float       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};
        float       b[] = {6.0, -11.0, -11.0, 19.0};

        x = imsl_f_lin_sol_posdef_band (n, a, ncoda, b, 0);

        imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
   Solution, x, of Ax = b
1           2           3           4
4          -6           2           9
```

Example 2

This example solves the same problem $Ax = b$ given in the first example. The solution is returned in user-allocated space and an estimate of $\kappa_1(A)$ is computed. Additionally, the R^TR factorization is returned. Then, knowing that $\kappa_1(A) = \|A\| \|A^{-1}\|$, the condition number is computed directly and compared to the estimate from Higham's method.

```
#include <imsl.h>

void main()
{
        int         n = 4;
        int         ncoda = 2;
        float       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};
        float       b[] = {6.0, -11.0, -11.0, 19.0};
        float       x[4];
        float       e_i[4];
        float      *factor;
        float       condition;
        float       column_norm;
        float       inverse_norm;
        int         i;
        int         j;

        imsl_f_lin_sol_posdef_band (n, a, ncoda, b,
                IMSL_FACTOR, &factor,
                IMSL_CONDITION, &condition,
                IMSL_RETURN_USER, x,
                0);

        imsl_f_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                        /*  find one norm of inverse */

        inverse_norm = 0.0;
        for (i=0; i<n; i++) {
                for (j=0; j<n; j++) e_i[j] = 0.0;
                e_i[i] = 1.0;

                        /*  determine one norm of each column of inverse */

                imsl_f_lin_sol_posdef_band (n, a, ncoda, e_i,
                        IMSL_FACTOR_USER, factor,
                        IMSL_SOLVE_ONLY,
                        IMSL_RETURN_USER, x,
                        0);
                column_norm = imsl_f_vector_norm (n, x,
                        IMSL_ONE_NORM,
```

```
                        0);

                        /*  the max of the column norms is the norm of
                            inv(A) */

                if (inverse_norm < column_norm)
                        inverse_norm = column_norm;
        }

                        /*  by observation, one norm of A is 11 */

        printf ("\nHigham's condition estimate = %f\n", condition);
        printf ("Direct condition estimate   = %f\n",
                11.0*inverse_norm);
}
```

Output

```
   Solution, x, of Ax = b
 1          2          3          4
 4         -6          2          9

Higham's condition estimate = 8.650485
Direct condition estimate   = 8.650485
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_NONPOSITIVE_MATRIX	The leading # by # submatrix of the input matrix is not positive definite.
IMSL_SINGULAR_MATRIX	The input matrix is singular.

lin_sol_posdef_band (complex)

Solves a complex Hermitian positive definite system of linear equations $Ax = b$ in band symmetric storage mode. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the $R^H R$ Cholesky factorization of A, computing the solution of $Ax = b$ given the Cholesky factorization of A, or estimating the L_1 condition number of A.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef_band` (*int* `n`, *f_complex* `a[]`, *int* `ncoda`, *f_complex* `b[]`, …, 0)

The type *double* procedure is `imsl_z_lin_sol_posdef_band`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

f_complex `a[]` (Input)
Array of size (*ncoda* + 1) × *n* containing the *n* × *n* positive definite band coefficient matrix in band symmetric storage mode.

int `ncoda` (Input)
Number of upper codiagonals of the matrix.

f_complex `b[]` (Input)
Array of size *n* containing the right-hand side.

Return Value

A pointer to the solution *x* of the linear system $Ax = b$. To release this space use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef_band` (*int* `n`, *f_complex* `a[]`, *int* `ncoda`,
f_complex `b[]`,
`IMSL_RETURN_USER`, *f_complex* `x[]`,
`IMSL_FACTOR`, *f_complex* `**p_factor`,
`IMSL_FACTOR_USER`, *f_complex* `factor[]`,
`IMSL_CONDITION`, *float* `*condition`,
`IMSL_FACTOR_ONLY`,
`IMSL_SOLVE_ONLY`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *f_complex* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_FACTOR`, *f_complex* `**p_factor` (Output)
The address of a pointer to an array of size (*ncoda* + 1) × *n* containing the $R^H R$ factorization of *A*. On return, the necessary space is allocated by `imsl_c_lin_sol_posdef_band`. Typically, *f_complex* `*p_factor` is declared and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *f_complex* `factor[]` (Input/Output)
A user-allocated array of size (*ncoda* + 1) × *n* containing the $R^H R$ factorization of *A* in band symmetric form. If *A* is not needed, `factor` and `a` can share the same storage.
These parameters are "Input" if `IMSL_SOLVE` is specified. They are "Output" otherwise.

IMSL_CONDITION, *float* *condition (Output)

A pointer to a scalar containing an estimate of the L_1 norm condition number of the matrix A. This option cannot be used with the option IMSL_SOLVE_ONLY.

IMSL_FACTOR_ONLY

Compute the $R^H R$ factorization of A. If IMSL_FACTOR_ONLY is used, either IMSL_FACTOR or IMSL_FACTOR_USER is required. The argument b is then ignored, and the returned value of imsl_c_lin_sol_posdef_band is NULL.

IMSL_SOLVE_ONLY

Solve $Ax = b$ given the $R^H R$ factorization previously computed by imsl_c_lin_sol_posdef_band. By default, the solution to $Ax = b$ is pointed to by imsl_c_lin_sol_posdef_band. If IMSL_SOLVE_ONLY is used, argument IMSL_FACTOR_USER is required and the argument a is ignored.

Description

The function imsl_c_lin_sol_posdef_band solves a system of linear algebraic equations with a real symmetric positive definite band coefficient matrix A. It computes the $R^H R$ Cholesky factorization of A. Argument R is an upper triangular band matrix.

When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The function imsl_c_lin_sol_posdef_band 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 function imsl_c_lin_sol_posdef_band is based partially on the LINPACK subroutines SPBFA and CPBSL; see Dongarra et al. (1979).

Examples

Example 1

Solve a linear system $Ax = b$ where

$$A = \begin{bmatrix} 2 & -1+i & 0 & 0 & 0 \\ -1-i & 4 & 1+2i & 0 & 0 \\ 0 & 1-2i & 10 & 4i & 0 \\ 0 & 0 & -4i & 6 & 1+i \\ 0 & 0 & 0 & 1-i & 9 \end{bmatrix}$$

```
#include <imsl.h>

void main()
```

```
{
        int         n = 5;
        int         ncoda = 1;
        f_complex  *x;

                        /* Note that a is in band storage mode */

        f_complex   a[] =
                    {{0.0, 0.0}, {-1.0, 1.0}, {1.0, 2.0}, {0.0, 4.0},
                        {1.0, 1.0},
                    {2.0, 0.0}, {4.0, 0.0}, {10.0, 0.0}, {6.0, 0.0},
                        {9.0, 0.0}};
        f_complex   b[] =
                    {{1.0, 5.0}, {12.0, -6.0}, {1.0, -16.0},{-3.0, -3.0},
                        {25.0, 16.0}};

        x = imsl_c_lin_sol_posdef_band (n, a, ncoda, b, 0);

        imsl_c_write_matrix ("Solution, x, of Ax = b", n, 1, x, 0);
}
```

Output

```
  Solution, x, of Ax = b
1  (         2,         1)
2  (         3,        -0)
3  (        -1,        -1)
4  (         0,        -2)
5  (         3,         2)
```

Example 2

This example solves the same problem $Ax = b$ given in the first example. The solution is returned in user-allocated space and an estimate of $\kappa_1(A)$ is computed. Additionally, the $R^H R$ factorization is returned. Then, knowing that $\kappa_1(A) = \|A\| \|A^{-1}\|$, the condition number is computed directly and compared to the estimate from Higham's method.

```
#include <imsl.h>
#include <math.h>

void main()
{
        int         n = 5;
        int         ncoda = 1;

                        /* Note that a is in band storage mode */

        f_complex   a[] =
                    {{0.0, 0.0}, {-1.0, 1.0}, {1.0, 2.0}, {0.0, 4.0},
                        {1.0, 1.0},
                    {2.0, 0.0}, {4.0, 0.0}, {10.0, 0.0}, {6.0, 0.0},
                        {9.0, 0.0}};
        f_complex   b[] =
                    {{1.0, 5.0}, {12.0, -6.0}, {1.0, -16.0},{-3.0, -3.0},
                        {25.0, 16.0}};
        f_complex   x[5];
        f_complex   e_i[5];
        f_complex  *factor;
        float       condition;
```

```
        float       column_norm;
        float       inverse_norm;
        int         i;
        int         j;

        imsl_c_lin_sol_posdef_band (n, a, ncoda, b,
                IMSL_FACTOR, &factor,
                IMSL_CONDITION, &condition,
                IMSL_RETURN_USER, x,
                0);

        imsl_c_write_matrix ("Solution, x, of Ax = b", 1, n, x, 0);

                        /*  Find one norm of inverse */

        inverse_norm = 0.0;
        for (i=0; i<n; i++) {
                for (j=0; j<n; j++) e_i[j] = imsl_cf_convert (0.0, 0.0);
                e_i[i] = imsl_cf_convert (1.0, 0.0);

                        /*  Determine one norm of each column of inverse */

                imsl_c_lin_sol_posdef_band (n, a, ncoda, e_i,
                        IMSL_FACTOR_USER, factor,
                        IMSL_SOLVE_ONLY,
                        IMSL_RETURN_USER, x,
                        0);

                column_norm = imsl_c_vector_norm (n, x,
                        IMSL_ONE_NORM,
                        0);

                        /*  The max of the column norms is the
                            norm of inv(A) */

                if (inverse_norm < column_norm)
                        inverse_norm = column_norm;
        }

                        /*  By observation, one norm of A is 14+sqrt(5) */

        printf ("\nHigham's condition estimate = %7.4f\n", condition);
        printf ("Direct condition estimate   = %7.4f\n",
                (14.0+sqrt(5.0))*inverse_norm);
}
```

Output

```
                        Solution, x, of Ax = b
                    1                           2                           3
(         2,         1)   (         3,        -0)   (        -1,        -1)

                    4                           5
(         0,        -2)   (         3,         2)

Higham's condition estimate = 19.3777
Direct condition estimate   = 19.3777
```

Warning Errors

IMSL_ILL_CONDITIONED	The input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is "rcond" = #. The solution might not be accurate.

Fatal Errors

IMSL_NONPOSITIVE_MATRIX	The leading # by # submatrix of the input matrix is not positive definite.
IMSL_SINGULAR_MATRIX	The input matrix is singular.

lin_sol_gen_coordinate

Solves a sparse system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include returning the LU factorization of A computing the solution of $Ax = b$ given an LU factorization setting drop tolerances, and controlling iterative refinement.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_gen_coordinate` (*int* `n`, *int* `nz`, *Imsl_f_sparse_elem* `*a`, *float* `*b`, ..., 0)

The type *double* function is `imsl_d_lin_sol_gen_coordinate`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

int `nz` (Input)
Number of nonzeros in the matrix.

Imsl_f_sparse_elem `*a` (Input)
Vector of length `nz` containing the location and value of each nonzero entry in the matrix.

float `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the sparse linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float *imsl_f_lin_sol_gen_coordinate (int n, int nz, Imsl_f_sparse_elem
        *a, float *b,
        IMSL_RETURN_SPARSE_LU_FACTOR,
                    Imsl_f_sparse_lu_factor *lu_factor,
        IMSL_SUPPLY_SPARSE_LU_FACTOR,
                    Imsl_f_sparse_lu_factor *lu_factor,
        IMSL_FREE_SPARSE_LU_FACTOR,
        IMSL_RETURN_SPARSE_LU_IN_COORD,
                    Imsl_f_sparse_elem **lu_coordinate,
                    int **row_pivots, int **col_pivots,
        IMSL_SUPPLY_SPARSE_LU_IN_COORD,
                    Imsl_f_sparse_elem *lu_coordinate, int *row_pivots,
                    int *col_pivots,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_RETURN_USER, float x[],
        IMSL_TRANSPOSE,
        IMSL_CONDITION, float *condition,
        IMSL_PIVOTING_STRATEGY, Imsl_pivot method,
        IMSL_NUM_OF_SEARCH_ROWS, int num_search_row,
        IMSL_ITERATIVE_REFINEMENT,
        IMSL_DROP_TOLERANCE, float tolerance,
        IMSL_HYBRID_FACTORIZATION, float density,
                    int order_bound,
        IMSL_STABILITY_FACTOR, float s_factor,
        IMSL_GROWTH_FACTOR_LIMIT, float gf_limit,
        IMSL_GROWTH_FACTOR, float *gf,
        IMSL_SMALLEST_PIVOT, float *small_pivot
        IMSL_NUM_NONZEROS_IN_FACTOR, int *num_nonzeros,
        IMSL_CSC_FORMAT, int *col_ptr, int *row_ind,
                    float *values,
        IMSL_MEMORY_BLOCK_SIZE, int block_size,
        0)
```

Optional Arguments

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_f_sparse_lu_factor* *lu_factor
(Output)
The address of a structure of type *Imsl_f_sparse_lu_factor*. The pointers within the structure are initialized to point to the *LU* factorization by imsl_f_lin_sol_gen_coordinate.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_sparse_lu_factor* *lu_factor (Input)
The address of a structure of type *Imsl_f_sparse_lu_factor*. This structure contains the *LU* factorization of the input matrix computed by imsl_f_lin_sol_gen_coordinate with the IMSL_RETURN_SPARSE_LU_FACTOR option.

IMSL_FREE_SPARSE_LU_FACTOR,
Before returning, free the linked list data structure containing the *LU* factorization of *A*. Use this option only if the factors are no longer required.

IMSL_RETURN_SPARSE_LU_IN_COORD,
Imsl_f_sparse_elem **lu_coordinate, *int* **row_pivots, *int* **col_pivots (Output)
The *LU* factorization is returned in coordinate form. This is more compact than the internal representation encapsulated in Imsl_f_sparse_lu. The disadvantage is that during a SOLVE_ONLY call, the internal representation of the factor must be reconstructed. If however, the factor is to be stored after the program exits, and loaded again at some subsequent run, the combination of IMSL_RETURN_LU_IN_COORD and IMSL_SUPPLY_LU_IN_COORD is probably the best choice, since the factors are in a format that is simple to store and read.

IMSL_SUPPLY_SPARSE_LU_IN_COORD,
Imsl_f_sparse_elem *lu_coordinate, *int* *row_pivots, *int* *col_pivots (Output)
Supply the *LU* factorization in coordinate form. See IMSL_RETURN_SPARSE_LU_IN_COORD for a description.

IMSL_FACTOR_ONLY,
Compute the *LU* factorization of the input matrix and return. The argument b is ignored.

IMSL_SOLVE_ONLY,
Solve $Ax = b$ given the *LU* factorization of *A*. This option requires the use of option IMSL_SUPPLY_SPARSE_LU_FACTOR or IMSL_SUPPLY_SPARSE_LU_IN_COORD.

IMSL_RETURN_USER, *float* x[] (Output)
A user-allocated array of length *n* containing the solution *x*.

IMSL_TRANSPOSE,
Solve the problem $A^T x = b$. This option can be used in conjunction with either of the options that supply the factorization.

IMSL_CONDITION, *float* *condition,
Estimate the L_1 condition number of *A* and return in the variable condition.

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method (Input)
Select the pivoting strategy by setting method to one of the following: IMSL_ROW_MARKOWITZ, IMSL_COLUMN_MARKOWITZ, or IMSL_SYMMETRIC_MARKOWITZ.
Default: IMSL_SYMMETRIC_MARKOWITZ.

IMSL_NUM_OF_SEARCH_ROWS, *int* num_search_row (Input)
The number of rows which have the least number of nonzero elements that will be searched for a pivot element.
Default: num_search_row = 3

IMSL_ITERATIVE_REFINEMENT,
Select this option if iterative refinement is desired.

IMSL_DROP_TOLERANCE, *float* tolerance (Input)
Possible fill-in is checked against tolerance. If the absolute value of the new element is less than tolerance, it will be discarded.
Default: tolerance = 0.0

IMSL_HYBRID_FACTORIZATION, *float* density, *int* order_bound,
Enable the function to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ density ≤ 1.0 and the order of the active submatrix is less than or equal to order_bound.

IMSL_STABILITY_FACTOR, *float* s_factor (Input)
The absolute value of the pivot element must be bigger than the largest element in absolute value in its row divided by s_factor.
Default: s_factor = 10.0

IMSL_GROWTH_FACTOR_LIMIT, *float* gf_limit (Input)
The computation stops if the growth factor exceeds gf_limit.
Default: gf_limit = 1.0e16

IMSL_GROWTH_FACTOR, *float* *gf (Output)
Argument gf is calculated as the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in *A*.

IMSL_SMALLEST_PIVOT, *float* *small_pivot (Output)
A pointer to the value of the pivot element of smallest magnitude that occurred during the factorization.

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros (Output)
A pointer to a scalar containing the total number of nonzeros in the factor.

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format. See the main "Introduction" chapter of this manual for a discussion of this storage scheme.

IMSL_MEMORY_BLOCKSIZE, *int* blocksize (Input)
If space must be allocated for fill-in, allocate enough space for blocksize new nonzero elements.
Default: blocksize = nz

Description

The function imsl_f_lin_sol_gen_coordinate (page 44) solves a system of linear equations $Ax = b$, where A is sparse. In its default use, it solves the so-called *one off* problem, by first performing an LU factorization of A using the improved generalized symmetric Markowitz pivoting scheme. The factor L is not stored explicitly because the saxpy operations performed during the elimination are extended to the right-hand side, along with any row interchanges. Thus, the system $Ly = b$ is solved implicitly. The

factor U is then passed to a triangular solver which computes the solution x from $Ux = y$.

If a sequence of systems $Ax = b$ are to be solved where A is unchanged, it is usually more efficient to compute the factorization once, and perform multiple forward and back solves with the various right-hand sides. In this case, the factor L is explicitly stored and a record of all row as well as column interchanges is made. The solve step then solves the two triangular systems $Ly = b$ and $Ux = y$. The user specifies either the `IMSL_RETURN_SPARSE_LU_FACTOR` or the `IMSL_RETURN_LU_IN_COORD` option to retrieve the factorization, then calls the function subsequently with different right-hand sides, passing the factorization back in using either `IMSL_SUPPLY_SPARSE_LU_FACTOR` or `IMSL_SUPPLY_SPARSE_LU_IN_COORD` in conjunction with `IMSL_SOLVE_ONLY`. If `IMSL_RETURN_SPARSE_LU_FACTOR` is used, the final call to `imsl_lin_sol_gen_coordinate` should include `IMSL_FREE_SPARSE_LU_FACTOR` to release the heap used to store L and U.

If the solution to $A^T x = b$ is required, specify the option `IMSL_TRANSPOSE`. This keyword only alters the forward elimination and back substitution so that the operations $U^T y = b$ and $L^T x = y$ are performed to obtain the solution. So, with one call to produce the factorization, solutions to both $Ax = b$ and $A^T x = b$ can be obtained.

The option `IMSL_CONDITION` is used to calculate and return an estimation of the L_1 condition number of A. The algorithm used is due to Higham. Specification of `IMSL_CONDITION` causes a complete L to be computed and stored, even if a one off problem is being solved. This is due to the fact that Higham's method requires solution to problems of the form $Az = r$ and $A^T z = r$.

The default pivoting strategy is symmetric Markowitz. If a row or column oriented problem is encountered, there may be some reduction in fill-in by selecting either `IMSL_ROW_MARKOWITZ` or `IMSL_COLUMN_MARKOWITZ`. The Markowitz strategy will search a pre-elected number of row or columns for pivot candidates. The default number is three, but this can be changed by using `IMSL_NUM_OF_SEARCH_ROWS`.

The option `IMSL_DROP_TOLERANCE` can be used to set a tolerance which can reduce fill-in. This works by preventing any new fill element which has magnitude less than the specified drop tolerance from being added to the factorization. Since this can introduce substantial error into the factorization, it is recommended that `IMSL_ITERATIVE_REFINEMENT` be used to recover more accuracy in the final solution. The trade-off is between space savings from the drop tolerance and the extra time needed in repeated solve steps needed for refinement.

The function `imsl_f_lin_sol_gen_coordinate` (page 44) provides the option of switching to a dense factorization method at some point during the decomposition. This option is enabled by choosing `IMSL_HYBRID_FACTORIZATION`. One of the two parameters required by this option, `density`, specifies a minimum density for the active submatrix before a format switch will occur. A density of 1.0 indicates complete fill-in. The other parameter, `order_bound`, places an upper bound of the order of the active submatrix which will be converted to dense format. This is used to prevent a switch from occurring too early, possibly when the $O(n^3)$ nature of the dense factoriza-

tion will cause performance degradation. Note that this option can significantly increase heap storage requirements.

Examples

Example 1

As an example, consider the following matrix:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

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.

```
#include <imsl.h>
#include <stdlib.h>
main()
{
        Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                                  1, 1, 10.0,
                                  1, 2, -3.0,
                                  1, 3, -1.0,
                                  2, 2, 15.0,
                                  3, 0, -2.0,
                                  3, 3, 10.0,
                                  3, 4, -1.0,
                                  4, 0, -1.0,
                                  4, 3, -5.0,
                                  4, 4, 1.0,
                                  4, 5, -3.0,
                                  5, 0, -1.0,
                                  5, 1, -2.0,
                                  5, 5, 6.0};

        float b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
        int n = 6;
        int nz = 15;
        float *x;

        x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b, 0);

        imsl_f_write_matrix ("solution", 1, n, x, 0);

        free (x);
}
```

Output

```
                        solution
 1           2           3           4           5           6
 1           2           3           4           5           6
```

Example 2

This examples sets $A = E(1000, 10)$. A linear system is solved and the LU factorization returned. Then a second linear system is solved, using the same coefficient matrix A just factored. Maximum absolute errors and execution time ratios are printed, showing that forward and back solves take approximately 10 percent of the computation time of a factor and solve. This ratio can vary greatly, depending on the order of the coefficient matrix, the initial number of nonzeros, and especially on the amount of fill-in produced during the elimination. Be aware that timing results are highly machine dependent.

```
#include <imsl.h>

#include <stdlib.h>
main()
{
        Imsl_f_sparse_elem          *a;
        Imsl_f_sparse_lu_factor      lu_factor;
        float                       *b;
        float                       *x;
        float                       *mod_five;
        float                       *mod_ten;
        float                        error_factor_solve;
        float                        error_solve;
        double                       time_factor_solve;
        double                       time_solve;
        int                          n = 1000;
        int                          c = 10;
        int                          i;
        int                          nz;
        int                          index;

                        /*  Get the coefficient matrix */

        a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

                        /*  Set two different predetermined solutions */

        mod_five = (float*) malloc (n*sizeof(*mod_five));
        mod_ten = (float*) malloc (n*sizeof(*mod_ten));
        for (i=0; i<n; i++) {
                mod_five[i] = (float) (i % 5);
                mod_ten[i] = (float) (i % 10);
        }

                        /*  Choose b so that x will approximate mod_five */

        b = imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, mod_five,
                0);
```

```
                        /*  Time the factor/solve */

        time_factor_solve = imsl_ctime();
        x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b,
                        IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor,
                        0);
        time_factor_solve = imsl_ctime() - time_factor_solve;

                        /*  Compute max abolute error */

        error_factor_solve = imsl_f_vector_norm (n, x,
                IMSL_SECOND_VECTOR, mod_five,
                IMSL_INF_NORM, &index,
                0);
        free (mod_five);
        free (b);
        free (x);

                        /*  Get new right hand side -- b = A * mod_ten */

        b = imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, mod_ten,
                0);

                        /*  Use the previously computed factorization
                            to solve Ax = b */

        time_solve = imsl_ctime();
        x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b,
                IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
                IMSL_SOLVE_ONLY,
                0);
        time_solve = imsl_ctime() - time_solve;
        error_solve = imsl_f_vector_norm (n, x,
                IMSL_SECOND_VECTOR, mod_ten,
                IMSL_INF_NORM, &index,
                0);
        free (mod_ten);
        free (b);
        free (x);

                        /*  Print errors and ratio of execution times */

        printf ("absolute error (factor/solve) = %e\n",
                error_factor_solve);
        printf ("absolute error (solve)        = %e\n", error_solve);
        printf ("time_solve/time_factor_solve  = %f\n",
                time_solve/time_factor_solve);
}
```

Output

```
absolute error (factor/solve) = 9.179115e-05
absolute error (solve)        = 2.160072e-04
time_solve/time_factor_solve  = 0.093750
```

Example 3

This example solves a system $Ax = b$, where $A = E$ (500, 50). Then, the same system is solved using a large drop tolerance. Finally, using the factorization just computed, the same linear system is solved with iterative refinement. Be aware that timing results are highly machine dependent.

```
#include <imsl.h>
#include <stdlib.h>

main()
{
        Imsl_f_sparse_elem      *a;
        Imsl_f_sparse_lu_factor  lu_factor;
        float                   *b;
        float                   *x;
        float                   *mod_five;
        float                    error_zero_drop_tol;
        float                    error_nonzero_drop_tol;
        float                    error_nonzero_drop_tol_IR;
        double                   time_zero_drop_tol;
        double                   time_nonzero_drop_tol;
        double                   time_nonzero_drop_tol_IR;
        int                      nz_nonzero_drop_tol;
        int                      nz_zero_drop_tol;
        int                      n = 500;
        int                      c = 50;
        int                      i;
        int                      nz;
        int                      index;

                        /*  Get the coefficient matrix */

        a = imsl_f_generate_test_coordinate (n, c, &nz, 0);
        for (i=0; i<nz; i++) a[i].val *= 0.05;

                        /*  Set a predetermined solution */

        mod_five = (float*) malloc (n*sizeof(*mod_five));
        for (i=0; i<n; i++)
                mod_five[i] = (float) (i % 5);

                        /*  Choose b so that x will approximate mod_five */

        b = imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, mod_five,
                0);

                        /*  Time the factor/solve */

        time_zero_drop_tol = imsl_ctime();
        x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b,
                IMSL_NUM_NONZEROS_IN_FACTOR, &nz_zero_drop_tol,
                0);
        time_zero_drop_tol = imsl_ctime() - time_zero_drop_tol;

                        /*  Compute max abolute error */
```

```
error_zero_drop_tol = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
free (x);

                /* Solve the same problem, with drop
                   tolerance = 0.005 */

time_nonzero_drop_tol = imsl_ctime();
x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b,
        IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor,
        IMSL_DROP_TOLERANCE, 0.005,
        IMSL_NUM_NONZEROS_IN_FACTOR, &nz_nonzero_drop_tol,
        0);
time_nonzero_drop_tol = imsl_ctime() - time_nonzero_drop_tol;

                /*  Compute max abolute error */

error_nonzero_drop_tol = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
free (x);

                /* Solve the same problem with IR, use last
                   factorization */

time_nonzero_drop_tol_IR = imsl_ctime();
x = imsl_f_lin_sol_gen_coordinate (n, nz, a, b,
        IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
        IMSL_SOLVE_ONLY,
        IMSL_ITERATIVE_REFINEMENT,
        0);
time_nonzero_drop_tol_IR = imsl_ctime() - time_nonzero_drop_tol_IR;

                /*  Compute max abolute error */

error_nonzero_drop_tol_IR = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
free (x);
free (b);

                /*  Print errors and ratio of execution times */

printf ("drop tolerance = 0.0\n");
printf ("\tabsolute error = %e\n", error_zero_drop_tol);
printf ("\tfillin         = %d\n\n", nz_zero_drop_tol);

printf ("drop tolerance = 0.005\n");
printf ("\tabsolute error = %e\n", error_nonzero_drop_tol);
printf ("\tfillin         = %d\n\n", nz_nonzero_drop_tol);

printf ("drop tolerance = 0.005 (with IR)\n");
printf ("\tabsolute error = %e\n", error_nonzero_drop_tol_IR);
printf ("\tfillin         = %d\n\n", nz_nonzero_drop_tol);
```

```
        printf ("time_nonzero_drop_tol/time_zero_drop_tol  = %f\n",
                time_nonzero_drop_tol/time_zero_drop_tol);
        printf ("time_nonzero_drop_tol_IR/time_zero_drop_tol  = %f\n",
                time_nonzero_drop_tol_IR/time_zero_drop_tol);
}
```

Output

```
drop tolerance = 0.0
        absolute error = 3.814697e-06
        fillin         = 9530

drop tolerance = 0.005
        absolute error = 2.699481e+00
        fillin         = 8656

drop tolerance = 0.005 (with IR)
        absolute error = 1.907349e-06
        fillin         = 8656

time_nonzero_drop_tol/time_zero_drop_tol  = 1.086957
time_nonzero_drop_tol_IR/time_zero_drop_tol  = 0.840580
```

Notice the absolute error when iterative refinement is not used. Also note that iterative refinement itself can be quite expensive. In this case, for example, the IR solve took approximately as much time as the factorization. For this problem the use of a drop high drop tolerance and iterative refinement was able to reduce fill-in by 10 percent at a time cost double that of the default usage. In tight memory situations, such a trade-off may be acceptable. Users should be aware that a drop tolerance can be chosen large enough, introducing large errors into *LU*, to prevent convergence of iterative refinement.

lin_sol_gen_coordinate (complex)

Solves a system of linear equations $Ax = b$, with sparse complex coefficient matrix A. Using optional arguments, any of several related computations can be performed. These extra tasks include returning the LU factorization of A, computing the solution of $Ax = b$ given an LU factorization, setting drop tolerances, and controlling iterative refinement.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_gen_coordinate` (*int* `n`, *int* `nz`,
Imsl_c_sparse_elem `*a`, *f_complex* `*b`, ..., 0)

The type *double* function is `imsl_z_lin_sol_gen_coordinate`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

int `nz` (Input)
Number of nonzeros in the matrix.

Imsl_c_sparse_elem `*a` (Input)
Vector of length `nz` containing the location and value of each nonzero entry in the matrix.

f_complex `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the sparse linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
f_complex *imsl_c_lin_sol_gen_coordinate (int n, int nz,
        Imsl_c_sparse_elem *a, f_complex *b,
        IMSL_RETURN_SPARSE_LU_FACTOR,
                    Imsl_c_sparse_lu_factor *lu_factor,
        IMSL_SUPPLY_SPARSE_LU_FACTOR,
                    Imsl_c_sparse_lu_factor *lu_factor,
        IMSL_FREE_SPARSE_LU_FACTOR,
        IMSL_RETURN_SPARSE_LU_IN_COORD,
                    Imsl_c_sparse_elem **lu_coordinate,
                    int **row_pivots, int **col_pivots,
        IMSL_SUPPLY_SPARSE_LU_IN_COORD,
                    Imsl_c_sparse_elem *lu_coordinate, int *row_pivots,
                    int *col_pivots,
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_RETURN_USER, f_complex x[],
        IMSL_TRANSPOSE,
        IMSL_CONDITION, float *condition,
        IMSL_PIVOTING_STRATEGY, Imsl_pivot method,
        IMSL_NUM_OF_SEARCH_ROWS, int num_search_row,
        IMSL_ITERATIVE_REFINEMENT,
        IMSL_DROP_TOLERANCE, float tolerance,
        IMSL_HYBRID_FACTORIZATION, float density,
                    int order_bound,
        IMSL_GROWTH_FACTOR_LIMIT, float gf_limit,
        IMSL_GROWTH_FACTOR, float *gf,
        IMSL_SMALLEST_PIVOT, float *small_pivot
        IMSL_NUM_NONZEROS_IN_FACTOR, int *num_nonzeros,
```

```
        IMSL_CSC_FORMAT, int *col_ptr, int *row_ind,
                    f_complex *values,
        IMSL_MEMORY_BLOCK_SIZE, int block_size,
        0)
```

Optional Arguments

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_c_sparse_lu_factor* *lu_factor
(Output)
The address of a structure of type *Imsl_c_sparse_lu_factor*. The pointers within the structure are initialized to point to the *LU* factorization by imsl_c_lin_sol_gen_coordinate.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_c_sparse_lu_factor* *lu_factor
(Input)
The address of a structure of type *Imsl_c_sparse_lu_factor*. This structure contains the *LU* factorization of the input matrix computed by imsl_c_lin_sol_gen_coordinate with the IMSL_RETURN_SPARSE_LU_FACTOR option.

IMSL_FREE_SPARSE_LU_FACTOR,
Before returning, free the linked list data structure containing the *LU* factorization of *A*. Use this option only if the factors are no longer required.

IMSL_RETURN_SPARSE_LU_IN_COORD,
Imsl_c_sparse_elem **lu_coordinate, *int* **row_pivots, *int* **col_pivots (Output)
The *LU* factorization is returned in coordinate form. This is more compact than the internal representation encapsulated in Imsl_c_sparse_lu. The disadvantage is that during a SOLVE_ONLY call, the internal representation of the factor must be reconstructed. If however, the factor is to be stored after the program exits, and loaded again at some subsequent run, the combination of IMSL_RETURN_LU_IN_COORD and IMSL_SUPPLY_LU_IN_COORD is probably the best choice, since the factors are in a format that is simple to store and read.

IMSL_SUPPLY_SPARSE_LU_IN_COORD, *Imsl_c_sparse_elem* *lu_coordinate, *int* *row_pivots, *int* *col_pivots (Output)
Supply the *LU* factorization in coordinate form. See IMSL_RETURN_SPARSE_LU_IN_COORD for a description.

IMSL_FACTOR_ONLY,
Compute the *LU* factorization of the input matrix and return. The argument b is ignored.

IMSL_SOLVE_ONLY,
Solve $Ax = b$ given the *LU* factorization of *A*. This option requires the use of option IMSL_SUPPLY_SPARSE_LU_FACTOR or IMSL_SUPPLY_SPARSE_LU_IN_COORD.

IMSL_RETURN_USER, *f_complex* x[] (Output)
A user-allocated array of length *n* containing the solution *x*.

IMSL_TRANSPOSE,
Solve the problem $A^T x = b$. This option can be used in conjunction with either of the options that supply the factorization.

IMSL_CONDITION, *float* *condition,
Estimate the L_1 condition number of *A* and return in the variable condition.

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method (Input)
Select the pivoting strategy by setting method to one of the following: IMSL_ROW_MARKOWITZ, IMSL_COLUMN_MARKOWITZ, or IMSL_SYMMETRIC_MARKOWITZ.
Default: IMSL_SYMMETRIC_MARKOWITZ.

IMSL_NUM_OF_SEARCH_ROWS, *int* num_search_row (Input)
The number of rows which have the least number of nonzero elements that will be searched for a pivot element.
Default: num_search_row = 3

IMSL_ITERATIVE_REFINEMENT,
Select this option if iterative refinement is desired.

IMSL_DROP_TOLERANCE, *float* tolerance (Input)
Possible fill-in is checked against tolerance. If the absolute value of the new element is less than tolerance, it will be discarded.
Default: tolerance = 0.0

IMSL_HYBRID_FACTORIZATION, *float* density, *int* order_bound,
Enable the code to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ density ≤ 1.0 and the order of the active submatrix is less than or equal to order_bound.

IMSL_GROWTH_FACTOR_LIMIT, *float* gf_limit (Input)
The computation stops if the growth factor exceeds gf_limit.
Default: gf_limit = 1.e16

IMSL_GROWTH_FACTOR, *float* *gf (Output)
gf is calculated as the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in *A*.

IMSL_SMALLEST_PIVOT, *float* *small_pivot (Output)
A pointer to the value of the pivot element of smallest magnitude.

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros (Output)
A pointer to a scalar containing the total number of nonzeros in the factor.

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *f_complex* *values (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format. See the main "Introduction" chapter at the beginning of this manual for a discussion of this storage scheme.

IMSL_FACTOR_RESIZE_INCREMENT, *int* increment (Input)

Supply the number of nonzeros which will be added to the factor if current allocations are inadequate.

Default: increment = nz

Description

The function imsl_c_lin_sol_gen_coordinate (page 44) solves a system of linear equations $Ax = b$, where A is sparse. In its default use, it solves the so-called *one off* problem, by first performing an LU factorization of A using the improved generalized symmetric Markowitz pivoting scheme. The factor L is not stored explicitly because the saxpy operations performed during the elimination are extended to the right-hand side, along with any row interchanges. Thus, the system $Ly = b$ is solved implicitly. The factor U is then passed to a triangular solver which computes the solution x from $Ux = y$.

If a sequence of systems $Ax = b$ are to be solved where A is unchanged, it is usually more efficient to compute the factorization once, and perform multiple forward and back solves with the various right-hand sides. In this case the factor L is explicitly stored and a record of all row as well as column interchanges is made. The solve step then solves the two triangular systems $Ly = b$ and $Ux = y$. The user specifies either the IMSL_RETURN_SPARSE_LU_FACTOR or the IMSL_RETURN_LU_IN_COORD option to retrieve the factorization, then calls the function subsequently with different right-hand sides, passing the factorization back in using either IMSL_SUPPLY_SPARSE_LU_FACTOR or IMSL_SUPPLY_SPARSE_LU_IN_COORD in conjunction with IMSL_SOLVE_ONLY. If IMSL_RETURN_SPARSE_LU_FACTOR is used, the final call to imsl_lin_sol_gen_coordinate should include IMSL_FREE_SPARSE_LU_FACTOR to release the heap used to store L and U.

If the solution to $A^T x = b$ is required, specify the option IMSL_TRANSPOSE. This keyword only alters the forward elimination and back substitution so that the operations $U^T y = b$ and $L^T x = y$ are performed to obtain the solution. So, with one call to produce the factorization, solutions to both $Ax = b$ and $A^T x = b$ can be obtained.

The option IMSL_CONDITION is used to calculate and return an estimation of the L_1 condition number of A. The algorithm used is due to Higham. Specification of IMSL_CONDITION causes a complete L to be computed and stored, even if a one off problem is being solved. This is due to the fact that Higham's method requires solution to problems of the form $Az = r$ and $A^T z = r$.

The default pivoting strategy is symmetric Markowitz. If a row or column oriented problem is encountered, there may be some reduction in fill-in by selecting either IMSL_ROW_MARKOWITZ or IMSL_COLUMN_MARKOWITZ. The Markowitz strategy will search a pre-elected number of row or columns for pivot candidates. The default number is three, by this can be changed by using IMSL_NUM_OF_SEARCH_ROWS.

The option IMSL_DROP_TOLERANCE can be used to set a tolerance which can reduce fill-in. This works by preventing any new fill element which has magnitude less than the specified drop tolerance from being added to the factorization. Since this can introduce substantial error into the factorization, it is recommended that

IMSL_ITERATIVE_REFINEMENT be used to recover more accuracy in the final solution. The trade-off is between space savings from the drop tolerance and the extra time needed in repeated solve steps needed for refinement.

The function imsl_c_lin_sol_gen_coordinate provides the option of switching to a dense factorization method at some point during the decomposition. This option is enabled by choosing IMSL_HYBRID_FACTORIZATION. One of the two parameters required by this option, density, specifies a minimum density for the active submatrix before a format switch will occur. A density of 1.0 indicates complete fill-in. The other parameter, order_bound, places an upper bound of the order of the active submatrix which will be converted to dense format. This is used to prevent a switch from occurring too early, possibly when the $O(n^3)$ nature of the dense factorization will cause performance degradation. Note that this option can significantly increase heap storage requirements.

Examples

Example 1

As an example, consider the following matrix:

$$A = \begin{bmatrix} 10+7i & 0 & 0 & 0 & 0 & 0 \\ 0 & 3+2i & -3 & -1+2i & 0 & 0 \\ 0 & 0 & 4+2i & 0 & 0 & 0 \\ -2-4i & 0 & 0 & 1+6i & -1+3i & 0 \\ -5+4i & 0 & 0 & -5 & 12+2i & -7+7i \\ -1+12i & -2+8i & 0 & 0 & 0 & 3+7i \end{bmatrix}$$

Let

$$x^T = (1 + i, 2 + 2i, 3 + 3i, 4 + 4i, 5 + 5i, 6 + 6i)$$

so that

$$Ax = (3 + 17i, -19 + 5i, 6 + 18i, -38 + 32i, -63 + 49i, -57 + 83i)^T$$

```
#include <imsl.h>
#include <stdlib.h>

main()
{
        static Imsl_c_sparse_elem a[] = {0, 0, {10.0, 7.0},
                                         1, 1, {3.0, 2.0},
                                         1, 2, {-3.0, 0.0},
                                         1, 3, {-1.0, 2.0},
                                         2, 2, {4.0, 2.0},
                                         3, 0, {-2.0, -4.0},
                                         3, 3, {1.0, 6.0},
```

```
                                3, 4, {-1.0, 3.0},
                                4, 0, {-5.0, 4.0},
                                4, 3, {-5.0, 0.0},
                                4, 4, {12.0, 2.0},
                                4, 5, {-7.0, 7.0},
                                5, 0, {-1.0, 12.0},
                                5, 1, {-2.0, 8.0},
                                5, 5, {3.0, 7.0}};

        static f_complex 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}};
        int              n = 6;
        int              nz = 15;
        f_complex       *x;

        x = imsl_c_lin_sol_gen_coordinate (n, nz, a, b, 0);

        imsl_c_write_matrix ("solution", n, 1, x, 0);

        free (x);
}
```

Output

```
          solution
1   (         1,         1)
2   (         2,         2)
3   (         3,         3)
4   (         4,         4)
5   (         5,         5)
6   (         6,         6)
```

Example 2

This examples sets $A = E(1000, 10)$. A linear system is solved and the LU factorization returned. Then a second linear system is solved using the same coefficient matrix A just factored. Maximum absolute errors and execution time ratios are printed showing that forward and back solves take a small percentage of the computation time of a factor and solve. This ratio can vary greatly, depending on the order of the coefficient matrix, the initial number of nonzeros, and especially on the amount of fill-in produced during the elimination. Be aware that timing results are highly machine dependent.

```
#include <imsl.h>
#include <stdlib.h>
main()
{
        Imsl_c_sparse_elem      *a;
        Imsl_c_sparse_lu_factor  lu_factor;
        f_complex               *b;
        f_complex               *x;
        f_complex               *mod_five;
        f_complex               *mod_ten;
        float                    error_factor_solve;
        float                    error_solve;
        double                   time_factor_solve;
        double                   time_solve;
        int                      n = 1000;
```

```
int                     c = 10;
int                     i;
int                     nz;
int                     index;

                /*  Get the coefficient matrix */

a = imsl_c_generate_test_coordinate (n, c, &nz, 0);

                /*  Set two different predetermined solutions */

mod_five = (f_complex*) malloc (n*sizeof(*mod_five));
mod_ten = (f_complex*) malloc (n*sizeof(*mod_ten));
for (i=0; i<n; i++) {
        mod_five[i] = imsl_cf_convert ((float)(i % 5), 0.0);
        mod_ten[i] = imsl_cf_convert ((float)(i % 10), 0.0);
}

                /*  Choose b so that x will approximate mod_five */

b = imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_five,
        0);

                /*  Time the factor/solve */

time_factor_solve = imsl_ctime();
x = imsl_c_lin_sol_gen_coordinate (n, nz, a, b,
                IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor,
                0);
time_factor_solve = imsl_ctime() - time_factor_solve;

                /*  Compute max abolute error */

error_factor_solve = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
free (b);
free (x);

                /*  Get new right hand side -- b = A * mod_ten */

b = imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_ten,
        0);

                /*  Use the previously computed factorization
                    to solve Ax = b */

time_solve = imsl_ctime();
x = imsl_c_lin_sol_gen_coordinate (n, nz, a, b,
        IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
        IMSL_SOLVE_ONLY,
        0);
time_solve = imsl_ctime() - time_solve;
error_solve = imsl_c_vector_norm (n, x,
```

```
                IMSL_SECOND_VECTOR, mod_ten,
                IMSL_INF_NORM, &index,
                0);
        free (b);
        free (x);

                        /*  Print errors and ratio of execution times */

        printf ("absolute error (factor/solve) = %e\n",
                error_factor_solve);
        printf ("absolute error (solve)        = %e\n", error_solve);
        printf ("time_solve/time_factor_solve  = %f\n",
                time_solve/time_factor_solve);
}
```

Output

```
absolute error (factor/solve) = 2.389053e-06
absolute error (solve)        = 7.656095e-06
time_solve/time_factor_solve  = 0.070313
```

lin_sol_posdef_coordinate

Solves a sparse real symmetric positive definite system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include returning the symbolic factorization of A, returning the numeric factorization of A, and computing the solution of $Ax = b$ given either the symbolic or numeric factorizations.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_posdef_coordinate` (*int* `n`, *int* `nz`,
Imsl_f_sparse_elem `*a`, *float* `*b`, ..., 0)

The type *double* function is `imsl_d_lin_sol_posdef_coordinate`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

int `nz` (Input)
Number of nonzeros in lower triangle of the matrix.

Imsl_f_sparse_elem `*a` (Input)
Vector of length `nz` containing the location and value of each nonzero entry in the lower triangle of the matrix.

float `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the sparse symmetric positive definite linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_lin_sol_posdef_coordinate (int n, int nz,
        Imsl_f_sparse_elem *a, float *b,
        IMSL_RETURN_SYMBOLIC_FACTOR,
                    Imsl_symbolic_factor *sym_factor,
        IMSL_SUPPLY_SYMBOLIC_FACTOR,
                    Imsl_symbolic_factor *sym_factor,
        IMSL_SYMBOLIC_FACTOR_ONLY,
        IMSL_RETURN_NUMERIC_FACTOR,
                    Imsl_f_numeric_factor *num_factor,

        IMSL_SUPPLY_NUMERIC_FACTOR,
                    Imsl_f_numeric_factor *num_factor,
        IMSL_NUMERIC_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_MULTIFRONTAL_FACTORIZATION,
        IMSL_RETURN_USER, float x[],
        IMSL_SMALLEST_DIAGONAL_ELEMENT, float *small_element,
        IMSL_LARGEST_DIAGONAL_ELEMENT, float *largest_element,
        IMSL_NUM_NONZEROS_IN_FACTOR, int *num_nonzeros,
        IMSL_CSC_FORMAT, int *col_ptr, int *row_ind,
                    float *values,
        0)
```

Optional Arguments

`IMSL_RETURN_SYMBOLIC_FACTOR`, *Imsl_symbolic_factor* `*sym_factor` (Output)
A pointer to a structure of type *Imsl_symbolic_factor* containing, on return, the symbolic factorization of the input matrix.

`IMSL_SUPPLY_SYMBOLIC_FACTOR`, *Imsl_symbolic_factor* `*sym_factor` (Input)
A pointer to a structure of type *Imsl_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by `imsl_f_lin_sol_posdef_coordinate` with the `IMSL_RETURN_SYMBOLIC_FACTOR` option.

`IMSL_SYMBOLIC_FACTOR_ONLY`,
Compute the symbolic factorization of the input matrix and return. The argument `b` is ignored.

`IMSL_RETURN_NUMERIC_FACTOR`, *Imsl_f_numeric_factor* `*num_factor` (Output)
A pointer to a structure of type *Imsl_f_numeric_factor* containing, on return, the numeric factorization of the input matrix.

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor (Input)
A pointer to a structure of type *Imsl_f_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by imsl_f_lin_sol_posdef_coordinate with the IMSL_RETURN_NUMERIC_FACTOR option.

IMSL_NUMERIC_FACTOR_ONLY,
Compute the numeric factorization of the input matrix and return. The argument b is ignored.

IMSL_SOLVE_ONLY,
Solve $Ax = b$ given the numeric or symbolic factorization of A. This option requires the use of either IMSL_SUPPLY_NUMERIC_FACTOR or IMSL_SUPPLY_SYMBOLIC_FACTOR.

IMSL_MULTIFRONTAL_FACTORIZATION,
Perform the numeric factorization using a multifrontal technique. By default, a standard factorization is computed based on a sparse compressed storage scheme.

IMSL_RETURN_USER, *float* x[] (Output)
A user-allocated array of length n containing the solution x.

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *small_element (Output)
A pointer to a scalar containing the smallest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to imsl_f_lin_sol_posdef_coordinate.

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *large_element (Output)
A pointer to a scalar containing the largest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to imsl_f_lin_sol_posdef_coordinate.

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros (Output)
A pointer to a scalar containing the total number of nonzeros in the factor.

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format. See the main "Introduction" main at the beginning of this manual for a discussion of this storage scheme.

Description

The function imsl_f_lin_sol_posdef_coordinate solves a system of linear algebraic equations having a sparse symmetric positive definite coefficient matrix A. In this function's default usage, a symbolic factorization of a permutation of the coefficient matrix is computed first. Then a numerical factorization is performed. The solution of the linear system is then found using the numeric factor.

The symbolic factorization step of the computation consists of determining a minimum degree ordering and then setting up a sparse data structure for the Cholesky factor, *L*. This step only requires the "pattern" of the sparse coefficient matrix, i.e., the locations of the nonzeros elements but not any of the elements themselves. Thus, the `val` field in the `Imsl_f_sparse_elem` structure is ignored. If an application generates different sparse symmetric positive definite coefficient matrices that all have the same sparsity pattern, then by using `IMSL_RETURN_SYMBOLIC_FACTOR` and `IMSL_SUPPLY_SYMBOLIC_FACTOR`, the symbolic factorization need only be computed once.

Given the sparse data structure for the Cholesky factor *L*, as supplied by the symbolic factor, the numeric factorization produces the entries in *L* so that

$$PAP^T = LL^T$$

Here *P* is the permutation matrix determined by the minimum degree ordering.

The numerical factorization can be carried out in one of two ways. By default, the standard factorization is performed based on a sparse compressed storage scheme. This is fully described in George and Liu (1981). Optionally, a multifrontal technique can be used. The multifrontal method requires more storage but will be faster in certain cases. The multifrontal factorization is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989).

If an application requires that several linear systems be solved where the coefficient matrix is the same but the right-hand sides change, the options `IMSL_RETURN_NUMERIC_FACTOR` and `IMSL_SUPPLY_NUMERIC_FACTOR` can be used to precompute the Cholesky factor. Then the `IMSL_SOLVE_ONLY` option can be used to efficiently solve all subsequent systems.

Given the numeric factorization, the solution *x* is obtained by the following calculations:

$$Ly_1 = Pb$$

$$L^T y_2 = y_1$$

$$x = P^T y_2$$

The permutation information, *P*, is carried in the numeric factor structure.

Examples

Example 1

As an example consider the 5 × 5 coefficient matrix:

$$a = \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

Let $x^T = (5, 4, 3, 2, 1)$ so that $Ax = (55, 83, 103, 97, 82)^T$. The number of nonzeros in the lower triangle of A is `nz` = 10. The sparse coordinate form for the lower triangle is given by the following:

row	0	1	2	2	3	3	4	4	4	4
col	0	1	0	2	2	3	0	1	3	4
val	10	20	1	30	4	40	2	3	5	50

Since this representation is not unique, an equivalent form would be as follows:

row	3	4	4	4	0	1	2	2	3	4
col	3	0	1	3	0	1	0	2	2	4
val	40	2	3	5	10	20	1	30	4	50

```
#include <imsl.h>
#include <stdlib.h>
main()
{
        Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                                  1, 1, 20.0,
                                  2, 0, 1.0,
                                  2, 2, 30.0,
                                  3, 2, 4.0,
                                  3, 3, 40.0,
                                  4, 0, 2.0,
                                  4, 1, 3.0,
                                  4, 3, 5.0,
                                  4, 4, 50.0};

        float   b[] = {55.0, 83.0, 103.0, 97.0, 82.0};
        int     n = 5;
        int     nz = 10;
        float  *x;
         x = imsl_f_lin_sol_posdef_coordinate (n, nz, a, b, 0);

        imsl_f_write_matrix ("solution", 1, n, x, 0);

        free (x);
}
```

Output

```
                  solution
 1           2           3           4           5
 5           4           3           2           1
```

Example 2

In this example, set $A = E(2500, 50)$. Then solve the system $Ax = b_1$ and return the numeric factorization resulting from that call. Then solve the system $Ax = b_2$ using the numeric factorization just computed. The ratio of execution time is printed. Be aware that timing results are highly machine dependent.

```
#include <imsl.h>

main()
{
        Imsl_f_sparse_elem     *a;
        Imsl_f_numeric_factor   numeric_factor;
        float                  *b_1;
        float                  *b_2;
        float                  *x_1;
        float                  *x_2;
        int                     n;
        int                     ic;
        int                     nz;
        double                  time_1;
        double                  time_2;

        ic = 50;
        n = ic*ic;

                        /*  Generate two right hand sides */

        b_1 = imsl_f_random_uniform (n*sizeof(*b_1), 0);
        b_2 = imsl_f_random_uniform (n*sizeof(*b_2), 0);

                        /*  Build coefficient matrix a */

        a = imsl_f_generate_test_coordinate (n, ic, &nz,
                IMSL_SYMMETRIC_STORAGE,
                0);

                        /*  Now solve Ax_1 = b_1 and return the numeric
                            factorization  */

        time_1 = imsl_ctime ();
        x_1 = imsl_f_lin_sol_posdef_coordinate (n, nz, a, b_1,
                IMSL_RETURN_NUMERIC_FACTOR, &numeric_factor,
                0);
        time_1 = imsl_ctime () - time_1;

                        /*  Now solve Ax_2 = b_2 given the numeric
                            factorization  */

        time_2 = imsl_ctime ();
        x_2 = imsl_f_lin_sol_posdef_coordinate (n, nz, a, b_2,
                IMSL_SUPPLY_NUMERIC_FACTOR, &numeric_factor,
                IMSL_SOLVE_ONLY,
                0);
        time_2 = imsl_ctime () - time_2;

        printf("time_2/time_1 = %lf\n", time_2/time_1);
}
```

Output

```
time_2/time_1 = 0.037037
```

lin_sol_posdef_coordinate (complex)

Solves a sparse Hermitian positive definite system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include returning the symbolic factorization of A, returning the numeric factorization of A, and computing the solution of $Ax = b$ given either the symbolic or numeric factorizations.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef_coordinate` (*int* `n`, *int* `nz`,
Imsl_c_sparse_elem `*a`, *f_complex* `*b`, ..., 0)

The type *d_complex* function is `imsl_z_lin_sol_posdef_coordinate`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

int `nz` (Input)
Number of nonzeros in the lower triangle of the matrix.

Imsl_c_sparse_elem `*a` (Input)
Vector of length `nz` containing the location and value of each nonzero entry in lower triangle of the matrix.

f_complex `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the sparse Hermitian positive definite linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_lin_sol_posdef_coordinate` (*int* `n`,
int `nz`, *Imsl_c_sparse_elem* `*a`, *f_complex* `*b`,
`IMSL_RETURN_SYMBOLIC_FACTOR`,
Imsl_symbolic_factor `*sym_factor`,
`IMSL_SUPPLY_SYMBOLIC_FACTOR`,
Imsl_symbolic_factor `*sym_factor`,
`IMSL_SYMBOLIC_FACTOR_ONLY`,

```
        IMSL_RETURN_NUMERIC_FACTOR,
                    Imsl_c_numeric_factor *num_factor,
        IMSL_SUPPLY_NUMERIC_FACTOR,
                    Imsl_c_numeric_factor *num_factor,
        IMSL_NUMERIC_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        IMSL_MULTIFRONTAL_FACTORIZATION,
        IMSL_RETURN_USER, f_complex x[],
        IMSL_SMALLEST_DIAGONAL_ELEMENT, float *small_element,
        IMSL_LARGEST_DIAGONAL_ELEMENT, float *largest_element,
        IMSL_NUM_NONZEROS_IN_FACTOR, int *num_nonzeros,
        IMSL_CSC_FORMAT, int *col_ptr, int *row_ind,
                    float *values,
        0)
```

Optional Arguments

`IMSL_RETURN_SYMBOLIC_FACTOR`, *Imsl_symbolic_factor* `*sym_factor` (Output)
A pointer to a structure of type *Imsl_symbolic_factor* containing, on return, the symbolic factorization of the input matrix.

`IMSL_SUPPLY_SYMBOLIC_FACTOR`, *Imsl_symbolic_factor* `*sym_factor` (Input)
A pointer to a structure of type *Imsl_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by `imsl_c_lin_sol_posdef_coordinate` with the `IMSL_RETURN_SYMBOLIC_FACTOR` option.

`IMSL_SYMBOLIC_FACTOR_ONLY`,
Compute the symbolic factorization of the input matrix and return. The argument `b` is ignored.

`IMSL_RETURN_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Output)
A pointer to a structure of type *Imsl_c_numeric_factor* containing, on return, the numeric factorization of the input matrix.

`IMSL_SUPPLY_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Input)
A pointer to a structure of type *Imsl_c_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by `imsl_c_lin_sol_posdef_coordinate` with the `IMSL_RETURN_NUMERIC_FACTOR` option.

`IMSL_NUMERIC_FACTOR_ONLY`,
Compute the numeric factorization of the input matrix and return. The argument `b` is ignored.

`IMSL_SOLVE_ONLY`,
Solve $Ax = b$ given the numeric or symbolic factorization of A. This option requires the use of either `IMSL_SUPPLY_NUMERIC_FACTOR` or `IMSL_SUPPLY_SYMBOLIC_FACTOR`.

IMSL_MULTIFRONTAL_FACTORIZATION,
Perform the numeric factorization using a multifrontal technique. By default a standard factorization is computed based on a sparse compressed storage scheme.

IMSL_RETURN_USER, *f_complex* x[] (Output)
A user-allocated array of length *n* containing the solution *x*.

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *small_element (Output)
A pointer to a scalar containing the smallest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to imsl_c_lin_sol_posdef_coordinate.

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *large_element (Output)
A pointer to a scalar containing the largest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to imsl_c_lin_sol_posdef_coordinate.

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros (Output)
A pointer to a scalar containing the total number of nonzeros in the factor.

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format. See the "Introduction" section at the beginnning of this manual for a discussion of this storage scheme.

Description

The function imsl_c_lin_sol_posdef_coordinate solves a system of linear algebraic equations having a sparse Hermitian positive definite coefficient matrix A. In this function's default use, a symbolic factorization of a permutation of the coefficient matrix is computed first. Then a numerical factorization is performed. The solution of the linear system is then found using the numeric factor.

The symbolic factorization step of the computation consists of determining a minimum degree ordering and then setting up a sparse data structure for the Cholesky factor, L. This step only requires the "pattern" of the sparse coefficient matrix, i.e., the locations of the nonzeros elements but not any of the elements themselves. Thus, the val field in the Imsl_c_sparse_elem structure is ignored. If an application generates different sparse Hermitian positive definite coefficient matrices that all have the same sparsity pattern, then by using IMSL_RETURN_SYMBOLIC_FACTOR and IMSL_SUPPLY_SYMBOLIC_FACTOR, the symbolic factorization need only be computed once.

Given the sparse data structure for the Cholesky factor L, as supplied by the symbolic factor, the numeric factorization produces the entries in L so that

$$PAP^T = LL^T$$

Here P is the permutation matrix determined by the minimum degree ordering.

The numerical factorization can be carried out in one of two ways. By default, the standard factorization is performed based on a sparse compressed storage scheme. This is fully described in George and Liu (1981). Optionally, a multifrontal technique can be used. The multifrontal method requires more storage but will be faster in certain cases. The multifrontal factorization is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989).

If an application requires that several linear systems be solved where the coefficient matrix is the same but the right-hand sides change, the options `IMSL_RETURN_NUMERIC_FACTOR` and `IMSL_SUPPLY_NUMERIC_FACTOR` can be used to precompute the Cholesky factor. Then the `IMSL_SOLVE_ONLY` option can be used to efficiently solve all subsequent systems.

Given the numeric factorization, the solution x is obtained by the following calculations:

$$Ly_1 = Pb$$

$$L^T y_2 = y_1$$

$$x = P^T y_2$$

The permutation information, P, is carried in the numeric factor structure.

Examples

Example 1

As a simple example of default use, consider the following Hermitian positive definite matrix

$$A = \begin{bmatrix} 2 & -1+i & 0 \\ -1-i & 4 & 1+2i \\ 0 & 1-2i & 10 \end{bmatrix}$$

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 is `nz` = 5.

```
#include <imsl.h>

main()
{
        Imsl_c_sparse_elem a[] = {0, 0, {2.0, 0.0},
                                  1, 1, {4.0, 0.0},
                                  2, 2, {10.0, 0.0},
```

```
                                1, 0, {-1.0, -1.0},
                                2, 1, {1.0, -2.0}};

        f_complex   b[] = {{-2.0, 2.0}, {5.0, 15.0}, {36.0, 28.0}};
        int         n = 3;
        int         nz = 5;
        f_complex  *x;

        x = imsl_c_lin_sol_posdef_coordinate (n, nz, a, b, 0);

        imsl_c_write_matrix ("Solution, x, of Ax = b", n, 1, x, 0);

        free (x);
}
```

Output

```
  Solution, x, of Ax = b
1  (         1,         1)
2  (         2,         2)
3  (         3,         3)
```

Example 2

Set $A = E(2500, 50)$. Then solve the system $Ax = b_1$ and return the numeric factorization resulting from that call. Then solve the system $Ax = b_2$ using the numeric factorization just computed. Absolute errors and execution time are printed.

```
#include <imsl.h>

main()
{
        Imsl_c_sparse_elem     *a;
        Imsl_c_numeric_factor   numeric_factor;
        f_complex               b_1[2500];
        f_complex               b_2[2500];
        f_complex              *x_1;
        f_complex              *x_2;
        int                     n;
        int                     ic;
        int                     nz;
        int                     i;
        int                     index;
        double                  time_1;
        double                  time_2;
        float                  *rand_vec;

        ic = 50;
        n = ic*ic;
        index = 0;

                        /*  Generate two right hand sides */

      rand_vec = imsl_f_random_uniform (4*n*sizeof(*rand_vec), 0);
        for (i=0; i<n; i++) {
                b_1[i].re = rand_vec[index++];
                b_1[i].im = rand_vec[index++];
                b_2[i].re = rand_vec[index++];
```

```
            b_2[i].im = rand_vec[index++];
        }
                        /*  Build coefficient matrix a */

        a = imsl_c_generate_test_coordinate (n, ic,
                &nz,
                IMSL_SYMMETRIC_STORAGE,
                0);

                        /*  Now solve Ax_1 = b_1 and return the numeric
                            factorization  */

        time_1 = imsl_ctime ();
        x_1 = imsl_c_lin_sol_posdef_coordinate (n, nz, a, b_1,
                IMSL_RETURN_NUMERIC_FACTOR, &numeric_factor,
                0);
        time_1 = imsl_ctime () - time_1;

                        /*  Now solve Ax_2 = b_2 given the numeric
                            factorization  */

        time_2 = imsl_ctime ();
        x_2 = imsl_c_lin_sol_posdef_coordinate (n, nz, a, b_2,
                IMSL_SUPPLY_NUMERIC_FACTOR, &numeric_factor,
                IMSL_SOLVE_ONLY,
                0);
        time_2 = imsl_ctime () - time_2;

        printf("time_2/time_1 = %lf\n", time_2/time_1);
}
```

Output

```
time_2/time_1 = 0.096386
```

lin_sol_gen_min_residual

Solves a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_gen_min_residual` (*int* `n`, *void* `amultp` (*float* `*p`, *float* `*z`), *float* `*b`, ..., 0)

The type *double* function is `imsl_d_lin_sol_gen_min_residual`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

void `amultp` (*float* `*p`, *float* `*z`)
User-supplied function which computes $z = Ap$.

float `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_lin_sol_gen_min_residual (int n, void amultp (), float *b,
        IMSL_RETURN_USER, float x[],
        IMSL_MAX_ITER, int *maxit,
        IMSL_REL_ERR, float tolerance,
        IMSL_PRECOND, void precond(),
        IMSL_MAX_KRYLOV_SUBSPACE_DIM, int kdmax,
        IMSL_HOUSEHOLDER_REORTHOG,
        IMSL_FCN_W_DATA, void amultp (), void *data,
        IMSL_PRECOND_W_DATA, void precond(), void *data,
        0)
```

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length `n` containing the solution x.

`IMSL_MAX_ITER`, *int* `*maxit` (Input/Output)
A pointer to an integer, initially set to the maximum number of GMRES iterations allowed. On exit, the number of iterations used is returned.
Default: `maxit` = 1000

`IMSL_REL_ERR`, *float* `tolerance` (Input)
The algorithm attempts to generate x such that $\|b - Ax\|_2 \leq \tau\|b\|_2$, where τ = `tolerance`.
Default: `tolerance` = `sqrt(imsl_f_machine(4))`

`IMSL_PRECOND`, *void* `precond` (*float* `*r`, *float* `*z`) (Input)
User supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix.

`IMSL_MAX_KRYLOV_SUBSPACE_DIM`, *int* `kdmax`, (Input)
The maximum Krylov subspace dimension, i.e., the maximum allowable number of GMRES iterations allowed before restarting.
Default: `kdmax` = `imsl_i_min(n, 20)`

`IMSL_HOUSEHOLDER_REORTHOG`,
Perform orthogonalization by Householder transformations, replacing the Gram-Schmidt process.

`IMSL_FCN_W_DATA`, *void* `amultp` (*float* `*p`, *float* `*z`, *void* `*data`), *void* `*data`, (Input)
User supplied function which computes $z = Ap$, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_PRECOND_W_DATA`, *void* `precond` (*float* `*r`, *float* `*z`, *void* `*data`), *void* `*data` (Input)
User supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_lin_sol_gen_min_residual`, based on the FORTRAN subroutine GMRES by H.F. Walker, solves the linear system $Ax = b$ using the GMRES method. This method is described in detail by Saad and Schultz (1986) and Walker (1988).

The GMRES method begins with an approximate solution x_0 and an initial residual $r_0 = b - Ax_0$. At iteration m, a correction z_m is determined in the Krylov subspace

$$\kappa_m(v) = \texttt{span}\,(v, Av, \ldots, A^{m-1}v)$$

$v = r_0$ which solves the least-squares problem

$$\min_{(z \in \kappa_m(r_0))} \left\| b - A(x_0 + z) \right\|_2$$

Then at iteration m, $x_m = x_0 + z_m$.

Orthogonalization by Householder transformations requires less storage but more arithmetic than Gram-Schmidt. However, Walker (1988) reports numerical experiments which suggest the Householder approach is more stable, especially as the limits of residual reduction are reached.

Examples

Example 1

As an example, consider the following matrix:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let $x^T = (1, 2, 3, 4, 5, 6)$ so that $Ax = (10, 7, 45, 33, -34, 31)^T$. The function `imsl_f_mat_mul_rect_coordinate` is used to form the product Ax.

```
#include <imsl.h>

void amultp (float*, float*);

main()
{
        float b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
        int n = 6;
        float *x;

        x = imsl_f_lin_sol_gen_min_residual (n, amultp, b,
                0);

        imsl_f_write_matrix ("Solution, x, to Ax = b", 1, n, x, 0);
}

void amultp (float *p, float *z)
{
        Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                                  1, 1, 10.0,
                                  1, 2, -3.0,
                                  1, 3, -1.0,
                                  2, 2, 15.0,
                                  3, 0, -2.0,
                                  3, 3, 10.0,
                                  3, 4, -1.0,
                                  4, 0, -1.0,
                                  4, 3, -5.0,
                                  4, 4, 1.0,
                                  4, 5, -3.0,
                                  5, 0, -1.0,
                                  5, 1, -2.0,
                                  5, 5, 6.0};
        int n = 6;
        int nz = 15;

        imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, p,
                IMSL_RETURN_USER_VECTOR, z,
                0);
}
```

Output

```
                Solution, x, to Ax = b
     1          2          3          4          5          6
     1          2          3          4          5          6
```

Example 2

In this example, the same system given in the first example is solved. This time a preconditioner is provided. The preconditioned matrix is chosen as the diagonal of A.

```
#include <imsl.h>

void amultp (float*, float*);
void precond (float*, float*);

main()
{
        float b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
        int n = 6;
        float *x;
        int maxit = 1000;

        x = imsl_f_lin_sol_gen_min_residual (n, amultp, b,
                IMSL_MAX_ITER, &maxit,
                IMSL_PRECOND, precond,
                0);

        imsl_f_write_matrix ("Solution, x, to Ax = b", 1, n, x, 0);
        printf ("\nNumber of iterations taken = %d\n", maxit);
}

                        /* Set z = Ap */

void amultp (float *p, float *z)
{
        static Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                                1, 1, 10.0,
                                1, 2, -3.0,
                                1, 3, -1.0,
                                2, 2, 15.0,
                                3, 0, -2.0,
                                3, 3, 10.0,
                                3, 4, -1.0,
                                4, 0, -1.0,
                                4, 3, -5.0,
                                4, 4, 1.0,
                                4, 5, -3.0,
                                5, 0, -1.0,
                                5, 1, -2.0,
                                5, 5, 6.0};
        int n = 6;
        int nz = 15;

        imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, p,
                IMSL_RETURN_USER_VECTOR, z,
                0);
```

```
}
                        /* Solve Mz = r */

void precond (float *r, float *z)
{
        static float diagonal_inverse[] =
                        {0.1, 0.1, 1.0/15.0, 0.1, 1.0, 1.0/6.0};
        int n = 6;
        int i;

        for (i=0; i<n; i++)
                z[i] = diagonal_inverse[i]*r[i];
}
```

Output

```
                    Solution, x, to Ax = b
 1           2           3           4           5           6
 1           2           3           4           5           6

Number of iterations taken = 5
```

lin_sol_def_cg

Solves a real symmetric definite linear system using a conjugate gradient method. Using optional arguments, a preconditioner can be supplied.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_def_cg` (*int* `n`, *void* `amultp` ( ), *float* `*b`, ..., 0)

The type *double* function is `imsl_d_lin_sol_def_cg`.

Required Arguments

int `n` (Input)
Number of rows in the matrix.

void `amultp` (*float* `*p`, *float* `*z`)
User-supplied function which computes $z = Ap$.

float `*b` (Input)
Vector of length `n` containing the right-hand side.

Return Value

A pointer to the solution x of the linear system $Ax = b$. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float *imsl_f_lin_sol_def_cg (int n, void amultp(), float *b,
        IMSL_RETURN_USER, float x[],
        IMSL_MAX_ITER, int *maxit,
        IMSL_REL_ERR, float relative_error,
        IMSL_PRECOND, void precond(),
        IMSL_JACOBI, float *diagonal,
        IMSL_FCN_W_DATA, void amultp(), void *data,
        IMSL_PRECOND_W_DATA, void precond(), void *data,
        0)
```

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length *n* containing the solution *x*.

`IMSL_MAX_ITER`, *int* `*maxit` (Input/Output)
A pointer to an integer, initially set to the maximum number of iterations allowed. On exit, the number of iterations used is returned.

`IMSL_REL_ERR`, *float* `relative_error` (Input)
The relative error desired.
Default: `relative_error = sqrt(imsl_f_machine(4))`

`IMSL_PRECOND`, *void* `precond` (*float* `*r`, *float* `*z`) (Input)
User supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix.

`IMSL_JACOBI`, *float* `diagonal[]` (Input)
Use the Jacobi preconditioner, i.e. $M = \text{diag}(A)$. The user-supplied vector `diagonal` should be set so that `diagonal[i]` $= A_{i,i}$.

`IMSL_FCN_W_DATA`, *void* `amultp` (*float* `*p`, *float* `*z`, *void* `*data`), *void* `*data`, (Input)
User supplied function which computes $z = Ap$, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_PRECOND_W_DATA`, *void* `precond` (*float* `*r`, *float* `*z`, *void* `*data`), *void* `*data`, (Input)
User supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_lin_sol_def_cg` solves the symmetric definite linear system $Ax = b$ using the conjugate gradient method with optional preconditioning. 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. In the default use of `imsl_f_lin_sol_def_cg`, $M = I$. If the option `IMSL_JACOBI` is selected, M is set to the diagonal of A.

The number of iterations needed depends on the matrix and the error tolerance. As a rough guide,

$$\texttt{itmax} = \sqrt{n} \text{ for } n >> 1$$

See the references mentioned above for details.

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
        $\beta_k = \left(z_k^T r_k\right) / \left(z_{k-1}^T r_{k-1}\right)$
        $p_k = z_k + \beta_k p_k$
    endif
    $z_k = Ap$
    $\alpha_k = \left(z_{k-1}^T z_{k-1}\right) / \left(z_k^T p_k\right)$
    $x_k = x_k + \alpha_k p_k$
    $r_k = r_k - \alpha_k z_k$
    if $\left(\| z_k \|_2 \le \tau(1-\lambda) \| x_k \|_2\right)$ then
        recompute λ
        if $\left(\| z_k \|_2 \le \tau(1-\lambda) \| x_k \|_2\right)$ exit
    endif
endfor

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, pp. 148-151)

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

where

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

It is also known that

$$\lambda_{\max}(T_1) \leq \lambda_{\max}(T_2) \leq \ldots \leq \lambda_{\max}(G) < 1$$

where the T_n are the symmetric, tridiagonal matrices

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

with $\mu_k = 1 - \beta_k/\alpha_{k-1} - 1/\alpha_k$, $\mu_1 = 1 - 1/\alpha_1$ and

$$\omega_k = \sqrt{B_k} / \alpha_{k-1}$$

Usually the eigenvalue computation is needed for only a few of the iterations.

Example 1

In this example, the solution to a linear system is found. The coefficient matrix is stored as a full matrix.

```
#include <imsl.h>

static void amultp (float*, float*);

void main()
{
        int n = 3;
        float b[] = {27.0, -78.0, 64.0};
        float *x;

        x = imsl_f_lin_sol_def_cg (n, amultp, b, 0);

        imsl_f_write_matrix ("x", 1, n, x, 0);
}

static void amultp (float *p, float *z)
{
        static float a[] = {1.0, -3.0, 2.0,
```

```
                        -3.0, 10.0, -5.0,
                        2.0, -5.0, 6.0};
        int n = 3;

        imsl_f_mat_mul_rect ("A*x",
                IMSL_A_MATRIX, n, n, a,
                IMSL_X_VECTOR, n, p,
                IMSL_RETURN_USER, z,
                0);
}
```

Output

```
          x
 1          2          3
 1         -4          7
```

Example 2

In this example, two different preconditioners are used to find the solution of a linear system which occurs in a finite difference solution of Laplace's equation on a regular $c \times c$ grid, $c = 100$. The matrix is $A = E(c^2, c)$. For the first solution, select Jacobi preconditioning and supply the diagonal, so $M = \texttt{diag}(A)$. The number of iterations performed and the maximum absolute error are printed. Next, use a more complicated preconditioning matrix, M, consisting of the symmetric tridiagonal part of A.

Notice that the symmetric positive definite band solver is used to factor M once, and subsequently just perform forward and back solves. Again, the number of iterations performed and the maximum absolute error are printed. Note the substantial reduction in iterations.

```
#include <imsl.h>

static void amultp (float*, float*);
static void precond (float*, float*);
static Imsl_f_sparse_elem *a;
static int n = 2500;
static int c = 50;
static int nz;

void main()
{
        int maxit = 1000;
        int i;
        int index;
        float *b;
        float *x;
        float *mod_five;
        float *diagonal;
        float norm;

        n = c*c;
        mod_five = (float*) malloc (n*sizeof(*mod_five));
        diagonal = (float*) malloc (n*sizeof(*diagonal));
        b = (float*) malloc (n*sizeof(*b));
```

```
                    /* Generate coefficient matrix */

        a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

                    /* Set a predetermined answer and diagonal */

        for (i=0; i<n; i++) {
                mod_five[i] = (float) (i % 5);
                diagonal[i] = 4.0;
        }

                    /* Get right hand side */

        amultp (mod_five, b);

                    /* Solve with jacobi preconditioning */

        x = imsl_f_lin_sol_def_cg (n, amultp, b,
                IMSL_MAX_ITER, &maxit,
                IMSL_JACOBI, diagonal,
                0);

                    /* Find max absolute error, print results */

        norm = imsl_f_vector_norm (n, x,
                IMSL_SECOND_VECTOR, mod_five,
                IMSL_INF_NORM, &index,
                0);
        printf ("iterations = %d, norm = %e\n", maxit, norm);
        free (x);

                    /* Solve same system, with different preconditioner */

        x = imsl_f_lin_sol_def_cg (n, amultp, b,
                IMSL_MAX_ITER, &maxit,
                IMSL_PRECOND, precond,
                0);

        norm = imsl_f_vector_norm (n, x,
                IMSL_SECOND_VECTOR, mod_five,
                IMSL_INF_NORM, &index,
                0);
        printf ("iterations = %d, norm = %e\n", maxit, norm);
}

                    /*  Set z = Ap */

static void amultp (float *p, float *z)
{
        imsl_f_mat_mul_rect_coordinate ("A*x",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, p,
                IMSL_RETURN_USER_VECTOR, z,
                0);
}

                   /* Solve Mz = r */
```

```
static void precond (float *r, float *z)
{
        static float *m;
        static float *factor;
        static int first = 1;
        float *null = (float*) 0;

        if (first) {

                    /* Factor the first time through */

                m = imsl_f_generate_test_band (n, 1,
                                               IMSL_SYMMETRIC_STORAGE, 0);
                imsl_f_lin_sol_posdef_band (n, m, 1, null,
                        IMSL_FACTOR, &factor,
                        IMSL_FACTOR_ONLY,
                        0);
                first = 1;
        }

                    /* Perform the forward and back solves */

        imsl_f_lin_sol_posdef_band (n, m, 1, r,
                IMSL_FACTOR_USER, factor,
                IMSL_SOLVE_ONLY,
                IMSL_RETURN_USER, z,
                0);
}
```

Output

```
iterations = 115, norm = 1.382828e-05
iterations = 75, norm = 7.319450e-05
```

lin_least_squares_gen

Solves a linear least-squares problem $Ax = b$. Using optional arguments, the QR factorization of A, $AP = QR$, and the solve step based on this factorization can be computed.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_least_squares_gen` (*int* `m`, *int* `n`, *float* `a[]`, *float* `b[]`, ..., 0)

The type *double* procedure is `imsl_d_lin_least_squares_gen`.

Required Arguments

int `m` (Input)
Number of rows in the matrix.

int `n` (Input)
Number of columns in the matrix.

float `a[]` (Input)
Array of size $m \times n$ containing the matrix.

float `b[]` (Input)
Array of size m containing the right-hand side.

Return Value

If no optional arguments are used, function `imsl_f_lin_least_squares_gen` returns a pointer to the solution x of the linear least-squares problem $Ax = b$. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float *imsl_f_lin_least_squares_gen (int m, int n, float a[], float b[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_RETURN_USER, float x[],
        IMSL_BASIS, float tol, int *kbasis,
        IMSL_RESIDUAL, float **p_res,
        IMSL_RESIDUAL_USER, float res[],
        IMSL_FACTOR, float **p_qraux, float **p_qr,
        IMSL_FACTOR_USER, float qraux[], float qr[],
        IMSL_FAC_COL_DIM, int qr_col_dim,
        IMSL_Q, float **p_q,
        IMSL_Q_USER, float q[],
        IMSL_Q_COL_DIM, int q_col_dim,
        IMSL_PIVOT, int pvt[],
        IMSL_FACTOR_ONLY,
        IMSL_SOLVE_ONLY,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of size n containing the least-squares solution x. If `IMSL_RETURN_USER` is used, the return value of the function is a pointer to the array `x`.

`IMSL_BASIS`, *float* `tol`, *int* `*kbasis` (Input, Input/Output)

`tol`: Nonnegative tolerance used to determine the subset of columns of A to be included in the solution.

Default: tol = sqrt (`imsl_amach`(4))

`kbasis`: Integer containing the number of columns used in the solution.

`kbasis` = k if $|r_{k+1,k+1}| < |tol|*|r_{1,1}|$ and $|r_{i,i}| \geq tol*|r_{1,1}|$ for $i = 1, 2, \ldots, k$. For more information on the use of this option, see "Description" on page 87.

Default: `kbasis` = min (m, n)

`IMSL_RESIDUAL`, *float* `**p_res` (Output)

The address of a pointer to an array of size m containing the residual vector $b - Ax$. On return, the necessary space is allocated by the function. Typically, *float* `*p_res` is declared, and `&p_res` is used as an argument.

`IMSL_RESIDUAL_USER`, *float* `res[]` (Output)

A user-allocated array of size m containing the residual vector $b - Ax$.

`IMSL_FACTOR`, *float* `**p_qraux`, *float* `**p_qr` (Output)

`**p_qraux`: The address of a pointer `qraux` to an array of size n containing the scalars τ_k of the Householder transformations in the first min (m, n) positions. On return, the necessary space is allocated by the function. Typically, *float* `*qraux` is declared, and `&qraux` is used as an argument.

`**p_qr`: The address of a pointer to an array of size $m \times n$ containing the Householder transformations that define the decomposition. The strictly lower-triangular part of this array contains the information to construct Q, and the upper-triangular part contains R. On return, the necessary space is allocated by the function. Typically, *float* `*qr` is declared, and `&qr` is used as an argument.

`IMSL_FACTOR_USER`, *float* `qraux[]`, *float* `qr[]` (Input /Output)

`qraux[]`: A user-allocated array of size n containing the scalars τ_k of the Householder transformations in the first min (m, n) positions.

`qr[]`: A user-allocated array of size $m \times n$ containing the Householder transformations that define the decomposition. The strictly lower-triangular part of this array contains the information to construct Q. The upper-triangular part contains R. If the data in `a` is not needed, `qr` can share the same storage locations as `a` by using `a` instead of the separate argument `qr`.

These parameters are "Input" if `IMSL_SOLVE` is specified; "Output" otherwise.

`IMSL_FAC_COL_DIM`, *int* `qr_col_dim` (Input)

The column dimension of the array containing QR factorization.
Default: `qr_col_dim` = n

`IMSL_Q`, *float* `**p_q` (Output)

The address of a pointer to an array of size $m \times m$ containing the orthogonal matrix of the factorization. On return, the necessary space is allocated by the function. Typically, *float* `*q` is declared, and `&q` is used as an argument.

`IMSL_Q_USER`, *float* `q[]` (Output)

A user-allocated array of size $m \times m$ containing the orthogonal matrix Q of the QR factorization.

IMSL_Q_COL_DIM, *int* q_col_dim (Input)

The column dimension of the array containing the Q matrix of the factorization.
Default: q_col_dim = m

IMSL_PIVOT, *int* pvt[] (Input/Output)

Array of size n containing the desired variable order and usage information. The argument is used with IMSL_FACTOR_ONLY or IMSL_SOLVE_ONLY.

On input, if pvt [k – 1] > 0, then column k of A is an initial column. If pvt [k – 1] = 0, then the column of A is a free column and can be interchanged in the column pivoting. If pvt [k – 1] < 0, then column k of A is a final column. If all columns are specified as initial (or final) columns, then no pivoting is performed. (The permutation matrix P is the identity matrix in this case.)

On output, pvt [k – 1] contains the index of the column of the original matrix that has been interchanged into column k.

Default: pvt [k – 1] = 0, k = 1, …, n

IMSL_FACTOR_ONLY

Compute just the QR factorization of the matrix AP with the permutation matrix P defined by pvt and by further pivoting involving free columns. If IMSL_FACTOR_ONLY is used, the additional arguments IMSL_PIVOT and IMSL_FACTOR are required. In that case, the required argument b is ignored, and the returned value of the function is NULL.

IMSL_SOLVE_ONLY

Compute the solution to the least-squares problem $Ax = b$ given the QR factorization previously computed by this function. If IMSL_SOLVE_ONLY is used, arguments IMSL_FACTOR, IMSL_PIVOT, and IMSL_BASIS are required, and the required argument a is ignored.

Description

The function imsl_f_lin_least_squares_gen solves a system of linear least-squares problems $Ax = b$ with column pivoting. It computes a QR factorization of the matrix AP, where P is the permutation matrix defined by the pivoting, and computes the smallest integer k satisfying $|r_{k+1,\,k+1}| < |tol|*|r_{1,1}|$ to the output variable kbasis. Householder transformations

$$Q_k = I - \tau_k u_k u_k^T Q$$

k = 1, …, min (m – 1, n)are used to compute the factorization. The decomposition is computed in the form $Q_{\min(m-1,\,n)} \ldots Q_1 AP = R$, so $AP = QR$ where $Q = Q_1 \ldots Q_{\min(m-1,\,n)}$. Since each Householder vector u_k has zeros in the first k – 1 entries, it is stored as part of column k of qr. The upper-trapezoidal matrix R is stored in the upper-trapezoidal part of the first min (m, n) rows of qr. The solution x to the least-squares problem is computed by solving the upper-triangular system of linear equations

$R(1{:}k, 1{:}k)\ y\ (1{:}k) = (Q^T b)\ (1{:}k)$ with k = `kbasis`. The solution is completed by setting $y(k + 1 : n)$ to zero and rearranging the variables, $x = Py$.

When `IMSL_FACTOR_ONLY` is specified, the function computes the QR factorization of AP with P defined by the input `pvt` and by column pivoting among ''free'' columns. Before the factorization, initial columns are moved to the beginning of the array `a` and the final columns to the end. Both initial and final columns are not permuted further during the computation. Just the free columns are moved.

If `IMSL_SOLVE_ONLY` is specified, then the function computes the least-squares solution to $Ax = b$ given the QR factorization previously defined. There are `kbasis` columns used in the solution. Hence, in the case that all columns are free, x is computed as described in the default case.

Examples

Example 1

This example illustrates the least-squares solution of four linear equations in three unknowns using column pivoting. The problem is equivalent to least-squares quadratic polynomial fitting to four data values. Write the polynomial as $p(t) = x_1 + tx_2 + t^2x_3$ and the data pairs (t_i, b_i), $t_i = 2i$, $i = 1, 2, 3, 4$. A pointer to the solution to $Ax = b$ is returned by the function `imsl_f_lin_least_squares_gen`.

```
#include <imsl.h>

float    a[] =  {1.0, 2.0, 4.0,
                 1.0, 4.0, 16.0,
                 1.0, 6.0, 36.0,
                 1.0, 8.0, 64.0};

float    b[] =  {4.999, 9.001, 12.999, 17.001};

main()
{
    int           m = 4, n = 3;
    float         *x;
                                    /* Solve Ax = b  for  x  */

    x = imsl_f_lin_least_squares_gen (m, n, a, b, 0);

                                    /* Print x */
    imsl_f_write_matrix ("Solution vector", 1, n, x, 0);
}
```

Output

```
Solution vector
    1           2           3
0.999       2.000       0.000
```

Example 2

This example uses the same coefficient matrix *A* as in the initial example. It computes the *QR* factorization of *A* with column pivoting. The final and free columns are specified by `pvt` and the column pivoting is done only among the free columns.

```
#include <imsl.h>

float    a[] =   {1.0, 2.0, 4.0,
                  1.0, 4.0, 16.0,
                  1.0, 6.0, 36.0,
                  1.0, 8.0, 64.0};

int      pvt[] =  {0, 0, -1};

main()
{
    int           m = 4, n = 3;
    float         *x, *b;
    float         *p_qraux, *p_qr;
    float         *p_q;
                                    /* Compute the QR factorization */
                                    /* of A with partial column */
                                    /* pivoting */
    x = imsl_f_lin_least_squares_gen (m, n, a, b,
                                      IMSL_PIVOT, pvt,
                                      IMSL_FACTOR, &p_qraux, &p_qr,
                                      IMSL_Q, &p_q,
                                      IMSL_FACTOR_ONLY,
                                      0);

                                    /* Print Q */
    imsl_f_write_matrix ("The matrix Q", m, m, p_q, 0);

                                    /* Print R */
    imsl_f_write_matrix ("The matrix R", m, n, p_qr,
                         IMSL_PRINT_UPPER,
                         0);

                                    /* Print pivots */
    imsl_i_write_matrix ("The Pivot Sequence", 1, n, pvt, 0);

}
```

Output

```
                The matrix Q
              1           2           3           4
1     -0.1826     -0.8165      0.5000     -0.2236
2     -0.3651     -0.4082     -0.5000      0.6708
3     -0.5477      0.0000     -0.5000     -0.6708
4     -0.7303      0.4082      0.5000      0.2236

             The matrix R
             1           2           3
1      -10.95       -1.83      -73.03
2                   -0.82       16.33
3                                8.00
```

```
The Pivot Sequence
     1   2   3
     2   1   3
```

Example 3

This example computes the *QR* factorization with column pivoting for the matrix *A* of the initial example. It computes the least-squares solutions to $Ax = b_i$ for $i = 1, 2, 3$.

```
#include <imsl.h>

float    a[]   = {1.0, 2.0, 4.0,
                  1.0, 4.0, 16.0,
                  1.0, 6.0, 36.0,
                  1.0, 8.0, 64.0};

float    b[]   = {4.999, 9.001, 12.999, 17.001,
                   2.0,   3.142,  5.11,   0.0,
                   1.34,  8.112,  3.76,  10.99};

int      pvt[] = {0, 0, 0};

main()
{
    int           m = 4, n = 3;
    int           i, k = 3;
    float         *p_qraux, *p_qr;
    float         tol = 1.e-4;
    int           *kbasis;
    float         *x, *p_res;
                                  /* Factor A with the given pvt */
                                  /* setting all variables to */
                                  /* be free */
    imsl_f_lin_least_squares_gen (m, n, a, b,
                              IMSL_BASIS, tol, &kbasis,
                              IMSL_PIVOT, pvt,
                              IMSL_FACTOR, &p_qraux, &p_qr,
                              IMSL_FACTOR_ONLY,
                              0);
                                  /* Print some factorization */
                                  /* information*/

    printf("Number of Columns in the base\n%2d", kbasis);
    imsl_f_write_matrix ("Upper triangular R Matrix", m, n, p_qr,
                              IMSL_PRINT_UPPER,
                              0);
    imsl_i_write_matrix ("The output column order ", 1, n, pvt, 0);

                                  /* Solve Ax = b  for each x  */
                                  /* given the factorization */
    for ( i = 0; i < k; i++) {
          x = imsl_f_lin_least_squares_gen (m, n, a, &b[i*m],
                              IMSL_BASIS, tol, &kbasis,
                              IMSL_PIVOT, pvt,
                              IMSL_FACTOR_USER, p_qraux, p_qr,
                              IMSL_RESIDUAL, &p_res,
                              IMSL_SOLVE_ONLY,
                              0);
                                  /* Print right-hand side, b */
```

```
                                /* and solution, x */
            imsl_f_write_matrix ("Right-hand side, b ", 1, m,
                                &b[i*m], 0);
            imsl_f_write_matrix ("Solution, x ", 1, n, x, 0);
                               /* Print residuals, b - Ax */
            imsl_f_write_matrix ("Residual, b - Ax ", 1, m, p_res,
                                0);
          }

}
```

Output

```
Number of Columns in the base
 3
      Upper triangular R Matrix
            1           2           3
1      -75.26      -10.63       -1.59
2                   -2.65       -1.15
3                                0.36

The output column order
        1   2   3
        3   2   1

             Right-hand side, b
          1           2           3           4
          5           9          13          17

           Solution, x
          1           2           3
      0.999       2.000       0.000

            Residual, b - Ax
          1           2           3           4
   -0.0004      0.0012     -0.0012      0.0004

             Right-hand side, b
          1           2           3           4
      2.000       3.142       5.110       0.000

           Solution, x
          1           2           3
    -4.244       3.706      -0.391

            Residual, b - Ax
          1           2           3           4
      0.395      -1.186       1.186      -0.395

             Right-hand side, b
          1           2           3           4
       1.34        8.11        3.76       10.99



           Solution, x
          1           2           3
     0.4735      0.9437      0.0286
```

```
           Residual, b - Ax
     1           2           3           4
-1.135       3.406      -3.406       1.135
```

Fatal Errors

`IMSL_SINGULAR_TRI_MATRIX`	The input triangular matrix is singular. The index of the first zero diagonal term is #.

lin_lsq_lin_constraints

Solves a linear least-squares problem with linear constraints.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_lsq_lin_constraints` (*int* `nra`, *int* `nca`, *int* `ncon`, *float* `a[]`, *float* `b[]`, *float* `c[]`, *float* `bl[]`, *float* `bu[]`, *int* `con_type[]`, *float* `xlb[]`, *float* `xub[]`, ..., 0)

The type double function is `imsl_d_lin_lsq_lin_constraints`.

Required Arguments

int `nra` (Input)
Number of least-squares equations.

int `nca` (Input)
Number of variables.

int `ncon` (Input)
Number of constraints.

float `a[]` (Input)
Array of size `nra` × `nca` containing the coefficients of the `nra` least-squares equations.

float `b[]` (Input)
Array of length `nra` containing the right-hand sides of the least-squares equations.

float `c[]` (Input)
Array of size `ncon` × `nca` containing the coefficients of the `ncon` constraints.

float `bl[]` (Input)
Array of length `ncon` containing the lower limit of the general constraints. If there is no lower limit on the *i*-th constraint, then `bl[i]` will not be referenced.

float `bu[]` (Input)
Array of length `ncon` containing the upper limit of the general constraints. 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.

int `con_type[]` (Input)
Array of length `ncon` indicating the type of constraints exclusive of simple bounds, where `con_type[i]` = 0, 1, 2, 3 indicates =, <=, >= and range constraints, respectively.

float `xlb[]` (Input)
Array of length `nca` containing the lower bound on the variables. If there is no lower bound on the *i*-th variable, then `xlb[i]` should be set to 1.0e30.

float `xub[]` (Input)
Array of length `nca` containing the upper bound on the variables. If there is no lower bound on the *i*-th variable, then `xub[i]` should be set to −1.0e30.

Return Value

A pointer to the to a vector of length `nca` containing the approximate solution. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float `*imsl_f_lin_lsq_lin_constraints` (*int* `nra`, *int* `nca`, *int* `ncon`, *float* `a[]`, *float* `b[]`, *float* `c[]`, *float* `bl[]`, *float* `bu[]`, *int* `con_type[]`, *float* `xlb[]`, *float* `xub[]`,
`IMSL_RETURN_USER`, *float* `x[]`,
`IMSL_RESIDUAL`, *float* `**residual`,
`IMSL_RESIDUAL_USER`, *float* `residual_user[]`,
`IMSL_PRINT`,
`IMSL_MAX_ITER`, *int* `max_iter`,
`IMSL_REL_FCN_TOL`, *float* `rel_tol`,
`IMSL_ABS_FCN_TOL`, *float* `abs_tol`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Store the solution in the user supplied vector `x` of length `nca`.

`IMSL_RESIDUAL`, *float* `**residual` (Output)
The address of a pointer to an array containing the residuals $b - Ax$ of the least-squares equations at the approximate solution.

`IMSL_RESIDUAL_USER`, *float* `residual_user[]` (Output)
Store the residuals in the user-supplied vector of length `nra`.

`IMSL_PRINT`,
Debug output flag. Choose this option if more detailed output is desired.

`IMSL_MAX_ITER`, *int* `max_iter` (Input)
Set the maximum number of add/drop iterations.
Default: `max_iter = 5*max(nra, nca)`

`IMSL_REL_FCN_TOL`, *float* `rel_tol` (Input)
Relative rank determination tolerance to be used.
Default: `rel_tol = sqrt(imsl_f_machine(4))`

`IMSL_ABS_FCN_TOL`, *float* `abs_tol` (Input)
Absolute rank determination tolerance to be used.
Default: `abs_tol = sqrt(imsl_f_machine(4))`

Description

The function `imsl_f_lin_lsq_lin_constraints` solves linear least-squares problems with linear constraints. These are systems of least-squares equations of the form

$$Ax \cong b$$

subject to

$$b^l \le Cx \le b_u$$

$$x_l \le x \le x_u$$

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

Examples

Example 1

In this example, the following problem is solved in the least-squares sense:

$$3x_1 + 2x_2 + x_3 = 3.3$$

$$4x_1 + 2x_2 + x_3 = 2.2$$

$$2x_1 + 2x_2 + x_3 = 1.3$$

$$x_1 + x_2 + x_3 = 1.0$$

Subject to

$$x_1 = x_2 + x_3 \leq 1$$

$$0 \leq x_1 \leq 0.5$$

$$0 \leq x_2 \leq 0.5$$

$$0 \leq x_3 \leq 0.5$$

```
#include <imsl.h>

main()
{
        int     nra = 4;
        int     nca = 3;

        int     ncon = 1;
        float  *x;
        float   a[] = {3.0, 2.0, 1.0,
                        4.0, 2.0, 1.0,
                        2.0, 2.0, 1.0,
                        1.0, 1.0, 1.0};
        float   b[] = {3.3, 2.3, 1.3, 1.0};
        float   c[] = {1.0, 1.0, 1.0};
        float   xlb[] = {0.0, 0.0, 0.0};
        float   xub[] = {0.5, 0.5, 0.5};
        int     con_type[] = {1};
        float   bc[] = {1.0};

        x = imsl_f_lin_lsq_lin_constraints (nra, nca, ncon, a, b, c,
                bc, bc, con_type, xlb, xub, 0);

        imsl_f_write_matrix ("Solution", 1, nca, x, 0);
}
```

Output

```
            Solution
         1           2           3
       0.5         0.3         0.2
```

Example 2

The same problem solved in the first example is solved again. This time residuals of the least-squares equations at the approximate solution are returned, and the norm of the residual vector is printed. Both the solution and residuals are returned in user-supplied space.

```
#include <imsl.h>
```

```
main()
{
        int     nra = 4;
        int     nca = 3;
        int     ncon = 1;
        float   x[3];
        float   residual[4];
        float   a[] = {3.0, 2.0, 1.0,
                       4.0, 2.0, 1.0,
                       2.0, 2.0, 1.0,
                       1.0, 1.0, 1.0};
        float   b[] = {3.3, 2.3, 1.3, 1.0};
        float   c[] = {1.0, 1.0, 1.0};
        float   xlb[] = {0.0, 0.0, 0.0};
        float   xub[] = {0.5, 0.5, 0.5};
        int     con_type[] = {1};
        float   bc[] = {1.0};

        imsl_f_lin_lsq_lin_constraints (nra, nca, ncon, a, b, c,
                bc, bc, con_type, xlb, xub,
                IMSL_RETURN_USER, x,
                IMSL_RESIDUAL_USER, residual,
                0);

        imsl_f_write_matrix ("Solution", 1, nca, x, 0);
        imsl_f_write_matrix ("Residual", 1, nra, residual, 0);
        printf ("\n\nNorm of residual = %f\n",
                 imsl_f_vector_norm (nra, residual, 0));
}
```

Output

```
              Solution
        1           2           3
      0.5         0.3         0.2

                 Residual
        1           2           3           4
     -1.0         0.5         0.5        -0.0

Norm of residual = 1.224745
```

lin_svd_gen

Computes the SVD, $A = USV^T$, of a real rectangular matrix A. An approximate generalized inverse and rank of A also can be computed.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_svd_gen` (*int* `m`, *int* `n`, *float* `a[]`, ..., 0)

The type *double* procedure is `imsl_d_lin_svd_gen`.

Required Arguments

int `m` (Input)
Number of rows in the matrix.

int `n` (Input)
Number of columns in the matrix.

float `a[]` (Input)
Array of size $m \times n$ containing the matrix.

Return Value

If no optional arguments are used, `imsl_f_lin_svd_gen` returns a pointer to an array of size min (m, n) containing the ordered singular values of the matrix. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_lin_svd_gen (int m, int n, float a[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_RETURN_USER, float s[],
        IMSL_RANK, float tol, int *rank,
        IMSL_U, float **p_u,
        IMSL_U_USER, float u[],
        IMSL_U_COL_DIM, int u_col_dim,
        IMSL_V, float **p_v,
        IMSL_V_USER, float v[],
        IMSL_V_COL_DIM, int v_col_dim,
        IMSL_INVERSE, float **p_gen_inva,
        IMSL_INVERSE_USER, float gen_inva[],
        IMSL_INV_COL_DIM, int gen_inva_col_dim,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

`IMSL_RETURN_USER`, *float* `s[]` (Output)
A user-allocated array of size min (m, n) containing the singular values of A in its first min (m, n) positions in nonincreasing order. If `IMSL_RETURN_USER` is used, the return value of `imsl_f_lin_svd_gen` is `s`.

`IMSL_RANK`, *float* `tol`, *int* `*rank` (Input/Output)

`tol`: Scalar containing the tolerance used to determine when a singular value is negligible and replaced by the value zero. If `tol` > 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq$ `tol`. If `tol` < 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq |\texttt{tol}| * \|A\|_\infty$. In this case, |`tol`| should be an estimate of relative error or uncertainty in the data.

`*rank`: Integer containing an estimate of the rank of A.

`IMSL_U`, *float* `**p_u` (Output)
`**p_u`: The address of a pointer to an array of size $m \times \min(m, n)$ containing the left- singular vectors of A. On return, the necessary space is allocated by `imsl_f_lin_svd_gen`. Typically, *float* `*p_u` is declared, and `&p_u` is used as an argument.

`IMSL_U_USER`, *float* `u[]` (Output)
`u[]`: A user-allocated array of size $m \times \min(m, n)$containing the left-singular vectors of A. If $m \geq n$, the left-singular vectors can be returned using the storage locations of the array `a`.

`IMSL_U_COL_DIM`, *int* `u_col_dim` (Input)
The column dimension of the array containing the left-singular vectors.
Default: `u_col_dim` = min (m, n)

`IMSL_V`, *float* `**p_v` (Output)
`**p_v`: The address of a pointer to an array of size $n \times \min(m, n)$ containing the right singular vectors of A. On return, the necessary space is allocated by `imsl_f_lin_svd_gen`. Typically, *float* `*p_v` is declared, and `&p_v` is used as an argument.

`IMSL_V_USER`, *float* `v[]` (Output)
`v[]`: A user-allocated array of size $n \times \min(m, n)$containing the right-singular vectors of A. The right-singular vectors can be returned using the storage locations of the array `a`. Note that the return of the left- and right-singular vectors cannot use the storage locations of `a` simultaneously.

`IMSL_V_COL_DIM`, *int* `v_col_dim` (Input)
The column dimension of the array containing the right-singular vectors.
Default: `v_col_dim` = min (m, n)

`IMSL_INVERSE`, *float* `**p_gen_inva` (Output)
The address of a pointer to an array of size $n \times m$ containing the generalized inverse of the matrix A. On return, the necessary space is allocated by `imsl_f_lin_svd_gen`. Typically, *float* `*p_gen_inva` is declared, and `&p_gen_inva` is used as an argument.

`IMSL_INVERSE_USER`, *float* `gen_inva[]` (Output)
A user-allocated array of size $n \times m$ containing the general inverse of the matrix A.

`IMSL_INV_COL_DIM`, *int* `gen_inva_col_dim` (Input)
The column dimension of the array containing the general inverse of the

matrix A.
Default: `gen_inva_col_dim` = m

Description

The function `imsl_f_lin_svd_gen` computes the singular value decomposition of a real matrix A. It first reduces the matrix A to a bidiagonal matrix B by pre- and post-multiplying Householder transformations. Then, the singular value decomposition of B is computed using the implicit-shifted QR algorithm. An estimate of the rank of the matrix A is obtained by finding the smallest integer k such that $s_{k,k} \leq$ `tol` or $s_{k,k} \leq |\texttt{tol}|*\|A\|_\infty$. Since $s_{i+1,\,i+1} \leq s_{i,i}$, it follows that all the $s_{i,\,i}$ satisfy the same inequality for $i = k, \ldots, \min(m, n) - 1$. The rank is set to the value $k - 1$. If $A = USV^T$, its generalized inverse is $A^+ = VS^+ U^T$. Here,

$$S^+ = diag\left(s_{1,1}^{-1}, \ldots, s_{i,i}^{-1}, 0, \ldots, 0\right)$$

Only singular values that are not negligible are reciprocated. If `IMSL_INVERSE` or `IMSL_INVERSE_USER` is specified, the function first computes the singular value decomposition of the matrix A. The generalized inverse is then computed. The function `imsl_f_lin_svd_gen` fails if the QR algorithm does not converge after 30 iterations isolating an individual singular value.

Examples

Example 1

This example computes the singular values of a real 6 × 4 matrix.

```
#include <imsl.h>

float a[] =   {1.0,  2.0,  1.0,  4.0,
               3.0,  2.0,  1.0,  3.0,
               4.0,  3.0,  1.0,  4.0,
               2.0,  1.0,  3.0,  1.0,
               1.0,  5.0,  2.0,  2.0,
               1.0,  2.0,  2.0,  3.0};

main()
{
    int          m = 6, n = 4;
    float        *s;
                              /* Compute singular values */
    s = imsl_f_lin_svd_gen (m, n, a, 0);
                              /* Print singular values */
    imsl_f_write_matrix ("Singular values", 1, n, s, 0);
}
```

Output

```
          Singular values
    1           2           3           4
11.49        3.27        2.65        2.09
```

Example 2

This example computes the singular value decomposition of the 6×4 real matrix A. The singular values are returned in the user-provided array. The matrices U and V are returned in the space provided by the function `imsl_f_lin_svd_gen`.

```
#include <imsl.h>

float a[] =  {1.0,  2.0,  1.0,  4.0,
              3.0,  2.0,  1.0,  3.0,
              4.0,  3.0,  1.0,  4.0,
              2.0,  1.0,  3.0,  1.0,
              1.0,  5.0,  2.0,  2.0,
              1.0,  2.0,  2.0,  3.0};

main()
{
    int           m = 6, n = 4;
    float        s[4], *p_u, *p_v;
                                    /* Compute SVD */
    imsl_f_lin_svd_gen (m, n, a,
                        IMSL_RETURN_USER, s,
                        IMSL_U, &p_u,
                        IMSL_V, &p_v,
                        0);
                                    /* Print decomposition*/

    imsl_f_write_matrix ("Singular values, S", 1, n, s, 0);
    imsl_f_write_matrix ("Left singular vectors, U", m, n, p_u, 0);
    imsl_f_write_matrix ("Right singular vectors, V", n, n, p_v, 0);
}
```

Output

```
              Singular values, S
         1           2           3           4
     11.49        3.27        2.65        2.09

             Left singular vectors, U
             1           2           3           4
1     -0.3805      0.1197      0.4391     -0.5654
2     -0.4038      0.3451     -0.0566      0.2148
3     -0.5451      0.4293      0.0514      0.4321
4     -0.2648     -0.0683     -0.8839     -0.2153
5     -0.4463     -0.8168      0.1419      0.3213
6     -0.3546     -0.1021     -0.0043     -0.5458

             Right singular vectors, 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
```

Example 3

This example computes the rank and generalized inverse of a 3×2 matrix A. The rank and the 2×3 generalized inverse matrix A^+ are printed.

```
#include <imsl.h>

float a[] =  {1.0,   0.0,
              1.0,   1.0,
            100.0, -50.0};

main()
{
    int           m = 3, n = 2;
    float         tol;
    float         gen_inva[6];
    float        *s;
    int          *rank;
                                  /* Compute generalized inverse */
    tol = 1.e-4;
    s = imsl_f_lin_svd_gen (m, n, a,
                            IMSL_RANK, tol, &rank,
                            IMSL_INVERSE_USER, gen_inva,
                            IMSL_INV_COL_DIM, m,
                            0);
                                  /* Print rank, singular values and  */
                                  /* generalized inverse. */


    printf ("Rank of matrix = %2d", rank);

    imsl_f_write_matrix ("Singular values", 1, n, s, 0);

    imsl_f_write_matrix ("Generalized inverse", n, m, gen_inva,
                          IMSL_A_COL_DIM, m,
                          0);
}
```

Output

```
Rank of matrix =  2
    Singular values
         1           2
     111.8         1.4

         Generalized inverse
            1           2           3
1       0.100       0.300       0.006
2       0.200       0.600      -0.008
```

Warning Errors

IMSL_SLOWCONVERGENT_MATRIX	Convergence cannot be reached after 30 iterations.

lin_svd_gen (complex)

Computes the SVD, $A = USV^H$, of a complex rectangular matrix A. An approximate generalized inverse and rank of A also can be computed.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_lin_svd_gen` (*int* `m`, *int* `n`, *f_complex* `a[]`, ..., 0)

The type *d_complex function* is `imsl_z_lin_svd_gen`.

Required Arguments

int `m` (Input)
Number of rows in the matrix.

int `n` (Input)
Number of columns in the matrix.

f_complex `a[]` (Input)
Array of size $m \times n$ containing the matrix.

Return Value

Using only required arguments, `imsl_c_lin_svd_gen` returns a pointer to a complex array of length min (m, n) containing the singular values of the matrix. To release this space, use `free`. If no value can be computed then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
f_complex *imsl_c_lin_svd_gen (int m, int n, f_complex a[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_RETURN_USER, f_complex s[],
        IMSL_RANK, float tol, int *rank,
        IMSL_U, f_complex **p_u,
        IMSL_U_USER, f_complex u[],
        IMSL_U_COL_DIM, int u_col_dim,
        IMSL_V, f_complex **p_v,
        IMSL_V_USER, f_complex v[],
        IMSL_V_COL_DIM, int v_col_dim,
        IMSL_INVERSE, f_complex **p_gen_inva,
        IMSL_INVERSE_USER, f_complex gen_inva[],
        IMSL_INV_COL_DIM, int gen_inva_col_dim,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

`IMSL_RETURN_USER`, *f_complex* `s[]` (Output)
A user-allocated array of length min (m, n) containing the singular values of A in its first min (m, n) positions in nonincreasing order. The complex entries are all real. If `IMSL_RETURN_USER` is used, the return value of `imsl_c_lin_svd_gen` is `s`.

`IMSL_RANK`, *float* `tol`, *int* `*rank` (Input/Output)

`tol`: Scalar containing the tolerance used to determine when a singular value is negligible and replaced by the value zero. If `tol` > 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq$ `tol`. If `tol` < 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq |\texttt{tol}|*\|A\|_\infty$. In this case, should be an estimate of relative error or uncertainty in the data.

`*rank`: Integer containing an estimate of the rank of A.

`IMSL_U`, *f_complex* `**p_u` (Output)
The address of a pointer to an array of size $m \times$ min (m, n) containing the left-singular vectors of A. On return, the necessary space is allocated by `imsl_c_lin_svd_gen`. Typically, *f_complex* `*p_u` is declared, and `&p_u` is used as an argument.

`IMSL_U_USER`, *f_complex* `u[]` (Output)
A user-allocated array of size $m \times$ min (m, n) containing the left-singular vectors of A. If $m \geq n$, the left-singular vectors can be returned using the storage locations of the array `a`.

`IMSL_U_COL_DIM`, *int* `u_col_dim` (Input)
The column dimension of the array containing the left-singular vectors.
Default: `u_col_dim` = min (m, n)

`IMSL_V`, *f_complex* `**p_v` (Output)
The address of a pointer to an array of size $n \times$ min (m, n) containing the right-singular vectors of A. On return, the necessary space is allocated by `imsl_c_lin_svd_gen`. Typically, *f_complex* `*p_v` is declared, and `&p_v` is used as an argument.

`IMSL_V_USER`, *f_complex* `v[]` (Output)
A user-allocated array of size $n \times$ min (m, n) containing the right-singular vectors of A. The right-singular vectors can be returned using the storage locations of the array `a`. Note that the return of the left and right-singular vectors cannot use the storage locations of `a` simultaneously.

`IMSL_V_COL_DIM`, *int* `v_col_dim` (Input)
The column dimension of the array containing the right-singular vectors.
Default: `v_col_dim` = min (m, n)

IMSL_INVERSE, *f_complex* **p_gen_inva (Output)
The address of a pointer to an array of size $n \times m$ containing the generalized inverse of the matrix A. On return, the necessary space is allocated by imsl_c_lin_svd_gen. Typically, *f_complex* *p_gen_inva is declared, and &p_gen_inva is used as an argument.

IMSL_INVERSE_USER, *f_complex* gen_inva[] (Output)
A user-allocated array of size $n \times m$ containing the general inverse of the matrix A.

IMSL_INV_COL_DIM, *int* gen_inva_col_dim (Input)
The column dimension of the array containing the general inverse of the matrix A.
Default: gen_inva_col_dim = m

Description

The function imsl_c_lin_svd_gen computes the singular value decomposition of a complex matrix A. It first reduces the matrix A to a bidiagonal matrix B by pre- and post-multiplying Householder transformations. Then, the singular value decomposition of B is computed using the implicit-shifted QR algorithm. An estimate of the rank of the matrix A is obtained by finding the smallest integer k such that $s_{k,k} \leq$ tol or $s_{k,k} \leq |\text{tol}|*\|A\|_\infty$. Since $s_{i+1,i+1} \leq s_{i,i}$, it follows that all the $s_{i,i}$ satisfy the same inequality for $i = k, \ldots, \min(m, n) - 1$. The rank is set to the value $k - 1$. If $A = USV^H$, its generalized inverse is $A^+ = VS^+ U^H$.

Here,

$$S^+ = \text{diag}\left(s_{1,1}^{-1}, \ldots, s_{i,i}^{-1}, 0, \ldots, 0\right)$$

Only singular values that are not negligible are reciprocated. If IMSL_INVERSE or IMSL_INVERSE_USER is specified, the function first computes the singular value decomposition of the matrix A. The generalized inverse is then computed. The function imsl_c_lin_svd_gen fails if the QR algorithm does not converge after 30 iterations isolating an individual singular value.

Examples

Example 1

This example computes the singular values of a 6×3 complex matrix.

```
#include <imsl.h>
 main()
{
    int               m = 6, n = 3;
    f_complex        *s;
    f_complex  a[] =  {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0},
                       {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0},
                       {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0},
```

```
                        {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0},
                        {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0},
                        {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}};
                                    /* Compute singular values */
    s = imsl_c_lin_svd_gen (m, n, a, 0);
                                    /* Print singular values */
    imsl_c_write_matrix ("Singular values", 1, n, s, 0);
}
```

Output

```
                          Singular values
                    1                       2                       3
(     11.77,      0.00)  (      9.30,      0.00)  (      4.99,      0.00)
```

Example 2

This example computes the singular value decomposition of the 6×3 complex matrix A. The singular values are returned in the user-provided array. The matrices U and V are returned in the space provided by the function `imsl_c_lin_svd_gen`.

```
#include <imsl.h>

main()
{
    int           m = 6, n = 3;
    f_complex     s[3], *p_u, *p_v;
    f_complex  a[] =  {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0},
                       {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0},


                       {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0},
                       {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0},
                       {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0},
                       {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}};
                                   /* Compute SVD of a */
        imsl_c_lin_svd_gen (m, n, a,
                            IMSL_RETURN_USER, s,
                            IMSL_U, &p_u,
                            IMSL_V, &p_v,
                            0);
                                    /* Print decomposition factors */
        imsl_c_write_matrix ("Singular values, S", 1, n, s, 0);
        imsl_c_write_matrix ("Left singular vectors, U", m, n, p_u, 0);
        imsl_c_write_matrix ("Right singular vectors, V", n, n, p_v, 0);
 }
```

Output

```
                          Singular values, S
                    1                       2                       3
(     11.77,      0.00)  (      9.30,      0.00)  (      4.99,      0.00)

                        Left singular vectors, U
                      1                       2                       3
1 (    0.1968,    0.2186)  (    0.5011,    0.0217)  (   -0.2007,   -0.1003)
2 (    0.3443,   -0.3542)  (   -0.2933,    0.0248)  (    0.1155,   -0.2338)
3 (    0.1457,    0.2307)  (   -0.5424,    0.1381)  (   -0.4361,   -0.4407)
4 (    0.3016,   -0.0844)  (    0.2157,    0.2659)  (   -0.0523,   -0.0894)
```

```
5 (    0.2283,   -0.6008)  (   -0.1325,    0.1433)  (    0.3152,   -0.0090)
6 (    0.2876,   -0.0350)  (    0.4377,   -0.0400)  (    0.0458,   -0.6205)

                        Right singular vectors, 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)
```

Example 3

This example computes the rank and generalized inverse of a 6×4 matrix A. The rank and the 4×6 generalized inverse matrix A^+ are printed.

```
#include <imsl.h>
main()
{
    int           m = 6, n = 4;
    int           *rank;
    float         tol;
    f_complex     gen_inv[24], *s;
    f_complex  a[] =  {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0}, {1.0,0.0},
                       {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0}, {0.0,1.0},
                       {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0}, {0.0,0.0},
                       {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0}, {2.0,1.0},
                       {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0}, {1.0,3.1},
                       {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}, {1.4,1.9}};
                                  /* Factor a */
    tol = 1.e-4;
    s = imsl_c_lin_svd_gen (m, n, a,
                            IMSL_RANK, tol, &rank,
                            IMSL_INVERSE_USER, gen_inv,
                            IMSL_INV_COL_DIM, m,
                            0);
                                   /* Print rank and generalized */
                                   /* inverse matrix */

    printf ("Rank = %2d", rank);

    imsl_c_write_matrix ("Singular values", 1, n, s, 0);

    imsl_c_write_matrix ("Generalized inverse", n, m, gen_inv,
                          IMSL_A_COL_DIM, m, 0);
}
```

Output

```
Rank =  4
                            Singular values
                       1                       2                       3
(     12.13,      0.00)  (      9.53,      0.00)  (      5.67,      0.00)

                       4
(      1.74,      0.00)

                          Generalized inverse
                       1                       2                       3
1 (    0.0266,    0.0164)  (   -0.0185,    0.0453)  (    0.0720,    0.0700)
2 (    0.0061,    0.0280)  (    0.0820,   -0.1156)  (   -0.0410,   -0.0242)
```

```
3 (   -0.0019,   -0.0572)  (    0.1174,    0.0812)  (    0.0499,    0.0463)
4 (    0.0380,    0.0298)  (   -0.0758,   -0.2158)  (    0.0356,   -0.0557)

                         4                        5                         6
1 (   -0.0220,   -0.0428)  (   -0.0003,   -0.0709)  (    0.0254,    0.1050)
2 (    0.0959,    0.0885)  (   -0.0187,    0.0287)  (   -0.0218,   -0.1109)
3 (   -0.0234,    0.1033)  (   -0.0769,    0.0103)  (    0.0810,   -0.1074)
4 (    0.2918,   -0.0763)  (    0.0881,    0.2070)  (   -0.1531,    0.0814)
```

Warning Errors

`IMSL_SLOWCONVERGENT_MATRIX`	Convergence cannot be reached after 30 iterations.

lin_sol_nonnegdef

Solves a real symmetric nonnegative definite system of linear equations $Ax = b$. Using options, computes a Cholesky factorization of the matrix A, such that $A = R^T R = LL^T$. Computes the solution to $Ax = b$ given the Cholesky factor.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_sol_nonnegdef` (*int* `n`, *float* `a[]`, *float* `b[]`, …, 0)

The type *double* function is `imsl_d_lin_sol_nonnegdef`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrix.

float `a[]` (Input)
Array of size $n \times n$ containing the matrix.

float `b[]` (Input)
Array of size n containing the right-hand side.

Return Value

Using required arguments, `imsl_f_lin_sol_nonnegdef` returns a pointer to a solution x of the linear system. To release this space, use `free`. If no value can be computed, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_lin_sol_nonnegdef` (*int* `n`, *float* `a[]`, *float* `b[]`,
`IMSL_RETURN_USER`, *float* `x[]`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_FACTOR`, *float* `**p_factor`,

```
IMSL_FACTOR_USER, float factor[],
IMSL_FAC_COL_DIM, int fac_col_dim,
IMSL_INVERSE, float **p_inva,
IMSL_INVERSE_USER, float inva[],
IMSL_INV_COL_DIM, int inv_col_dim,
IMSL_TOLERANCE, float tol,
IMSL_FACTOR_ONLY,
IMSL_SOLVE_ONLY,
IMSL_INVERSE_ONLY,
0)
```

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length n containing the solution x. When this option is specified, no storage is allocated for the solution, and `imsl_f_lin_sol_nonnegdef` returns a pointer to the array `x`.

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of the array `a`.
Default: `a_col_dim` = n

`IMSL_FACTOR`, *float* `**p_factor` (Output)
The address of a pointer to an array of size $n \times n$ containing the LL^T factorization of A. When this option is specified, the space for the factor matrix is allocated by `imsl_f_lin_sol_nonnegdef`. The lower-triangular part of the factor array contains L, and the upper-triangular part contains $L^T R$. Typically, *float* `*p_factor` is declared, and `&p_factor` is used as an argument.

`IMSL_FACTOR_USER`, *float* `factor[]` (Input/Output)
A user-allocated array of size $n \times n$ containing the LL^T factorization of A. The lower-triangular part of `factor` contains L, and the upper-triangular part contains L^T. If `a` is not needed, `a` and `factor` can be the same storage locations. If `IMSL_SOLVE` is specified, this parameter is *input*; otherwise, it is *output*.

`IMSL_FAC_COL_DIM`, *int* `fac_col_dim` (Input)
The column dimension of the array containing the LL^T factorization.
Default: `fac_col_dim` = n

`IMSL_INVERSE`, *float* `**p_inva` (Output)
The address of a pointer to an array of size $n \times n$ containing the inverse of A. The space for this array is allocated by `imsl_f_lin_sol_nonnegdef`. Typically, *float* `*p_inva` is declared, and `&p_inva` is used as an argument.

`IMSL_INVERSE_USER`, *float* `inva[]` (Output)
A user-allocated array of size $n \times n$ containing the inverse of A. If `a` is not needed, `a` and `factor` can be the same storage locations. The storage locations for A cannot be the factorization and the inverse of A at the same time.

IMSL_INV_COL_DIM, *int* inva_col_dim (Input)
Default: inva_col_dim = *n*

IMSL_TOLERANCE, *float* tol (Input)
Default: tol = 100* imsl_f_machine(4)
See the documentation for imsl_f_machine in Chapter 12, "Utilities."

IMSL_FACTOR_ONLY
Compute the LL^T factorization of A only. The argument b is ignored, and either the optional argument IMSL_FACTOR or IMSL_FACTOR_USER is required.

IMSL_SOLVE_ONLY
Solve $Ax = b$ using the factorization previously computed by this function. The argument a is ignored, and the optional argument IMSL_FACTOR_USER is required.

IMSL_INVERSE_ONLY
Compute the inverse of A only. The argument b is ignored, and either the optional argument IMSL_INVERSE or IMSL_INVERSE_USER is required.

Description

The function imsl_f_lin_sol_nonnegdef solves a system of linear algebraic equations having a symmetric nonnegative definite (positive semidefinite) coefficient matrix. It first computes a Cholesky (LL^T or R^TR) factorization of the coefficient matrix A.

The factorization algorithm is based on the work of Healy (1968) and proceeds sequentially by columns. The i-th column is declared to be linearly dependent on the first $i - 1$ columns if

$$\left| a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2 \right| \leq \varepsilon \left| a_{ii} \right|$$

where ε (specified in tol) may be set by the user. When a linear dependence is declared, all elements in the i-th row of R (column of L) are set to zero.

Modifications due to Farebrother and Berry (1974) and Barrett and Healy (1978) for checking for matrices that are not nonnegative definite also are incorporated. The function imsl_f_lin_sol_nonnegdef declares A to not be nonnegative definite and issues an error message if either of the following conditions are satisfied:

1. $a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2 < -\varepsilon \left| a_{ii} \right|$
2. $r_{ii} = 0$ and $\left| a_{ik} - \sum_{j=1}^{i-1} r_{ji} r_{jk} \right| > \varepsilon \sqrt{a_{ii} a_{kk}}, k > i$

Healy's (1968) algorithm and the function `imsl_f_lin_sol_nonnegdef` permit the matrices A and R to occupy the same storage. Barrett and Healy (1978) in their remark neglect this fact. The function `imsl_f_lin_sol_nonnegdef` uses

$$\sum_{j=1}^{i-1} r_{ij}^2$$

for a_{ii} in the above condition 2 to remedy this problem.

If an inverse of the matrix A is required and the matrix is not (numerically) positive definite, then the resulting inverse is a symmetric g_2 inverse of A. For a matrix G to be a g_2 inverse of a matrix A, G must satisfy conditions 1 and 2 for the Moore-Penrose inverse, but generally fail conditions 3 and 4. The four conditions for G to be a Moore-Penrose inverse of A are as follows:

1. $AGA = A$
2. $GAG = G$
3. AG is symmetric
4. GA is symmetric

The solution of the linear system $Ax = b$ is computed by solving the factored version of the linear system $R^T Rx = b$ as two successive triangular linear systems. In solving the triangular linear systems, if the elements of a row of R are all zero, the corresponding element of the solution vector is set to zero. For a detailed description of the algorithm, see Section 2 in Sallas and Lionti (1988).

Examples

Example 1

A solution to a system of four linear equations is obtained. Maindonald (1984, pp. 83–86 and 104–105) discusses the computations for the factorization and solution to this problem.

```
#include <imsl.h>

main()
{
    int         n = 4;
    float       *x;
    float       a[] = {36.0, 12.0, 30.0,  6.0,
                       12.0, 20.0,  2.0, 10.0,
                       30.0,  2.0, 29.0,  1.0,
                        6.0, 10.0,  1.0, 14.0};
    float       b[] = {18.0, 22.0,  7.0, 20.0};

                                /* Solve Ax = b  for  x  */
    x = imsl_f_lin_sol_nonnegdef(n, a, b, 0);
                                /* Print solution, x, of Ax = b */
    imsl_f_write_matrix("Solution, x", 1, n, x, 0);
}
```

Output

```
              Solution, x
    1           2           3           4
0.167       0.500       0.000       1.000
```

Example 2

The symmetric nonnegative definite matrix in the initial example is used to compute the factorization only in the first call to `lin_sol_nonnegdef`. The space needed for the factor is provided by the user. On the second call, both the LL^T factorization and the right-hand side vector in the first example are used as the input to compute a solution x. It also illustrates another way to obtain the solution array x.

```
#include <imsl.h>

main()
{
    int         n = 4,  a_col_dim = 6;
    float       factor[36], x[5];
    float       a[] = {36.0, 12.0, 30.0,  6.0,
                       12.0, 20.0,  2.0, 10.0,
                       30.0,  2.0, 29.0,  1.0,
                        6.0, 10.0,  1.0, 14.0};
    float       b[] = {18.0, 22.0,  7.0, 20.0};
                                /* Factor A */
    imsl_f_lin_sol_nonnegdef(n, a, b,
                             IMSL_FACTOR_USER, factor,
                             IMSL_FAC_COL_DIM, a_col_dim,
                             IMSL_FACTOR_ONLY,
                             0);
                                /* NULL is returned in  */
                                /* this case.  Another  */
                                /* way to obtain the    */
                                /* factor is to use the */
                                /* IMSL_FACTOR option.  */
    imsl_f_write_matrix("factor", n, n, factor,
                        IMSL_A_COL_DIM, a_col_dim,
                        0);
                                /* Get the solution using */
                                /* the factorized matrix. */
    imsl_f_lin_sol_nonnegdef(n, a, b,
                             IMSL_FACTOR_USER, factor,
                             IMSL_FAC_COL_DIM, a_col_dim,
                             IMSL_RETURN_USER, x,
                             IMSL_SOLVE_ONLY,
                             0);
    imsl_f_write_matrix("Solution, x, of Ax = b", 1, n, x, 0);
}
```

Output

```
                  factor
            1           2           3           4
1           6           2           5           1
2           2           4          -2           2
3           5          -2           0           0
```

```
4           1           2           0           3

            Solution, x, of Ax = b
          1           2           3           4
      0.167       0.500       0.000       1.000
```

Example 3

This example uses the IMSL_INVERSE option to compute the symmetric *g* inverse of the symmetric nonnegative matrix in the first example. Maindonald (1984, p. 106) discusses the computations for this problem.

```
#include <stdio.h>
#include <imsl.h>

void main()
{
    int         n = 4;
    float       *p_a_inva, *p_a_inva_a, *p_inva;
    float       a[] = {36.0, 12.0, 30.0,  6.0,
                       12.0, 20.0,  2.0, 10.0,
                       30.0,  2.0, 29.0,  1.0,
                        6.0, 10.0,  1.0, 14.0};
                                /* Get g2_inverse(a) */
    imsl_f_lin_sol_nonnegdef(n, a, NULL,
                             IMSL_INVERSE, &p_inva,
                             IMSL_INVERSE_ONLY,
                             0);
                                /* Form a*g2_inverse(a) */
    p_a_inva = imsl_f_mat_mul_rect("A*B",
                                   IMSL_A_MATRIX, n, n, a,
                                   IMSL_B_MATRIX, n, n, p_inva,
                                   0);
                                /* Form a*g2_inverse(a)*a */
    p_a_inva_a = imsl_f_mat_mul_rect("A*B",
                                     IMSL_A_MATRIX, n, n, p_a_inva,
                                     IMSL_B_MATRIX, n, n, a,
                                      0);
    imsl_f_write_matrix("The g2 inverse of a", n, n, p_inva, 0);
    imsl_f_write_matrix("a*g2_inverse(a)\nviolates condition 3 of"
                        " the M-P inverse", n, n, p_a_inva, 0);
    imsl_f_write_matrix("a = a*g2_inverse(a)*a\ncondition 1 of"
                        " the M-P inverse", n, n, p_a_inva_a, 0);
}
```

Output

```
               The g2 inverse of a
            1           2           3           4
1      0.0347     -0.0208      0.0000      0.0000
2     -0.0208      0.0903      0.0000     -0.0556
3      0.0000      0.0000      0.0000      0.0000
4      0.0000     -0.0556      0.0000      0.1111

                a*g2_inverse(a)
     violates condition 3 of the M-P inverse
            1           2           3           4
```

```
1          1.0         -0.0          0.0          0.0
2          0.0          1.0          0.0          0.0
3          1.0         -0.5          0.0          0.0
4          0.0         -0.0          0.0          1.0

          a = a*g2_inverse(a)*a
        condition 1 of the M-P inverse
             1            2            3            4
1           36           12           30            6
2           12           20            2           10
3           30            2           29            1
4            6           10            1           14
```

Warning Errors

IMSL_INCONSISTENT_EQUATIONS_2	The linear system of equations is inconsistent.
IMSL_NOT_NONNEG_DEFINITE	The matrix A is not nonnegative definite.

Chapter 2: Eigensystem Analysis

Routines

2.1 Linear Eigensystem Problems

General Matrices
Eigenvalues and eigenvectors..... eig_gen 118
Eigenvalues and eigenvectors..... eig_gen (complex) 120

Real Symmetric Matrices
Eigenvalues and eigenvectors.....eig_sym 123

Complex Hermitian Matrices
Eigenvalues and eigenvectors..... eig_herm (complex) 126

2.2 Generalized Eigensystem Problems

Real Symmetric Matrices and B Positive Definite
Eigenvalues and eigenvectors.....eig_symgen 129
Real matrices ..... geneig 132
Complex matrices..... geneig (complex) 135

Usage Notes

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 functions, we have chosen this factor so that x has Euclidean length one, and the component of x of largest magnitude is positive. The eigenvalues and corresponding eigenvectors are sorted then returned in the order of largest to smallest complex magnitude. If x is a complex vector, this component of largest magnitude is scaled to be real and positive. The entry where this component occurs can be arbitrary for eigenvectors having nonunique maximum magnitude values.

A generalized linear eigensystem problem is represented by $Ax = \lambda Bx$ where A and B are $n \times n$ matrices. The value λ is a generalized eigenvalue, and x is the corresponding generalized eigenvector. The generalized eigenvectors are normalized in the same manner as the ordinary eigensystem problem.

Error Analysis and Accuracy

The remarks in this section are for ordinary eigenvalue problems. Except in special cases, functions will not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem $Ax = \lambda x$. Typically, the computed pair

$$\tilde{x}, \tilde{\lambda}$$

are an exact eigenvector-eigenvalue pair for a "nearby" matrix $A + E$. Information about E is known only in terms of bounds of the form $\|E\|_2 \leq f(n)\,\|A\|_2\varepsilon$. The value of $f(n)$ depends on the algorithm, but is typically a small fractional power of n. The parameter ε is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan 1989, p. 342),

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

where $\sigma(A)$ is the set of all eigenvalues of A (called the *spectrum* of A), X is the matrix of eigenvectors, $\|\cdot\|_2$ is Euclidean length, 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 accuracy of the computed eigenvalues

$$\tilde{\lambda}_j$$

and eigenvectors

$$\tilde{x}_j$$

can be checked by computing their performance index τ. The performance index is defined to be

$$\tau = \max_{1 \leq j \leq n} \frac{\left\|A\tilde{x}_j - \tilde{\lambda}_j\tilde{x}_j\right\|_2}{n\varepsilon\left\|A\right\|_2\left\|\tilde{x}_j\right\|_2}$$

where ε is again the machine precision.

The performance index τ is related to the error analysis because

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

where E is the "nearby" matrix discussed above.

While the exact value of τ is precision and data dependent, the performance of an eigensystem analysis function is defined as excellent if $\tau < 1$, good if $1 \leq \tau \leq 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of τ can serve as a warning that there is a significant error in the calculation.

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 often difficult for users to understand. Suppose the accuracy of an individual eigenvalue is desired. This can be answered approximately by computing the *condition number of an individual eigenvalue*

(see Golub and Van Loan 1989, pp. 344–345). For matrices A, such that the computed array of normalized eigenvectors X is invertible, the condition number of λ_j is

$$\kappa_j = \left\| e_j^T X^{-1} \right\|$$

the Euclidean length of the j-th row of X^{-1}. Users can choose to compute this matrix using function `imsl_c_lin_sol_gen` in 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. Occasionally, changes of variables 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, assuming A is definite, 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.

Now suppose that B is nonsingular. The user can solve the ordinary eigenvalue problem $Cx = \lambda x$ where $C = B^{-1}A$. The matrix C is 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 $F = P^{-1}AQ^{-1}$ and 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 function `imsl_f_eig_symgen` (page 129), uses $P = R^T$ and $Q = R$ where R is an upper-triangular matrix obtained from a Cholesky decomposition, $B = R^T R$. The matrix $F = R^{-T}AR^{-1}$ is symmetric and real. Computation of the eigenvalue-eigenvector expansion for F is based on function `imsl_f_eig_sym` (page 123).

eig_gen

Computes the eigenexpansion of a real matrix *A*.

Synopsis

#include `<imsl.h>`

f_complex *imsl_f_eig_gen (*int* n, *float* *a, …, 0)

The type *d_complex* function is `imsl_d_eig_gen`.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

float `*a` (Input)
An array of size $n \times n$ containing the matrix.

Return Value

A pointer to the *n* complex eigenvalues of the matrix. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
f_complex *imsl_f_eig_gen (int n, float *a,
       IMSL_VECTORS, f_complex **evec,
       IMSL_VECTORS_USER, f_complex evecu[],
       IMSL_RETURN_USER, f_complex evalu[],
       IMSL_A_COL_DIM, int a_col_dim,
       IMSL_EVECU_COL_DIM, int evecu_col_dim,
       0)
```

Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the matrix of eigenvectors is returned in the space `evecu`.

`IMSL_RETURN_USER`, *f_complex* `evalu[]` (Output)
Store the *n* eigenvalues in the space `evalu`.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of a.
Default: a_col_dim = *n*

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of evecu.
Default: evecu_col_dim = *n*

Description

Function imsl_f_eig_gen computes the eigenvalues of a real matrix by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary orthogonal or Gauss similarity transformations. Then, eigenvalues are computed using a *QR* or combined *LR*-*QR* algorithm (Golub and Van Loan 1989, pp. 373–382, and Watkins and Elsner 1990). The combined *LR*-*QR* algorithm is based on an implementation by Jeff Haag and David Watkins. Eigenvectors are then calculated as required. When eigenvectors are computed, the *QR* algorithm is used to compute the eigenexpansion. When only eigenvalues are required, the combined *LR*-*QR* algorithm is used.

Examples

Example 1

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       a[] =  {8.0, -1.0, -5.0,
                        -4.0,  4.0, -2.0,
                        18.0, -5.0, -7.0};
    f_complex   *eval;
                                /* Compute eigenvalues of A */
    eval = imsl_f_eig_gen (n, a, 0);
                                /* Print eigenvalues */
    imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
                          Eigenvalues
                    1                       2                       3
(         2,         4)   (         2,        -4)   (         1,         0)
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

main()
{
    int         n = 3;
```

```
    float       a[] =  {8.0, -1.0, -5.0,
                        -4.0,  4.0, -2.0,
                        18.0, -5.0, -7.0};
    f_complex   *eval;
    f_complex   *evec;
                                /* Compute eigenvalues of A */
    eval = imsl_f_eig_gen (n, a,
                           IMSL_VECTORS, &evec,
                           0);
                                /* Print eigenvalues and eigenvectors */
    imsl_c_write_matrix ("Eigenvalues",  1, n, eval, 0);
    imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
                                Eigenvalues
                      1                         2                         3
(         2,         4)   (         2,        -4)   (         1,         0)

                                Eigenvectors
                        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)
```

Warning Errors

IMSL_SLOW_CONVERGENCE_GEN	The iteration for an eigenvalue did not converge after # iterations.

eig_gen (complex)

Computes the eigenexpansion of a complex matrix *A*.

Synopsis

#include <imsl.h>

f_complex *imsl_c_eig_gen (*int* n, *f_complex* *a, …, 0)

The type *d_complex* procedure is imsl_z_eig_gen.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

f_complex *a (Input)
Array of size $n \times n$ containing the matrix.

Return Value

A pointer to the *n* complex eigenvalues of the matrix. To release this space, use free. If no value can be computed, then NULL is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_eig_gen` (*int* `n`, *f_complex* `*a`,
`IMSL_VECTORS`, *f_complex* `**evec`,
`IMSL_VECTORS_USER`, *f_complex* `evecu[]`,
`IMSL_RETURN_USER`, *f_complex* `evalu[]`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim`,
0)

Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* *evecu is declared, and &evecu is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the matrix of eigenvectors is returned in the space evecu.

`IMSL_RETURN_USER`, f_complex evalu[] (Output)
Store the n eigenvalues in the space evalu.

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of A.
Default: `a_col_dim` = n

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = n

Description

The function `imsl_c_eig_gen` computes the eigenvalues of a complex matrix by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary Gauss transformations. Then, the eigenvalues are computed using an explicitly shifted *LR* algorithm. Eigenvectors are calculated during the iterations for the eigenvalues (Martin and Wilkinson 1971).

Examples

Example 1

```
#include <imsl.h>

main()
{
    int         n = 4;
    f_complex   a[] =   { {5,9}, {5,5},  {-6,-6}, {-7,-7},
                          {3,3}, {6,10}, {-5,-5}, {-6,-6},
```

```
                          {2,2}, {3,3},  {-1, 3}, {-5,-5},
                          {1,1}, {2,2},  {-3,-3}, { 0, 4} };
    f_complex   *eval;
                                /* Compute eigenvalues */
    eval = imsl_c_eig_gen (n, a, 0);
                                /* Print eigenvalues */
    imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
                              Eigenvalues
                       1                       2                       3
(         4,         8)  (         3,         7)  (         2,         6)

                       4
(         1,         5)
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

main()
{
    int         n = 4;
    f_complex   a[] =   { {5,9}, {5,5},  {-6,-6}, {-7,-7},
                          {3,3}, {6,10}, {-5,-5}, {-6,-6},
                          {2,2}, {3,3},  {-1, 3}, {-5,-5},
                          {1,1}, {2,2},  {-3,-3}, { 0, 4} };
    f_complex   *eval;
    f_complex   *evec;
                           /* Compute eigenvalues and eigenvectors */
    eval = imsl_c_eig_gen (n, a,
                           IMSL_VECTORS, &evec,
                           0);
                           /* Print eigenvalues and eigenvectors */
    imsl_c_write_matrix ("Eigenvalues",  1, n, eval, 0);
    imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
                              Eigenvalues
                       1                       2                       3
(         4,         8)  (         3,         7)  (         2,         6)

                       4
(         1,         5)

                              Eigenvectors
                         1                       2                       3
1  (   0.5773,   -0.0000)  (   0.5774    0.0000)  (   0.3780,   -0.0000)
2  (   0.5773,   -0.0000)  (   0.5773,  -0.0000)  (   0.7559,    0.0000)
3  (   0.5774,    0.0000)  (  -0.0000,  -0.0000)  (   0.3780,    0.0000)
4  (  -0.0000,   -0.0000)  (   0.5774,   0.0000)  (   0.3780,   -0.0000)
```

```
                          4
1   (     0.7559,     0.0000)
2   (     0.3780,     0.0000)
3   (     0.3780,     0.0000)
4   (     0.3780,     0.0000)
```

Fatal Errors

IMSL_SLOW_CONVERGENCE_GEN	The iteration for an eigenvalue did not converge after # iterations.

eig_sym

Computes the eigenexpansion of a real symmetric matrix A.

Synopsis

#*include* <imsl.h>

float *imsl_f_eig_sym (*int* n, *float* *a, …, 0)

The type *double* procedure is imsl_d_eig_sym.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

float *a (Input)
Array of size $n \times n$ containing the symmetric matrix.

Return Value

A pointer to the n eigenvalues of the symmetric matrix. To release this space, use free. If no value can be computed, then NULL is returned.

Synopsis with Optional Arguments

#*include* <imsl.h>

```
float *imsl_f_eig_sym (int n, float *a,
        IMSL_VECTORS, float **evec,
        IMSL_VECTORS_USER, float evecu[],
        IMSL_RETURN_USER, float evalu[],
        IMSL_RANGE, float elow, float ehigh,
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_EVECU_COL_DIM, int evecu_col_dim,
        IMSL_RESULT_NUMBER, int *n_eval,
        0)
```

Optional Arguments

`IMSL_VECTORS`, *float* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing the eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *float* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *float* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the orthogonal matrix of eigenvectors is returned in the space `evecu`.

`IMSL_RETURN_USER`, *float* `evalu[]` (Output)
Store the n eigenvalues in the space `evalu`.

`IMSL_RANGE`, *float* `elow`, *float* `ehigh` (Input)
Return eigenvalues and optionally eigenvectors that lie in the interval with lower limit `elow` and upper limit `ehigh`.
Default: (`elow`, `ehigh`) = $(-\infty, +\infty)$

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of `a`.
Default: `a_col_dim` = n

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = n

`IMSL_RESULT_NUMBER`, *int* `*n_eval` (Output)
The number of output eigenvalues and eigenvectors in the range `low`, `ehigh`.

Description

The function `imsl_f_eig_sym` computes the eigenvalues of a symmetric real matrix by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational *QR* or bisection algorithm. Eigenvectors are calculated as required (Parlett 1980, pp. 169–173).

Examples

Example 1

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       a[] =   {7.0,  -8.0,  -8.0,
                        -8.0, -16.0, -18.0,
                        -8.0, -18.0,  13.0};
    float       *eval;
                                /* Compute eigenvalues */
    eval = imsl_f_eig_sym(n, a, 0);
                                /* Print eigenvalues */
```

```
    imsl_f_write_matrix ("Eigenvalues", 1, 3, eval, 0);
}
```

Output

```
            Eigenvalues
         1           2           3
    -27.90       22.68        9.22
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       a[] =   {7.0,  -8.0,  -8.0,
                        -8.0, -16.0, -18.0,
                        -8.0, -18.0,  13.0};
    float       *eval;
    float       *evec;
                                /* Compute eigenvalues and eigenvectors */
    eval = imsl_f_eig_sym(n, a,
                          IMSL_VECTORS, &evec,
                          0);
                                /* Print eigenvalues and eigenvectors */
    imsl_f_write_matrix ("Eigenvalues",  1, n, eval, 0);
    imsl_f_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
            Eigenvalues
         1           2           3
    -27.90       22.68        9.22



            Eigenvectors
            1           2           3
1      0.2945     -0.2722      0.9161
2      0.8521     -0.3591     -0.3806
3      0.4326      0.8927      0.1262
```

Warning Errors

IMSL_SLOW_CONVERGENCE_SYM	The iteration for the eigenvalue failed to converge in 100 iterations before deflating.
IMSL_SLOW_CONVERGENCE_2	Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue.
IMSL_LOST_ORTHOGONALITY_2	The eigenvectors have lost orthogonality.

IMSL_NO_EIGENVALUES_RETURNED	The number of eigenvalues in the specified interval exceeds mxeval. The argument n_eval contains the number of eigenvalues in the interval. No eigenvalues will be returned.

eig_herm (complex)

Computes the eigenexpansion of a complex Hermitian matrix A.

Synopsis

#include <imsl.h>

float *imsl_c_eig_herm (*int* n, *f_complex* *a, …, 0)

The type *double* procedure is imsl_d_eig_herm.

Required Arguments

int n (Input)
Number of rows and columns in the matrix.

f_complex *a (Input)
Array of size $n \times n$ containing the matrix.

Return Value

A pointer to the n eigenvalues of the matrix. To release this space, use free. If no value can be computed, then NULL is returned.

Synopsis with Optional Arguments

#include <imsl.h>

```
float *imsl_c_eig_herm (int n, f_complex *a,
        IMSL_VECTORS, f_complex **evec,
        IMSL_VECTORS_USER, f_complex evecu[],
        IMSL_RETURN_USER, float evalu[],
        IMSL_RANGE, float elow, float ehigh,
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_EVECU_COL_DIM, int evecu_col_dim,
        IMSL_RESULT_NUMBER, int *n_eval,
        0)
```

Optional Arguments

IMSL_VECTORS, *f_complex* **evec (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* *evec is declared, and &evec is used as an argument.

IMSL_VECTORS_USER, *f_complex* evecu[] (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the unitary matrix of eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *float* evalu[] (Output)
Store the n eigenvalues in the space evalu.

IMSL_RANGE, *float* elow, *float* ehigh (Input)
Return eigenvalues and optionally eigenvectors that lie in the interval with lower limit elow and upper limit ehigh.
Default: (elow, ehigh) = $(-\infty, +\infty)$.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of A.
Default: a_col_dim = n

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of X.
Default: evecu_col_dim = n

IMSL_RESULT_NUMBER, *int* *n_eval (Output)
The number of output eigenvalues and eigenvectors in the range elow, ehigh.

Description

The function imsl_c_eig_herm computes the eigenvalues of a complex Hermitian matrix by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational QR or bisection algorithm. Eigenvectors are calculated as required.

Examples

Example 1

```
#include <imsl.h>

main()
{
    int         n = 3;
    f_complex   a[] =   { {1,0}, {1,-7}, {0,-1},
                          {1,7}, {5,0}, {10,-3},
                          {0,1}, {10,3}, {-2,0} };
    float       *eval;
                                    /* Compute eigenvalues */
    eval = imsl_c_eig_herm(n, a, 0);
                                    /* Print eigenvalues */
    imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
            Eigenvalues
         1           2           3
     15.38      -10.63       -0.75
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

main()
{
    int         n = 3;
    f_complex   a[] =   { {1,0}, {1,-7},  {0,-1},
                          {1,7}, {5,0},  {10,-3},
                          {0,1}, {10,3}, {-2,0} };
    float       *eval;
    f_complex   *evec;
                                /* Compute eigenvalues and eigenvectors */
    eval = imsl_c_eig_herm(n, a,
                           IMSL_VECTORS, &evec,
                           0);
                                /* Print eigenvalues and eigenvectors */
    imsl_f_write_matrix ("Eigenvalues",  1, n, eval, 0);
    imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
            Eigenvalues
        1           2           3
     15.38      -10.63       -0.75

                              Eigenvectors
                        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)
```

Warning Errors

IMSL_LOST_ORTHOGONALITY	The iteration for at least one eigenvector failed to converge. Some of the eigenvectors may be inaccurate.
IMSL_NEVAL_MXEVAL_MISMATCH	The determined number of eigenvalues in the interval (#, #) is #. However, the input value for the maximum number of eigenvalues in this interval is #.

Fatal Errors

IMSL_SLOW_CONVERGENCE_GEN	The iteration for the eigenvalues did not converge.
IMSL_HERMITIAN_DIAG_REAL	The matrix element A (#, #) = #. The diagonal of a Hermitian matrix must be real.

eig_symgen

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. The matrices A and B are real and symmetric, and B is positive definite.

Synopsis

#*include* `<imsl.h>`

float *imsl_f_eig_symgen (*int* n, *float* *a, *float* *b, …, 0)

The type *double* procedure is `imsl_d_eig_symgen`.

Required Arguments

int `n` (Input)
Number of rows and columns in the matrices.

float `*a` (Input)
Array of size $n \times n$ containing the symmetric coefficient matrix A.

float `*b` (Input)
Array of size $n \times n$ containing the positive definite symmetric coefficient matrix B.

Return Value

A pointer to the n eigenvalues of the symmetric matrix. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#*include* `<imsl.h>`

```
float *imsl_f_eig_symgen (int n, float *a, float *b,
        IMSL_VECTORS, float **evec,
        IMSL_VECTORS_USER, float evecu[],
        IMSL_RETURN_USER, float evalu[],
        IMSL_RANGE, float elow, float ehigh,
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_B_COL_DIM, int b_col_dim,
        IMSL_EVECU_COL_DIM, int evecu_col_dim,
        0)
```

Optional Arguments

`IMSL_VECTORS`, *float* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the problem. On return, the necessary space is allocated by the function. Typically, *float* `*evec` is declared, and `&evec` is used as an argument.

IMSL_VECTORS_USER, *float* evecu[] (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the matrix of generalized eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *float* evalu[] (Output)
Store the n eigenvalues in the space evalu.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of A.
Default: a_col_dim = n

IMSL_B_COL_DIM, *int* b_col_dim (Input)
The column dimension of B.
Default: b_col_dim = n

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of evecu.
Default: evecu_col_dim = n

Description

The function imsl_f_eig_symgen computes the eigenvalues of a symmetric, positive definite eigenvalue problem by a three-phase process (Martin and Wilkinson 1971). The matrix B is reduced to factored form using the Cholesky decomposition. These factors are used to form a congruence transformation that yields a symmetric real matrix whose eigenexpansion is obtained. The problem is then transformed back to the original coordinates. Eigenvectors are calculated and transformed as required.

Examples

Example 1

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       a[] =   {1.1, 1.2, 1.4,
                         1.2, 1.3, 1.5,
                         1.4, 1.5, 1.6};
    float       b[] =   {2.0, 1.0, 0.0,
                         1.0, 2.0, 1.0,
                         0.0, 1.0, 2.0};
    float       *eval;
                                /* Solve for eigenvalues */
    eval = imsl_f_eig_symgen (n, a, b, 0);
                                /* Print eigenvalues */
    imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
           Eigenvalues
        1           2           3
    1.386      -0.058      -0.003
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

main()
{
    int         n = 3;
    float       a[] =   {1.1, 1.2, 1.4,
                         1.2, 1.3, 1.5,
                         1.4, 1.5, 1.6};
    float       b[] =   {2.0, 1.0, 0.0,
                         1.0, 2.0, 1.0,
                         0.0, 1.0, 2.0};
    float       *eval;
    float       *evec;
                              /* Solve for eigenvalues and eigenvectors */
    eval = imsl_f_eig_symgen (n, a, b,
                              IMSL_VECTORS, &evec,
                              0);
                              /* Print eigenvalues and eigenvectors */
    imsl_f_write_matrix ("Eigenvalues",  1, n, eval, 0);
    imsl_f_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
            Eigenvalues
        1           2           3
    1.386      -0.058      -0.003

            Eigenvectors
            1           2           3
1      0.6431     -0.1147     -0.6817
2     -0.0224     -0.6872      0.7266
3      0.7655      0.7174     -0.0858
```

Warning Errors

IMSL_SLOW_CONVERGENCE_SYM	The iteration for an eigenvalue failed to converge in 100 iterations before deflating.

Fatal Errors

IMSL_SUBMATRIX_NOT_POS_DEFINITE	The leading # by # submatrix of the input matrix is not positive definite.
IMSL_MATRIX_B_NOT_POS_DEFINITE	Matrix B is not positive definite.

geneig

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B real.

Synopsis

#include `<imsl.h>`

void `imsl_f_geneig` (*int* `n`, *float* `*a`, *float* `*b`, *f_complex* `*alpha`, *float* `*beta`, ..., 0)

The *double* analogue is `imsl_d_geneig`.

Required Arguments

int `n` (Input)
Number of rows and columns in A and B.

float `*a` (Input)
Array of size $n \times n$ containing the coefficient matrix A.

float `*b` (Input)
Array of size $n \times n$ containing the coefficient matrix B.

f_complex `*alpha` (Output)
Vector of size n containing scalars α_i. If $\beta_i \neq 0$, $\lambda_i = \alpha_i/\beta_i$ for $i = 0, \ldots, n-1$ are the eigenvalues of the system.

float `*beta` (Output)
Vector of size n.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `imsl_f_geneig` (*int* `n`, *float* `*a`, *float* `*b`,
`IMSL_VECTORS`, *f_complex* `**evec`,
`IMSL_VECTORS_USER`, *f_complex* `evecu[]`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_B_COL_DIM`, *int* `b_col_dim`,
`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim`,
0)

Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the problem. Each vector is normalized to have Euclidean length equal to the value one. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the matrix

of generalized eigenvectors is returned in the space `evecu`. Each vector is normalized to have Euclidean length equal to the value one.

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of *A*.
Default: `a_col_dim` = *n*

`IMSL_B_COL_DIM`, *int* `b_col_dim` (Input)
The column dimension of *B*.
Default: `b_col_dim` = *n*.

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = *n*

Description

The function `imsl_f_geneig` uses the QZ algorithm to compute the 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 α and β are returned instead of λ. If β is nonzero, $\lambda = \alpha/\beta$.

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 and eigenvectors for the reduced problem are then computed.

The function `imsl_f_geneig` 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).

Examples

Example 1

In this example, the eigenvalue, λ, of system $Ax = \lambda Bx$ is computed, where

$$A = \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ -10.0 & 2.0 & 0.0 \\ 5.0 & 1.0 & 0.5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 3.0 & 3.0 & 0.0 \\ 4.0 & 0.5 & 1.0 \end{bmatrix}$$

```
#include <imsl.h>

main()
{
        int             n = 3;
        f_complex       alpha[3];
        float           beta[3];
        int             i;
        f_complex       eval[3];
        float           a[] = {1.0, 0.5, 0.0,
                              -10.0, 2.0, 0.0,
                               5.0, 1.0, 0.5};
```

```
        float           b[] = {0.5, 0.0, 0.0,
                               3.0, 3.0, 0.0,
                               4.0, 0.5, 1.0};

                                /* Compute eigenvalues */

        imsl_f_geneig (n, a, b, alpha, beta, 0);

        for (i=0; i<n; i++)
                if (beta[i] != 0.0)
                        eval[i] = imsl_c_div(alpha[i],
                                imsl_cf_convert(beta[i], 0.0));
                else
                        printf ("Infinite eigenvalue\n");

                                /* Print eigenvalues */

        imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
                          Eigenvalues
                  1                       2                       3
(     0.833,    1.993)  (    0.833,   -1.993)  (    0.500,    0.000)
```

Example 2

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

```
#include <imsl.h>

main()
{
        int             n = 3;
        f_complex       alpha[3];
        float           beta[3];
        int             i;
        f_complex       eval[3];
        f_complex      *evec;
        float           a[] = {1.0, 0.5, 0.0,
                              -10.0, 2.0, 0.0,
                               5.0, 1.0, 0.5};
        float           b[] = {0.5, 0.0, 0.0,
                               3.0, 3.0, 0.0,
                               4.0, 0.5, 1.0};

        imsl_f_geneig (n, a, b, alpha, beta,
                IMSL_VECTORS, &evec,
                0);

        for (i=0; i<n; i++)
                if (beta[i] != 0.0)
                        eval[i] = imsl_c_div(alpha[i],
                                imsl_cf_convert(beta[i], 0.0));
                else
                        printf ("Infinite eigenvalue\n");
```

```
                                /* Print eigenvalues */

        imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);

                                /* Print eigenvectors */

        imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
                                Eigenvalues
                      1                         2                         3
(     0.833,     1.993)   (     0.833,    -1.993)   (     0.500,    -0.000)

                                 Eigenvectors
                        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)
```

geneig (*complex*)

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B complex.

Synopsis

#*include* `<imsl.h>`

void `imsl_c_geneig` (*int* `n`, *f_complex* `*a`, *f_complex* `*b`, *f_complex* `*alpha`, *float* `*beta`, ..., 0)

The *double* analogue is `imsl_z_geneig`.

Required Arguments

int `n` (Input)
Number of rows and columns in A and B.

f_complex `*a` (Input)
Array of size $n \times n$ containing the coefficient matrix A.

f_complex `*b` (Input)
Array of size $n \times n$ containing the coefficient matrix B

f_complex `*alpha` (Output)
Vector of size n containing scalars α_i. If $\beta_i \neq 0$, $\lambda_i = \alpha_i/\beta_i$ for $i = 0, \ldots, n-1$ are the eigenvalues of the system.

f_complex `*beta` (Output)
Vector of size n.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `imsl_c_geneig` (*int* `n`, *f_complex* `*a`, *f_complex* `*b`, *f_complex* `*alpha`, *f_complex* `*beta`
`IMSL_VECTORS`, *f_complex* `**evec`,
`IMSL_VECTORS_USER`, *f_complex* `evecu[]`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_B_COL_DIM`, *int* `b_col_dim`,
`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim`,
0)

Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size $n \times n$ containing eigenvectors of the problem. Each vector is normalized to have Euclidean length equal to the value one. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size $n \times n$ containing the matrix of generalized eigenvectors is returned in the space evecu. Each vector is normalized to have Euclidean length equal to the value one.

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of A.
Default: `a_col_dim` =

`IMSL_B_COL_DIM`, *int* `b_col_dim` (Input)
The column dimension of B.
Default: `b_col_dim` = `n`.

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = `n`.

Description

The function `imsl_c_geneig` uses the QZ algorithm to compute the eigenvalues and eigenvectors 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 α and β are returned instead of λ. If β is nonzero, $\lambda = \alpha/\beta$.

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 and eigenvectors for the reduced problem are then computed.

The function `imsl_c_geneig` is based on the QZ algorithm due to Moler and Stewart (1973).

Examples

Example 1

In this example, the eigenvalue, λ, of system $Ax = \lambda Bx$ is solved, where

$$A = \begin{bmatrix} 1 & 0.5+i & 5i \\ -10 & 2+i & 0 \\ 5+i & 1 & 0.5+3i \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0 & 0 \\ 3+3i & 3+3i & i \\ 4+2i & 0.5+i & 1+i \end{bmatrix}$$

```
#include <imsl.h>

main()
{
        int             n = 3;
        f_complex       alpha[3];
        f_complex       beta[3];
        int             i;
        f_complex       eval[3];
        f_complex       zero = {0.0, 0.0};
        f_complex       a[] = {{1.0, 0.0}, {0.5, 1.0}, {0.0, 5.0},
                               {-10.0, 0.0}, {2.0, 1.0}, {0.0, 0.0},
                               {5.0, 1.0}, {1.0, 0.0}, {0.5, 3.0}};
        f_complex       b[] = {{0.5, 0.0}, {0.0, 0.0}, {0.0, 0.0},
                               {3.0, 3.0}, {3.0, 3.0}, {0.0, 1.0},
                               {4.0, 2.0}, {0.5, 1.0}, {1.0, 1.0}};

                                /* Compute eigenvalues */

        imsl_c_geneig (n, a, b, alpha, beta, 0);

        for (i=0; i<n; i++)
                if (!imsl_c_eq(beta[i], zero))
                        eval[i] = imsl_c_div(alpha[i], beta[i]);
                else
                        printf ("Infinite eigenvalue\n");

                                /* Print eigenvalues */

        imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

Output

```
                              Eigenvalues
                    1                         2                         3
(     -8.18,    -25.38)  (      2.18,      0.61)  (      0.12,     -0.39)
```

Example 2

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

```
#include <imsl.h>

main()
{
```

```
        int             n = 3;
        f_complex       alpha[3];
        f_complex       beta[3];
        int             i;
        f_complex       eval[3];
        f_complex      *evec;
        f_complex       zero = {0.0, 0.0};
        f_complex       a[] = {{1.0, 0.0}, {0.5, 1.0}, {0.0, 5.0},
                               {-10.0, 0.0}, {2.0, 1.0}, {0.0, 0.0},
                               {5.0, 1.0}, {1.0, 0.0}, {0.5, 3.0}};
        f_complex       b[] = {{0.5, 0.0}, {0.0, 0.0}, {0.0, 0.0},
                               {3.0, 3.0}, {3.0, 3.0}, {0.0, 1.0},
                               {4.0, 2.0}, {0.5, 1.0}, {1.0, 1.0}};

                                /* Compute eigenvalues and eigenvectors */

        imsl_c_geneig (n, a, b, alpha, beta,
                IMSL_VECTORS, & evec,
                0);

        for (i=0; i<n; i++)
                if (!imsl_c_eq(beta[i], zero))
                        eval[i] = imsl_c_div(alpha[i], beta[i]);
                else
                        printf ("Infinite eigenvalue\n");

                                /* Print eigenvalues */

        imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);

                                /*Print eigenvectors */

        imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

Output

```
                              Eigenvalues
                     1                          2                          3
(     -8.18,   -25.38)  (      2.18,     0.61)  (      0.12,    -0.39)

                               Eigenvectors
                     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)
```

Chapter 3: Interpolation and Approximation

Routines

3.1 Cubic Spline Interpolation
Derivative end conditions ..... cub_spline_interp_e_cnd 145
Shape preserving ..... cub_spline_interp_shape 152

3.2 Cubic Spline Evaluation and Integration
Evaluation and differentiation ..... cub_spline_value 157
Integration..... cub_spline_integral 160

3.3 Spline Interpolation
One-dimensional interpolation..... spline_interp 161
Knot sequence given interpolation data ..... spline_knots 167
Two-dimensional, tensor-product interpolation ..... spline_2d_interp 171

3.4 Spline Evaluation and Integration
One-dimensional evaluation and differentiation ..... spline_value 177
One-dimensional integration ..... spline_integral 180
Two-dimensional evaluation and differentiation ..... spline_2d_value 182
Two-dimensional integration ..... spline_2d_integral 186

3.5 Least-Squares Approximation and Smoothing
General functions ..... user_fcn_least_squares 189
Splines with fixed knots ..... spline_least_squares 193
Tensor-product splines with fixed knots ..... spline_2d_least_squares 199
Cubic smoothing spline ..... cub_spline_smooth 205
Splines with constraints ..... spline_lsq_constrained 209
Smooth one-dimensional data by error detection..... smooth_1d_data 216

3.6 Scattered Data Interpolation
Akima's surface-fitting method ..... scattered_2d_interp 220

3.7 Scattered Data Least Squares
Fit using radial-basis functions ..... radial_scattered_fit 225
Evaluate radial-basis fit ..... radial_evaluate 231

Usage Notes

The majority of the functions in this chapter produce cubic piecewise polynomial or general spline functions that either interpolate or approximate given data or support the evaluation and integration of these functions. Two major subdivisions of functions are provided. The cubic spline functions begin with the prefix "`cub_spline_`" and use the piecewise polynomial representation described below. The spline functions begin with the prefix "`spline_`" and use the B-spline representation described below. Most of the spline functions are based on routines in the book by de Boor (1978).

We provide a few general purpose routines for general least-squares fit to data and a routine that produces an interpolant to two-dimensional scattered data.

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 (ppoly) function is given by

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

The breakpoint sequence ξ is assumed to be strictly increasing, and we extend the ppoly function to the entire real axis by extrapolation from the first and last intervals. This representation is redundant when the ppoly 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:

$$c_{1,i+1} = p(\xi_{i+1}) = \sum_{j=1}^{k} c_{ji} \frac{(\xi_{i+1}-\xi_i)^{j-1}}{(j-1)!}$$

For smooth ppoly, we prefer to use the nonredundant representation in terms of the "basis" or B-splines, at least when such a function is first to be determined.

Splines and B-Splines

B-splines provide a particularly convenient and suitable basis for a given class of smooth ppoly functions. Such a class is specified by giving its breakpoint sequence, its order k, 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 as follows: 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, choose the first k knots equal to ξ_1 and the last k knots equal to ξ_n. This can be done because the B-splines are defined to be right continuous near ξ_1 and left continuous near ξ_n.

When the above construction is completed, a knot sequence **t** of length M is generated, and there are $m := M - k$ B-splines of order k, for example $B_0, \ldots, B_{m-1}$, spanning the ppoly functions on the interval with the indicated smoothness. That is, each ppoly function in this class has a unique representation

$$p = a_0 B_0 + a_1 B_1 + \ldots + a_{m-1} B_{m-1}$$

as a linear combination of B-splines. A B-spline is a particularly compact ppoly function. B_i is a nonnegative function that is nonzero only on the interval $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. More precisely, the support of the i-th B-spline is $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. No ppoly 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 $B_{i,k,\mathbf{t}}$ to denote the i-th B-spline of order k for the knot sequence **t**.

Cubic Splines

Cubic splines are smooth (i.e., C^1 or C^2), fourth-order ppoly functions. For historical and other reasons, cubic splines are the most heavily used ppoly functions. Therefore, we provide special functions for their construction and evaluation. These routines use the ppoly representation as described above for general ppoly functions (with $k = 4$).

We provide two cubic spline interpolation functions: `imsl_f_cub_spline_interp_e_cnd` (page 145) and `imsl_f_cub_spline_interp_shape` (page 152). The function `imsl_f_cub_spline_interp_e_cnd` 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 function by setting the second derivative to zero at both endpoints. The function `imsl_f_cub_spline_interp_shape` (page 152) is designed so that the shape of the curve matches the shape of the data. In particular, one option of this function preserves the convexity of the data while the default attempts to minimize oscillations.

It is possible that the cubic spline interpolation functions will produce unsatisfactory results. For example, the interpolant may not have the shape required by the user, or the data may be noisy and require a least-squares fit. The interpolation function `imsl_f_spline_interp` (page 161) is more flexible, as it allows you to choose the knots and order of the spline interpolant. We encourage the user to use this routine and exploit the flexibility provided.

Tensor Product Splines

The simplest method of obtaining multivariate interpolation and approximation functions is to take univariate methods and form a multivariate method via tensor products. In the case of two-dimensional spline interpolation, the derivation 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 following form.

$$\sum_{m=0}^{N_y-1}\sum_{n=0}^{N_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y)$$

Given two sets of points

$$\{x_i\}_{i=1}^{N_x}$$

and

$$\{y_i\}_{i=1}^{N_y}$$

for which the corresponding univariate interpolation problem can be solved, the tensor-product interpolation problem finds the coefficients c_{nm} so that

$$\sum_{m=0}^{N_y-1}\sum_{n=0}^{N_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) = f_{ij}$$

This problem can be solved efficiently by repeatedly solving univariate interpolation problems as described in de Boor (1978, p. 347). Three-dimensional interpolation can be handled in an analogous manner. This chapter provides functions that compute the two-dimensional, tensor-product spline coefficients given two-dimensional interpolation data (`imsl_f_spline_2d_interp` (page 171) and that compute the two-dimensional, tensor-product spline coefficients for a tensor-product, least-squares problem (`imsl_f_spline_2d_least_squares` (page 199)). In addition, we provide evaluation, differentiation, and integration functions for the two-dimensional, tensor-product spline functions. The relevant functions are `imsl_f_spline_2d_value` (page 182)) and `imsl_f_spline_2d_integral` (page 186).

Scattered Data Interpolation

The IMSL C/Math/Library provides one function, `imsl_f_scattered_2d_interp` (page 220), that returns values of an interpolant to scattered data in the plane. This function is based on work by Akima (1978), which uses C^1 piecewise quintics on a triangular mesh.

Least Squares

The IMSL C/Math/Library includes functions for smoothing noisy data. The function `imsl_f_user_fcn_least_squares` (page 189) computes regressions with user-supplied functions. The function `imsl_f_spline_least_squares` (page 193) computes a least-squares fit using splines with fixed knots or variable knots. These functions produce cubic spline, least-squares fit by default. Optional arguments allow the user to choose the order and the knot sequence. IMSL C/Math/Library also includes a tensor-product spline regression function (`imsl_f_spline_2d_least_squares`), (page 199), mentioned above. The function `imsl_f_radial_scattered_fit` (page 225) computes an approximation to scattered data in $\mathbf{R}^N$ using radial-basis functions.

In addition to the functions listed above, several functions in Chapter 10, "Statistics and Random Number Generation", provide for polynomial regression and general linear regression.

Smoothing by Cubic Splines

One ''smoothing spline'' function is provided. The default action of `imsl_f_cub_spline_smooth` estimates a smoothing parameter by cross-validation and then returns the cubic spline that smooths the data. If the user wishes to supply a smoothing parameter, then this function returns the appropriate cubic spline.

Structures for Splines and Piecewise Polynomials

This optional section includes more details concerning the structures for splines and piecewise polynomials.

A spline may be viewed as a mapping with domain $\mathbf{R}^d$ and target $\mathbf{R}^r$, where d and r are positive integers. For this version of the IMSL C/Math/Library, only $r = 1$ is supported. Thus, if s is a spline, then for some d and r

$$s : \mathbf{R}^d \rightarrow \mathbf{R}^r$$

This implies that such a spline s must have d knot sequences and orders (one for each domain dimension). Thus, associated with s, we have knots and orders

$$\mathbf{t}^0, \ldots, \mathbf{t}^{d-1}$$

$$k_0, \ldots, k_{d-1}$$

The precise form of the spline follows:

$$s(x) = (s_0(x), \ldots, s_{r-1}(x)) \qquad x = (x_1, \ldots, x_d) \in \mathbf{R}^d$$

where the following equation is true.

$$s_i(x) := \sum_{j_{d-1}=0}^{n_{d-1}-1} \cdots \sum_{j_0=0}^{n_0-1} c^i_{j_0,\ldots,j_{d-1}} B_{j_0,k_0,\mathbf{t}^0} \cdots B_{j_{d-1},k_{d-1},\mathbf{t}^{d-1}}$$

Note that n_i is the number of knots in $\mathbf{t}^i$ minus the order k_i.

We store all the information for a spline in one structure called *Imsl_f_spline*. (If the type is double, then the structure name is *Imsl_d_spline*, and the *float* becomes *double*.) The specification for this structure follows:

```
typedef struct {
    int   domain_dim;
    int   target_dim;
    int   *order;
    int   *num_coef;
    int   *num_knots;
    float **knots;
    float **coef;
} Imsl_f_spline;
```

Explicitly, if `sp` is a pointer to *Imsl_f_spline*, then

`sp-> domain_dim`	$= d$
`sp-> target_dim`	$= r$
`sp-> order [i]`	$= k_i \quad i = 0, \ldots, d-1$
`sp-> num_coef [i]`	$= m_i \quad i = 0, \ldots, d-1$
`sp-> num_knots [i]`	$= n_i + k_i \quad i = 0, \ldots, d-1$
`sp-> knots [i] [j]`	$= t_j^i \; I = 0, \ldots, d-1 \; j = 0, \ldots, n_i + k_i - 1$
`sp-> coef [i] [j]`	$= c_j^i \; I = 0, \ldots, r-1 \; j = j_0 + j_1 \, n_0 + \ldots + j_{d-1} \, n_0 \ldots n_{d-2}$

For `ppoly` functions, we view a `ppoly` as a mapping with domain $\mathbf{R}^d$ and target $\mathbf{R}^r$ where d and r are positive integers. Thus, if p is a `ppoly`, then for some d and r the following is true.

$$p : \mathbf{R}^d \rightarrow \mathbf{R}^r$$

For this version of the C/Math/Library, only $r = 1$ is supported. This implies that such a `ppoly` p must have d breakpoint sequences and orders (one for each domain dimension). Thus, associated with p, we have breakpoints and orders

$$\xi^1, \ldots, \xi^d$$
$$k_1, \ldots, k_d$$

The precise form of the `ppoly` follows:

$$p(x) = (p_0(x), \ldots, p_r(x)) \; x = (x_1, \ldots, x_d) \in \mathbf{R}^d$$

where

$$p_i(x) := \sum_{l_d=0}^{k_d-1} \cdots \sum_{l_1=0}^{k_1-1} c^i_{L^1, \ldots, L^d, l_1, \ldots, l_d} \frac{\left(x_1 - \xi^1_{L^1}\right)}{l_1!} \cdots \frac{\left(x_d - \xi^d_{L^d}\right)^{l_d}}{l_d!}$$

with

$$L^j := \max \{1, \min \{M^j, n_j - 1\}\}$$

where M^J is chosen so that

$$\xi^j_{M^j} \le x_j < \xi^j_{M^{j+1}} \qquad j = 1, \ldots, d$$

with

$$\xi^j_0 = -\infty \text{ and } \xi^j_{n_j+1} = \infty$$

Note that n_j is the number of breakpoints in ξ^j.

We store all the information for a ppoly in one structure called *Imsl_f_ppoly*. (If the type is *double*, then the structure name is *Imsl_d_ppoly*, and the *float* becomes *double*.) The following is the specification for this structure.

```
typedef struct {
    int   domain_dim;
    int   target_dim;
    int   *order;
    int   *num_coef;
    int   *num_breakpoints;
    float **breakpoints;
    float **coef;
} Imsl_f_ppoly;
```

In particular, if `ppoly` is a pointer to the structure of type *Imsl_f_ppoly*, then

`ppoly-> domain_dim`	$= d$
`ppoly-> target_dim`	$= r$
`ppoly-> order [i]`	$= k_i \quad i = 0, \ldots, d-1$
`ppoly-> num_coef [i]`	$= k_i (n_i - 1) \quad i = 0, \ldots, d-1$
`ppoly-> num_breakpoints [i]`	$= n_i \quad i = 0, \ldots, d-1$
`ppoly-> breakpoints [i] [j]`	$= \xi_j^i \quad i = 0, \ldots, d-1 \quad j = 0, \ldots, n_i - 1$
`ppoly->coef [i[ [j]`	$= c_j^i \, i = 0, \ldots, r-1 j = 0, \ldots, k_0(n_0 - 1) \ldots k_{d-1}(n_{d-1} - 1)$

cub_spline_interp_e_cnd

Computes a cubic spline interpolant, specifying various endpoint conditions. The default interpolant satisfies the "not-a-knot" condition.

Synopsis

```
#include <imsl.h>
Imsl_f_ppoly *imsl_f_cub_spline_interp_e_cnd (int ndata,
      float xdata[], float fdata[], ..., 0)
```

The type *Imsl_d_ppoly* function is `imsl_d_cub_spline_interp_e_cnd`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the interpolation problem.

float `fdata[]` (Input)
Array with `ndata` components containing the ordinates for the interpolation problem.

Return Value

A pointer to the structure that represents the cubic spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

```
#include <imsl.h>
```

Imsl_f_ppoly `*imsl_f_cub_spline_interp_e_cnd` (*int* `ndata`, *float* `xdata[]`,
float `fdata[]`,
`IMSL_LEFT`, *int* `ileft`, *float* `left`,
`IMSL_RIGHT`, *int* `iright`, *float* `right`,
`IMSL_PERIODIC`,
0)

Optional Arguments

`IMSL_LEFT`, *int* `ileft`, *float* `left` (Input)
Set the value for the first or second derivative of the interpolant at the left endpoint. If `ileft` = *i*, then the interpolant *s* satisfies

$$s^{(i)}(x_L) = \texttt{left}$$

where x_L is the leftmost abscissa. The only valid values for `ileft` are 1 or 2.

`IMSL_RIGHT`, *int* `iright`, *float* `right` (Input)
Set the value for the first or second derivative of the interpolant at the right endpoint. If `iright` = *i*, then the interpolant *s* satisfies

$$s^{(i)}(x_R) = \texttt{right}$$

where x_R is the rightmost abscissa. The only valid values for `iright` are 1 or 2.

`IMSL_PERIODIC`
Compute the C^2 periodic interpolant to the data. That is, we require

$$s^{(i)}(x_L) = s^{(i)}(x_R) \qquad i = 0, 1, 2$$

where *s*, x_L, and x_R are defined above.

Description

The function `imsl_f_cub_spline_interp_e_cnd` computes a C^2 cubic spline interpolant to a set of data points (x_i, f_i) for $i = 0, \ldots,$ `ndata` $- 1 = n$. The breakpoints of the spline are the abscissas. We emphasize here that for all the univariate interpolation functions, the abscissas need not be sorted. Endpoint conditions are to be selected by the user. The user may specify "not-a-knot" or first derivative or second derivative at each endpoint, or C^2 periodicity may be requested (see de Boor 1978, Chapter 4). If no defaults are selected, then the "not-a-knot" spline interpolant is computed. If the `IMSL_PERIODIC` keyword is selected, then all other keywords are ignored; and a C^2 periodic interpolant is computed. In this case, if the `fdata` values at the left and right endpoints are not the same, then a warning message is issued; and we set the right value equal to the left. If `IMSL_LEFT` or `IMSL_RIGHT` are selected (in the absence of `IMSL_PERIODIC`), then the user has the ability to select the values of the first or second derivative at either endpoint. The default case (when the keyword is not used) is the

"not-a-knot" condition on that endpoint. Thus, when no optional arguments are chosen, this function produces the "not-a-knot" interpolant.

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

$$\|f - s\|_{[\xi_0, \xi_n]} \leq C \left\| f^{(4)} \right\|_{[\xi_0, \xi_n]} |\xi|^4$$

where

$$|\xi| := \max_{i=0, \ldots, n-1} |\xi_{i+1} - \xi_i|$$

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

The return value for this function is a pointer to the structure *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*ppoly`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this function. For example, the following code sequence evaluates this spline at x and returns the value in y

```
y = imsl_f_cub_spline_value (x, ppoly, 0)
```

The difference between the default ("not-a-knot") spline and the interpolating cubic spline, which has first derivative set to 1 at the left end and the second derivative set to –90 at the right end, is illustrated in the following figure.

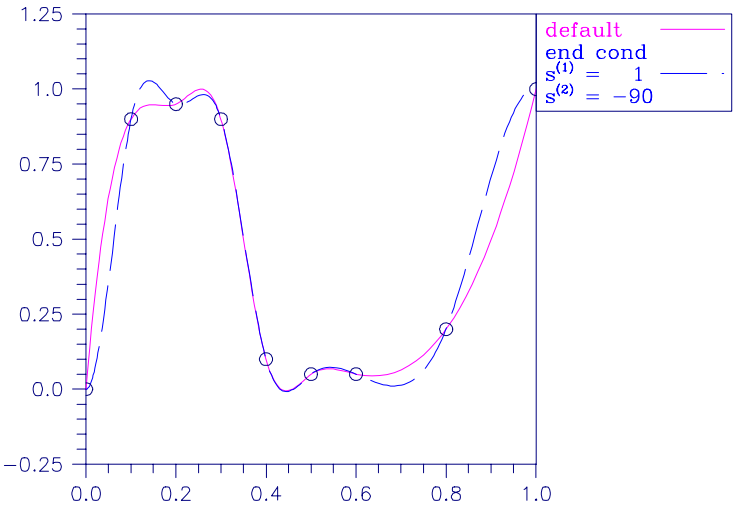


Figure 3-1 Two Interpolating Splines

Examples

Example 1

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. Since we are using the default settings, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA  11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *ppoly;
                                /* Compute xdata and fdata  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute  cubic spline interpolant */
    ppoly = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);

                                /* Print results */
    printf("     x          F(x)       Interpolant    Error\n");
    for (i = 0;  i < 2*NDATA-1;  i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x,ppoly,0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                fabs(F(x)-y));
    }
}
```

Output

```
    x         F(x)     Interpolant    Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.809       0.1270
0.100       0.997        0.997       0.0000
0.150       0.778        0.723       0.0552
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.549       0.0228
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.843       0.0162
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.441       0.0093
0.500       0.938        0.938       0.0000
0.550       0.923        0.903       0.0199
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.315       0.0049
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.938       0.0295
0.800      -0.537       -0.537       0.0000
```

```
0.850       0.183       0.148       0.0347
0.900       0.804       0.804       0.0000
0.950       0.994       1.086       0.0926
1.000       0.650       0.650       0.0000
```

Example 2

In this example, 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.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  NDATA  11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int              i, ileft, iright;
    float            left, right, x, y, fdata[NDATA], xdata[NDATA];
    Imsl_f_ppoly     *pp;
                               /* Compute xdata and fdata  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
                                /* Specify end conditions */
    ileft   = 1;
    left   = 0.0;
    iright  = 2;
    right  =-225.0*sin(15.0);
                                /* Compute cubic spline interpolant */
    pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                        IMSL_LEFT,  ileft,  left,
                                        IMSL_RIGHT, iright, right,
                                        0);
                                /* Print results for first half */
                                /* of interval */
    printf("     x          F(x)       Interpolant    Error\n\n");
    for (i=0;  i<NDATA;  i++){
        x = (float)(i)/(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x,pp,0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                  fabs(F(x)-y));
    }
}
```

Output

```
    x         F(x)     Interpolant    Error
0.000       0.000       0.000       0.0000
0.050       0.682       0.438       0.2441
0.100       0.997       0.997       0.0000
0.150       0.778       0.822       0.0442
0.200       0.141       0.141       0.0000
```

```
0.250      -0.572       -0.575       0.0038
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.836       0.0233
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.439       0.0111
0.500       0.938        0.938       0.0000
```

Example 3

This example computes 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, the exact function values are computed with the values of the spline.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  NDATA  11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int              i, ileft, iright;
    float            left, right, x, y, fdata[NDATA],
                     xdata[NDATA];
    Imsl_f_ppoly     *pp;
                                /* Compute xdata and fdata  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
                                /* Specify end conditions  */
    ileft   = 2;
    left   = 0.0;
    iright  = 2;
    right  = 0.0;
                                /* Compute cubic spline interpolant  */
    pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                        IMSL_LEFT,  ileft,  left,
                                        IMSL_RIGHT, iright, right,
                                        0);
                                /* Print results for first half */
                                /* of interval */
    printf("     x         F(x)       Interpolant    Error\n\n");
    for (i = 0;  i < NDATA;  i++){
        x = (float)(i)/(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x,pp,0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                 fabs(F(x)-y));
    }
}
```

Output

```
    x         F(x)     Interpolant    Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.667       0.0150
0.100       0.997        0.997       0.0000
0.150       0.778        0.761       0.0172
```

```
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.559       0.0126
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.840       0.0189
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.440       0.0098
0.500       0.938        0.938       0.0000
```

Example 4

This example computes the cubic spline interpolant to a functions, and imposes the periodic end conditions $s(a) = s(b)$, $s'(a) = s'(b)$, and $s''(a) = s''(b)$, where a is the leftmost abscissa and b is the rightmost abscissa.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  NDATA  11
                                /* Define function*/
#define F(x)    (float)(sin(x))

main()
{
    int                 i;
    float               x, y, twopi, fdata[NDATA], xdata[NDATA];
    Imsl_f_ppoly        *pp;
                                /* Compute xdata and fdata */
    twopi = 2.0*imsl_f_constant("pi", 0);
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = twopi*(float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
    fdata[NDATA-1] = fdata[0];
                                /* Compute periodic cubic spline */
                                /* interpolant */
    pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                        IMSL_PERIODIC,
                                        0);
                                /* Print results for first half */
                                /* of interval */
    printf("     x         F(x)       Interpolant    Error\n\n");
    for (i = 0;  i < NDATA;  i++){
        x = (twopi/20.)*i;
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n",x,F(x), y,
                                                fabs(F(x)-y));
    }
}
```

Output

```
    x         F(x)     Interpolant    Error
0.000       0.000        0.000       0.0000
0.314       0.309        0.309       0.0001
0.628       0.588        0.588       0.0000
0.942       0.809        0.809       0.0004
1.257       0.951        0.951       0.0000
1.571       1.000        1.000       0.0004
1.885       0.951        0.951       0.0000
```

```
2.199       0.809        0.809       0.0004
2.513       0.588        0.588       0.0000
2.827       0.309        0.309       0.0001
3.142      -0.000       -0.000       0.0000
```

Warning Errors

IMSL_NOT_PERIODIC	The data is not periodic. The rightmost fdata value is set to the leftmost fdata value.

Fatal Errors

IMSL_DUPLICATE_XDATA_VALUES	The xdata values must be distinct.

cub_spline_interp_shape

Computes a shape-preserving cubic spline.

Synopsis

#include `<imsl.h>`

Imsl_f_ppoly `*imsl_f_cub_spline_interp_shape` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`, …, 0)

The type *Imsl_d_ppoly* function is `imsl_d_cub_spline_interp_shape`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the interpolation problem.

float `fdata[]` (Input)
Array with `ndata` components containing the ordinates for the interpolation problem.

Return Value

A pointer to the structure that represents the cubic spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_ppoly `*imsl_f_cub_spline_interp_shape` (*int* `ndata`,
float `xdata[]`, *float* `fdata[]`,
`IMSL_CONCAVE`,
`IMSL_CONCAVE_ITMAX`, *int* `itmax`,
0)

Optional Arguments

`IMSL_CONCAVE`

This option produces a cubic interpolant that will preserve the concavity of the data.

`IMSL_CONCAVE_ITMAX`, *int* `itmax` (Input)

This option allows the user to set the maximum number of iterations of Newton's Method. Default: `itmax` = 25.

Description

The function `imsl_f_cub_spline_interp_shape` computes a C^1 cubic spline interpolant to a set of data points(x_i, f_i) for $i = 0, \ldots,$ `ndata` $- 1 = n$. The breakpoints of the spline are the abscissas. This computation is based on a method by Akima (1970) to combat wiggles in the interpolant. Endpoint conditions are automatically determined by the program; see Akima (1970) or de Boor (1978).

If the optional argument `IMSL_CONCAVE` is chosen, then this function computes a cubic spline interpolant to the data. 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

$$\int_{x_1}^{x_n} \left(g''\right)^2$$

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, refer to Michelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this function 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. This function should be used when it is important to preserve the convex and concave regions implied by the data.

Both methods are nonlinear, and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.) This explains the theoretical error estimate below.

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 either of the above spline interpolants. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{[\xi_0, \xi_m]} \leq C \left\| f^{(2)} \right\|_{[\xi_0, \xi_m]} |\xi|^2$$

where

$$|\xi| := \max_{i=0, \ldots, m-1} |\xi_{i+1} - \xi_i|$$

and ξ_m is the last breakpoint.

The return value for this function is a pointer of the type *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*ppoly`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y*.

```
y = imsl_f_cub_spline_value (x, ppoly, 0)
```

The difference between the convexity-preserving spline and Akima's spline is illustrated in the following figure. Note that the convexity-preserving interpolant exhibits linear segments where the convexity constraints are binding.

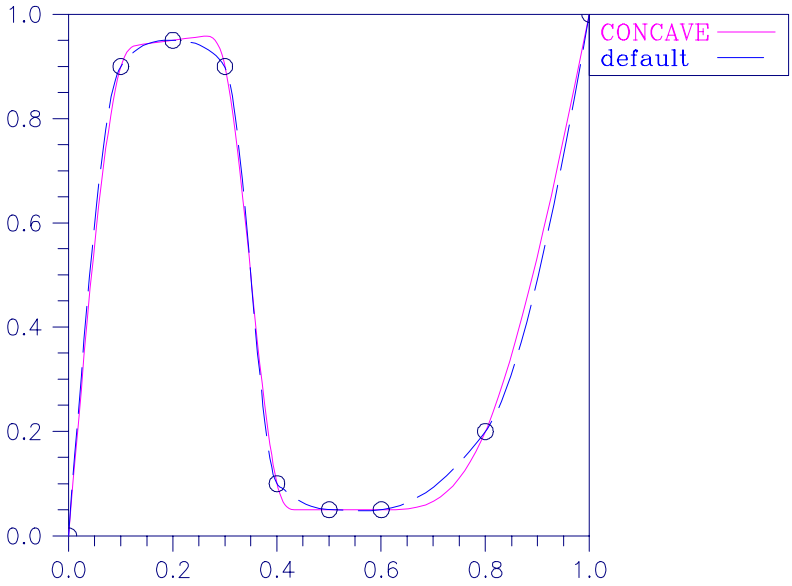


Figure 3-2 Two Shape-Preserving Splines

Examples

Example 1

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.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  NDATA  11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
```

```
    Imsl_f_ppoly        *pp;
                                /* Compute xdata and fdata */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = (float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant  */
    pp = imsl_f_cub_spline_interp_shape(NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x         F(x)       Interpolant    Error\n\n");
    for (i = 0;  i < 2*NDATA-1;  i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
    }
}
```

Output

```
  x         F(x)       Interpolant    Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.818       0.1360
0.100       0.997        0.997       0.0000
0.150       0.778        0.615       0.1635
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.478       0.0934
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.812       0.0464
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.386       0.0645
0.500       0.938        0.938       0.0000
0.550       0.923        0.854       0.0683
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.276       0.0433
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.889       0.0789
0.800      -0.537       -0.537       0.0000
0.850       0.183        0.149       0.0338
0.900       0.804        0.804       0.0000
0.950       0.994        0.932       0.0613
1.000       0.650        0.650       0.0000
```

Example 2

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.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int                 i;
```

```
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *pp;
                                /* Compute xdata and fdata */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant  */
    pp = imsl_f_cub_spline_interp_shape(NDATA, xdata, fdata,
                                        IMSL_CONCAVE,
                                        0);
                               /* Print results */
    printf("     x         F(x)       Interpolant    Error\n\n");
    for (i = 0;  i < 2*NDATA-1;  i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
    }
}
```

Output

```
    x         F(x)     Interpolant    Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.667       0.0150
0.100       0.997        0.997       0.0000
0.150       0.778        0.761       0.0172
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.559       0.0126
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.840       0.0189
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.440       0.0098
0.500       0.938        0.938       0.0000
0.550       0.923        0.902       0.0208
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.311       0.0086
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.952       0.0156
0.800      -0.537       -0.537       0.0000
0.850       0.183        0.200       0.0174
0.900       0.804        0.804       0.0000
0.950       0.994        0.892       0.1020
1.000       0.650        0.650       0.0000
```

Warning Errors

IMSL_MAX_ITERATIONS_REACHED	The maximum number of iterations has been reached. The best approximation is returned.

Fatal Errors

IMSL_DUPLICATE_XDATA_VALUES	The `xdata` values must be distinct.

cub_spline_value

Computes the value of a cubic spline or the value of one of its derivatives.

Synopsis

#include `<imsl.h>`

float `imsl_f_cub_spline_value` (*float* `x`, `Imsl_f_ppoly *ppoly`, ..., 0)

The type *double* function is `imsl_d_cub_spline_value`.

Required Arguments

float `x` (Input)
Evaluation point for the cubic spline.

Imsl_f_ppoly `*ppoly` (Input)
Pointer to the piecewise polynomial structure that represents the cubic spline.

Return Value

The value of a cubic spline or one of its derivatives at the point *x*. If no value can be computed, then NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_cub_spline_value` (*float* `x`, *Imsl_f_ppoly* `*ppoly`,
`IMSL_DERIV`, *int* `deriv`,
`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value`,
`IMSL_GRID_USER`, *int* `n`, *float* `*xvec`, *float* `value_user[]`,
0)

Optional Arguments

`IMSL_DERIV`, *int* `deriv` (Input)
Let *d* = `deriv` and let *s* be the cubic spline that is represented by the structure `*ppoly`, then this option produces the *d*-th derivative of *s* at *x*, $s^{(d)}(x)$.

`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value` (Input/Output)
The array `xvec` of length `n` contains the points at which the cubic spline is to be evaluated. The *d*-th derivative of the spline at the points in `xvec` is returned in `value`.

`IMSL_GRID_USER`, *int* `n`, *float* `*xvec`, *float* `value_user[]` (Input/Output)
The array `xvec` of length `n` contains the points at which the cubic spline is to be evaluated. The *d*-th derivative of the spline at the points in `xvec` is returned in the user-supplied space `value_user`.

Description

The function `imsl_f_cub_spline_value` computes the value of a cubic spline or one of its derivatives. The first and last pieces of the cubic spline are extrapolated. As a result, the cubic spline structures returned by the cubic spline routines are defined and can be evaluated on the entire real line. This routine is based on the routine PPVALU by de Boor (1978, p. 89).

Examples

Example 1

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. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA    11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *pp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                               /* Compute cubic spline interpolant */
    pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x          F(x)       Interpolant    Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                  fabs(F(x)-y));
    }
}
```

Output

```
    x        F(x)      Interpolant    Error
0.250      -0.572       -0.549       0.0228
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.843       0.0162
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.441       0.0093
0.500       0.938        0.938       0.0000
0.550       0.923        0.903       0.0199
```

```
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.315       0.0049
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.938       0.0295
```

Example 2

Recall that in the first example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. This example compares the values of the first derivatives.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA    11
                                /* Define functions */
#define F(x)     (float)(sin(15.0*x))
#define FP(x)    (float)(15.*cos(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *pp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata,fdata, 0);
                                /* Print results */
    printf("     x         FP(x)      Interpolant   Deriv Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp,
                                    IMSL_DERIV, 1,
                                    0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, FP(x), y,
                                                 fabs(FP(x)-y));
        }
}
```

Output

```
   x        FP(x)   Interpolant   Deriv Error
0.250     -12.308      -12.559       0.2510
0.300      -3.162       -3.218       0.0560
0.350       7.681        7.796       0.1151
0.400      14.403       13.919       0.4833
0.450      13.395       13.530       0.1346
0.500       5.200        5.007       0.1926
0.550      -5.786       -5.840       0.0535
0.600     -13.667      -13.201       0.4660
0.650     -14.214      -14.393       0.1798
0.700      -7.133       -6.734       0.3990
0.750       3.775        3.911       0.1359
```

cub_spline_integral

Computes the integral of a cubic spline.

Synopsis

#include `<imsl.h>`

float `imsl_f_cub_spline_integral` (*float* `a`, *float* `b`, *Imsl_f_ppoly* `*ppoly`)

The type *double* function is `imsl_d_cub_spline_integral`.

Required Arguments

float `a` (Input)

float `b` (Input)
Endpoints for integration.

Imsl_f_ppoly `*ppoly` (Input)
Pointer to the piecewise polynomial structure that represents the cubic spline.

Return Value

The integral from *a* to *b* of the cubic spline. If no value can be computed, then NaN is returned.

Description

The function `imsl_f_cub_spline_integral` computes the integral of a cubic spline from *a* to *b*.

$$\int_a^b s(x)\,dx$$

Example

In this example, a cubic spline interpolant to a function *f* is computed. The values of the integral of this spline are then compared with the exact integral values. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   21
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))
                                /* Integral from 0 to x */
#define FI(x)   (float)((1.-cos(15.0*x))/15.)

main()
{
    int                 i;
```

```
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *pp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x         FI(x)      Interpolant   Integral Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_integral(0.0, x, pp);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, FI(x), y,
                                                    fabs(FI(x)-y));
        }
}
```

Output

```
  x         FI(x)   Interpolant   Integral Error
0.250       0.121        0.121       0.0001
0.275       0.104        0.104       0.0001
0.300       0.081        0.081       0.0001
0.325       0.056        0.056       0.0001
0.350       0.033        0.033       0.0001
0.375       0.014        0.014       0.0002
0.400       0.003        0.003       0.0002
0.425       0.000        0.000       0.0002
0.450       0.007        0.007       0.0002
0.475       0.022        0.022       0.0001
0.500       0.044        0.044       0.0001
0.525       0.068        0.068       0.0001
0.550       0.092        0.092       0.0001
0.575       0.113        0.113       0.0001
0.600       0.127        0.128       0.0001
0.625       0.133        0.133       0.0001
0.650       0.130        0.130       0.0001
0.675       0.118        0.118       0.0001
0.700       0.098        0.098       0.0001
0.725       0.075        0.075       0.0001
0.750       0.050        0.050       0.0001
```

spline_interp

Compute a spline interpolant.

Synopsis

#include `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_interp` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`, ..., 0)

The type *Imsl_d_spline* function is `imsl_d_spline_interp`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the interpolation problem.

float `fdata[]` (Input)
Array with `ndata` components containing the ordinates of the interpolation problem.

Return Value

A pointer to the structure that represents the spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_interp` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`,
`IMSL_ORDER`, *int* `order`,
`IMSL_KNOTS`, *float* `knots[]`,
0)

Optional Arguments

`IMSL_ORDER`, *int* `order` (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: `order` = 4, i.e., cubic splines

`IMSL_KNOTS`, *float* `knots[]` (Input)
This option requires the user to provide the knots.
Default: knots are selected by the function `imsl_f_spline_knots` using its defaults.

Description

Given the data points x = `xdata`, f = `fdata`, and the number n = `ndata` of elements in `xdata` and `fdata`, the default action of `imsl_f_spline_interp` computes a cubic (k = 4) spline interpolant s to the data using the default knot sequence generated by `imsl_f_spline_knots`.

The optional argument `IMSL_ORDER` allows the user to choose the order of the spline interpolant. The optional argument `IMSL_KNOTS` allows user specification of knots.

The function `imsl_f_spline_interp` is based on the routine SPLINT by de Boor (1978, p. 204).

First, `imsl_f_spline_interp` 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 $i = 0$ to $n - 1$.

The following preliminary checks are performed on the data. We verify that

$$x_i < x_{i+1} \qquad i = 0, \ldots, n-2$$
$$\mathbf{t}_i < \mathbf{t}_{i+k} \qquad i = 0, \ldots, n-1$$
$$\mathbf{t}_i < \mathbf{t}_{i+1} \qquad i = 0, \ldots, n+k-2$$

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 $\mathbf{t}_{k-1} \le x_i \le \mathbf{t}_n$ for $i = 0$ to $n - 1$. 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,

  $$B_{j-k+1}, \ldots, B_j \quad \text{where } j \text{ satisfies } \mathbf{t}_j \le x_i < \mathbf{t}_{j+1}$$

  be well-defined (that is, $j - k + 1 \ge 0$).

General conditions are not known for the exact behavior of the error in spline interpolation; however, if $\mathbf{t}$ and x are selected properly and the data points arise from the values of a smooth (say C^k) function f, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. The maximum absolute error satisfies

$$\|f - s\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} \le C \left\| f^{(k)} \right\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} |\mathbf{t}|^k$$

where

$$|\mathbf{t}| := \max_{i=k-1, \ldots, n-1} |\mathbf{t}_{i+1} - \mathbf{t}_i|$$

For more information on this problem, see de Boor (1978, Chapter 13) and his reference. This function can be used in place of the IMSL function `imsl_f_cub_spline_interp`.

The return value for this function is a pointer of type *Imsl_f_spline*. The calling program must receive this in a pointer *Imsl_f_spline* `*sp`. This structure contains all the information to determine the spline (stored as a linear combination of B-splines) that is computed by this function. For example, the following code sequence evaluates this spline at x and returns the value in y.

```
y = imsl_f_spline_value (x, sp, 0)
```

Three spline interpolants of order 2, 3, and 5 are plotted. These splines use the default knots.

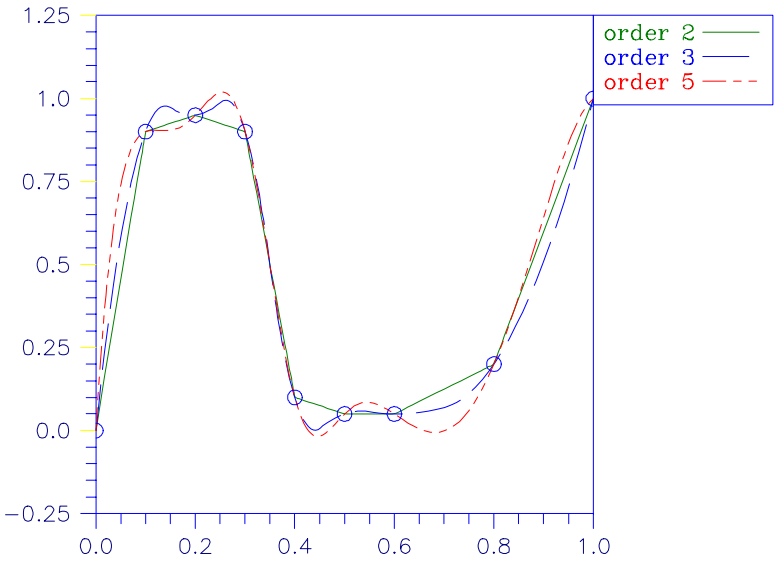


Figure 3-3 Three Spline Interpolants

Examples

Example 1

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. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int                 i;
    float               xdata[NDATA], fdata[NDATA], x, y;
    Imsl_f_spline       *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x          F(x)        Interpolant    Error\n");
    for (i = 0;  i < 2*NDATA-1;  i++){
```

```
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_spline_value(x, sp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
    }
}
```

Output

```
    x        F(x)    Interpolant      Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.809       0.1270
0.100       0.997        0.997       0.0000
0.150       0.778        0.723       0.0552
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.549       0.0228
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.843       0.0162
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.441       0.0093
0.500       0.938        0.938       0.0000
0.550       0.923        0.903       0.0199
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.315       0.0049
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.938       0.0295
0.800      -0.537       -0.537       0.0000
0.850       0.183        0.148       0.0347
0.900       0.804        0.804       0.0000
0.950       0.994        1.086       0.0926
1.000       0.650        0.650       0.0000
```

Example 2

Recall that in the first example, a cubic spline interpolant to a function f is computed. The values of this spline are then compared with the exact function values. This example chooses to use a quadratic ($k = 3$) and a quintic $k = 6$ spline interpolant to the data instead of the default values.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA  11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

main()
{
    int                 i, order;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_spline       *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
    for (order =3; order<7; order += 3) {
```

```
                                /* Compute cubic spline interpolant */
        sp = imsl_f_spline_interp (NDATA, xdata, fdata,
                                   IMSL_ORDER, order,
                                   0);
                                /* Print results */
        printf("\nThe order of the spline is %d\n", order);
        printf("     x         F(x)       Interpolant    Error\n");
        for (i = NDATA/2;  i < 3*NDATA/2;  i++){
            x = (float) i /(float)(2*NDATA-2);
            y = imsl_f_spline_value(x,sp,0);
            printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
        }
    }
}
```

Output

```
The order of the spline is 3
     x         F(x)       Interpolant    Error
   0.250      -0.572       -0.542       0.0299
   0.300      -0.978       -0.978       0.0000
   0.350      -0.859       -0.819       0.0397
   0.400      -0.279       -0.279       0.0000
   0.450       0.450        0.429       0.0210
   0.500       0.938        0.938       0.0000
   0.550       0.923        0.879       0.0433
   0.600       0.412        0.412       0.0000
   0.650      -0.320       -0.305       0.0149
   0.700      -0.880       -0.880       0.0000
   0.750      -0.968       -0.922       0.0459

The order of the spline is 6
     x         F(x)       Interpolant    Error
   0.250      -0.572       -0.573       0.0016
   0.300      -0.978       -0.978       0.0000
   0.350      -0.859       -0.856       0.0031
   0.400      -0.279       -0.279       0.0000
   0.450       0.450        0.448       0.0020
   0.500       0.938        0.938       0.0000
   0.550       0.923        0.922       0.0003
   0.600       0.412        0.412       0.0000
   0.650      -0.320       -0.322       0.0025
   0.700      -0.880       -0.880       0.0000
   0.750      -0.968       -0.959       0.0090
```

Warning Errors

IMSL_ILL_COND_INTERP_PROB	The interpolation matrix is ill-conditioned. The solution might not be accurate.

Fatal Errors

IMSL_DUPLICATE_XDATA_VALUES	The `xdata` values must be distinct.
IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.

IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.
IMSL_KNOT_XDATA_INTERLACING	The i-th smallest element of xdata (x_i) must satisfy $\mathbf{t}_i \le x_i < \mathbf{t}_{i+order}$ where $\mathbf{t}$ is the knot sequence.
IMSL_XDATA_TOO_LARGE	The array xdata must satisfy $\text{xdata}_i \le \mathbf{t}_{ndata}$, for $i = 1, \ldots,$ ndata.
IMSL_XDATA_TOO_SMALL	The array xdata must satisfy $\text{xdata}_i \ge \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ ndata.

spline_knots

Computes the knots for a spline interpolant

Synopsis

```
#include <imsl.h>
float *imsl_f_spline_knots (int ndata, float xdata[], ..., 0)
```

The type *double* function is `imsl_d_spline_knots`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the interpolation problem.

Return Value

A pointer to the knots. If the knots cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

```
#include <imsl.h>
float *imsl_f_spline_knots (int ndata, float xdata[],
        IMSL_ORDER, int order,
        IMSL_OPT,
        IMSL_OPT_ITMAX, int itmax,
        IMSL_RETURN_USER, float knots[],
        0)
```

Optional Arguments

`IMSL_ORDER`, *int* `order` (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: `order` = 4, i.e., cubic splines

`IMSL_OPT`
This option produces knots that satisfy an optimality criterion.

`IMSL_OPT_ITMAX`, *int* `itmax` (Input)
This option allows the user to set the maximum number of iterations of Newton's method.
Default: `itmax` = 10

`IMSL_RETURN_USER`, *float* `knots[]` (Output)
This option requires the user to provide the space for the return knots. For example, the user could declare `float knots`[100]; and pass in `knots`. The return value is then also set to `knots`.

Description

Given the data points x = `xdata`, the order of the spline k = `order`, and the number n = `ndata` of elements in `xdata`, the default action of `imsl_f_spline_knots` returns a pointer to a knot sequence that is appropriate for interpolation of data on x by splines of order k (the default order is $k = 4$). The knot sequence is contained in its first $n + k$ positions. If k is even, and we assume that the entries in the input vector x are increasing, then the resulting knot sequence $\mathbf{t}$ is returned as

$$\mathbf{t}_i = x_0 \qquad \text{for } i = 0, \ldots, k-1$$
$$\mathbf{t}_i = x_{i-k/2-1} \qquad \text{for } i = k, \ldots, n-1$$
$$\mathbf{t}_i = x_{n-1} \qquad \text{for } i = n, \ldots, n+k-1$$

There is some discussion concerning this selection of knots in de Boor (1978, p. 211). If k is odd, then $\mathbf{t}$ is returned as

$$\mathbf{t}_i = x_0 \qquad \text{for } i = 0, \ldots, k-1$$
$$\mathbf{t}_i = (x_{i-\frac{k-1}{2}-1} + x_{i-1-\frac{k-2}{2}})/2 \qquad \text{for } i = k, \ldots, n-1$$
$$\mathbf{t}_i = x_{n-1} \qquad \text{for } i = n, \ldots, n = k-1$$

It is not necessary to sort the values in `xdata`.

If the option `IMSL_OPT` is selected, then the knot sequence returned minimizes the constant c 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, p. 204), which in turn is based on a theorem of Micchelli et al. (1976).

Examples

Example 1

In this example, knots for a cubic spline are generated and printed. Notice that the knots are stacked at the endpoints and that the second and next to last data points are not knots.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA  6

main()
{
    int         i;
    float       *knots, xdata[NDATA];

    for(i = 0;  i < NDATA;  i++)
        xdata[i] = i;
    knots = imsl_f_spline_knots(NDATA, xdata, 0);
    imsl_f_write_matrix("The knots for the cubic spline are:\n",
                        1, NDATA+4, knots,
                        IMSL_COL_NUMBER_ZERO,
                        0);
}
```

Output

```
      The knots for the cubic spline are:

          0           1           2           3           4           5
          0           0           0           0           2           3

          6           7           8           9
          5           5           5           5
```

Example 2

This is a continuation of the examples for `imsl_f_spline_interp` (page 161). Recall that in these examples, a cubic spline interpolant to a function f is computed first. The values of this spline are then compared with the exact function values. The second example uses a quadratic ($k = 3$) and a quintic ($k = 6$) spline interpolant to the data. Now, instead of using the default knots, select the "optimal" knots as described above. Notice that the error is actually worse in this case.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>
```

```
#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int                 i, order;
    float               fdata[NDATA], xdata[NDATA], *knots, x, y;
    Imsl_f_spline       *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
    for(order = 3;  order < 7;  order += 3) {
        knots = imsl_f_spline_knots(NDATA, xdata, IMSL_ORDER, order,
                                    IMSL_OPT,
                                    0);
                                /* Compute spline interpolant */
        sp = imsl_f_spline_interp (NDATA, xdata,fdata,
                                   IMSL_ORDER, order,
                                   IMSL_KNOTS, knots,
                                   0);
                                /* Print results */
        printf("\nThe order of the spline is %d\n", order);
        printf("     x         F(x)       Interpolant    Error\n");
        for (i = NDATA/2;  i < 3*NDATA/2;  i++) {
            x = (float) i /(float)(2*NDATA-2);
            y = imsl_f_spline_value(x, sp, 0);
            printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
        }
    }
}
```

Output

```
The order of the spline is 3
     x         F(x)       Interpolant    Error
   0.250      -0.572       -0.543       0.0290
   0.300      -0.978       -0.978       0.0000
   0.350      -0.859       -0.819       0.0401
   0.400      -0.279       -0.279       0.0000
   0.450       0.450        0.429       0.0210
   0.500       0.938        0.938       0.0000
   0.550       0.923        0.879       0.0433
   0.600       0.412        0.412       0.0000
   0.650      -0.320       -0.305       0.0150
   0.700      -0.880       -0.880       0.0000
   0.750      -0.968       -0.920       0.0478

The order of the spline is 6
     x         F(x)       Interpolant    Error
   0.250      -0.572       -0.578       0.0061
   0.300      -0.978       -0.978       0.0000
   0.350      -0.859       -0.854       0.0054
   0.400      -0.279       -0.279       0.0000
   0.450       0.450        0.448       0.0019
```

```
0.500       0.938        0.938       0.0000
0.550       0.923        0.920       0.0022
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.317       0.0020
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.966       0.0023
```

Warning Errors

IMSL_NO_CONV_NEWTON	Newton's method iteration did not converge.

Fatal Errors

IMSL_DUPLICATE_XDATA_VALUES	The xdata values must be distinct.
IMSL_ILL_COND_LIN_SYS	Interpolation matrix is singular. The xdata values may be too close together.

spline_2d_interp

Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data.

Synopsis

#include <imsl.h>

Imsl_f_spline *imsl_f_spline_2d_interp (*int* num_xdata, *float* xdata[], *int* num_ydata, *float* ydata[], *float* fdata[], ..., 0)

The type *Imsl_d_spline* function is imsl_d_spline_2d_interp.

Required Arguments

int num_xdata (Input)
Number of data points in the *X* direction.

float xdata[] (Input)
Array with num_xdata components containing the data points in the *X* direction.

int num_ydata (Input)
Number of data points in the *Y* direction.

float ydata[] (Input)
Array with num_ydata components containing the data points in the *Y* direction.

float fdata[] (Input)
Array of size num_xdata × num_ydata containing the values to be interpolated. fdata[i][j] is the value at (xdata[i], ydata[j]).

Return Value

A pointer to the structure that represents the tensor-product spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_2d_interp` (*int* `num_xdata`, *float* `xdata[]`, *int* `num_ydata`, *float* `ydata[]`, *float* `fdata[]`,
`IMSL_ORDER`, *int* `xorder`, *int* `yorder`,
`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]`,
`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim`,
0)

Optional Arguments

`IMSL_ORDER`, *int* `xorder`, *int* `yorder` (Input)
This option is used to communicate the order of the spline subspace.
Default: `xorder`, `yorder = 4`, (i.e., tensor-product cubic splines)

`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]` (Input)
This option requires the user to provide the knots. The default knots are selected by the function `imsl_f_spline_knots` using its defaults.

`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim` (Input)
The column dimension of the matrix `fdata`.
Default: `fdata_col_dim = num_ydata`

Description

The function `imsl_f_spline_2d_interp` computes a tensor-product spline interpolant. The tensor-product spline interpolant to data $\{(x_i, y_j, f_{ij})\}$, where $0 \le i \le n_x - 1$ and $0 \le j \le n_y - 1$ has the form

$$\sum_{m=0}^{n_y-1} \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x) B_{m,k_y,\mathbf{t}_y}(y)$$

where k_x and k_y are the orders of the splines. These numbers are defaulted to be 4, but can be set to any positive integer using the keyword, `IMSL_ORDER`. Likewise, $\mathbf{t}_x$ and $\mathbf{t}_y$ are the corresponding knot sequences (`xknots` and `yknots`). These values are defaulted to the knots returned by `imsl_f_spline_knots`. The algorithm requires that

$$\mathbf{t}_x(k_x - 1) \le x_i \le \mathbf{t}_x(n_x) \quad 0 \le i \le n_x - 1$$

$$\mathbf{t}_y(k_y - 1) \le y_j \le \mathbf{t}_y(n_y - 1) \qquad 0 \le j \le n_y - 1$$

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

$$\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i)$$

note that for each fixed i from 0 to $n_x - 1$, we have n_y linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=0}^{n_y-1} h_{mi} B_{m,k_y,\mathbf{t}_y}(y_i) = f_{ij}$$

$$\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) B_{m,k_y,\mathbf{t}_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i)$$

note that for each fixed i from 1 to $n_x - 1$, we have $n_y - 1$ linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=0}^{n_y-1} h_{mi} B_{m,k_y,\mathbf{t}_y}(y_i) = f_{ij}$$

The same matrix appears in all of the equations above:

$$\left[B_{m,k_y,\mathbf{t}_y}(y_j)\right] \qquad 1 \le m, j \le n_y - 1$$

Thus, only factor this matrix once and then apply this factorization to the n_x right-hand sides. Once this is done and h_{mi} is computed, then solve for the coefficients c_{nm} using the relation

$$\sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) = h_{mi}$$

for m from 0 to $n_y - 1$, which again involves one factorization and n_y solutions to the different right-hand sides. The function `imsl_f_spline_2d_interp` is based on the routine SPLI2D by de Boor (1978, p. 347).

The return value for this function is a pointer to the structure `imsl_f_spline`. The calling program must receive this in a pointer `imsl_f_spline *sp`. This structure contains all the information to determine the spline (stored in B-spline format) that is computed by this procedure. For example, the following code sequence evaluates this spline at (x,y) and returns the value in z.

```
z = imsl_f_spline_2d_value (x, y, sp, 0);
```

Examples

Example 1

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.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           11
#define OUTDATA          2
                                /* Define function  */
#define F(x, y)          (float)(x*x*x+y*y)

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline        *sp;
                                /* Set up grid  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = ydata[i] = (float)i / ((float)(NDATA-1));
    }
    for (i = 0;  i < NDATA;  i++) {
        for (j = 0;  j < NDATA;  j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                        ydata, fdata, 0);
                                /* Print results */
    printf("     x       y        F(x, y)    Interpolant    Error \n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float) i / (float) (OUTDATA);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float) j / (float) (OUTDATA);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                   x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

Output

```
   x       y       F(x, y)    Interpolant     Error
0.000   0.000       0.000        0.000       0.0000
0.000   0.500       0.250        0.250       0.0000
0.500   0.000       0.125        0.125       0.0000
0.500   0.500       0.375        0.375       0.0000
```

Example 2

Recall that in the first 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.

Notice that the first interpolant with `order` = 3 does not reproduce the cubic data, while the second interpolant with `order` = 6 does reproduce the data.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           7
#define OUTDATA         4
                                /* Define function  */
#define F(x,y)          (float)(x*x*x+y*y)

main()
{
    int                 i, j, num_xdata, num_ydata, order;
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline        *sp;
                                /* Set up grid  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = ydata[i] = (float) i / ((float) (NDATA - 1));
    }
    for (i = 0;  i < NDATA;  i++) {
        for (j = 0; j < NDATA; j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;

    for(order = 3;  order < 7;  order += 3) {
                                /* Compute tensor-product interpolant */
        sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                          ydata, fdata,
                                     IMSL_ORDER, order, order,
                                     0);
                                /* Print results */
       printf("\nThe order of the spline is %d \n", order);
       printf("     x       y        F(x, y)    Interpolant    Error\n");
       for (i = 0;  i < OUTDATA;  i++) {
            x = (float) i / (float) (OUTDATA);
            for (j = 0;  j < OUTDATA;  j++) {
                y = (float) j / (float) (OUTDATA);
                z = imsl_f_spline_2d_value(x, y, sp, 0);
                printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f   \n",
                       x, y, F(x,y), z, fabs(F(x,y)-z));
            }
        }
    }
}
```

Output

```
The order of the spline is 3
     x       y        F(x, y)    Interpolant    Error
   0.000   0.000       0.000        0.000       0.0000
   0.000   0.250       0.062        0.063       0.0000
   0.000   0.500       0.250        0.250       0.0000
   0.000   0.750       0.562        0.562       0.0000
   0.250   0.000       0.016        0.016       0.0002
   0.250   0.250       0.078        0.078       0.0002
   0.250   0.500       0.266        0.266       0.0002
   0.250   0.750       0.578        0.578       0.0002
   0.500   0.000       0.125        0.125       0.0000
   0.500   0.250       0.188        0.188       0.0000
   0.500   0.500       0.375        0.375       0.0000
   0.500   0.750       0.688        0.687       0.0000
   0.750   0.000       0.422        0.422       0.0002
   0.750   0.250       0.484        0.484       0.0002
   0.750   0.500       0.672        0.672       0.0002
   0.750   0.750       0.984        0.984       0.0002

The order of the spline is 6
     x       y        F(x, y)    Interpolant    Error
   0.000   0.000       0.000        0.000       0.0000
   0.000   0.250       0.062        0.063       0.0000
   0.000   0.500       0.250        0.250       0.0000
   0.000   0.750       0.562        0.562       0.0000
   0.250   0.000       0.016        0.016       0.0000
   0.250   0.250       0.078        0.078       0.0000
   0.250   0.500       0.266        0.266       0.0000
   0.250   0.750       0.578        0.578       0.0000
   0.500   0.000       0.125        0.125       0.0000
   0.500   0.250       0.188        0.188       0.0000
   0.500   0.500       0.375        0.375       0.0000
   0.500   0.750       0.688        0.688       0.0000
   0.750   0.000       0.422        0.422       0.0000
   0.750   0.250       0.484        0.484       0.0000
   0.750   0.500       0.672        0.672       0.0000
   0.750   0.750       0.984        0.984       0.0000
```

Warning Errors

IMSL_ILL_COND_INTERP_PROB	The interpolation matrix is ill-conditioned. The solution might not be accurate.

Fatal Errors

IMSL_XDATA_NOT_INCREASING	The `xdata` values must be strictly increasing.
IMSL_YDATA_NOT_INCREASING	The `ydata` values must be strictly increasing.
IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.

IMSL_KNOT_DATA_INTERLACING	The *i*-th smallest element of the data arrays xdata and ydata must satisfy $\mathbf{t}_i \le \text{data}_i < \mathbf{t}_{i+order}$, where **t** is the knot sequence.
IMSL_DATA_TOO_LARGE	The data arrays xdata and ydata must satisfy $\text{data}_i \le \mathbf{t}_{num\_data}$, for $i = 1, \ldots,$ num_data.
IMSL_DATA_TOO_SMALL	The data arrays xdata and ydata must satisfy $\text{data}_i \ge \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ num_data.

spline_value

Computes the value of a spline or the value of one of its derivatives.

Synopsis

#include <imsl.h>

float imsl_f_spline_value (*float* x, *Imsl_f_spline* *sp, …, 0)

The type *double* function is imsl_d_spline_value.

Required Arguments

float x (Input)
Evaluation point for the spline.

Imsl_f_spline *sp (Input)
Pointer to the structure that represents the spline.

Return Value

The value of a spline or one of its derivatives at the point *x*. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float imsl_f_spline_value (*float* x, *Imsl_f_spline* *sp,
IMSL_DERIV, *int* deriv,
IMSL_GRID, *int* n, *float* *xvec, *float* **value,
IMSL_GRID_USER, *int* n, *float* *xvec, *float* value_user[],
0)

Optional Arguments

IMSL_DERIV, *int* deriv (Input)
Let *d* = deriv and let *s* be the spline that is represented by the structure *sp. Then, this option produces the *d*-th derivative of *s* at *x*, $s^{(d)}(x)$.
Default: deriv = 0

IMSL_GRID, *int* n, *float* *xvec, *float* **value (Input/Output)

The argument xvec is the array of length n containing the points at which the spline is to be evaluated. The *d*-th derivative of the spline at the points in xvec is returned in value.

IMSL_GRID_USER *int* n, *float* *xvec, *float* value_user[] (Input/Output)

The argument xvec is the array of length n containing the points at which the spline is to be evaluated. The *d*-th derivative of the spline at the points in xvec is returned in value_user.

Description

The function imsl_f_spline_value computes the value of a spline or one of its derivatives. This function is based on the routine BVALUE by de Boor (1978, p. 144).

Examples

Example 1

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. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_spline       *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    sp = imsl_f_spline_interp (NDATA, xdata,fdata, 0);
                                /* Print results */
    printf("     x         F(x)       Interpolant    Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_spline_value(x, sp, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
    }
}
```

Output

```
   x         F(x)      Interpolant    Error
0.250      -0.572       -0.549       0.0228
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.843       0.0162
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.441       0.0093
0.500       0.938        0.938       0.0000
0.550       0.923        0.903       0.0199
0.600       0.412        0.412       0.0000
0.650      -0.320       -0.315       0.0049
0.700      -0.880       -0.880       0.0000
0.750      -0.968       -0.938       0.0295
```

Example 2

Recall that in the first example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. This example compares the values of the first derivatives.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))
#define FP(x)   (float)(15.*cos(15.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_spline        *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x         FP(x)      Interpolant   Deriv Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_spline_value(x, sp, IMSL_DERIV, 1, 0);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f   \n", x, FP(x), y,
                                                    fabs(FP(x)-y));
    }
}
```

Output

```
   x         FP(x)   Interpolant   Deriv Error
0.250     -12.308      -12.559       0.2510
0.300      -3.162       -3.218       0.0560
0.350       7.681        7.796       0.1151
0.400      14.403       13.919       0.4833
```

```
0.450      13.395       13.530       0.1346
0.500       5.200        5.007       0.1926
0.550      -5.786       -5.840       0.0535
0.600     -13.667      -13.201       0.4660
0.650     -14.214      -14.393       0.1798
0.700      -7.133       -6.734       0.3990
0.750       3.775        3.911       0.1359
```

Fatal Errors

`IMSL_KNOT_MULTIPLICITY`	Multiplicity of the knots cannot exceed the order of the spline.
`IMSL_KNOT_NOT_INCREASING`	The knots must be nondecreasing.

spline_integral

Computes the integral of a spline.

Synopsis

#include `<imsl.h>`

float `imsl_f_spline_integral` (*float* `a`, *float* `b`, `Imsl_f_spline *sp`)

The type *double* function is `imsl_d_spline_integral`.

Required Arguments

float `a` (Input)

float `b` (Input)
Endpoints for integration.

Imsl_f_spline `*sp` (Input)
Pointer to the structure that represents the spline.

Return Value

The integral of a spline. If no value can be computed, then NaN is returned.

Description

The function `imsl_f_spline_integral` computes the integral of a spline from *a* to *b*

$$\int_a^b s(x)dx$$

This routine uses the identity (22) on page 151 of de Boor (1978).

Example

In this example, a cubic spline interpolant to a function *f* is computed. The values of the integral of this spline are then compared with the exact integral values. Since the

default settings are used, the interpolant is determined by the “not-a-knot” condition (see de Boor 1978).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA    21
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))
                                /* Integral from 0 to x */
#define FI(x)    (float)((1.-cos(15.0*x))/15.)

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_spline       *sp;
                                /* Set up a grid */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                /* Compute cubic spline interpolant */
    sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                                /* Print results */
    printf("     x         FI(x)      Interpolant   Integral Error\n");
    for (i = NDATA/2;  i < 3*NDATA/2;  i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_spline_integral(0.0, x, sp);
        printf("  %6.3f  %10.3f   %10.3f   %10.4f   \n", x, FI(x), y,
                                                    fabs(FI(x)-y));
    }
}
```

Output

```
   x        FI(x)    Interpolant  Integral Error
0.250       0.121        0.121       0.0001
0.275       0.104        0.104       0.0001
0.300       0.081        0.081       0.0001
0.325       0.056        0.056       0.0001
0.350       0.033        0.033       0.0001
0.375       0.014        0.014       0.0002
0.400       0.003        0.003       0.0002
0.425       0.000        0.000       0.0002
0.450       0.007        0.007       0.0002
0.475       0.022        0.022       0.0001
0.500       0.044        0.044       0.0001
0.525       0.068        0.068       0.0001
0.550       0.092        0.092       0.0001
0.575       0.113        0.113       0.0001
0.600       0.127        0.128       0.0001
0.625       0.133        0.133       0.0001
0.650       0.130        0.130       0.0001
0.675       0.118        0.118       0.0001
0.700       0.098        0.098       0.0001
0.725       0.075        0.075       0.0001
0.750       0.050        0.050       0.0001
```

Warning Errors

IMSL_SPLINE_SMLST_ELEMNT	The data arrays xdata and ydata must satisfy $\texttt{data}_i \le \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ num_data.
IMSL_SPLINE_EQUAL_LIMITS	The upper and lower endpoints of integration are equal. The indefinite integral is zero.
IMSL_LIMITS_LOWER_TOO_SMALL	The left endpoint is less than $\mathbf{t}_{order-1}$. Integration occurs only from $\mathbf{t}_{order-1}$ to b.
IMSL_LIMITS_UPPER_TOO_SMALL	The right endpoint is less than $\mathbf{t}_{order-1}$. Integration occurs only from $\mathbf{t}_{order-1}$ to a.
IMSL_LIMITS_UPPER_TOO_BIG	The right endpoint is greater than $\mathbf{t}_{spline\_space\_dim-1}$. Integration occurs only from a to $\mathbf{t}_{spline\_space\_dim-1}$.
IMSL_LIMITS_LOWER_TOO_BIG	The left endpoint is greater than $\mathbf{t}_{spline\_space\_dim-1}$. Integration occurs only from b to $\mathbf{t}_{spline\_space\_dim-1}$.

Fatal Errors

IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.

spline_2d_value

Computes the value of a tensor-product spline or the value of one of its partial derivatives.

Synopsis

#include <imsl.h>

float imsl_f_spline_2d_value (*float* x, *float* y, *Imsl_f_spline* *sp, …, 0)

The type *double* function is imsl_d_spline_2d_value.

Required Arguments

float x (Input)

float y (Input)
The (x, y) coordinates of the evaluation point for the tensor-product spline.

Imsl_f_spline *sp (Input)
Pointer to the structure that represents the spline.

Return Value

The value of a tensor-product spline or one of its derivatives at the point (x, y).

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_spline_2d_value` (*float* `x`, *float* `y`, *Imsl_f_spline* `*sp`,
`IMSL_DERIV`, *int* `x_partial`, *int* `y_partial`,
`IMSL_GRID`, *int* `nx`, *float* `*xvec`, *int* `ny`, *float* `*yvec`,
float `**value`,
`IMSL_GRID_USER`, *int* `nx`, *float* `*xvec`, *int* `ny`, *float* `*yvec`, *float* `value_user[]`,
0)

Optional Arguments

`IMSL_DERIV`, *int* `x_partial`, *int* `y_partial` (Input)
Let p = `x_partial` and q = `y_partial`, and let s be the spline that is represented by the structure `*sp`, then this option produces the (p, q)-th derivative of s at (x, y), $s^{(p, q)}$ (x, y).
Default: `x_partial` = `y_partial` = 0

`IMSL_GRID`, *int* `nx`, *float* `*xvec`, *int* `ny`, *float* `*yvec`, *float* `**value` (Input/Output)
The argument `xvec` is the array of length `nx` containing the X coordinates at which the spline is to be evaluated. The argument `yvec` is the array of length `ny` containing the Y coordinates at which the spline is to be evaluated. The value of the spline on the `nx` by `ny` grid is returned in `value`.

`IMSL_GRID_USER`, *int* `nx`, *float* `*xvec`, *int* `ny`, *float* `*yvec`, *float* `value_user[]` (Input/Output)
The argument `xvec` is the array of length `nx` containing the X coordinates at which the spline is to be evaluated. The argument `yvec` is the array of length `ny` containing the Y coordinates at which the spline is to be evaluated. The value of the spline on the `nx` by `ny` grid is returned in the user-supplied space `value_user`.

Description

The function `imsl_f_spline_2d_value` computes the value of a tensor-product spline or one of its derivatives. This function is based on the discussion in de Boor (1978, pp. 351–353).

Examples

Example 1

In this example, a spline interpolant s to a function f is constructed. Using the procedure `imsl_f_spline_2d_interp` to compute the interpolant,

imsl_f_spline_2d_value is employed to compute $s(x, y)$. The values of this partial derivative and the error are computed on a 4×4 grid and then displayed.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           11
#define OUTDATA          2
                                /* Define function  */
#define F(x,y)          (float)(x*x*x+y*y)

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline       *sp;
                                /* Set up grid  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = ydata[i] = (float) i / ((float) (NDATA - 1));
    }
    for (i = 0;  i < NDATA;  i++) {
        for (j = 0;  j < NDATA;  j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                        ydata, fdata, 0);
                                /* Print results */
    printf("     x       y       F(x, y)        Value       Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float) (1+i) / (float) (OUTDATA+1);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float) (1+j) / (float) (OUTDATA+1);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                   x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

Output

```
    x       y      F(x, y)       Value        Error
0.333   0.333       0.148        0.148       0.0000
0.333   0.667       0.481        0.481       0.0000
0.667   0.333       0.407        0.407       0.0000
0.667   0.667       0.741        0.741       0.0000
```

Example 2

In this example, a spline interpolant s to a function f is constructed.Using function imsl_f_spline_2d_interp to compute the interpolant, then imsl_f_spline_2d_value 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.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           11
#define OUTDATA          2
                                /* Define function  */
#define F(x, y)         (float)(x*x*x*y*y)
#define F21(x,y)        (float)(6.*x*2.*y)

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline       *sp;
                                /* Set up grid  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = ydata[i] = (float)i / ((float)(NDATA-1));
    }
    for (i = 0;  i < NDATA;  i++) {
        for (j = 0;  j < NDATA;  j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                    ydata, fdata, 0);
                                /* Print results */
    printf("     x       y       F21(x, y)   21InterpDeriv   Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float) (1+i) / (float) (OUTDATA+1);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float) (1+j) / (float) (OUTDATA+1);
            z = imsl_f_spline_2d_value(x, y, sp,
                                       IMSL_DERIV, 2, 1,
                                       0);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                   x, y, F21(x, y), z, fabs(F21(x,y)-z));
        }
    }
}
```

Output

```
    x       y     F21(x, y)   21InterpDeriv   Error
0.333   0.333       1.333        1.333       0.0000
0.333   0.667       2.667        2.667       0.0000
0.667   0.333       2.667        2.667       0.0000
0.667   0.667       5.333        5.333       0.0001
```

Warning Errors

IMSL_X_NOT_WITHIN_KNOTS	The value of x does not lie within the knot sequence.

IMSL_Y_NOT_WITHIN_KNOTS	The value of y does not lie within the knot sequence.

Fatal Errors

IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.

spline_2d_integral

Evaluates the integral of a tensor-product spline on a rectangular domain.

Synopsis

#include <imsl.h>

float imsl_f_spline_2d_integral (*float* a, *float* b, *float* c, *float* d, *Imsl_f_spline* *sp)

The type *double* function is imsl_d_spline_2d_integral.

Required Arguments

float a (Input)

float b (Input)
The integration limits for the first variable of the tensor-product spline.

float c (Input)

float d (Input)
The integration limits for the second variable of the tensor-product spline.

Imsl_f_spline *sp (Input)
Pointer to the structure that represents the spline.

Return Value

The value of the integral of the tensor-product spline over the rectangle $[a, b] \times [c, d]$. If no value can be computed, NaN is returned.

Description

The function imsl_f_spline_2d_integral computes the integral of a tensor-product spline. If s is the spline, then this function returns

$$\int_a^b \int_c^d s(x, y) dy dx$$

This function uses the (univariate integration) identity (22) in de Boor (1978, p. 151)

$$\int_{t_0}^{x} \sum_{i=0}^{n-1} \alpha_i B_{i,k}(\tau) d\tau = \sum_{i=0}^{r-1} \left[\sum_{j=0}^{i} \alpha_j \frac{\mathbf{t}_{j+k} - \mathbf{t}_j}{k} \right] B_{i,k+1}(x)$$

where $\mathbf{t}_0 \le x \le \mathbf{t}_r$.
It assumes (for all knot sequences) that the first and last k knots are stacked, that is, $\mathbf{t}_0 = \ldots = \mathbf{t}_{k-1}$ and $\mathbf{t}_n = \ldots = \mathbf{t}_{n+k-1}$, where k is the order of the spline in the x or y direction.

Example

This example integrates a two-dimensional, tensor-product spline over the rectangle $[0, x] \times [0, y]$.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           11
#define OUTDATA          2
                                /* Define function  */
#define F(x,y)          (float)(x*x*x+y*y)
                                /* The integral of F from 0 to x */
                                /* and 0 to y */
#define FI(x,y)         (float)(y*x*x*x*x/4. + x*y*y*y/3.)

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline       *sp;
                                /* Set up grid  */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = ydata[i] = (float) i / ((float)(NDATA-1));
    }
    for (i = 0;  i < NDATA;  i++) {
        for (j = 0; j < NDATA;  j++) {
            fdata[i][j] = F(xdata[i],ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                        ydata, fdata, 0);
                                /* Print results */
    printf("     x       y       FI(x, y)       Integral    Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float) (1+i) / (float) (OUTDATA+1);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float) (1+j) / (float) (OUTDATA+1);
            z = imsl_f_spline_2d_integral(0.0, x, 0.0, y, sp);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                   x, y, FI(x, y), z, fabs(FI(x,y)-z));
        }
    }
}
```

Output

```
    x       y     FI(x, y)      Integral       Error
0.333   0.333       0.005         0.005      0.0000
0.333   0.667       0.035         0.035      0.0000
0.667   0.333       0.025         0.025      0.0000
0.667   0.667       0.099         0.099      0.0000
```

Warning Errors

IMSL_SPLINE_LEFT_ENDPT	The left endpoint of *X* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to b.
IMSL_SPLINE_RIGHT_ENDPT	The right endpoint of *X* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to a.
IMSL_SPLINE_LEFT_ENDPT_1	The left endpoint of *X* integration is not within the knot sequence. Integration occurs only from b to $\mathbf{t}_{spline\_space\_dim-1}$.
IMSL_SPLINE_RIGHT_ENDPT_1	The right endpoint of *X* integration is not within the knot sequence. Integration occurs only from a to $\mathbf{t}_{spline\_space\_dim-1}$.
IMSL_SPLINE_LEFT_ENDPT_2	The left endpoint of *Y* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to d.
IMSL_SPLINE_RIGHT_ENDPT_2	The right endpoint of *Y* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to c.
IMSL_SPLINE_LEFT_ENDPT_3	The left endpoint of *Y* integration is not within the knot sequence. Integration occurs only from d to $\mathbf{t}_{spline\_space\_dim-1}$.
IMSL_SPLINE_RIGHT_ENDPT_3	The right endpoint of *Y* integration is not within the knot sequence. Integration occurs only from c to $\mathbf{t}_{spline\_space\_dim-1}$.

Fatal Errors

IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.

user_fcn_least_squares

Computes a least-squares fit using user-supplied functions.

Synopsis

#include <imsl.h>

float *imsl_f_user_fcn_least_squares (*float* fcn (*int* k, *float* x), *int* nbasis, *int* ndata, *float* xdata[], *float* ydata[], ..., 0)

The type *double* function is imsl_d_user_fcn_least_squares.

Required Arguments

float fcn (*int* k, *float* x) (Input)
User-supplied function that defines the subspace from which the least-squares fit is to be performed. The *k*-th basis function evaluated at x is f(k, x) where *k* = 1, 2, …, nbasis.

int nbasis (Input)
Number of basis functions.

int ndata (Input)
Number of data points.

float xdata[] (Input)
Array with ndata components containing the abscissas of the least-squares problem.

float ydata[] (Input)
Array with ndata components containing the ordinates of the least-squares problem.

Return Value

A pointer to the vector containing the coefficients of the basis functions. If a fit cannot be computed, then NULL is returned. To release this space, use free.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_user_fcn_least_squares (), *int* nbasis, *int* ndata, *float* xdata[], *float* ydata[],
IMSL_RETURN_USER, *float* coef[],
IMSL_INTERCEPT, *float* *intercept,
IMSL_SSE, *float* *ssq_err,
IMSL_WEIGHTS, *float* weights[],
IMSL_FCN_W_DATA, *float* fcn ( ), *void* *data,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `coef[]` (Output)
The coefficients are stored in the user-supplied array.

`IMSL_INTERCEPT`, *float* `*intercept` (Output)
This option adds an intercept to the model. Thus, the least-squares fit is computed using the user-supplied basis functions augmented by the constant function. The coefficient of the constant function is stored in `intercept`.

`IMSL_SSE`, *float* `*ssq_err` (Output)
This option returns the error sum of squares.

`IMSL_WEIGHTS`, *float* `weights[]` (Input)
This option requires the user to provide the weights.
Default: all weights equal one

`IMSL_FCN_W_DATA`, `fcn` (*int* `k`, *float* `x`, *float* `*data`), *void* `*data`, (Input)
User supplied function that defines the subspace from which the least-squares fit is to be performed, which also accepts a pointer to data that is supplied by the user. `.data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_user_fcn_least_squares` computes a best least-squares approximation to given univariate data of the form

$$\{(x_i, f_i)\}_{i=0}^{n-1}$$

by M basis functions

$$\{F_j\}_{j=1}^{M}$$

(where M = `nbasis`). In particular, the default for this function returns the coefficients a which minimize

$$\sum_{i=0}^{n-1} w_i \left[f_i - \sum_{j=1}^{M} a_{j-1} F_j(x_i) \right]^2$$

where w = `weights`, n = `ndata`, x = `xdata`, and f = `ydata`.

If the optional argument `IMSL_INTCERCEPT` is chosen, then an intercept is placed in the model, and the coefficients a, returned by `imsl_f_user_fcn_least_squares`, minimize the error sum of squares as indicated below.

$$\sum_{i=0}^{n-1} w_i \left[f_i - \text{intercept} - \sum_{j=1}^{M} a_{j-1} F_j(x_i) \right]^2$$

Examples

Example 1

This example fits the following two functions (indexed by δ):

$$1 + \sin x + 7\sin 3x + \delta\varepsilon$$

where ε is a 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]. Four basis functions are used: 1, sin*x*, sin2*x*, sin3*x*.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

float            fcn(int n, float x);

main()
{
    int         nbasis = 4, i, delta;
    float       ydata[NDATA], xdata[NDATA], *random, *coef;
                                /* Generate random numbers */
    imsl_random_seed_set(1234567);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for(delta = 0;  delta < 2;  delta++) {
        for (i = 0; i < NDATA; i++) {
            xdata[i] = 6.*(float)i /((float)(NDATA-1));
            ydata[i] = F(xdata[i]) + (delta)*2.*(random[i]-.5);
        }
        coef = imsl_f_user_fcn_least_squares(fcn, nbasis, NDATA, xdata,
                                                          ydata, 0);
        printf("\nFor delta = %1d", delta);
        imsl_f_write_matrix("the computed coefficients are\n",
                            1, nbasis,  coef, 0);
    }
}


float fcn(int n, float x)
{
    return  (n == 1) ? 1.0 : sin((n-1)*x);
}
```

Output

```
For delta = 0
        the computed coefficients are

         1           2           3           4
         1           1          -0           7

For delta = 1
        the computed coefficients are
```

```
          1           2           3           4
      0.979       0.998       0.096       6.839
```

Example 2

Recall that the first example fitted the following two functions (indexed by δ):

$$1 + \sin x + 7\sin 3x + \delta\varepsilon$$

where ε is a 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]. Previously, the four basis functions were used: 1, sin*x*, sin2*x*, sin3*x*. This example uses the four basis functions: sin*x*, sin2*x*, sin3*x*, sin4*x*, combined with the intercept option.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)     (float)(1.+ sin(x)+7.*sin(3.0*x))

float            fcn(int n, float x);

main()
{
    int         nbasis = 4, i, delta;
    float       ydata[NDATA], xdata[NDATA], *random, *coef, intercept;
                                  /* Generate random numbers */
    imsl_random_seed_set(1234567);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for(delta = 0;  delta < 2;  delta++){
        for (i = 0;  i < NDATA;  i++) {
            xdata[i] = 6.*(float)i /((float)(NDATA-1));
            ydata[i] = F(xdata[i]) + (delta)*2.*(random[i]-.5);
        }
        coef = imsl_f_user_fcn_least_squares(fcn, nbasis, NDATA, xdata,
                                                                ydata,
                                             IMSL_INTERCEPT, &intercept,
                                             0);
        printf("\nFor delta = %1d\n", delta);
        printf("The predicted intercept value is  %10.3f\n" ,
                intercept);
        imsl_f_write_matrix("the computed coefficients are\n",
                            1, nbasis,  coef, 0);
    }
}


float fcn(int n, float x)
{
    return sin(n*x);
}
```

Output

```
For delta = 0
The predicted intercept value is       1.000

        the computed coefficients are

         1           2           3           4
         1           0           7          -0

For delta = 1
The predicted intercept value is       0.978

        the computed coefficients are

         1           2           3           4
     0.998       0.097       6.841       0.075
```

Warning Errors

IMSL_LINEAR_DEPENDENCE	Linear dependence of the basis functions exists. One or more components of coef are set to zero.
IMSL_LINEAR_DEPENDENCE_CONST	Linear dependence of the constant function and basis functions exists. One or more components of coef are set to zero.

Fatal Errors

IMSL_NEGATIVE_WEIGHTS_2	All weights must be greater than or equal to zero.

spline_least_squares

Computes a least-squares spline approximation.

Synopsis

#*include* <imsl.h>

Imsl_f_spline *imsl_f_spline_least_squares (*int* ndata, *float* xdata[], *float* fdata[], *int* spline_space_dim, ..., 0)

The type *Imsl_d_spline* function is imsl_d_spline_least_squares.

Required Arguments

int ndata (Input)
Number of data points.

float xdata[] (Input)
Array with ndata components containing the abscissas of the least-squares problem.

float `fdata[]` (Input)
Array with `ndata` components containing the ordinates of the least-squares problem.

int `spline_space_dim` (Input)
The linear dimension of the spline subspace. It should be smaller than `ndata` and greater than or equal to `order` (whose default value is 4).

Return Value

A pointer to the structure that represents the spline fit. If a fit cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_least_squares` (*int* `ndata`, *float* `xdata[]`,
float `fdata[]`, *int* `spline_space_dim`,
`IMSL_SSE`, *float* `*sse_err`,
`IMSL_WEIGHTS`, *float* `weights[]`,
`IMSL_ORDER`, *int* `order`,
`IMSL_KNOTS`, *float* `knots[]`,
`IMSL_OPTIMIZE`,
0)

Optional Arguments

`IMSL_SSE`, *float* `*sse` (Output)
This option places the weighted error sum of squares in the place pointed to by `sse`.

`IMSL_WEIGHTS`, *float* `weights[]` (Input)
This option requires the user to provide the weights.
Default: all weights equal one.

`IMSL_ORDER`, *int* `order` (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: `order` = 4, (i.e., cubic splines).

`IMSL_KNOTS`, *float* `knots[]` (Input)
This option requires the user to provide the knots. The user must provide a knot sequence of length `spline_space_dimension` + `order`.
Default: an appropriate knot sequence is selected. See below for more details.

`IMSL_OPTIMIZE`
This option optimizes the knot locations, by attempting to minimize the least-squares error as a function of the knots. The optimal knots are available in the returned spline structure.

Description

Let's make the identifications

n = `ndata`

x = `xdata`

f = `fdata`

m = `spline_space_dim`

k = `order`

For convenience, we assume that the sequence x is increasing, although the function does not require this.

By default, $k = 4$, and the knot sequence we select equally distributes the knots through the distinct x_i's. In particular, the $m + k$ knots will be generated in $[x_1, x_n]$ with k knots stacked at each of the extreme values. The interior knots will be equally spaced in the interval.

Once knots **t** and weights w are determined (and assuming that the option `IMSL_OPTIMIZE` is not chosen), then the function computes the spline least-squares fit to the data by minimizing over the linear coefficients a_j

$$\sum_{i=0}^{n-1} w_i \left[f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right]^2$$

where the $B_j, j = 1, \ldots, m$ are a (B-spline) basis for the spline subspace.

The optional argument `IMSL_ORDER` allows the user to choose the order of the spline fit. The optional argument `IMSL_KNOTS` allows user specification of knots. The function `imsl_f_spline_least_squares` is based on the routine L2APPR by de Boor (1978, p. 255).

If the option `IMSL_OPTIMIZE` is chosen, then the procedure attempts to find the best placement of knots that will minimize the least-squares error to the given data by a spline of order k with m coefficients. For this problem to make sense, it is necessary that $m > k$. We then attempt to find the minimum of the functional

$$F(a, \mathbf{t}) = \sum_{i=0}^{n-1} w_i \left[f_i - \sum_{j=0}^{m-1} a_j B_{j,k,\mathbf{t}}(x_i) \right]$$

The technique employed here uses the fact that for a fixed knot sequence **t** the minimization in a is a linear least-squares problem that can be easily solved. 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, pp. 184 and 258–261).

The initial guess, $\mathbf{t}^g$, for the knot sequence is either provided by the user or is the default. This guess must be a *valid* knot sequence for splines of order k with

$$\mathbf{t}_0^g \leq \ldots \leq \mathbf{t}_{k-1}^g \leq x_i \leq \mathbf{t}_m^g \leq \ldots \leq \mathbf{t}_{m+k-1}^g \qquad i = 1, \ldots, M$$

with $\mathbf{t}^g$ nondecreasing, and

$$\mathbf{t}_i^g < \mathbf{t}_{i+k}^g \text{ for } i = 0, \ldots, m-1$$

In regard to execution speed, this function can be several orders of magnitude slower than a simple least-squares fit.

The return value for this function is a pointer of type `Imsl_f_spline`. The calling program must receive this in a pointer `Imsl_f_spline *sp`. This structure contains all the information to determine the spline (stored in B-spline form) that is computed by this function. For example, the following code sequence evaluates this spline a x and returns the value in y.

```
y = imsl_f_spline_value (x, sp, 0);
```

In the figure below two cubic splines are fit to

$$\sqrt{|x|}$$

Both splines are cubics with the same `spline_space_dim` = 8. The first spline is computed with the default settings, while the second spline is computed by optimizing the knot locations using the keyword `IMSL_OPTIMIZE`.

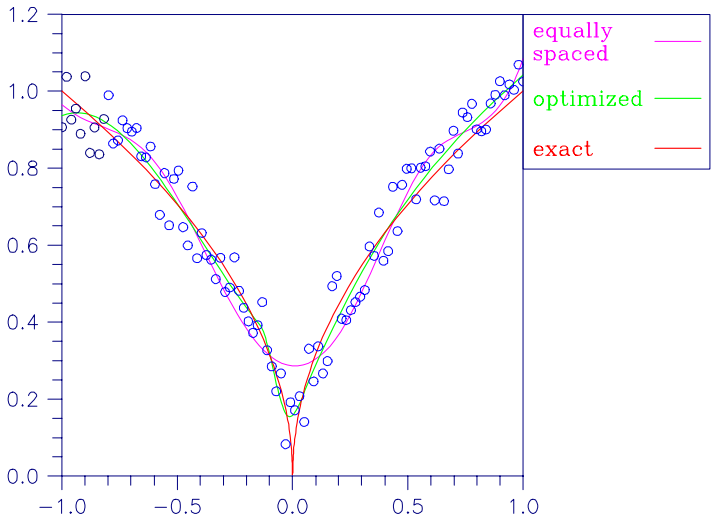


Figure 3-4 Two Fits to Noisy $\sqrt{|x|}$

Examples

Example 1

This example fits data generated from a trigonometric polynomial

$$1 + \sin x + 7\sin 3x + \varepsilon$$

where ε is a random uniform deviate over the range [–1, 1]. The data are obtained by evaluating this function at 90 equally spaced points on the interval [0, 6]. This data is fitted with a cubic spline with 12 degrees of freedom (eight equally spaced interior knots). The error at 10 equally spaced points is printed out.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

main()
{
    int                 i, spline_space_dim = 12;
    float               fdata[NDATA], xdata[NDATA], *random;
    Imsl_f_spline       *sp;
                                /* Generate random numbers */
    imsl_random_seed_set(123457);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = 6.*(float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]) + 2.*(random[i]-.5);
    }
    sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                        spline_space_dim, 0);
    printf("       x         error  \n");
    for(i = 0;  i < 10;  i++) {
        float x, error;
        x = 6.*i/9.;
        error = F(x) - imsl_f_spline_value(x, sp, 0);
        printf("%10.3f  %10.3f\n", x, error);
    }
}
```

Output

```
   x        Error
0.000      -0.356
0.667      -0.004
1.333       0.434
2.000      -0.069
2.667      -0.494
3.333       0.362
4.000      -0.273
4.667      -0.247
5.333       0.303
6.000       0.578
```

Example 2

This example continues with the first example in which we fit data generated from the trigonometric polynomial

$$1 + \sin x + 7\sin 3x + \varepsilon$$

where ε is random uniform deviate over the range [−1, 1]. The data is obtained by evaluating this function at 90 equally spaced points on the interval [0, 6]. This data was fitted with a cubic spline with 12 degrees of freedom (in this case, the default gives us eight equally spaced interior knots) and the error sum of squares was printed. In this example, the knot locations are optimized and the error sum of squares is printed. Then, the error at 10 equally spaced points is printed.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)        (float)(1.+ sin(x)+7.*sin(3.0*x))

main()
{
    int                 i, spline_space_dim = 12;
    float               fdata[NDATA], xdata[NDATA], *random, sse1, sse2;
    Imsl_f_spline        *sp;
                                /* Generate random numbers */
    imsl_random_seed_set(123457);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = 6.*(float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]) + 2.*(random[i]-.5);
    }
    sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                                    spline_space_dim,
                                    IMSL_SSE, &sse1,
                                    0);
    sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                                      spline_space_dim,
                                    IMSL_OPTIMIZE,
                                    IMSL_SSE, &sse2,
                                    0);
    printf("The error sum of squares before optimizing is %10.1f\n",
           sse1);
    printf("The error sum of squares  after optimizing is %10.1f\n\n",
           sse2);
    printf("       x         error\n");
    for(i = 0; i < 10;  i++){
        float   x, error;
        x = 6.*i/9.;
        error = F(x) - imsl_f_spline_value(x, sp, 0);
        printf("%10.3f  %10.3f\n", x, error);
    }
}
```

Output

```
The error sum of squares before optimizing is       32.6
The error sum of squares  after optimizing is       27.0

       x         Error
   0.000       -0.656
   0.667        0.107
   1.333        0.055
   2.000       -0.243
   2.667       -0.063
   3.333       -0.015
   4.000       -0.424
   4.667       -0.138
   5.333        0.133
   6.000        0.494
```

Warning Errors

IMSL_OPT_KNOTS_STACKED_1	The knots found to be optimal are stacked more than order. This indicates fewer knots will produce the same error sum of squares. The knots have been separated slightly.

Fatal Errors

IMSL_XDATA_TOO_LARGE	The array xdata must satisfy $\texttt{xdata}_i \le \mathbf{t}_{ndata}$, for $i = 1, \ldots,$ ndata.
IMSL_XDATA_TOO_SMALL	The array xdata must satisfy $\texttt{xdata}_i \ge \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ ndata.
IMSL_NEGATIVE_WEIGHTS	All weights must be greater than or equal to zero.
IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.
IMSL_OPT_KNOTS_STACKED_2	The knots found to be optimal are stacked more than order. This indicates fewer knots will produce the same error sum of squares.

spline_2d_least_squares

Computes a two-dimensional, tensor-product spline approximant using least squares.

Synopsis

#include <imsl.h>

Imsl_f_spline *imsl_f_spline_2d_least_squares (*int* num_xdata, *float* xdata[], *int* num_ydata, *float* ydata[], *float* fdata[], *int* x_spline_space_dim, *int* y_spline_space_dim, ..., 0)

The type *Imsl_d_spline* function is `imsl_d_spline_2d_least_squares`.

Required Arguments

int `num_xdata` (Input)
Number of data points in the *X* direction.

float `xdata[]` (Input)
Array with `num_xdata` components containing the data points in the *X* direction.

int `num_ydata` (Input)
Number of data points in the *Y* direction.

float `ydata[]` (Input)
Array with `num_ydata` components containing the data points in the *Y* direction.

float `fdata[]` (Input)
Array of size `num_xdata` × `num_ydata` containing the values to be approximated. `fdata[i][j]` is the (possibly noisy) value at (`xdata[i]`, `ydata[j]`).

int `x_spline_space_dim` (Input)
The linear dimension of the spline subspace for the *x* variable. It should be smaller than `num_xdata` and greater than or equal to `xorder` (whose default value is 4).

int `y_spline_space_dim` (Input)
The linear dimension of the spline subspace for the *y* variable. It should be smaller than `num_ydata` and greater than or equal to `yorder` (whose default value is 4).

Return Value

A pointer to the structure that represents the tensor-product spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_2d_least_squares` (*int* `num_xdata`, *float* `xdata[]`, *int* `num_ydata`, *float* `ydata[]`, *float* `fdata[]`, *int* `x_spline_space_dim`, *int* `y_spline_space_dim`,
`IMSL_SSE`, *float* `*sse`,
`IMSL_ORDER`, *int* `xorder`, *int* `yorder`,
`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]`,
`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim`,
`IMSL_WEIGHTS`, *float* `xweights[]`, *float* `yweights[]`,
0)

Optional Arguments

`IMSL_SSE`, *float* `*sse` (Output)
This option places the weighted error sum of squares in the place pointed to by `sse`.

`IMSL_ORDER`, *int* `xorder`, *int* `yorder` (Input)
This option is used to communicate the order of the spline subspace.
Default: `xorder, yorder` = 4 (i.e., tensor-product cubic splines)

`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]` (Input)
This option requires the user to provide the knots.
Default: The default knots are equally spaced in the x and y dimensions.

`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim` (Input)
The column dimension of `fdata`.
Default: `fdata_col_dim = num_ydata`

`IMSL_WEIGHTS`, *float* `xweights[]`, *float* `yweights[]` (Input)
This option requires the user to provide the weights for the least-squares fit.
Default: all weights are equal to 1.

Description

The `imsl_f_spline_2d_least_squares` procedure computes a tensor-product spline least-squares approximation to weighted tensor-product data. The input for this function consists of data vectors to specify the tensor-product grid for the data, two vectors with the weights (optional, the default is 1), the values of the surface on the grid, and the specification for the tensor-product spline (optional, a default is chosen). The grid is specified by the two vectors x = `xdata` and y = `ydata` of length n = `num_xdata` and m = `num_ydata`, respectively. A two-dimensional array f = `fdata` contains the data values which are to be fit. The two vectors w_x = `xweights` and w_y = `yweights` contain the weights for the weighted least-squares problem. The information for the approximating tensor-product spline can be provided using the keywords `IMSL_ORDER` and `IMSL_KNOTS`. This information is contained in k_x = `xorder`, $\mathbf{t}_x$ = `xknots`, and N = `xspline_space_dim` for the spline in the first variable, and in k_y = `yorder`, $\mathbf{t}_y$ = `yknots` and M = `y_spline_space_dim` for the spline in the second variable.

This function computes coefficients for the tensor-product spline 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 Grosse (1980).

As the computation proceeds, we obtain coefficients c minimizing

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} w_x(i)\,w_y(j)\left[\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl}B_{kl}(x_i,y_i) - f_{ij}\right]^2$$

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

$$\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl}B_{kl}$$

and its partial derivatives can be evaluated using `imsl_f_spline_2d_value`.

The return value for this function is a pointer to the structure *Imsl_f_spline*. The calling program must receive this in a pointer of type *Imsl_f_spline*. This structure contains all the information to determine the spline that is computed by this procedure. For example, the following code sequence evaluates this spline (stored in the structure `sp` at (x, y) and returns the value in `v`.

```
v = imsl_f_spline_2d_value (x, y, sp, 0)
```

Examples

Example 1

The data for this example comes from the function $e^x \sin(x + y)$ on the rectangle $[0, 3] \times [0, 5]$. This function is sampled on a 50×25 grid. Next try to recover it by using tensor-product cubic splines. The values of the function $e^x \sin(x + y)$ are printed on a 2×2 grid and compared with the values of the tensor-product spline least-squares fit.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NXDATA          50
#define NYDATA          25
#define OUTDATA          2
                                /* Define function  */
#define F(x,y)          (float)(exp(x)*sin(x+y))

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NXDATA][NYDATA];
    float               xdata[NXDATA], ydata[NYDATA], x, y, z;
    Imsl_f_spline       *sp;
                                /* Set up grid  */
    for (i = 0;  i < NXDATA;  i++) {
        xdata[i] = 3.*(float) i / ((float)(NXDATA-1));
    }
    for (i = 0;  i < NYDATA;  i++) {
        ydata[i] = 5.*(float) i / ((float)(NYDATA-1));
    }
                                /* Compute function values on grid  */
    for (i = 0;  i < NXDATA;  i++) {
        for (j = 0;  j < NYDATA;  j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = NXDATA;
```

```
    num_ydata = NYDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_least_squares(num_xdata, xdata, num_ydata,
                                                    ydata, fdata, 5, 7, 0);
                                /* Print results */
    printf("     x       y        F(x, y)   Fitted Values    Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float)i / (float)(OUTDATA);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float)j / (float)(OUTDATA);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                               x, y, F(x, y), z, fabs(F(x,y)-z));
        }

    }
}
```

Output

```
  x      y         F(x, y)   Fitted Values    Error
0.000   0.000       0.000       -0.020       0.0204
0.000   0.500       0.479        0.500       0.0208
0.500   0.000       0.790        0.816       0.0253
0.500   0.500       1.387        1.384       0.0031
```

Example 2

The same data is used as in the previous example. Optional argument `IMSL_SSE` is used to return the error sum of squares.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NXDATA         50
#define NYDATA         25
#define OUTDATA         2
                                /* Define function  */
#define F(x,y)         (float)(exp(x)*sin(x+y))

main()
{
    int                 i, j, num_xdata, num_ydata;
    float               fdata[NXDATA][NYDATA];
    float               xdata[NXDATA], ydata[NYDATA], x, y, z;
    Imsl_f_spline       *sp;
                                /* Set up grid  */
    for (i = 0;  i < NXDATA;  i++) {
        xdata[i] = 3.*(float) i / ((float) (NXDATA - 1));
    }
    for (i = 0;  i < NYDATA;  i++) {
        ydata[i] = 5.*(float) i / ((float) (NYDATA - 1));
    }
                                /*  Compute function values on grid  */
    for (i = 0;  i < NXDATA;  i++) {
        for (j = 0;  j < NYDATA;  j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
```

```
    }
    num_xdata = NXDATA;
    num_ydata = NYDATA;
                                /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_least_squares(num_xdata, xdata, num_ydata,
                                                    ydata, fdata, 5, 7,
                                        IMSL_SSE, &x,
                                        0);
                                /* Print results */
    printf("The error sum of squares is %10.3f\n\n", x);
    printf("     x       y        F(x, y)   Fitted Values    Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        x = (float) i / (float) (OUTDATA);
        for (j = 0;  j < OUTDATA;  j++) {
            y = (float) j / (float) (OUTDATA);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                               x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

Output

```
The error sum of squares is      3.753

    x       y      F(x, y)   Fitted Values    Error
0.000   0.000       0.000       -0.020       0.0204
0.000   0.500       0.479        0.500       0.0208
0.500   0.000       0.790        0.816       0.0253
0.500   0.500       1.387        1.384       0.0031
```

Warning Errors

IMSL_ILL_COND_LSQ_PROB	The least-squares matrix is ill-conditioned. The solution might not be accurate.
IMSL_SPLINE_LOW_ACCURACY	There may be less than one digit of accuracy in the least-squares fit. Try using a higher precision if possible.

Fatal Errors

IMSL_KNOT_MULTIPLICITY	Multiplicity of the knots cannot exceed the order of the spline.
IMSL_KNOT_NOT_INCREASING	The knots must be nondecreasing.
IMSL_SPLINE_LRGST_ELEMNT	The data arrays xdata and ydata must satisfy $\texttt{data}_i \le \mathbf{t}_{spline\_space\_dim}$, for $i = 1, \ldots,$ num_data.
IMSL_SPLINE_SMLST_ELEMNT	The data arrays xdata and ydata must satisfy $\texttt{data}_i \ge \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ num_data.

IMSL_NEGATIVE_WEIGHTS	All weights must be greater than or equal to zero.
IMSL_DATA_DECREASING	The xdata values must be nondecreasing.

cub_spline_smooth

Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.

Synopsis

#*include* <imsl.h>

Imsl_f_ppoly *imsl_f_cub_spline_smooth (*int* ndata, *float* xdata[], *float* fdata[], ..., 0)

The type *Imsl_d_ppoly* function is imsl_d_cub_spline_smooth.

Required Arguments

int ndata (Input)
Number of data points.

float xdata[] (Input)
Array with ndata components containing the abscissas of the problem.

float fdata[] (Input)
Array with ndata components containing the ordinates of the problem.

Return Value

A pointer to the structure that represents the cubic spline. If a smoothed cubic spline cannot be computed, then NULL is returned. To release this space, use free.

Synopsis with Optional Arguments

#*include* <imsl.h>

Imsl_f_ppoly *imsl_f_cub_spline_smooth (*int* ndata, *float* xdata[], *float* fdata[],
IMSL_WEIGHTS, *float* weights[],
IMSL_SMOOTHING_PAR, *float* sigma,
0)

Optional Arguments

IMSL_WEIGHTS, *float* weights[] (Input)
This option requires the user to provide the weights.
Default: all weights are equal to 1.

IMSL_SMOOTHING_PAR, *float* sigma (Input)
This option sets the smoothing parameter σ = sigma explicitly.

Description

The function `imsl_f_cub_spline_smooth` 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*.

Consider first the situation when the optional argument `IMSL_SMOOTHING_PAR` is selected. Then, a natural cubic spline with knots at all the data abscissas x = `xdata` is computed, 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

$$\sum_{i=0}^{n-1} \left| \left(s(x_i) - f_i \right) w_i \right|^2 \leq \sigma$$

where w = `weights`, σ = `sigma` 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 the inverse of 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,

$$n - \sqrt{2n} \leq \sigma \leq n + \sqrt{2n}$$

The function `imsl_f_cub_spline_smooth` is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pp. 235–243).

The default for this function chooses the smoothing parameter σ by a statistical technique called *cross-validation*. For more information on this topic, refer to Craven and Wahba (1979).

The return value for this function is a pointer to the structure *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*pp`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this procedure. For example, the following code sequence evaluates this spline at x and returns the value in `y`.

```
y = imsl_f_cub_spline_value (x, pp, 0);
```

Examples

Example 1

In this example, function values are contaminated by adding a small "random" amount to the correct values. The function `imsl_f_cub_spline_smooth` is used to approximate the original, uncontaminated data.

```
#include <imsl.h>
#include <stdio.h>
```

```
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], *random;
    Imsl_f_ppoly        *pp;
                                /* Generate random numbers */
    imsl_random_seed_set(123457);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = 6.*(float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]) + .5*(random[i]-.5);
    }
    pp = imsl_f_cub_spline_smooth(NDATA, xdata, fdata,  0);
    printf("       x         error  \n");
    for(i = 0;  i < 10;  i++){
        float x, error;
        x = 6.*i/9.;
        error = F(x) - imsl_f_cub_spline_value(x, pp, 0);
        printf("%10.3f  %10.3f\n", x, error);
    }
}
```

Output

```
   x        Error
0.000      -0.201
0.667       0.070
1.333      -0.008
2.000      -0.058
2.667      -0.025
3.333       0.076
4.000      -0.002
4.667      -0.008
5.333       0.045
6.000       0.276
```

Example 2

Recall that in the first example, function values are contaminated by adding a small "random" amount to the correct values. Then, `imsl_f_cub_spline_smooth` is used to approximate the original, uncontaminated data. This example explicitly inputs the value of the smoothing parameter to be 5.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
                                /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))
```

```
main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], *random;
    Imsl_f_ppoly        *pp;
                                /* Generate random numbers */
    imsl_random_seed_set(123457);
    random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
    for (i = 0;  i < NDATA;  i++) {
        xdata[i] = 6.*(float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]) + .5*(random[i]-.5);
    }
    pp = imsl_f_cub_spline_smooth(NDATA, xdata, fdata,
                                  IMSL_SMOOTHING_PAR, 5.0,
                                  0);
    printf("       x         error  \n");
    for(i = 0;  i < 10;  i++){
        float  x, error;
        x = 6.*i/9.;
        error = F(x) - imsl_f_cub_spline_value(x, pp, 0);
        printf("%10.3f  %10.3f\n", x, error);
    }
}
```

Output

```
  x         Error
0.000      -0.593
0.667       0.230
1.333      -0.116
2.000      -0.106
2.667       0.176
3.333      -0.071
4.000      -0.171
4.667       0.196
5.333      -0.036
6.000       0.971
```

Warning Errors

IMSL_MAX_ITERATIONS_REACHED	The maximum number of iterations has been reached. The best approximation is returned.

Fatal Errors

IMSL_DUPLICATE_XDATA_VALUES	The `xdata` values must be distinct.
IMSL_NEGATIVE_WEIGHTS	All weights must be greater than or equal to zero.

spline_lsq_constrained

Computes a least-squares constrained spline approximation.

Synopsis

#*include* `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_lsq_constrained` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`, *int* `spline_space_dim`, *int* `num_con_pts`, `f_constraint_struct constraints[]`, ..., 0)

The type *Imsl_d_spline* function is `imsl_d_spline_lsq_constrained`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the least-squares problem.

float `fdata[]` (Input)
Array with `ndata` components containing the ordinates of the least-squares problem.

int `spline_space_dim` (Input)
The linear dimension of the spline subspace. It should be smaller than `ndata` and greater than or equal to `order` (whose default value is 4).

int `num_con_pts` (Input)
The number of points in the vector constraints.

f_constraint_struct `constraints[]` (Input)
A structure containing the abscissas at which the fit is to be constrained, the derivative of the spline that is to be constrained, the type of constraints, and any lower or upper limits. A description of the structure fields follows:

Field	Description
`xval`	point at which fit is constrained
`der`	derivative value of the spline to be constrained
`type`	types of the general constraints
`bl`	lower limit of the general constraints
`bu`	upper limit of the general constraints

Notes: If you want to constrain the integral of the spline over the closed interval (c, d), then set `constraints[i].der` = `constraints[i+1].der` = –1 and `constraints[i].xval` = c and `constraints[i+1].xval` = d. For consistency, insist that

`constraints[i].type` = `constraints[i+1].type` ≥ 0 and $c \le d$. Note that every `der` must be at least –1.

`constraints [i].type`	*i*-th constraint
1	$bl_i = f^{(d_i)}(x_i)$
2	$f^{(d_i)}(x_i) \le bu_i$
3	$f^{(d_i)}(x_i) \ge bl_i$
4	$bl_i \le f^{(d_i)}(x_i) \le bu_i$
5	$bl_i = \int_c^d f(t)dt$
6	$\int_c^d f(t)dt \le bu_i$
7	$\int_c^d f(t)dt \ge bl_i$
8	$bl_i \le \int_c^d f(t)dt \le bu_i$
20	periodic end conditions
99	disregard this constraint

In order to have two point constraints, must have

`constraints[i].type = constraints[i+1].type`

`constraints [i]. type`	*i*-th constraint
9	$bl_i = f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1})$
10	$f^{(d_i)}(x_i) \le bu_i$
11	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \ge bl_i$
12	$bl_i \le f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \le bu_i$

Return Value

A pointer to the structure that represents the spline fit. If a fit cannot be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#*include* `<imsl.h>`

Imsl_f_spline `*imsl_f_spline_lsq_constrained` (*int* `ndata`, *float* `xdata[]`,
float `fdata[]`, *int* `spline_space_dim`, *int* `num_con_pts`,
f_constraint_struct `constraints[]`,
`IMSL_NHARD`, *int* `nhard`,
`IMSL_WEIGHTS`, *float* `weights[]`,
`IMSL_ORDER`, *int* `order`,
`IMSL_KNOTS`, *float* `knots[]`,
0)

Optional Arguments

IMSL_NHARD, *int* nhard (Output)

The argument nhard is the number of entries of constraints involved in the "hard" constraints. Note that 0 ≤ nhard ≤ num_con_pts. The default, nhard = 0, always results in a fit, while setting nhard = num_con_pts forces all constraints to be met. The "hard" constraints must be met, or else the function signals failure. The "soft" constraints need not be satisfied, but there will be an attempt to satisfy the "soft" constraints. The constraints must be listed in terms of priority with the most important constraints first. Thus, all of the "hard" constraints must precede the "soft" constraints. If infeasibility is detected among the "soft" constraints, we satisfy, in order, as many of the "soft" constraints as possible.
Default: nhard = 0

IMSL_WEIGHTS, *float* weights[] (Input)

This option requires the user to provide the weights.
Default: all weights equal one

IMSL_ORDER, *int* order (Input)

The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: order = 4(i.e., cubic splines)

IMSL_KNOTS, *float* knots[] (Input)

This option requires the user to provide the knots. The user must provide a knot sequence of length spline_space_dimension + order.
Default: an appropriate knot sequence is selected. See below for more details.

Description

The function imsl_f_spline_lsq_constrained 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 functions are of four types:

$E_p[f]$	$= f^{(j_p)}(y_p)$
or	$= f^{(j_p)}(y_p) - f^{(j_{p+1})}(y_{p+1})$
or	$= \int_{y_p}^{y_{p+1}} f(t)dt$
or	= periodic end conditions

An interval, I_p (which may be a point, a finite interval, or a semi-infinite interval), is associated with each of these constraints.
The input for this function 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, defaulting to 1). The vector constraints contains the abscissas of the points involved in specifying the constraints, as well as information relating the type of constraints and the constraint interval.

Let n_f denote the number of feasible constraints as described above. Then, the function solved the problem

$$\sum_{i=1}^{n} \left| f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right|^2 w_i$$

subject to

$$E_p \left[\sum_{j=1}^{m} a_j B_j \right] \in I_p \qquad p = 1, \ldots, n_f$$

This linearly constrained least-squares problem is treated as a quadratic program and is solved by invoking the function `imsl_f_quadratic_prog`.

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.

Examples

Example 1

This is a simple application of `imsl_f_lsq_constrained`. Data is generated from the function

$$\frac{x}{2} + \sin\left(\frac{x}{2}\right)$$

and contaminated with random noise and fit with cubic splines. The function is increasing, so least-squares fit should also be increasing. This is not the case for the unconstrained least-squares fit generated by `imsl_f_spline_least_squares`. Then, the derivative is forced to be greater than 0 at `num_con_pts` = 15 equally spaced points and `imsl_f_lsq_constrained` is called. The resulting curve is monotone. The error is printed for the two fits averaged over 100 equally spaced points.

```
#include <imsl.h>
#include <math.h>

#define MXKORD   4
#define MXNCOF  20
#define MXNDAT  51
#define MXNXVL  15

main()
{
    f_constraint_struct constraint[MXNXVL];
    int   i, korder, ncoef, ndata, nxval;
    float *noise, errlsq, errnft, grdsiz, x;
    float fdata[MXNDAT], xdata[MXNDAT];
    Imsl_f_spline *sp, *spls;
```

```
#define F1(x)    (float)(.5*(x) + sin( .5*(x) ))

    korder = 4;
    ndata = 15;
    nxval = 15;
    ncoef = 8;
    /*
     * Compute original xdata and fdata with random noise.
     */
    imsl_random_seed_set (234579);
    noise = imsl_f_random_uniform (ndata, 0);
    grdsiz = 10.0;
    for (i = 0; i < ndata; i++) {
        xdata[i] = grdsiz * ((float) (i) / (float) (ndata - 1));
        fdata[i] = F1 (xdata[i]) + (noise[i] - .5);
    }

    /* Compute least-squares fit. */

    spls = imsl_f_spline_least_squares (ndata, xdata, fdata, ncoef, 0);
    /*
     * Construct the constraints.
     */
    for (i = 0; i < nxval; i++) {
        constraint[i].xval = grdsiz * (float)(i) / (float)(nxval - 1);
        constraint[i].type = 3;
        constraint[i].der = 1;
        constraint[i].bl = 0.0;
    }
    /* Compute constrained least-squares fit. */
    sp = imsl_f_spline_lsq_constrained (ndata, xdata, fdata, ncoef,
              nxval, constraint, 0);
    /*
     * Compute the average error of 100 points in the interval.
     */
    errlsq = 0.0;
    errnft = 0.0;
    for (i = 0; i < 100; i++) {
        x = grdsiz * (float) (i) / 99.0;
        errnft += fabs (F1 (x) - imsl_f_spline_value(x,sp,0));
        errlsq += fabs (F1 (x) - imsl_f_spline_value(x,spls,0));
    }
    /* Print results */
    printf (" Average error with spline_least_squares fit:    %8.5f\n",
              errlsq / 100.0);
    printf (" Average error with spline_lsq_constrained fit:  %8.5f\n",
              errnft / 100.0);
}
```

Output

```
Average error with spline_least_squares fit:     0.20250
Average error with spline_lsq_constrained fit:   0.14334
```

Example 2

Now, try to recover the function

$$\frac{1}{1+x^4}$$

from noisy data. First, try the unconstrained least-squares fit using `imsl_f_spline_least_squares`. Finding that fit somewhat unsatisfactory, several constraints are applied using `imsl_f_spline_lsq_constrained`. 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, the cubic spline is constrained 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, the fit is constrained (called *s*) as follows:

$$s(-7) \geq 0$$

$$\int_{-7}^{7} s(x)\, dx \leq 2.3$$

$$s(-7) = s(7)$$

Notice that the last constraint was generated using the periodic option (requiring only the *zero*-th derivative to be periodic). The error is printed for the two fits averaged over 100 equally spaced points.

```
#include <imsl.h>
#include <math.h>

#define KORDER   4
#define NDATA   51
#define NXVAL   12
#define NCOEF   13

main()
{
    f_constraint_struct constraint[NXVAL];
    int   i;
    float *noise, errlsq, errnft, grdsiz, x;
    float fdata[NDATA], xdata[NDATA], xknot[NDATA+KORDER];
    Imsl_f_spline *sp, *spls;


#define F1(x)    (float)(1.0/(1.0+x*x*x*x))

     /* Compute original xdata and fdata with random noise */

    imsl_random_seed_set (234579);
    noise = imsl_f_random_uniform (NDATA, 0);
    grdsiz = 14.0;
    for (i = 0; i < NDATA; i++) {
        xdata[i] = grdsiz * ((float)(i)/(float)(NDATA - 1))
                   - grdsiz/2.0;
        fdata[i] = F1 (xdata[i]) + 0.125*(noise[i] - .5);
    }
```

```
/* Generate knots. */
    for (i = 0; i < NCOEF-KORDER+2; i++) {
        xknot[i+KORDER-1] = grdsiz * ((float)(i)/
                            (float)(NCOEF-KORDER+1)) - grdsiz/2.0;
    }
    for (i = 0; i < KORDER - 1; i++) {
        xknot[i] = xknot[KORDER-1];
        xknot[i+NCOEF+1] = xknot[NCOEF];
    }

    /* Compute spline_least_squares fit */

    spls = imsl_f_spline_least_squares (NDATA, xdata, fdata, NCOEF,
                IMSL_KNOTS, xknot, 0);

     /* Construct the constraints for CONFT */

    for (i = 0; i < 4; i++) {
        constraint[i].xval = xknot[KORDER+i-1];
        constraint[i+4].xval = xknot[NCOEF-3+i];
        constraint[i].itype = 1;
        constraint[i+4].itype = 1;
        constraint[i].ider = 2;
        constraint[i+4].ider = 2;
        constraint[i].bl = 0.0;
        constraint[i+4].bl = 0.0;
    }
    constraint[8].xval = -7.0;
    constraint[8].itype = 3;
    constraint[8].ider = 0;
    constraint[8].bl = 0.0;

    constraint[9].xval = -7.0;
    constraint[9].itype = 6;
    constraint[9].bu = 2.3;

    constraint[10].xval = 7.0;
    constraint[10].itype = 6;
    constraint[10].bu = 2.3;

    constraint[11].xval = -7.0;
    constraint[11].itype = 20;
    constraint[11].ider = 0;

    sp = imsl_f_spline_lsq_constrained (NDATA, xdata, fdata, NCOEF,
               NXVAL, constraint, IMSL_KNOTS, xknot, 0);

    /* Compute the average error of 100 points in the interval */

    errlsq = 0.0;
    errnft = 0.0;
    for (i = 0; i < 100; i++) {
        x = grdsiz * (float) (i) / 99.0 - grdsiz/2.0;
        errnft += fabs (F1 (x) - imsl_f_spline_value(x,sp,0));
        errlsq += fabs (F1 (x) - imsl_f_spline_value(x,spls,0));
    }
    /* Print results */
    printf (" Average error with BSLSQ fit:  %8.5f\n",
```

```
            errlsq / 100.0);
    printf (" Average error with CONFT fit:  %8.5f\n",
            errnft / 100.0);
}
```

Output

```
Average error with BSLSQ fit:   0.01783
Average error with CONFT fit:   0.01339
```

smooth_1d_data

Smooth one-dimensional data by error detection.

Synopsis

#include `<imsl.h>`

float `*imsl_f_smooth_1d_data` (*int* `ndata`,
float `xdata[]`, *float* `fdata[]`, ..., 0)

The type *double* function is `imsl_d_smooth_1d_data`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the data points.

float `ydata[]` (Input)
Array with `ndata` components containing the ordinates of the data points.

Return Value

A pointer to the vector of length `ndata` containing the smoothed data.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*` `imsl_f_smooth_1d_data` (*int* `ndata`,
float `xdata[]`, *float* `fdata[]`,
`IMSL_RETURN_USER`, *float* `sdata[]`,
`IMSL_ITMAX`, *int* `itmax`,
`IMSL_DISTANCE`, *float* `dis`,
`IMSL_STOPPING_CRITERION`, *float* `sc`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `sdata[]` (Output)
The smoothed data is stored in the user-supplied array.

`IMSL_ITMAX`, *int* `itmax` (Input)
The maximum number of iterations allowed.
Default: `itmax` = 500

`IMSL_DISTANCE`, *float* `dis` (Input)
Proportion of the distance the ordinate in error is moved to its interpolating curve. It must be in the range 0.0 to 1.0.
Default: `dis` = 1.0

`IMSL_STOPPING_CRITERION`, *float* `sc` (Input)
The stopping criterion. `sc` should be greater than or equal to zero.
Default: `sc` = 0.0

Algorithm

The function `imsl_f_smooth_1d_data` 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 `imsl_f_smooth_1d_data` is based on an algorithm of Guerra and Tapia (1974).

Setting `ndata` = n, `ydata` = 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 $i = 4, \ldots, n - 3$. 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 `itmax` iterations have taken place or if

$$\left|pe_i\right| \leq sc\left(x_{i+3} - x_{i-3}\right)/6 \qquad i = 4, \ldots, n-3$$

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 `itmax` 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 `itmax` to be the number of data points in error. On the other hand, if no such specific information is available, then choose $d = .5$, `itmax` $\leq 2n$, and

$$sc = .5\frac{\max s - \min s}{(x_n - x_1)}$$

In any case, we would encourage the user to experiment with these values.

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.

```
#include "imsl.h"
#include "stdlib.h"
#include "math.h"

#define NDATA 91
#define F(X) (X*X*sin((double)(X))+5.0)/X + 5.0

main()
{
  int i, maxit;
  int isub[10] = {5, 16, 25, 33, 41, 48, 55, 61, 74, 82};
  float dis, fdata[NDATA], sc, *sdata=NULL;
  float xdata[NDATA], s_user[NDATA];
  float rnoise[10] = {2.5, -3., -2., 2.5, 3.,
                      -2., -2.5, 2., -2., 3.};

  /* Example 1: No specific information available. */
  dis = .5;
  sc = .56;
  maxit = 182;

  /* Set values for xdata and fdata. */
  xdata[0] = 1.;
  fdata[0] = F(xdata[0]);
  for (i=1;i<NDATA;i++) {
    xdata[i] = xdata[i-1]+.1;
    fdata[i] = F(xdata[i]);
  }

  /* Contaminate the data. */
  for (i=0;i<10;i++) fdata[isub[i]] += rnoise[i];

  /* Smooth the data. */
```

```
  sdata = imsl_f_smooth_1d_data(NDATA, xdata, fdata,
                                IMSL_DISTANCE, dis,
                                IMSL_STOPPING_CRITERION, sc,
                                IMSL_ITMAX, maxit,
                                0);

  /* Output the result. */
  printf("Case A - No specific information available. \n");
  printf("   F(X)        F(X)+noise         sdata\n");

  for (i=0;i<10;i++) printf("%7.3f\t%15.3f\t%15.3f\n",
                            F(xdata[isub[i]]),
                            fdata[isub[i]],
                            sdata[isub[i]]);

  /* Example 2: No specific information is available. */
  dis = 1.0;
  sc = 0.0;
  maxit = 10;

  /*
   * A warning message is produced because the maximum
   * number of iterations is reached.
   */

  /* Smooth the data. */
sdata = imsl_f_smooth_1d_data(NDATA, xdata, fdata,
                                IMSL_DISTANCE, dis,
                                IMSL_STOPPING_CRITERION, sc,
                                IMSL_ITMAX, maxit,
                                IMSL_RETURN_USER, s_user,
                                0);

  /* Output the result. */
  printf("Case B - Specific information available. \n");
  printf("   F(X)        F(X)+noise         sdata\n");

  for (i=0;i<10;i++) printf("%7.3f\t%15.3f\t%15.3f\n",
                            F(xdata[isub[i]]),
                            fdata[isub[i]],
                            s_user[isub[i]]);
}
```

Output

```
Case A - No specific information available.
   F(X)        F(X)+noise         sdata
  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 IMSL_ITMAX_EXCEEDED from imsl_f_smooth_1d_data.
*** Maximum  number of iterations limit "itmax" = 10 exceeded.
*** The best answer found is returned.

Case B - Specific information available.
   F(X)        F(X)+noise         sdata
  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
```

scattered_2d_interp

Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

Synopsis

#include `<imsl.h>`

float `*imsl_f_scattered_2d_interp` (*int* `ndata`, *float* `xydata[]`, *float* `fdata[]`, *int* `nx_out`, *int* `ny_out`, *float* `x_out[]`, *float* `y_out[]`, ..., 0)

The type *double* function is `imsl_d_scattered_2d_interp`.

Required Arguments

int `ndata` (Input)
Number of data points.

float `xydata[]` (Input)
Array with `ndata*2` components containing the data points for the interpolation problem. The i-th data point (x_i, y_i) is stored consecutively in the $2i$ and $2i + 1$ positions of `xydata`.

float `fdata[]` (Input)
Array of size `ndata` containing the values to be interpolated.

int `nx_out` (Input)
Number of data points in the x direction for the output grid.

int `ny_out` (Input)
Number of data points in the y direction for the output grid.

float `x_out[]` (Input)
Array of length `nx_out` specifying the x values for the output grid. It must be strictly increasing.

float `y_out[]` (Input)
Array of length `ny_out` specifying the y values for the output grid. It must be strictly increasing.

Return Value

A pointer to the `nx_out` × `ny_out` grid of values of the interpolant. If no answer can be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_scattered_2d_interp` (*int* `ndata`, *float* `xydata[]`, *float* `fdata[]`, *int* `nx_out`, *int* `ny_out`, *float* `x_out[]`, *float* `y_out[]`, `IMSL_RETURN_USER`, *float* `surface[]`, `IMSL_SUR_COL_DIM`, *int* `surface_col_dim`, 0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `surface[]` (Output)
This option allows the user to provide his own space for the result. In this case, the answer will be returned in `surface`.

`IMSL_SUR_COL_DIM`, *int* `surface_col_dim` (Input)
This option requires the user to provide the column dimension of the two-

dimensional array `surface`.
Default: `surface_col_dim` = `ny_out`

Description

The function `imsl_f_scattered_2d_interp` computes a C^1 interpolant to scattered data in the plane. Given the data points

$$\{(x_i, y_i, f_i)\}_{i=0}^{n-1}$$

in $\mathbf{R}^3$ where n = `ndata`, `imsl_f_scattered_2d_interp` returns the values of the interpolant s on the user-specified grid. The computation of s is as follows: First the Delaunay triangulation of the points

$$\{(x_i, y_i)\}_{i=0}^{n-1}$$

is computed. On each triangle T in this triangulation, s has the form

$$s(x,y) = \sum_{m+n\le 5} c_{mn}^T x^m y^n \qquad \forall x, y \in T$$

Thus, s is a bivariate quintic polynomial on each triangle of the triangulation. In addition, we have

$$s(x_i, y_i) = f_i \qquad \text{for } i = 0, \ldots, n-1$$

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 procedure, refer to the article by Akima (1978). The output grid is specified by the two integer variables `nx_out` and `ny_out` that represent the number of grid points in the first (second) variable and by two real vectors that represent the first (second) coordinates of the grid.

Examples

Example 1

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. Then the values are printed on a 3×3 grid.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           20
#define OUTDATA          3
                              /* Define function  */
#define F(x,y)          (float)(3.+7.*x+2.*y)

#define SURF(I,J)       surf[(J) +(I)*OUTDATA]
```

```
main()
{
    int                 i, j;
    float               fdata[NDATA], xydata[2*NDATA], *surf;
    float               x, y, z, x_out[OUTDATA], y_out[OUTDATA],  pi;

    pi = imsl_f_constant("pi", 0);
                                /* Set up output grid  */
    for (i = 0;  i < OUTDATA;  i++) {
        x_out[i] = y_out[i] = (float) i / ((float) (OUTDATA - 1));
    }
    for (i = 0;  i < 2*NDATA;  i += 2) {
        xydata[i]   = 3.*cos(pi*i/NDATA);
        xydata[i+1] = 3.*sin(pi*i/NDATA);
        fdata[i/2]  = F(xydata[i], xydata[i+1]);
    }
                                /* Compute scattered data interpolant */
    surf = imsl_f_scattered_2d_interp (NDATA, xydata, fdata, OUTDATA,
                                       OUTDATA, x_out, y_out, 0);
                                /* Print results */
    printf("     x       y        F(x, y)    Interpolant    Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        for (j = 0;  j < OUTDATA;  j++) {
            x = x_out[i];
            y = y_out[j];
            z = SURF(i,j);
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                              x, y, F(x,y), z, fabs(F(x,y)-z));
        }

    }
}
```

Output

```
    x       y      F(x, y)    Interpolant    Error
0.000   0.000       3.000        3.000       0.0000
0.000   0.500       4.000        4.000       0.0000
0.000   1.000       5.000        5.000       0.0000
0.500   0.000       6.500        6.500       0.0000
0.500   0.500       7.500        7.500       0.0000
0.500   1.000       8.500        8.500       0.0000
1.000   0.000      10.000       10.000       0.0000
1.000   0.500      11.000       11.000       0.0000
1.000   1.000      12.000       12.000       0.0000
```

Example 2

Recall that in the first 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. This example used the optional arguments to indicate that the answer is stored noncontiguously in a two-dimensional array `surf` with column dimension equal to 11.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>
```

```
#define NDATA           20
#define OUTDATA          3
#define COLDIM          11
                                /* Define function  */
#define F(x,y)          (float)(3.+7.*x+2.*y)

main()
{
    int                 i, j;
    float               fdata[NDATA], xydata[2*NDATA];
    float               surf[OUTDATA][COLDIM];
    float               x, y, z, x_out[OUTDATA], y_out[OUTDATA], pi;

    pi = imsl_f_constant("pi", 0);
                                /* Set up output grid  */
    for (i = 0;  i < OUTDATA;  i++) {
        x_out[i] = y_out[i] = (float) i / ((float) (OUTDATA - 1));
    }
    for (i = 0;  i < 2*NDATA;  i += 2) {
        xydata[i]   = 3.*cos(pi*i/NDATA);
        xydata[i+1] = 3.*sin(pi*i/NDATA);
        fdata[i/2]  = F(xydata[i], xydata[i+1]);
    }
                                /* Compute scattered data interpolant */
    imsl_f_scattered_2d_interp (NDATA, xydata, fdata, OUTDATA,
                                                  OUTDATA, x_out, y_out,
                                IMSL_RETURN_USER, surf,
                                IMSL_SUR_COL_DIM, COLDIM,
                                0);
                                /* Print results */
    printf("     x       y        F(x, y)    Interpolant    Error\n");
    for (i = 0;  i < OUTDATA;  i++) {
        for (j = 0;  j < OUTDATA;  j++) {
            x = x_out[i];
            y = y_out[j];
            z = surf[i][j];
            printf("  %6.3f  %6.3f  %10.3f   %10.3f   %10.4f\n",
                              x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

Output

```
    x       y      F(x, y)    Interpolant    Error
0.000   0.000       3.000        3.000       0.0000
0.000   0.500       4.000        4.000       0.0000
0.000   1.000       5.000        5.000       0.0000
0.500   0.000       6.500        6.500       0.0000
0.500   0.500       7.500        7.500       0.0000
0.500   1.000       8.500        8.500       0.0000
1.000   0.000      10.000       10.000       0.0000
1.000   0.500      11.000       11.000       0.0000
1.000   1.000      12.000       12.000       0.0000
```

Fatal Errors

IMSL_DUPLICATE_XYDATA_VALUES	The two-dimensional data values must be distinct.
IMSL_XOUT_NOT_STRICTLY_INCRSING	The vector x_out must be strictly increasing.
IMSL_YOUT_NOT_STRICTLY_INCRSING	The vector y_out must be strictly increasing.

radial_scattered_fit

Computes an approximation to scattered data in $\mathbf{R}^n$ for $n \geq 1$ using radial-basis functions.

Synopsis

#include <imsl.h>

Imsl_f_radial_basis_fit *imsl_f_radial_scattered_fit (*int* dimension, *int* num_points, *float* abscissae[], *float* fdata[], *int* num_centers, ..., 0)

The type *Imsl_d_radial_basis_fit* function is imsl_d_radial_scattered_fit.

Required Arguments

int dimension (Input)
Number of dimensions.

int num_points (Input)
The number of data points.

float abscissae[] (Input)
Array of size dimension × num_points containing the abscissae of the data points. The argument abscissae[i][j] is the abscissa value of the (i+1)-th data point in the (j+1)-th dimension.

float fdata[] (Input)
Array with num_points components containing the ordinates for the problem.

int num_centers (Input)
The number of centers to be used when computing the radial-basis fit. The argument num_centers should be less than or equal to num_points.

Return Value

A pointer to the structure that represents the radial-basis fit. If a fit cannot be computed, then NULL is returned. To release this space, use free.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_radial_basis_fit `*imsl_f_radial_scattered_fit` (*int* `dimension`, *int* `num_points`, *float* `abscissae[]`, *float* `fdata[]`,
int `num_centers`,
`IMSL_CENTERS`, *float* `centers[]`,
`IMSL_CENTERS_RATIO`, *float* `ratio`,
`IMSL_RANDOM_SEED`, *int* `seed`,
`IMSL_SUPPLY_BASIS`, *float* `radial_function()`,
`IMSL_SUPPLY_BASIS_W_DATA`, *float* `radial_function()`, *void* `*data`,
`IMSL_SUPPLY_DELTA`, *float* `delta`,
`IMSL_WEIGHTS`, *float* `weights[]`,
`IMSL_NO_SVD`,
0)

Optional Arguments

`IMSL_CENTERS` (Input)
User-supplied centers. See the "Description" (page 227) section of this function for details.

`IMSL_CENTERS_RATIO`, *float* `ratio` (Input)
The desired ratio of centers placed on an evenly spaced grid to the total number of centers. The condition that the same number of centers placed on a grid for each dimension must be equal. Thus, the actual number of centers placed on a grid is usually less than `ratio*num_centers`, but will never be more than `ratio*num_centers`. The remaining centers are randomly chosen from the set of abscissae given in `abscissae`.
Default: `ratio` = 0.5

`IMSL_RANDOM_SEED`, *int* `seed`
The value of the random seed used when determining the random subset of abscissae to use as centers. By changing the value of seed on different calls to `imsl_f_radial_scattered_fit`, with the same data set, a different set of random centers will be chosen. Setting `seed` to zero forces the random number seed to be based on the system clock, so a possibly different set of centers will be chosen each time the program is executed.
Default: `seed` = 234579

`IMSL_SUPPLY_BASIS`, *float* `radial_function` (*float* `distance`) (Input)
User-supplied function to compute the values of the radial functions.
Default: Hardy multiquadric

`IMSL_SUPPLY_BASIS_W_DATA`, *float* `radial_function` (*float* `distance`, *void* `*data`), *void* `*data` (Input)
User-supplied function to compute the values of the radial functions, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the "*Introduction,*

Passing Data to User-Supplied Functions" at the beginning of this manual for more details.

Default: Hardy multiquadric

`IMSL_SUPPLY_DELTA`, *float* `delta` (Input)
The delta used in the default basis function

$$\phi(r) = \sqrt{r^2 + \delta^2}$$

Default: `delta` = 1

`IMSL_WEIGHTS`, *float* `weights[]`
This option requires the user to provide the weights.
Default: all weights equal one

`IMSL_NO_SVD`
This option forces the use of a *QR* decomposition instead of a singular value decomposition. This may result in space savings for large problems.

Description

The function `imsl_f_radial_scattered_fit` computed a least-squares fit to scattered data in $\mathbf{R}^d$ where d = `dimension`. More precisely, let n = `ndata`, x = `abscissae`, f = `fdata`, and d = `dimension`. Then we have

$$x^0, \ldots, x^{n-1} \subset \mathbf{R}^d \; f_0, \ldots, f_{n-1} \subset \mathbf{R}^1$$

This function computes a function F which approximates the above data in the sense that it minimizes the sum-of-squares error

$$\sum_{i=0}^{n-1} w_i \left(F\left(x^i\right) - f_i \right)^2$$

where w = `weights`. Of course, we must restrict the functional form of F. This is done as follows:

$$F(x) := \sum_{j=0}^{k-1} \alpha_j \phi\left(\sqrt{\left\| x - c_j \right\|^2 + \delta^2} \right)$$

The function ϕ is called the radial function. It maps $\mathbf{R}^1$ into $\mathbf{R}^1$, only defined for the nonnegative reals. For the purpose of this routine, the user-supplied function

$$\phi(r) = \sqrt{\left(r^2 + \delta^2\right)}$$

Note that the value of delta is defaulted to 1. It can be set by the user by using the keyword `IMSL_DELTA`. The parameter δ is used to scale the problem. Generally choose δ to be near the minimum spacing of the centers.

The default basis function is called the Hardy multiquadric, and it is defined as

$$\phi(r) = \sqrt{\left(r^2 + \delta^2\right)}$$

A key feature of this routine is the user's control over the selection of the basis function.

To obtain the default selection of centers, we first compute the number of centers that will be on a grid and how many are on a random subset of the abscissae. Next, we compute those centers on a grid. Finally, a random subset of abscissa are obtained determining where the centers are placed. Let us examine the selection of centers in more detail.

First, we restrict the computed grid to have the same number of grid values in each of the `dimension` directions. Then, the number of centers placed on a grid, `num_gridded`, is computed as follows:

$$\alpha = (\texttt{centers\_ratio})\ \ (\texttt{num\_centers})$$

$$\beta = \lfloor \alpha^{1/dimension} \rfloor$$

$$\texttt{num\_gridded} = \beta^{dimension}$$

Note that there are β grid values in each of the `dimension` directions. Then we have

$$\texttt{num\_random} = (\texttt{num\_centers}) - (\texttt{num\_gridded})$$

Now we know how many centers will be placed on a grid and how many will be placed on a random subset of the abscissae. The gridded centers are computed such that they are equally spaced in each of the `dimension` directions. The last problem is to compute a random subset, without replacement, of the abscissa. The selection is based on a random seed. The default seed is 234579. The user can change this using the optional argument `IMSL_RANDOM_SEED`. Once the subset is computed, we use the abscissae as centers.

Since the selection of good centers for a specific problem is an unsolved problem at this time, we have given the ultimate flexibility to the user. That is, you can select your own centers using the keyword `IMSL_CENTERS`. As a rule of thumb, the centers should be interspersed with the abscissae.

The return value for this function is a pointer to the structure, which contains all the information necessary to evaluate the fit. This pointer is then passed to the function `imsl_f_radial_evaluate` to produce values of the fitted function.

Examples

Example 1

This example, generates data from a function and contaminates it with noise on a grid of 10 equally spaced points.The fit is evaluated on a finer grid and compared with the actual function values.

```
#include <imsl.h>
#include <math.h>

#define NDATA          10
```

```
#define NUM_CENTERS    5
#define NOISE_SIZE     0.25
#define F(x)           ((float)(sin(2*pi*x)))

main ()
{
    int        i;
    int        dim = 1;
    float      fdata[NDATA];
    float      *fdata2;
    float      xdata[NDATA];
    float      xdata2[2*NDATA];
    float      pi;
    float      *noise;
    Imsl_f_radial_basis_fit   *radial_fit;

    pi = imsl_f_constant ("pi", 0);

    imsl_random_seed_set (234579);
    noise = imsl_f_random_uniform(NDATA, 0);

/* Set up the sampled data points with noise. */

    for (i = 0; i <  NDATA; ++i) {
       xdata[i] = (float)(i)/(float)(NDATA-1);
       fdata[i] = F(xdata[i]) + NOISE_SIZE*(1.0 - 2.0*noise[i]);
    }
/* Compute the radial fit. */

    radial_fit = imsl_f_radial_scattered_fit (dim, NDATA, xdata,
                 fdata, NUM_CENTERS, 0);

/* Compare result to the original function at twice as many values as
   there were original data points. */

    for (i = 0; i < 2*NDATA; ++i)
        xdata2[i] = (float)(i/(float)(2*(NDATA-1)));
/* Evaluate the fit at these new points. */

    fdata2 = imsl_f_radial_evaluate(2*NDATA, xdata2, radial_fit, 0);

    printf("    I     TRUE       APPROX     ERROR\n");
    for (i = 0; i < 2*NDATA; ++i)
    printf("%5d %10.5f %10.5f %10.5f\n",i+1,F(xdata2[i]), fdata2[i],
            F(xdata2[i])-fdata2[i]);
  }
```

Output

```
 I     TRUE       APPROX     ERROR
 1    0.00000   -0.08980    0.08980
 2    0.34202    0.38795   -0.04593
 3    0.64279    0.75470   -0.11191
 4    0.86603    0.99915   -0.13312
 5    0.98481    1.11597   -0.13116
 6    0.98481    1.10692   -0.12211
 7    0.86603    0.98183   -0.11580
 8    0.64279    0.75826   -0.11547
 9    0.34202    0.46078   -0.11876
```

```
10   -0.00000    0.11996   -0.11996
11   -0.34202   -0.23007   -0.11195
12   -0.64279   -0.55348   -0.08931
13   -0.86603   -0.81624   -0.04979
14   -0.98481   -0.98752    0.00271
15   -0.98481   -1.04276    0.05795
16   -0.86603   -0.96471    0.09868
17   -0.64279   -0.74472    0.10193
18   -0.34202   -0.38203    0.04001
19    0.00000    0.11600   -0.11600
20    0.34202    0.73553   -0.39351
```

Example 2

This example generates data from a function and contaminates it with noise.We fit this data successively on grids of size 10, 20, …, 100. Now interpolate and print the 2-norm of the difference between the interpolated result and actual function values. Note that double precision is used for higher accuracy.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA           100
#define NUM_CENTERS     100
#define NRANDOM         200
#define NOISE_SIZE      1.0
#define G(x,y)          (exp((y)/2.0)*sin(x) - cos((y)/2.0))

double radial_function (double r);

main()
{
    int        i;
    int        ndata;
    double     *fit;
    double     ratio;
    double     fdata[NDATA+1];
    double     xydata[2 * NDATA+1];
    double     pi;
    double     *noise;
    int        num_centers;
    Imsl_d_radial_basis_fit *radial_struct;

    pi = imsl_d_constant ("pi", 0);

    /* Get the random numbers used for the noise. */

    imsl_random_seed_set (234579);
    noise = imsl_d_random_uniform (NRANDOM+1, 0);
    for (i = 0; i < NRANDOM; ++i) noise[i] = 1.0 - 2.0 * noise[i];
    printf("    NDATA          || Error ||_2 \n");

    for (ndata = 10; ndata <= 100 ; ndata += 10) {
        num_centers = ndata;

    /* Set up the sampled data points with noise. */
        for (i = 0; i < 2 * ndata; i += 2) {
```

```
            xydata[i] = 3. * (noise[i]);
            xydata[i + 1] = 3. * (noise[i + 1]);
            fdata[i / 2] = G(xydata[i], xydata[i + 1])
                         + NOISE_SIZE * noise[i];
        }

    /* Compute the radial fit. */
        ratio = 0.5;
        radial_struct= imsl_d_radial_scattered_fit (2, ndata, xydata,
                     fdata, num_centers,
                     IMSL_CENTERS_RATIO, ratio,
                     IMSL_SUPPLY_BASIS, radial_function,
                     0);
        fit = imsl_d_radial_evaluate (ndata, xydata, radial_struct, 0);

        for (i = 0; i < ndata; ++i) fit[i] -= fdata[i];

        printf("%8d %17.8f \n", ndata,
                imsl_d_vector_norm(ndata, fit, 0));
    }

}

double radial_function (double r)
{
    return log(1.0+r);
}
```

Output

```
NDATA          || Error ||_2
 10         0.00000000
 20         0.00000000
 30         0.00000000
 40         0.00000000
 50         0.00000000
 60         0.00000000
 70         0.00000000
 80         0.00000000
 90         0.00000000
100         0.00000000
```

radial_evaluate

Evaluates a radial-basis fit.

Synopsis

#include `<imsl.h>`

float `*imsl_f_radial_evaluate` (*int* `n`, *float* `x[]`,
Imsl_d_radial_basis_fit `*radial_fit`, ..., 0)

The type *double* function is `imsl_d_evaluate`.

Required Arguments

int `n` (Input)
The number of points at which the fit will be evaluated.

float `x[]` (Input)
Array of size (`radial_fit->dimension`) × *n* containing the abscissae of the data points at which the fit will be evaluated. The argument x[`i`][`j`] is the abscissa value of the (`i`+1)-th data point in the (`j`+1)-th dimension.

Imsl_f_radial_basis_fit `*radial_fit` (Input)
A pointer to radial-basis structure to be used for the evaluation. (Input).

Return Value

A pointer to an array of length `n` containing the values of the radial-basis fit at the desired values. If no value can be computed, then `NULL` is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_radial_evaluate` (*int* `n`, *float* `x[]`,
Imsl_f_radial_basis_fit `*radial_fit`
`IMSL_RETURN_USER`, *float* `value[]`,
0)

Optional Arguments

`IMSL_RETURN_USER, value[]` (Input)
A user-allocated array of length `n` containing the returned values.

Description

The function `imsl_f_radial_evaluate` evaluates a radial-basis fit from data generated by `imsl_f_radial_scattered_fit`.

Example

```
#include <imsl.h>
#include <math.h>

#define NDATA          10
#define NUM_CENTERS    5
#define NOISE_SIZE     0.25
#define F(x)           ((float)(sin(2*pi*x)))

main ()
{
    int        i;
    int        dim = 1;
    float      fdata[NDATA];
    float      *fdata2;
    float      xdata[NDATA];
```

```
    float      xdata2[2*NDATA];
    float      pi;
    float      *noise;
    Imsl_f_radial_basis_fit   *radial_fit;

    pi = imsl_f_constant ("pi", 0);

    imsl_random_seed_set (234579);
    noise = imsl_f_random_uniform(NDATA, 0);

/* Set up the sampled data points with noise */

    for (i = 0; i <  NDATA; ++i) {
       xdata[i] = (float)(i)/(float)(NDATA-1);
       fdata[i] = F(xdata[i]) + NOISE_SIZE*(1.0 - 2.0*noise[i]);
    }
/* Compute the radial fit */

    radial_fit = imsl_f_radial_scattered_fit (dim, NDATA, xdata,
                 fdata, NUM_CENTERS, 0);

/* Compare result to the original function at twice as many values as there
   were original data points */

    for (i = 0; i < 2*NDATA; ++i)
        xdata2[i] = (float)(i/(float)(2*(NDATA-1)));

/* Evaluate the fit at these new points */

    fdata2 = imsl_f_radial_evaluate(2*NDATA, xdata2, radial_fit, 0);

    printf("    I     TRUE       APPROX     ERROR\n");
    for (i = 0; i < 2*NDATA; ++i)
    printf("%5d %10.5f %10.5f %10.5f\n",i+1,F(xdata2[i]), fdata2[i],
            F(xdata2[i])-fdata2[i]);
  }
```

Output

```
 I     TRUE       APPROX     ERROR
 1    0.00000   -0.08980    0.08980
 2    0.34202    0.38795   -0.04593
 3    0.64279    0.75470   -0.11191
 4    0.86603    0.99915   -0.13312
 5    0.98481    1.11597   -0.13116
 6    0.98481    1.10692   -0.12211
 7    0.86603    0.98183   -0.11580
 8    0.64279    0.75826   -0.11547
 9    0.34202    0.46078   -0.11876
10   -0.00000    0.11996   -0.11996
11   -0.34202   -0.23007   -0.11195
12   -0.64279   -0.55348   -0.08931
13   -0.86603   -0.81624   -0.04979
14   -0.98481   -0.98752    0.00271
15   -0.98481   -1.04276    0.05795
16   -0.86603   -0.96471    0.09868
17   -0.64279   -0.74472    0.10193
18   -0.34202   -0.38203    0.04001
```

```
19    0.00000    0.11600   -0.11600
20    0.34202    0.73553   -0.39351
```

Chapter 4: Quadrature

Routines

4.1 Univariate Quadrature

Adaptive general-purpose endpoint singularity ..... int_fcn_sing 237
Adaptive general purpose..... int_fcn 241
Adaptive general-purpose points of singularity..... int_fcn_sing_pts 245
Adaptive weighted algebraic singularities..... int_fcn_alg_log 249
Adaptive infinite interval..... int_fcn_inf 253
Adaptive weighted oscillatory (trigonometric)..... int_fcn_trig 257
Adaptive weighted Fourier (trigonometric) ..... int_fcn_fourier 261
Cauchy principal value ..... int_fcn_cauchy 265
Nonadaptive general purpose ..... int_fcn_smooth 268

4.2 Multivariate Quadrature

Two-dimensional iterated integral ..... int_fcn_2d 272
Iterated integral using product Gauss formulas ..... int_fcn_hyper_rect 276
Iterated integral using a quasi-Monte Carlo method ..... int_fcn_qmc 279

4.3 Gauss Quadrature

Gauss quadrature formulas..... gauss_quad_rule 282

4.4 Differentiation

First, second, or third derivative of a function..... fcn_derivative 286

Usage Notes

Univariate Quadrature

The first nine functions in this chapter are designed to compute approximations to integrals of the form

$$\int_c^b f(x)w(x)\,dx$$

The weight function w is used to incorporate known singularities (either algebraic or logarithmic) or to incorporate oscillations. For general-purpose integration, we recommend the use of `imsl_f_int_fcn_sing` (even if no endpoint singularities are

present). If more efficiency is desired, then the use of one of the more specialized functions should be considered. These functions are organized as follows:

- $w = 1$

  `imsl_f_int_fcn_sing`

  `imsl_f_int_fcn`

  `imsl_f_int_fcn_sing_pts`

  `imsl_f_int_fcn_inf`

  `imsl_f_int_fcn_smooth`

- $w(x) = \sin\omega x$ or $w(x) = \cos\omega x$

  `imsl_f_int_fcn_trig` (for a finite interval)

  `imsl_f_int_fcn_fourier` (for an infinite interval)

- $w(x) = (x-a)^{\alpha}(b-x)^{\beta}\ln(x-a)\ln(b-x)$ where the ln factors are optional

  `imsl_f_int_fcn_alg_log`

- $w(x) = 1/(x-c)$

  `imsl_f_int_fcn_cauchy`

The calling sequences for these functions are very similar. The function to be integrated is always `fcn`, and the lower and upper limits are `a` and `b`, respectively. The requested absolute error ε is `err_abs`, while the requested relative error ρ is `err_rel`. These quadrature functions return the estimated answer R. An optional value `err_est` = E estimates the error. These numbers are related as follows:

$$\left|\int_a^b f(x)w(x)\,dx - R\right| \le E \le \max\{\varepsilon, \rho\left|\int_a^b f(x)w(x)\,dx\right|\}$$

Several of the univariate quadrature functions have arguments of type `Imsl_quad`, which is defined in `imsl.h`.

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The functions 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 functions with one of the spline integration functions, which can be found in Chapter 3, "Interpolation and Approximation."

Multivariate Quadrature

Two functions have been included in this chapter that are of use in approximating certain multivariate integrals. In particular, the function `imsl_f_int_fcn_2d` 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 function, `imsl_f_int_fcn_hpyer_rect`, returns an approximation to the integral of a function of *n* variables over a hyper-rectangle

$$\int_{a_1}^{b_1} \ldots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_n \ldots dx_1$$

When working with two-dimensional tensor-product tabular data, use the IMSL spline interpolation function `imsl_f_spline_2d_interp`, followed by the IMSL spline integration function `imsl_f_spline_2d_integral` described in Chapter 3, "Interpolation and Approximation".

Gauss Quadrature

Before computing Gauss quadratures, you must compute so-called Gauss quadrature rules that integrate polynomials of as high degree as possible. These quadrature rules can be easily computed using the function `imsl_f_gauss_quad_rule`, which produces the points $\{w_i\}$ 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
$1/(\sqrt{1-x^2})$	(–1, 1)	Chebyshev 1st kind
$\sqrt{1-x^2}$	(–1, 1)	Chebyshev 2nd kind
e^{-x^2}	(–∞, ∞)	Hermite
$(1+x)^{\alpha}(1-x)^{\beta}$	(–1, 1)	Jacobi
$e^{-x}x^{a}$	(0, ∞)	Generalized Laguerre
1/cosh (*x*)	(–∞, ∞)	Hyperbolic cosine

Where permissible, `imsl_f_gauss_quad_rule` also computes Gauss-Radau and Gauss-Lobatto quadrature rules.

int_fcn_sing

Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules.

Synopsis

#include `<imsl.h>`

float `imsl_f_int_fcn_sing` (*float* `fcn()`, *float* `a`, *float* `b`, ..., 0)

The type *double* function is `imsl_d_int_fcn_sing`.

Required Arguments

float `fcn` (*float* `x`) (input)
User-supplied function to be integrated.

float `a` (Input)
Lower limit of integration.

float `b` (Input)
Upper limit of integration.

Return Value

An estimate of

$$\int_a^b \text{fcn}(x)dx$$

If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_sing` (*float* `fcn()`, *float* `a`, *float* `b`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_ERR_REL`, *float* `err_rel`,
`IMSL_ERR_EST`, *float* `*err_est`,
`IMSL_MAX_SUBINTER`, *int* `max_subinter`,
`IMSL_N_SUBINTER`, *int* `*n_subinter`,
`IMSL_N_EVALS`, *int* `*n_evals`,
`IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: `err_rel` = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

This function is designed to handle functions with endpoint singularities. However, the performance on functions that are well-behaved at the endpoints is also quite good.

The function `imsl_f_int_fcn_sing` 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 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. 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. This function uses an extrapolation procedure known as the ε-algorithm.

The function `imsl_f_int_fcn_sing` is based on the subroutine QAGS by Piessens et al. (1983).

Examples

Example 1

The value of

$$\int_0^1 \ln(x)\, x^{-1/2}\, dx = -4$$

is estimated.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_sing (fcn, 0.0, 1.0, 0);
                                /* Print the result and */
                                /*the exact answer */
    exact = -4.0;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}
```

```
float fcn(float x)
{
    return log(x)/sqrt(x);
}
```

Output

```
integral  =     -4.000
exact     =     -4.000
```

Example 2

The value of

$$\int_0^1 \ln(x) x^{-1/2} dx = -4$$

is again estimated. The values of the actual and estimated errors are printed as well. Note that these numbers are machine dependent. Furthermore, usually the error estimate is pessimistic. That is, the actual error is usually smaller than the error estimate as is in this example.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact, err_est, exact_err;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_sing (fcn, 0.0, 1.0,
                             IMSL_ERR_EST, &err_est,
                             0);
                                /* Print the result and */
                                /* the exact answer */
    exact = -4.0;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
}

float fcn(float x)
{
    return log(x)/sqrt(x);
}
```

Output

```
integral  =     -4.000
exact     =     -4.000
error estimate   = 3.175735e-04
exact error      = 6.556511e-05
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.
IMSL_EXTRAPOLATION_ROUNDOFF	Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

Fatal Errors

IMSL_DIVERGENT	Integral is probably divergent or slowly convergent.
IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn

Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.

Synopsis

#include `<imsl.h>`

float `imsl_f_int_fcn` (*float* `fcn()`, *float* `a`, *float* `b`, …, 0)

The type *double* function is `imsl_d_int_fcn`.

Required Arguments

float `fcn` (*float* *x*) (Input)
User-supplied function to be integrated.

float `a` (Input)
Lower limit of integration.

float `b` (Input)
Upper limit of integration.

Return Value

The value of

$$\int_a^b \mathrm{fcn}(x)dx$$

is returned. If no value can be computed, then NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float imsl_f_int_fcn (float fcn(float x), float a, float b,
        IMSL_RULE, int rule,
        IMSL_ERR_ABS, float err_abs,
        IMSL_ERR_REL, float err_rel,
        IMSL_ERR_EST, float *err_est,
        IMSL_MAX_SUBINTER, int max_subinter,
        IMSL_N_SUBINTER, int *n_subinter,
        IMSL_N_EVALS, int *n_evals,
        IMSL_FCN_W_DATA, float fcn(), void *data,
        0)
```

Optional Arguments

IMSL_RULE, *int* rule (Input)
Choice of quadrature rule.

rule	Gauss-Kronrod Rule
1	7-15 points
2	10-21 points
3	15-31 points
4	20-41 points
5	25-51 points
6	30-61 points

Default: rule = 1

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_int_fcn 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 ($2k$ + 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 function imsl_f_int_fcn is based on the subroutine QAG by Piessens et al. (1983).

Should imsl_f_int_fcn fail to produce acceptable results, consider one of the more specialized functions documented in this chapter.

Examples

Example 1

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is computed. Since the integrand is not oscillatory, all of the default values are used. The values of the actual and estimated error are machine dependent.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);
float           q;
float           exact;

 main()
{
                    /* evaluate the integral */
    q = imsl_f_int_fcn (fcn, 0.0, 2.0, 0);
                    /* print the result and the exact answer */
    exact = exp(2.0) + 1.0;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    float y;
    y =  x *  (exp(x));
    return y;
}
```

Output

```
integral  =      8.389
exact     =      8.389
```

Example 2

The value of

$$\int_0^1 \sin(1/x)dx$$

is computed. Since the integrand is oscillatory, `rule` = 6 is used. The exact value is 0.50406706. The values of the actual and estimated error are machine dependent.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
float           q, err_est, err_abs= 0.0001, exact = 0.50406706, error;

                    /* intergrate fcn(x) from 0 to 1 */
    q = imsl_f_int_fcn (fcn, 0.0, 1.0,
                        IMSL_ERR_ABS,    err_abs,/* set abs error value*/
                        IMSL_RULE,       6,
                        IMSL_ERR_EST,    &err_est, /* pass in address */
                        0);
    error = q - exact;
                    /* print the result and the exact answer */
    printf(" integral = %10.3f\n    exact = %10.3f\n    error = %10.3f\n ",
            q, exact , error);
    printf("   err_est = %g\n", err_est);
}

float fcn(float x)
{
                    /* compute sin(1/x), avoiding division by zero */
    return      ((x)>1.0e-5) ? sin(1.0/(x)) : 0.0;
}
```

Output

```
integral =      0.504
   exact =      0.504
   error =      0.000
   err_est = 0.000170593
```

Warning Errors

`IMSL_ROUNDOFF_CONTAMINATION`	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
`IMSL_PRECISION_DEGRADATION`	A degradation in precision has been detected.

Fatal Errors

`IMSL_MAX_SUBINTERVALS`	The maximum number of subintervals allowed has been reached.

int_fcn_sing_pts

Integrates a function with singularity points given.

Synopsis

#include `<imsl.h>`

float `imsl_f_int_fcn_sing_pts` (*float* `fcn()`, *float* `a`, *float* `b`, *int* `npoints`, *float* `points[]`, ..., 0)

The type *double* function is `imsl_d_int_fcn_sing_pts`.

Required Arguments

float `fcn` (*float* `x`) (Input)
User-supplied function to be integrated.

float `a` (Input)
Lower limit of integration.

float `b` (Input)
Upper limit of integration.

int `npoints` (Input)
The number of singularities of the integrand.

float `points[]` (Input)
The abscissas of the singularities. These values should be interior to the interval [*a, b*].

Return Value

The value of

$$\int_a^b \mathrm{fcn}(x)dx$$

is returned. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_sing_pts` (*float* `fcn()`, *float* `a`, *float* `b`, *int* `npoints`, *float* `points[]`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_ERR_REL`, *float* `err_rel`,
`IMSL_ERR_EST`, *float* `*err_est`,
`IMSL_MAX_SUBINTER`, *int* `max_subinter`,
`IMSL_N_SUBINTER`, *int* `*n_subinter`,

```
IMSL_N_EVALS, int *n_evals,
IMSL_FCN_W_DATA, float fcn(),void *data,
0)
```

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: `err_rel` = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_sing_pts` is a special-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error. It subdivides the interval [*a*, *b*] into `npoints` + 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. 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. This function uses an extrapolation procedure known as the ε-algorithm.

The function `imsl_f_int_fcn_sing_pts` is based on the subroutine QAGP by Piessens et al. (1983).

Examples

Example 1

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 computed. The values of the actual and estimated error are machine dependent. Note that this function never evaluates the user-supplied function at the user-supplied breakpoints.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    int         npoints = 2;
    float       q, exact, points[2];
                                /* Set singular points */
    points[0] = 1.0;
    points[1] = sqrt(2.);
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_sing_pts (fcn, 0.0, 3.0, npoints, points, 0);
                                /* print the result and */
                                /* the exact answer */
    exact = 61.*log(2.) + (77./4)*log(7.) - 27.;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  x*x*x*(log(fabs((x*x-1.)*(x*x-2.))));
}
```

Output

```
integral  =     52.741
exact     =     52.741
```

Example 2

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 again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    int         n_evals, npoints = 2;
    float       q, exact, err_est, exact_err, points[2];
                                /* Set singular points */
    points[0] = 1.0;
    points[1] = sqrt(2.);
                                /* Evaluate the integral and get the */
                                /* error estimate and the number of */
                                /* evaluations */
    q = imsl_f_int_fcn_sing_pts (fcn, 0.0, 3.0, npoints, points,
                                 IMSL_ERR_EST, &err_est,
                                 IMSL_N_EVALS, &n_evals,
                                 0);
                                /* Print the result and the */
                                /* exact answer */
    exact = 61.*log(2.) + (77./4)*log(7.) - 27.;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
            exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    return  x*x*x*(log(fabs((x*x-1.)*(x*x-2.))));
}
```

Output

```
integral  =     52.741
exact     =     52.741
error estimate   = 1.258850e-04
exact error      = 3.051758e-05
The number of function evaluations  =  819
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.
IMSL_EXTRAPOLATION_ROUNDOFF	Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

Fatal Errors

IMSL_DIVERGENT	Integral is probably divergent or slowly convergent.
IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_alg_log

Integrates a function with algebraic-logarithmic singularities.

Synopsis

#include <imsl.h>

float imsl_f_int_fcn_alg_log (*float* fcn(), *float* a, *float* b, *Imsl_quad* weight, *float* alpha, *float* beta, ..., 0)

The type *double* function is imsl_d_int_fcn_alg_log.

Required Arguments

float fcn (*float* x) (Input)
User-supplied function to be integrated.

float a (Input)
Lower limit of integration.

float b (Input)
Upper limit of integration.

Imsl_quad weight, *float* alpha, *float* beta (Input)
These three parameters are used to describe the weight function that may have algebraic or logarithmic singularities at the endpoints. The parameter weight can take on four values as described below. The parameters alpha = α and beta = β specify the strength of the singularities at a or b and hence, must be greater than -1.

weight	Integration Weight
IMSL_ALG	$(x-a)^{\alpha}\,(b-x)^{\beta}$
IMSL_ALG_LEFT_LOG	$(x-a)^{\alpha}\,(b-x)^{\beta}\log(x-a)$
IMSL_ALG_RIGHT_LOG	$(x-a)^{\alpha}\,(b-x)^{\beta}\log(b-x)$
IMSL_ALG_LOG	$(x-a)^{\alpha}\,(b-x)^{\beta}\log(x-a)\log(b-x)$

Return Value

The value of

$$\int_a^b \text{fcn}(x)w(x)\,dx$$

is returned where $w(x)$ is one of the four weights above. If no value can be computed, then NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_alg_log` (*float* `fcn`(*float* `x`), *float* `a`, *float* `b`,
Imsl_quad `weight`, *float* `alpha`, *float* `beta`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_ERR_REL`, *float* `err_rel`,
`IMSL_ERR_EST`, *float* `*err_est`,
`IMSL_MAX_SUBINTER`, *int* `max_subinter`,
`IMSL_N_SUBINTER`, *int* `*n_subinter`,
`IMSL_N_EVALS`, *int* `*n_evals`,
`IMSL_FCN_W_DATA`, *float* `fcn`(), *void* `*data`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the

user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_alg_log` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It 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. This function is based on the subroutine `QAWS`, which is fully documented by Piessens et al. (1983).

Examples

Example 1

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 computed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_alg_log (fcn, 0.0, 1.0,
                                IMSL_ALG_LEFT_LOG,  1.0, 0.5,
                                0);
                                /* Print the result and the */
                                /* exact answer */
    exact = (3.*log(2.)-4.)/9.;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  sqrt(1+x);
}
```

Output

```
integral  =     -0.213
exact     =     -0.213
```

Example 2

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 again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    int         n_evals;
    float       q, exact, err_est, exact_err;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_alg_log (fcn, 0.0, 1.0,
                                IMSL_ALG_LEFT_LOG, 1.0, 0.5,
                                IMSL_ERR_EST, &err_est,
                                IMSL_N_EVALS, &n_evals,
                                0);
                                /* Print the result and the */
                                /* exact answer */
    exact = (3.*log(2.)-4.)/9.;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    return  sqrt(1+x);
}
```

Output

```
integral  =     -0.213
exact     =     -0.213
error estimate   = 3.725290e-09
exact error      = 1.490116e-08
The number of function evaluations  =  50
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.

IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.

Fatal Errors

IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_inf

Integrates a function over an infinite or semi-infinite interval.

Synopsis

#include <imsl.h>

float imsl_f_int_fcn_inf (*float* fcn(), *float* bound, *Imsl_quad* interval, …, 0)

The type *double* procedure is imsl_d_int_fcn_inf.

Required Arguments

float fcn (*float* x) (Input)
User-supplied function to be integrated.

float bound (Input)
Finite limit of integration. This argument is ignored if interval has the value IMSL_INF_INF.

Imsl_quad interval (Input)
Flag indicating integration limits. The following settings are allowed:

interval	Integration Limits
IMSL_INF_BOUND	(–∞, bound)
IMSL_BOUND_INF	(bound, ∞)
IMSL_INF_INF	(–∞, ∞)

Return Value

The value of

$$\int_a^b \text{fcn}(x)\,dx$$

is returned where *a* and *b* are appropriate integration limits. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float imsl_f_int_fcn_inf (*float* fcn, *float* bound, *Imsl_quad* interval, IMSL_ERR_ABS, *float* err_abs,

```
        IMSL_ERR_REL, float err_rel,
        IMSL_ERR_EST, float *err_est,
        IMSL_MAX_SUBINTER, int max_subinter,
        IMSL_N_SUBINTER, int *n_subinter,
        IMSL_N_EVALS, int *n_evals,
        IMSL_FCN_W_DATA, float fcn(), void *data,
        0)
```

Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_int_fcn_inf is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It initially transforms an infinite or semi-infinite interval into the finite interval [0, 1]. It then uses the same strategy as the function imsl_f_int_fcn_sing.

The function imsl_f_int_fcn_inf is based on the subroutine QAGI by Piessens et al. (1983).

Examples

Example 1

The value of

$$\int_0^\infty \frac{\ln(x)}{1+(10x)^2}dx = \frac{-\pi \ln(10)}{20}$$

is computed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact, pi;

    pi = imsl_f_constant("pi", 0);
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_inf (fcn, 0.0,
                            IMSL_BOUND_INF,
                            0);
                                /* Print the result and the */
                                /* exact answer */
    exact = -pi*log(10.)/20.;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    float       z;
    z = 10.*x;
    return  log(x)/(1+ z*z);
}
```

Output

```
integral  =     -0.362
exact     =     -0.362
```

Example 2

The value of

$$\int_0^\infty \frac{\ln x}{1+(10x)^2}dx = \frac{-\pi \ln(10)}{20}$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

 main()
{
    int         n_evals;
    float       q, exact, err_est, exact_err, pi;

    pi = imsl_f_constant("pi", 0);
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_inf (fcn, 0.0,
                            IMSL_BOUND_INF,
                            IMSL_ERR_EST, &err_est,
                            IMSL_N_EVALS, &n_evals,
                            0);
                                /* Print the result and the */
                                /* exact answer */
    exact = -pi*log(10.)/20.;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    float       z;
    z = 10.*x;
    return  log(x)/(1+ z*z);
}
```

Output

```
integral  =     -0.362
exact     =     -0.362
error estimate   = 2.801418e-06
exact error      = 2.980232e-08
The number of function evaluations  =  285
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.
IMSL_EXTRAPOLATION_ROUNDOFF	Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

Fatal Errors

IMSL_DIVERGENT	Integral is probably divergent or slowly convergent.
IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_trig

Integrates a function containing a sine or a cosine factor.

Synopsis

#*include* <imsl.h>

float imsl_f_int_fcn_trig (*float* fcn(), *float* a, *float* b, *Imsl_quad* weight, *float* omega, ..., 0)

The type *double* function is imsl_d_int_fcn_trig.

Required Arguments

float fcn (*float* x) (Input)
User-supplied function to be integrated.

float a (Input)
Lower limit of integration.

float b (Input)
Upper limit of integration.

Imsl_quad weight and *float* omega (Input)
These two parameters are used to describe the trigonometric weight. The parameter weight can take on the two values described below, and the parameter omega = ω specifies the frequency of the trigonometric weighting function.

weight	Integration Weight
IMSL_COS	cos (ω*x*)
IMSL_SIN	sin (ω*x*)

Return Value

The value of

$$\int_a^b \text{fcn}(x)\cos(\omega x)\,dx$$

is returned if weight = IMSL_COS. If weight = IMSL_SIN, then the cosine factor is replaced with a sine factor. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
float imsl_f_int_fcn_trig (float fcn(), float a, float b, Imsl_quad weight,
        float omega,
        IMSL_ERR_ABS, float err_abs,
        IMSL_ERR_REL, float err_rel,
        IMSL_ERR_EST, float *err_est,
        IMSL_MAX_SUBINTER, int max_subinter,
        IMSL_N_SUBINTER, int *n_subinter,
        IMSL_N_EVALS, int *n_evals,
        IMSL_MAX_MOMENTS, int max_moments,
        IMSL_FCN_W_DATA, float fcn(), void *data,
        0)
```

Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_MAX_MOMENTS, *int* max_moments (Input)
This is an upper bound on the number of Chebyshev moments that can be stored. Increasing (decreasing) this number may increase (decrease) execution speed and space used.

Default: max_moments = 21

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_trig` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form $w(x)f(x)$ where $w(x)$ is either $\cos(\omega x)$ or $\sin(\omega x)$. Depending on the length of the subinterval in relation to the size of ω, either a modified Clenshaw-Curtis procedure or a Gauss-Kronrod 7/15 rule is employed to approximate the integral on a subinterval. This function uses the general strategy of the function `imsl_f_int_fcn_sing`. The function `imsl_f_int_fcn_trig` is based on the subroutine QAWO by Piessens et al. (1983).

Examples

Example 1

The value of

$$\int_0^1 \ln(x)\sin(10\pi x)\,dx$$

is computed. Notice that we have coded around the singularity at zero. This is necessary since this procedure evaluates the integrand at the two endpoints.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact, omega;

    omega = 10*imsl_f_constant("pi", 0);
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_trig (fcn, 0.0, 1.0,
                             IMSL_SIN, omega,
                             0);
                                /* Print the result and the */
                                /* exact answer */
    exact = -.1281316;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  (x==0.0) ? 0.0 : log(x);
}
```

Output

```
integral  =     -0.128
exact     =     -0.128
```

Example 2

The value of

$$\int_0^1 \ln(x)\sin(10\pi x)\,dx$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, it is usually the case that the error estimate is pessimistic. That is, the actual error is usually smaller than the error estimate as is the case in this example. The number of function evaluations are also printed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

 main()
{
    int         n_evals;
    float       q, exact, omega, err_est, exact_err;

    omega = 10*imsl_f_constant("pi", 0);
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_trig (fcn, 0.0, 1.0,
                             IMSL_SIN, omega,
                             IMSL_ERR_EST, &err_est,
                             IMSL_N_EVALS, &n_evals,
                             0);
                                /* Print the result and the */
                                /* exact answer */
    exact = -.1281316;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
            exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    return  (x==0.0) ? 0.0 : log(x);
}
```

Output

```
integral  =     -0.128
exact     =     -0.128
error estimate   = 7.504603e-05
exact error      = 5.245209e-06
The number of function evaluations  =  215
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.
IMSL_EXTRAPOLATION_ROUNDOFF	Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

Fatal Errors

IMSL_DIVERGENT	Integral is probably divergent or slowly convergent.
IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_fourier

Computes a Fourier sine or cosine transform.

Synopsis

#include <imsl.h>

float imsl_f_int_fcn_fourier (*float* fcn(), *float* a, *Imsl_quad* weight, *float* omega, ..., 0)

The type *double* function is imsl_d_int_fcn_fourier.

Required Arguments

float fcn (*float* x) (Input)
User-supplied function to be integrated.

float a (Input)
Lower limit of integration. The upper limit of integration is ∞.

Imsl_quad weight and *float* omega (Input)
These two parameters are used to describe the trigonometric weight. The parameter weight can take on the two values described below, and the parameter omega = ∞ specifies the frequency of the trigonometric weighting function.

weight	Integration Weight
IMSL_COS	cos (ω*x*)
IMSL_SIN	sin (ω*x*)

Return Value

The return value is

$$\int_a^{\infty} \text{fcn}(x)\cos(\omega x)\,dx$$

if `weight` = `IMSL_COS`. If `weight` = `IMSL_SIN`, then the cosine factor is replaced with a sine factor. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_fourier` (*float* `fcn()`, *float* `a`, *Imsl_quad* `weight`,
float `omega`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_ERR_EST`, *float* `*err_est`,
`IMSL_MAX_SUBINTER`, *int* `max_subinter`,
`IMSL_MAX_CYCLES`, *int* `max_cycles`,
`IMSL_MAX_MOMENTS`, *int* `max_moments`,
`IMSL_N_CYCLES`, *int* `*n_cycles`,
`IMSL_N_EVALS`, *int* `*n_evals`,
`IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_MAX_CYCLES`, *int* `max_cycles` (Input)
Number of cycles allowed.
Default: `max_subinter` = 50

`IMSL_MAX_MOMENTS`, *int* `max_moments` (Input)
Number of subintervals allowed in the partition of each cycle.
Default: `max_moments` = 21

`IMSL_N_CYCLES`, *int* `*n_cycles` (Output)
Address to store the number of cycles generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_fourier` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It 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 function `imsl_f_int_fcn_trig` followed by extrapolation.

The function `imsl_f_int_fcn_fourier` is based on the subroutine QAWF by Piessens et al. (1983).

Examples

Example 1

The value of

$$\int_0^{\infty} x^{-1/2} \cos\left(\pi x/2\right) dx = 1$$

is computed. Notice that the integrand is coded to protect for the singularity at zero.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact, omega;

    omega = imsl_f_constant("pi",0) / 2.;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_fourier (fcn, 0.0,
                                IMSL_COS, omega,
                                0);
                                /* Print the result and the */
                                /* exact answer */
    exact = 1.0;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  (x==0.) ? 0. : 1./sqrt(x);
}
```

Output

```
integral  =      1.000
exact     =      1.000
```

Example 2

The value of

$$\int_0^\infty x^{-1/2} \cos\left(\pi x/2\right) dx = 1$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example.The number of function evaluations also are printed. Notice that the integrand is coded to protect for the singularity at zero.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    int         n_evals;
    float       q, exact, omega, err_est, exact_err;

    omega = imsl_f_constant("pi",0) / 2.0;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_fourier (fcn, 0.0,
                                IMSL_COS, omega,
                                IMSL_ERR_EST, &err_est,
                                IMSL_N_EVALS, &n_evals,
                                0);
                                /* Print the result and the */
                                /* exact answer */
    exact = 1.;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    return  (x==0.) ? 0. : 1./sqrt(x);
}
```

Output

```
integral  =      1.000
exact     =      1.000
error estimate   = 1.803637e-04
exact error      = 1.013279e-06
The number of function evaluations  =  405
```

Warning Errors

IMSL_BAD_INTEGRAND_BEHAVIOR	Bad integrand behavior occurred in one or more cycles.
IMSL_EXTRAPOLATION_PROBLEMS	Extrapolation table constructed for convergence acceleration of the series formed by the integral contributions of the cycles does not converge to the requested accuracy.

Fatal Errors

IMSL_MAX_CYCLES	Maximum number of cycles allowed has been reached.

int_fcn_cauchy

Computes integrals of the form

$$\int_a^b \frac{f(x)}{x-c}\,dx$$

in the Cauchy principal value sense.

Synopsis

#include `<imsl.h>`

float `imsl_f_int_fcn_cauchy` (*float* `fcn()`, *float* `a`, *float* `b`, *float* `c`, ..., 0)

The type *double* function is `imsl_d_int_fcn_cauchy`.

Required Arguments

float `fcn` (*float* `x`) (Input)
User-supplied function to be integrated.

float `a` (Input)
Lower limit of integration.

float `b` (Input)
Upper limit of integration.

float `c` (Input)
Singular point, *c* must not equal *a* or *b*.

Return Value

The value of

$$\int_a^b \frac{\mathrm{fcn}(x)}{x-c}\,dx$$

is returned. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
float imsl_f_int_fcn_cauchy (float fcn(), float a, float b, float c,
        IMSL_ERR_ABS, float err_abs,
        IMSL_ERR_REL, float err_rel,
        IMSL_ERR_EST, float *err_est,
        IMSL_MAX_SUBINTER, int max_subinter,
        IMSL_N_SUBINTER, int *n_subinter,
        IMSL_N_EVALS, int *n_evals,
        IMSL_FCN_W_DATA, float fcn(), void *data,
        0)
```

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_cauchy` uses a globally adaptive scheme in an attempt to reduce the absolute error. It 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.

The function `imsl_f_int_fcn_cauchy` is an implementation of the subroutine QAWC by Piessens et al. (1983).

Examples

Example 1

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 computed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_cauchy (fcn, -1.0, 5.0, 0.0, 0);
                                /* Print the result and the */
                                /* exact answer */
    exact = log(125./631.)/18.;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  1.0/(5.0*x*x*x+6.0);
}
```

Output

```
integral  =     -0.090
exact     =     -0.090
```

Example 2

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 again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example. The number of function evaluations also are printed.

```
#include <math.h>
#include <imsl.h>
```

```
float           fcn(float x);

main()
{
    int         n_evals;
    float       q, exact, err_est, exact_err;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_cauchy (fcn, -1.0, 5.0, 0.0,
                               IMSL_ERR_EST, &err_est,
                               IMSL_N_EVALS, &n_evals,
                               0);
                                /* Print the result and the */
                                /* exact answer */
    exact = log(125./631.)/18.;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
           exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x)
{
    return  1.0/(5.0*x*x*x+6.0);
}
```

Output

```
integral  =     -0.090
exact     =     -0.090
error estimate   = 2.160174e-06
exact error      = 0.000000e+00
The number of function evaluations  =  215
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.

Fatal Errors

IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_smooth

Integrates a smooth function using a nonadaptive rule.

Synopsis

#include <imsl.h>

float `imsl_f_int_fcn_smooth` (*float* `fcn()`, *float* `a`, *float* `b`, ..., 0)

The type *double* function is `imsl_d_int_fcn_smooth`.

Required Arguments

float `fcn` (*float* `x`) (Input)
User-supplied function to be integrated.

float `a` (Input)
Lower limit of integration.

float `b` (Input)
Upper limit of integration.

Return Value

The value of

$$\int_a^b \text{fcn}(x)\,dx$$

is returned. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_smooth` (*float* `fcn()`, *float* `a`, *float* `b`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_ERR_REL`, *float* `err_rel`,
`IMSL_ERR_EST`, *float* `*err_est`,
`IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the

user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_smooth` is designed to integrate smooth functions. It 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 function `imsl_f_int_fcn_smooth` applies these rules successively, estimating the error, until either the error estimate satisfies the user-supplied constraints or the last rule is applied.

This function is not very robust, but for certain smooth functions it can be efficient. If `imsl_f_int_fcn_smooth` should not perform well, we recommend the use of the function `imsl_f_int_fcn_sing`.

The function `imsl_f_int_fcn_smooth` is based on the subroutine QNG by Piessens et al. (1983).

Examples

Example 1

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is computed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

main()
{
    float       q, exact;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_smooth (fcn, 0., 2., 0);
                                /* Print the result and the */
                                /* exact answer */
    exact = exp(2.0) + 1.0;
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return  x * exp(x);
}
```

Output

```
integral  =      8.389
exact     =      8.389
```

Example 2

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x);

 main()
{
    float       q, exact, err_est, exact_err;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_smooth (fcn, 0.0, 2.0,
                               IMSL_ERR_EST, &err_est,
                               0);
                                /* Print the result and the */
                                /* exact answer */
    exact = exp(2.0) + 1.0;
    exact_err = fabs(exact - q);
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
}

float fcn(float x)
{
    return  x * exp(x);
}
```

Output

```
integral  =      8.389
exact     =      8.389
error estimate   = 5.000267e-05
exact error      = 9.536743e-07
```

Fatal Errors

IMSL_MAX_STEPS	The maximum number of steps allowed have been taken. The integrand is too difficult for this routine.

int_fcn_2d

Computes a two-dimensional iterated integral.

Synopsis

#*include* <imsl.h>

float imsl_f_int_fcn_2d (*float* fcn(), *float* a, *float* b, *float* gcn (*float* x), *float* hcn (*float* x), …, 0)

The type *double* function is imsl_d_int_fcn_2d.

Required Arguments

float fcn (*float* x, *float* y) (Input)
User-supplied function to be integrated.

float a (Input)
Lower limit of outer integral.

float b (Input)
Upper limit of outer integral.

float gcn (*float* x) (Input)
User-supplied function to evaluate the lower limit of the inner integral.

float hcn (*float* x) (Input)
User-supplied function to evaluate the upper limit of the inner integral.

Return Value

The value of

$$\int_a^b \int_{gcn(x)}^{hcn(x)} \mathrm{fcn}(x, y)\, dydx$$

is returned. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#*include* <imsl.h>

float imsl_f_int_fcn_2d (*float* fcn(), *float* a, *float* b, *float* gcn (), *float* hcn (),
IMSL_ERR_ABS, *float* err_abs,
IMSL_ERR_REL, *float* err_rel,
IMSL_ERR_EST, *float* *err_est,
IMSL_MAX_SUBINTER, *int* max_subinter,
IMSL_N_SUBINTER, *int* *n_subinter,
IMSL_N_EVALS, *int* *n_evals,
IMSL_FCN_W_DATA, *float* fcn(), *void* *data,
IMSL_GCN_W_DATA, *float* gcn(), *void* *data,
IMSL_HCN_W_DATA, *float* hcn(), *void* *data,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *float* `y`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_GCN_W_DATA`, *float* `gcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to evaluate the lower limit of the inner integral, which also accepts a pointer to data that is supplied by the user. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_HCN_W_DATA`, *float* `hcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)
User supplied function to evaluate the upper limit of the inner integral, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_2d` approximates the two-dimensional iterated integral

$$\int_a^b \int_{g(x)}^{h(x)} f(x,y)\, dydx$$

An estimate of the error is returned in `err_est`. The lower-numbered rules are used for less smooth integrands while the higher-order rules are more efficient for smooth (oscillatory) integrands.

Examples

Example 1

In this example, compute the value of the integral

$$\int_0^1 \int_1^3 y\cos\left(x+y^2\right) dydx$$

```
#include <math.h>
#include <imsl.h>

float           fcn(float x, float y), gcn(float x), hcn(float x);

main()
{
    float       q, exact;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_2d (fcn, 0.0, 1.0, gcn, hcn, 0);
                    /* print the result and the exact answer */
    exact = 0.5*(cos(9.0)+cos(2.0)-cos(10.0)-cos(1.0));
    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x, float y)
{
    return  y * cos(x+y*y);
}

float gcn(float x)
{
    return 1.0;
}

float hcn(float x)
{
    return 3.0;
}
```

Output

```
integral  =     -0.514
exact     =     -0.514
```

Example 2

In this example, compute the value of the integral

$$\int_0^1 \int_1^3 y\cos\left(x+y^2\right) dydx$$

The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example. The number of function evaluations also are printed.

```
#include <math.h>
#include <imsl.h>

float           fcn(float x, float y), gcn(float x), hcn(float x);

main()
{
    int         n_evals;
    float       q, exact, err_est, exact_err;
                                /* Evaluate the integral */
    q = imsl_f_int_fcn_2d (fcn, 0., 1., gcn, hcn,
                           IMSL_ERR_EST, &err_est,
                           IMSL_N_EVALS, &n_evals,
                           0);
                                /* Print the result and the */
                                /* exact answer */
    exact = 0.5*(cos(9.0)+cos(2.0)-cos(10.0)-cos(1.0));
    exact_err = fabs(exact - q);


    printf("integral  = %10.3f\nexact     = %10.3f\n", q, exact);
    printf("error estimate   = %e\nexact error      = %e\n", err_est,
             exact_err);
    printf("The number of function evaluations  =  %d\n", n_evals);
}

float fcn(float x, float y)
{
    return  y * cos(x+y*y);
}

float gcn(float x)
{
    return 1.0;
}

float hcn(float x)
{
    return 3.0;
}
```

Output

```
integral  =     -0.514
exact     =     -0.514
error estimate   = 3.065193e-06
exact error      = 1.192093e-07
The number of function evaluations  =  441
```

Warning Errors

IMSL_ROUNDOFF_CONTAMINATION	Roundoff error, preventing the requested tolerance from being achieved, has been detected.
IMSL_PRECISION_DEGRADATION	A degradation in precision has been detected.

Fatal Errors

IMSL_MAX_SUBINTERVALS	The maximum number of subintervals allowed has been reached.

int_fcn_hyper_rect

Integrate a function on a hyper-rectangle,

$$\int_{a_0}^{b_0} \cdots \int_{a_{n-1}}^{b_{n-1}} f\left(x_0, \ldots, x_{n-1}\right) dx_{n-1} \ldots dx_0$$

Synopsis

#include <imsl.h>

float imsl_f_int_fcn_hyper_rect (*float* fcn(), *int* ndim, *float* a[], *float* b[], ..., 0)

The type *double* function is imsl_d_int_fcn_hyper_rect.

Required Arguments

float fcn (*int* ndim, *float* x[]) (Input)
User-supplied function to be integrated.

int ndim (Input)
The dimension of the hyper-rectangle.

float a[] (Input)
Lower limits of integration.

float b[] (Input)
Upper limits of integration.

Return Value

The value of

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f\left(x_0, \ldots, x_{n-1}\right) dx_{n-1} \ldots dx_0$$

is returned. If no value can be computed, then NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_hyper_rect` (*float* `fcn()`, *int* `ndim`, *float* `a[]`, *float* `b[]`, `IMSL_ERR_ABS`, *float* `err_abs`, `IMSL_ERR_REL`, *float* `err_rel`, `IMSL_ERR_EST`, *float* `*err_est`, `IMSL_MAX_EVALS`, *int* `max_evals`, `IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`, 0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: err_abs = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: err_rel = $\sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_EVALS`, *int* `max_evals` (Input)
Number of evaluations allowed.

Default: `max_evals` = 32^n.

`IMSL_FCN_W_DATA`, *float* `fcn` (*int* `ndim`, *float* `x[]`, *void* `*data`), *void* `*data` (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_int_fcn_hyper_rect` approximates the *n*-dimensional iterated integral

$$\int_{a_0}^{b} \ldots \int_{a_{n-1}}^{b_{n-1}} f\left(x_0, \ldots, x_{n-1}\right) dx_{n-1} \ldots dx_0$$

An estimate of the error is returned in the optional argument `err_est`. 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 function calculates a new estimate by doubling the number of points in the i-th direction, then 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; the number of Gauss points in one direction exceeds 256; or the number of function evaluations needed to complete a sweep exceeds `max_evals`.

Example

In this example, we compute 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 $\mathbf{R}^3$ is $\pi^{3/2}$.

```
#include <math.h>
#include <imsl.h>

float           fcn(int n, float x[]);

main()
{
    int         i, j, ndim = 3;
    float       q, limit, a[3], b[3];

    printf("       integral       limit \n");
    limit = pow(imsl_f_constant("pi",0), 1.5);
                                /* Evaluate the integral */
    for (i = 0;  i < 6;  i++) {
        for (j = 0; j < 3;  j++) {
            a[j] = -(i+1)/2.;
            b[j] = (i+1)/2.;
        }
        q = imsl_f_int_fcn_hyper_rect (fcn, ndim, a, b, 0);
                                /* Print the result and the */
                                /* limiting answer */
        printf("   %10.3f    %10.3f\n", q, limit);
    }
}

float fcn(int n, float x[])
{
    float     s;
    s = x[0]*x[0] + x[1]*x[1] + x[2]*x[2];
    return  exp(-s);
}
```

Output

```
integral        limit
 0.785          5.568
 3.332          5.568
 5.021          5.568
 5.491          5.568
 5.561          5.568
 5.568          5.568
```

Warning Errors

IMSL_MAX_EVALS_TOO_LARGE	The argument max_evals was set greater than 2^{8n}.

Fatal Errors

IMSL_NOT_CONVERGENT	The maximum number of function evaluations has been reached, and convergence has not been attained.

int_fcn_qmc

Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.

Synopsis

#include <imsl.h>

float imsl_f_int_fcn_qmc (*float* fcn(), *int* ndim, *float* a[], *float* b[], ..., 0)

The type *double* function is imsl_d_int_fcn_qmc.

Required Arguments

float fcn (*int* ndim, *float* x[]) (Input)
User-supplied function to be integrated.

int ndim (Input)
The dimension of the hyper-rectangle.

float a[] (Input)
Lower limits of integration.

float b[] (Input)
Upper limits of integration.

Return Value

The value of

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f\left(x_0, \ldots, x_{n-1}\right) dx_{n-1} \ldots dx_0$$

is returned. If no value can be computed, then NaN is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_int_fcn_qmc (*float* fcn(), *int* ndim, *float* a[], *float* b[],
IMSL_ERR_ABS, *float* err_abs,
IMSL_ERR_REL, *float* err_rel,
IMSL_ERR_EST, *float* *err_est,
IMSL_MAX_EVALS, *int* max_evals,
IMSL_BASE, *int* base,
IMSL_SKIP, *int* skip,
IMSL_FCN_W_DATA, *float* fcn(), *void* *data,
0)

Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: err_abs = 1.0e-4.

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: err_abs = 1.0e-4.

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_EVALS, *int* max_evals (Input)
Number of evaluations allowed.
Default: No limit.

IMSL_MAX_EVALS, *int* max_evals (Input)
Number of evaluations allowed.
Default: No limit.

IMSL_BASE, *int* base (Input)
The value of IMSL_BASE used to compute the Faure sequence.

IMSL_SKIP, *int* skip (Input)
The value of IMSL_SKIP used to compute the Faure sequence.

IMSL_FCN_W_DATA, *float* fcn (*int* ndim, *float* x[], *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Integration of functions over hypercubes by direct methods, such as imsl_f_fcn_hyper_rect, is practical only for fairly low dimensional hypercubes. This is because the amount of work required increases exponential as the dimension increases.

An alternative to direct methods is Monte Carlo, 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 $1/n^{1/2}$, where n is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo 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 at 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 `imsl__f_faure_next_point`.

Example

```
#include <imsl.h>
#include <math.h>

float fcn(int ndim, float x[]);

main()
{
    int         k, ndim = 10;
    float       q, a[10], b[10];

    for (k = 0;  k < ndim;  k++) {
        a[k] = 0.0;
        b[k] = 1.0;
    }

    q = imsl_f_int_fcn_qmc (fcn, ndim, a, b, 0);
    printf ("integral=%10.3f\n", q);
}

float fcn (int ndim, float x[])
{
    int         i, j;
    float       prod, sum = 0.0, sign = -1.0;

    for (i = 0;  i < ndim;  i++) {
        prod = 1.0;
        for (j = 0;  j <= i;  j++) {
            prod *= x[j];
        }
        sum += sign * prod;
        sign = -sign;
    }
    return sum;
}
```

Output

```
q = -0.333
```

Fatal Errors

IMSL_NOT_CONVERGENT	The maximum number of function evaluations has been reached and convergence has not been attained.

gauss_quad_rule

Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

Synopsis

#include <imsl.h>

void imsl_f_gauss_quad_rule (*int* n, *float* weights[], *float* points[], ..., 0)

The type *double* procedure is imsl_d_gauss_quad_rule.

Required Arguments

int n (Input)
Number of quadrature points.

float weights[] (Output)
Array of length *n* containing the quadrature weights.

float points[] (Output)
Array of length *n* containing quadrature points. The default action of this routine is to produce the Gauss Legendre points and weights.

Synopsis with Optional Arguments

#include <imsl.h>

void imsl_f_gauss_quad_rule (*int* n, *float* weights[], *float* points[],
IMSL_CHEBYSHEV_FIRST,
IMSL_CHEBYSHEV_SECOND,
IMSL_HERMITE,
IMSL_COSH,
IMSL_JACOBI, *float* alpha, *float* beta,
IMSL_GEN_LAGUERRE, *float* alpha,
IMSL_FIXED_POINT, *float* a,
IMSL_TWO_FIXED_POINTS, *float* a, *float* b,
0)

Optional Arguments

IMSL_CHEBYSHEV_FIRST
Compute the Gauss points and weights using the weight function

$$1/\sqrt{1-x^2}$$

on the interval (–1, 1).

`IMSL_CHEBYSHEV_SECOND`

Compute the Gauss points and weights using the weight function

$$\sqrt{1-x^2}$$

on the interval (–1, 1).

`IMSL_HERMITE`

Compute the Gauss points and weights using the weight function $\exp(-x^2)$ on the interval $(-\infty, \infty)$.

`IMSL_COSH`

Compute the Gauss points and weights using the weight function $1 / (\cosh(x))$ on the interval $(-\infty, \infty)$.

`IMSL_JACOBI`, *float* `alpha`, *float* `beta` (Input)

Compute the Gauss points and weights using the weight function $(1-x)^{\alpha}(1+x)^{\beta}$ on the interval (–1, 1).

`IMSL_GEN_LAGUERRE`, *float* `alpha` (Input)

Compute the Gauss points and weights using the weight function $\exp(-x)x^{\alpha}$ on the interval $(0, \infty)$.

`IMSL_FIXED_POINT`, *float* `a` (Input)

Compute the Gauss-Radau points and weights using the specified weight function and the fixed point *a*. This formula will integrate polynomials of degree less than $2n - 1$ exactly.

`IMSL_TWO_FIXED_POINTS`, *float* `a`, *float* `b` (Input)

Compute the Gauss-Lobatto points and weights using the specified weight function and the fixed points *a* and *b*. This formula will integrate polynomials of degree less than $2n - 2$ exactly.

Description

The function `imsl_f_gauss_quad_rule` produces the points and weights for the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature formulas for some of the most popular weights. The default weight is the weight function identically equal to 1 on the interval (–1, 1). 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 function is a modification of the subroutine GAUSSQUADRULE (Golub and Welsch 1969).

In the default case, the function returns points in *x* = `points` and weights in *w* = weights 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 the keyword IMSL_FIXED_POINT is specified, then one of the above x_i is equal to a. Similarly, if the keyword IMSL_TWO_FIXED_POINTS is specified, then two of the components of x are equal to a and b. In general, the accuracy of the above quadrature formula degrades when n increases. The quadrature rule will integrate all functions f that are polynomials of degree less than $2n - F$, where F is the number of fixed points.

Examples

Example 1

The three-point Gauss Legendre quadrature points and weights are computed and used to approximate the integrals

$$\int_{-1}^{1} x^i dx \qquad i = 0, \ldots, 6$$

Notice that the integrals are exact for the first six monomials, but that the last approximation is in error. In general, the Gauss rules with k points integrate polynomials with degree less than $2k$ exactly.

```
#include <math.h>
#include <imsl.h>

#define  QUADPTS  3
#define  POWERS   7

main()
{
    int         i, j;
    float       weights[QUADPTS], points[QUADPTS], s[POWERS];
                                /* Produce the Gauss Legendre */
                                /* quadrature points */
    imsl_f_gauss_quad_rule (QUADPTS, weights, points, 0);
                                /* integrate the functions */
                                /* 1, x, ..., pow(x,POWERS-1) */
    for(i = 0;  i < POWERS;  i++) {
        s[i] = 0.0;
        for(j = 0;  j < QUADPTS;  j++) {
            s[i] += weights[j]*imsl_fi_power(points[j], i);
        }
    }
    printf("The integral from -1 to 1 of pow(x, i) is\n");
    printf("Function            Quadrature    Exact\n\n");
    for(i = 0;  i < POWERS;  i++){
        float   z;
        z = (1-i%2)*2./(i+1.);
        printf("pow(x, %d)        %10.3f  %10.3f\n", i, s[i], z);
    }
}
```

Output

```
The integral from -1 to 1 of pow(x, i) is
Function            Quadrature    Exact

pow(x, 0)             2.000       2.000
pow(x, 1)             0.000       0.000
pow(x, 2)             0.667       0.667
pow(x, 3)             0.000       0.000
pow(x, 4)             0.400       0.400
pow(x, 5)             0.000       0.000
pow(x, 6)             0.240       0.286
```

Example 2

The three-point Gauss Laguerre quadrature points and weights are computed and used to approximate the integrals

$$\int_0^\infty x^i x e^{-x} dx = i! \qquad i = 0, \ldots, 6$$

Notice that the integrals are exact for the first six monomials, but that the last approximation is in error. In general, the Gauss rules with k points integrate polynomials with degree less than $2k$ exactly.

```
#include <math.h>
#include <imsl.h>

#define  QUADPTS  3
#define  POWERS   7

main()
{
    int         i, j;
    float       weights[QUADPTS], points[QUADPTS], s[POWERS], z;
                                /* Produce the Gauss Legendre */
                                /* quadrature points */
    imsl_f_gauss_quad_rule (QUADPTS, weights, points,
                            IMSL_GEN_LAGUERRE, 1.0,

                            0);
                                /* Integrate the functions */
                                /* 1, x, ..., pow(x,POWERS-1) */
    for(i = 0;  i < POWERS;  i++) {
        s[i] = 0.0;
        for(j = 0;  j < QUADPTS;  j++){
            s[i] += weights[j]*imsl_fi_power(points[j], i);
        }
    }
    printf("The integral from 0 to infinity of pow(x, i)*x*exp(x) is\n");
    printf("Function            Quadrature    Exact\n\n");
    for(z = 1.0, i = 0;  i < POWERS;  i++){
        z *= (i+1);
        printf("pow(x, %d)        %10.3f  %10.3f  \n", i, s[i], z);
    }
}
```

Output

```
The integral from 0 to infinity of pow(x, i)*x*exp(x) is
Function            Quadrature    Exact

pow(x, 0)             1.000       1.000
pow(x, 1)             2.000       2.000
pow(x, 2)             6.000       6.000
pow(x, 3)            24.000      24.000
pow(x, 4)           120.000     120.000
pow(x, 5)           720.000     720.000
pow(x, 6)          4896.000    5040.000
```

fcn_derivative

Computes the first, second, or third derivative of a user-supplied function.

Synopsis

#include `<imsl.h>`

float `imsl_f_fcn_derivative` (*float* `fcn()`, *float* `x`, …, 0)

The type *double* procedure is `imsl_d_fcn_derivative`.

Required Arguments

float `fcn`(*float* `x`) (Input)
User-supplied function whose derivative at `x` will be computed.

float `x` (Input)
Point at which the derivative will be evaluated.

Return Value

An estimate of the first, second or third derivative of `fcn` at `x`. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`
float `imsl_f_fcn_derivative` (*float* `fcn()`, *float* `x`,
`IMSL_ORDER`, *int* `order`,
`IMSL_INITIAL_STEPSIZE`, *float* `stepize`,
`IMSL_RELATIVE_ERROR`, *float* `tolerance`,
`IMSL_FCN_W_DATA`, *float* `fcn`(), *void* `*data`,
0)

Optional Arguments

`IMSL_ORDER`, *int* `order` (Input)
The order of the desired derivative (1, 2 or 3).
Default: `order` = 1.

IMSL_INITIAL_STEPSIZE, *float* stepsize (Input)

Beginning value used to compute the size of the interval for approximating the derivative. Stepsize must be chosen small enough that fcn is defined and reasonably smooth in the interval (x – 4.0*stepsize, x + 4.0*stepsize), yet large enough to avoid roundoff problems.

Default: stepsize = .01

IMSL_RELATIVE_ERROR, *float* tolerance (Input)

The relative error desired in the derivative estimate. Convergence is assumed when (2/3) $|d_2 - d_1| <$ tolerance, for two successive derivative estimates, d_1 and d_2.

Default: tolerance = $\sqrt[4]{\varepsilon}$

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)

User supplied function whose derivative at x will be computed, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_fcn_derivative 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 value within the interval (x – 4.0*stepsize, x + 4.0*stepsize), then differentiating the spline at x.

Examples

Example 1

This example obtains the approximate first derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
#include <imsl.h>
#include <math.h>

void main()
{
    float fcn(float);
    float x;
    float deriv;

    x = 2.0;

    deriv = imsl_f_fcn_derivative(fcn, x, 0);
    printf ("f'(x) = %7.4f\n", deriv);
}

float fcn(float x)
{
```

```
    return -2.0*sin(1.5*x);
}
```

Output

```
f'(x) =  2.9701
```

Example 2

This example obtains the approximate first, second, and third derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
#include "imsl.h"
#include <math.h>

void main()
{
        double fcn(double);
        double x;
        double tolerance;
        double deriv;

        x = 2.0;

        deriv = imsl_d_fcn_derivative(fcn, x,
                0);
        printf ("f'(x)   = %7.3f, error = %5.2e\n", deriv,
                fabs(deriv+3.0*cos(1.5*x)));

        deriv = imsl_d_fcn_derivative(fcn, x,
                IMSL_ORDER, 2,
                0);
        printf ("f''(x)  = %7.4f, error = %5.2e\n", deriv,
                fabs(deriv-4.5*sin(1.5*x)));

        deriv = imsl_d_fcn_derivative(fcn, x,
                IMSL_ORDER, 3,
                0);
        printf ("f'''(x) = %7.4f, error = %5.2e\n", deriv,
                fabs(deriv-6.75*cos(1.5*x)));
}

double fcn(double x)
{
        return -2.0*sin(1.5*x);
}
```

Output

```
f'(x)   =   2.970, error = 1.11e-07
f''(x)  =  0.6350, error = 8.52e-09
f'''(x) = -6.6824, error = 1.12e-08
```

Chapter 5: Differential Equations

Routines

Runge-Kutta method ..... ode_runge_kutta 291
Adam's or Gear's method..... ode_adams_gear 297
Method of lines ..... pde_method_of_lines 304
Solves a parameterized system of differential equations
with boundary condiditons at two points ..... bvp_finite_difference 321
Fast Poisson solver ..... fast_poisson_2d 332

Usage Notes

Ordinary Differential Equations

An *ordinary differential equation* is an equation involving one or more dependent variables called y_i, one independent variable, t, and derivatives of the y_i with respect to t.

In the *initial-value problem* (IVP), the initial or starting values of the dependent variables y_i at a known value $t = t_0$ are given. Values of $y_i(t)$ for $t > t_0$ or $t < t_0$ are required.

The functions `imsl_f_ode_runge_kutta` and `imsl_f_ode_adams_gear` solve the IVP for ODEs of the form

$$\frac{dy_i}{dt} = y_i' = f_i\left(t, y_1, \ldots, y_N\right) \qquad i = 1, \ldots, N$$

with $y_i = (t = t_0)$ specified. Here, f_i is a user-supplied function that must be evaluated at any set of values $(t, y_1, \ldots, y_N)$, $i = 1, \ldots, N$.

This problem statement is abbreviated by writing it as a *system* of first-order ODEs, $y(t) = [y_1(t), \ldots, y_N(t)]^T$, $f(t, y) = [f_1(t, y), \ldots, f_N(t, y)]^T$, so that the problem becomes $y' = f(t, y)$ with initial values $y(t_0)$.

The system

$$\frac{dy}{dt} = y' = f\left(t, y\right)$$

is said to be *stiff* if some of the eigenvalues of the Jacobian matrix

$$\left\{\partial y_i' / \partial y_j\right\}$$

are large and negative. This is frequently the case for differential equations modeling the behavior of physical systems, such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reactions in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*).

Users typically identify stiff systems by the fact that numerical differential equation solvers such as `imsl_f_ode_runge_kutta` are inefficient, or else completely fail. Special methods are often required. The most common inefficiency is that a large number of evaluations of $f(t, y)$ (and hence an excessive amount of computer time) are required to satisfy the accuracy and stability requirements of the software. In such cases, use the IMSL function `imsl_f_ode_adams_gear`. For more discussion about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

Partial Differential Equations

The routine `imsl_f_pde_method_of_lines`, page 304, solves the IVP problem for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

subject to the boundary conditions

$$\begin{aligned} \alpha_1^{(i)} u_i(a) + \beta_1^{(i)} \frac{\partial u_i}{\partial x}(a) &= \gamma_1(t) \\ \alpha_2^{(i)} u_i(b) + \beta_2^{(i)} \frac{\partial u_i}{\partial x}(b) &= \gamma_2(t) \end{aligned}$$

and subject to the initial conditions

$$u_i(x, t = t_0) = g_i(x)$$

for $i = 1, \ldots, N$. Here, f_i, g_i,

$$\alpha_j^{(i)}, \text{ and } \beta_j^{(i)}$$

are user-supplied, $j = 1, 2$.

The routine `imsl_f_bvp_finite_difference`, page 321, solves the boundary value problem (BVP) for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

subject to the boundary conditions

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

and subject to the initial conditions

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

for $i = 1, \ldots, N$. Here, $\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$ are user-supplied.

In this formulation, p is an optional continuation parameter. It can be useful in solving nonlinear problems. When used, p=0 corresponds to an easy-to-solve problem and p=1 corresponds to the actual problem to be solved.

The routine `imsl_f_fast_poisson_2d`, page 332, solves Laplace's, Poisson's, or Helmholtz's equation in two dimensions. This routine uses a fast Poisson method to solve a PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = f(x, y)$$

over a rectangle, subject to boundary conditions on each of the four sides. The scalar constant c and the function f are user specified.

ode_runge_kutta

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

Synopsis

#include `<imsl.h>`

float `imsl_f_ode_runge_kutta_mgr` (*int* `task`, *void* `**state`, ..., 0)

void `imsl_f_ode_runge_kutta` (*int* `neq`, *float* `*t`, *float* `tend`, *float* `y[]`, `void *state, void fcn()`)

The type *double* functions are `imsl_d_ode_runge_kutta_mgr` and `imsl_d_ode_runge_kutta`.

Required Arguments for imsl_ f_ode_runge_kutta_mgr

int task (Input)
This function must be called with `task` set to `IMSL_ODE_INITIALIZE` to set up for solving an ODE system and with `task` equal to `IMSL_ODE_RESET` to clean up after it has been solved. These values for `task` are defined in the include file, `imsl.h`.

void `**state` (Input/Output)
The current state of the ODE solution is held in a structure pointed to by `state`. It cannot be directly manipulated.

Required Arguments for imsl_f_ode_runge_kutta

int `neq` (Input)
Number of differential equations.

float `*t` (Input/Output)
Independent variable. On input, `t` is the initial independent variable value. On output, `t` is replaced by `tend`, unless error conditions arise.

float `tend` (Input)
Value of `t` at which the solution is desired. The value `tend` may be less than the initial value of `t`.

float `y[]` (Input/Output)
Array with `neq` components containing a vector of dependent variables. On input, `y` contains the initial values. On output, `y` contains the approximate solution.

void `*state` (Input/Output)
The current state of the ODE solution is held in a structure pointed to by `state`. It must be initialized by a call to `imsl_f_ode_runge_kutta_mgr`. It cannot be directly manipulated.

void `fcn` (*int* `neq`, *float* `t`, *float* `*y`, *float* `*yprime`)
User-supplied function to evaluate the right-hand side where
float `*yprime` (Output)

Array with `neq` components containing the vector y'. This function computes

$$\texttt{yprime} = \frac{dy}{dt} = y' = f(t, y)$$

and `neq`, `t`, and `*y` are defined immediately preceding this function.

Synopsis with Optional Arguments

```
#include <imsl.h>
float imsl_f_ode_runge_kutta_mgr (int task, void **state,
        IMSL_TOL, float tol,
        IMSL_HINIT, float hinit,
        IMSL_HMIN, float hmin,
        IMSL_HMAX, float hmax,
        IMSL_MAX_NUMBER_STEPS, int max_steps,
        IMSL_MAX_NUMBER_FCN_EVALS, int max_fcn_evals,
        IMSL_SCALE, float scale,
        IMSL_NORM, int norm,
        IMSL_FLOOR, float floor,
        IMSL_NSTEP, int *nstep,
        IMSL_NFCN, int *nfcn,
```

```
IMSL_HTRIAL, float *htrial,
IMSL_FCN_W_DATA, void fcn (), void *data,
0)
```

Optional Arguments

`IMSL_TOL`, *float* `tol` (Input)
Tolerance for error control. An attempt is made to control the norm of the local error such that the global error is proportional to `tol`.
Default: `tol` = 100.0*`imsl_f_machine`(4)

`IMSL_HINIT`, *float* `hinit` (Input)
Initial value for the step size *h*. Steps are applied in the direction of integration.
Default: `hinit` = 0.001|`tend` − `t`|

`IMSL_HMIN`, *float* `hmin` (Input)
Minimum value for the step size *h*.
Default: `hmin` – 0.0

`IMSL_HMAX`, *float* `hmax` (Input)
Maximum value for the step size *h*.
Default: `hmax` = 2.0

`IMSL_MAX_NUMBER_STEPS`, *int* `max_steps` (Input)
Maximum number of steps allowed.
Default: `max_steps` = 500

`IMSL_MAX_NUMBER_FCN_EVALS`, *int* `max_fcn_evals` (Input)
Maximum number of function evaluations allowed.
Default: `max_fcn_evals` = No enforced limit

`IMSL_SCALE`, *float* `scale` (Input)
A measure of the scale of the problem, such as an approximation to the Jacobian along the trajectory.
Default: `scale` = 1

`IMSL_NORM`, *int* `norm` (Input)
Switch determining the error norm. In the following, e_i is the absolute value of the error estimate for y_i.

0	minimum of the absolute error and the relative error, equals the maximum of e_i / max (|y_i|, 1) for i = 1, …, `neq`.
1	absolute error, equals $\max_i e_i$.
2	$\max_i(e_i / w_i)$ where w_i = max (|y_i|, `floor`). The value of `floor` is reset using `IMSL_FLOOR`.

Default: `norm` = 0

`IMSL_FLOOR`, *float* `floor` (Input)
This is used with `IMSL_NORM`. It provides a positive lower bound for the error norm option with value 2.
Default: `floor` = 1.0

IMSL_NSTEP, *int* *nstep (Output)
Returns the number of steps taken.

IMSL_NFCN, *int* *nfcn (Output)
Returns the number of function evaluations used.

IMSL_HTRIAL, *float* *htrial (Output)
Returns the current trial step size.

IMSL_FCN_W_DATA, *void* fcn (*int* neq, *float* t, *float* *y, *float* *yprime, *void* *data), *void* *data, (Input)
User-supplied function to evaluate the right-hand side, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_ode_runge_kutta finds an approximation to the solution of a system of first-order differential equations of the form

$$\frac{dy}{dt} = y' = f(t, y)$$

with given initial conditions for y at the starting value for t. The function attempts to keep the global error proportional to a user-specified tolerance. The proportionality depends on the differential equation and the range of integration.

The function imsl_f_ode_runge_kutta is efficient for nonstiff systems where the evaluations of $f(t, y)$ are not expensive. The code is based on an algorithm designed by Hull et al. (1976, 1978). It uses Runge-Kutta formulas of order five and six developed by J.H. Verner.

Examples

Example 1

This example solves

$$\frac{dy}{dt} = -y$$

over the interval [0, 1] with the initial condition $y(0) = 1$. The solution is $y(t) = e^{-t}$.

The ODE solver is initialized by a call to imsl_f_ode_runge_kutta_mgr with IMSL_ODE_INITIALIZE. This is the simplest use of the solver, so none of the default values are changed. The function imsl_f_ode_runge_kutta is then called to integrate from $t = 0$ to $t = 1$.

```
#include <imsl.h>
#include <math.h>

void        fcn (int neq, float t, float y[], float yprime[]);

main()
{
    int         neq = 1;        /* Number of ode's */
    float       t = 0.0;        /* Initial time */
```

```
    float       tend = 1.0;     /* Final time */
    float       y[1] = {1.0};   /* Initial condition */
    void        *state;
                                /* Initialize the ODE solver */
    imsl_f_ode_runge_kutta_mgr(IMSL_ODE_INITIALIZE, &state, 0);
                                /* Integrate from t=0 to tend=1 */
    imsl_f_ode_runge_kutta (neq, &t, tend, y, state, fcn);
                                /* Print the solution and error */
    printf("y[%f] = %f\n", t, y[0]);
    printf("Error is: %e\n", exp( (double)(-tend) )-y[0]);
}

void fcn (int neq, float t, float y[], float yprime[])
{
    yprime[0] = -y[0];
}
```

Output

```
y[1.000000] = 0.367879
Error is: -9.149755e-09
```

Example 2

Consider a predator-prey problem with rabbits and foxes. Let r be the density of rabbits, and let f be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. Mathematically, the model without species interaction is approximated by the equation

$$r' = 2r$$

$$f' = -f$$

With species interaction, the rate at which the rabbits are consumed by the foxes is assumed to equal the value $2rf$. The rate at which the foxes increase, because they are consuming the rabbits, is equal to rf. Thus, the model differential equations to be solved are

$$r' = 2r - 2rf$$

$$f' = -f + rf$$

For illustration, the initial conditions are taken to be $r(0) = 1$ and $f(0) = 3$. The interval of integration is $0 \le t \le 10$. In the program, $y[0] = r$ and $y[1] = f$. The ODE solver is initialized by a call to `imsl_f_ode_runge_kutta_mgr`. The error tolerance is set to 0.0005. Absolute error control is selected by setting `IMSL_NORM` to the value one. We also request that `nstep` be set to the current number of steps in the integration. The function `imsl_f_ode_runge_kutta` is then called in a loop to integrate from $t = 0$ to $t = 10$ in steps of $\delta t = 1$. At each step, the solution is printed. Note that `nstep` is updated even though it is not an argument to this function. Its address has been stored within `imsl_f_ode_runge_kutta_mgr` into the area pointed to by `state`. The last call to `imsl_f_ode_runge_kutta_mgr` with `IMSL_ODE_RESET` releases workspace.

```
#include <imsl.h>

void            fcn(int neq, float t, float y[], float yprime[]);

main()
{
    int         neq = 2;
    float       t = 0.0;            /* Initial time */
    float       tend;               /* Final time */
    float       y[2] = {1.0, 3.0};  /* Initial conditions */
    int         k;
    int         nstep;
    void        *state;
                                    /* Initialize the ODE solver */
    imsl_f_ode_runge_kutta_mgr(IMSL_ODE_INITIALIZE, &state,
                               IMSL_TOL,    0.0005,
                               IMSL_NSTEP,  &nstep,
                               IMSL_NORM,   1,
                               0);

    printf("\n Start    End     Density of  Density of    Number of"  );
    printf("\n Time     Time      Rabbits     Foxes        Steps\n\n");

    for (k = 0;   k < 10;   k++) {
        tend = k + 1;
        imsl_f_ode_runge_kutta (neq, &t, tend, y, state, fcn);
        printf("%3d %12.3f %12.3f %12.3f %12d\n", k, t, y[0], y[1], nstep);
    }
    imsl_f_ode_runge_kutta_mgr(IMSL_ODE_RESET, &state, 0);
}

void fcn (int neq, float t, float y[], float yprime[])
{
                                    /* Density change rate for Rabbits: */
    yprime[0] = 2*y[0]*(1 - y[1]);
                                    /* Density change rate for Foxes: */
    yprime[1] =  -y[1]*(1 - y[0]);
}
```

Output

```
Start       End       Density of   Density of      Number of
Time        Time       Rabbits       Foxes          Steps

 0        1.000         0.078        1.465            4
 1        2.000         0.085        0.578            6
 2        3.000         0.292        0.250            7
 3        4.000         1.449        0.187            8
 4        5.000         4.046        1.444           11
 5        6.000         0.176        2.256           15
 6        7.000         0.066        0.908           18
 7        8.000         0.148        0.367           20
 8        9.000         0.655        0.188           21
 9       10.000         3.157        0.352           23
```

Fatal Errors

IMSL_ODE_TOO_MANY_EVALS	Completion of the next step would make the number of function evaluations #, but only # evaluations are allowed.
IMSL_ODE_TOO_MANY_STEPS	Maximum number of steps allowed, #, used. The problem may be stiff.
IMSL_ODE_FAIL	Unable to satisfy the error requirement. "tol" = # may be too small.

ode_adams_gear

Solves a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods.

Synopsis

```
#include <imsl.h>
float imsl_f_ode_adams_gear_mgr (int task, void **state, ..., 0)
void imsl_f_ode_adams_gear (int neq, float *t, float tend, float y[],
        void *state, void fcn())
```

The type *double* functions are `imsl_d_ode_adams_gear_mgr` and `imsl_d_ode_adams_gear`.

Required Arguments for imsl_f_ode_adams_gear_mgr

int `task` (Input)
This function must be called with `task` set to `IMSL_ODE_INITIALIZE` to set up for solving an ODE system and with `task` equal to `IMSL_ODE_RESET` to clean up after it has been solved. These values for `task` are defined in the included file, `imsl.h`.

void `**state` (Input/Output)
The current state of the ODE solution is held in a structure pointed to by `state`. It cannot be directly manipulated.

Required Arguments for imsl_f_ode_adams_gear

int `neq` (Input)
Number of differential equations.

float `*t` (Input/Output)
Independent variable. On input, `t` is the initial independent variable value. On output, `t` is replaced by `tend` unless error conditions arise.

float `tend` (Input)
Value of `t` at which the solution is desired. The value `tend` may be less than the initial value of `t`.

float `y[]` (Input/Output)
Array with `neq` components containing a vector of dependent variables. On input, `y` contains the initial values. On output, `y` contains the approximate solution.

void `*state` (Input/Output)
The current state of the ODE solution is held in a structure pointed to by `state`. It must be initialized by a call to `imsl_f_ode_adams_gear_mgr`. It cannot be directly manipulated.

void `fcn` (*int* `neq`, *float* `t`, *float* `*y`, *float* `*yprime`)
User-supplied function to evaluate the right-hand side where
float `*yprime` (Output)

Array with `neq` components containing the vector y'. This function computes

$$\texttt{yprime} = \frac{dy}{dt} = y' = f(t, y)$$

and `neq`, `t`, and `*y` are defined immediately preceding this function.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_ode_adams_gear_mgr` (*int* `task`, *void* `**state`,
`IMSL_JACOBIAN`, *void* `fcnj ()`,
`IMSL_METHOD`, *int* `method`,
`IMSL_MAXORD`, *int* `maxord`,
`IMSL_MITER`, *int* `miter`,
`IMSL_TOL`, *float* `tol`,
`IMSL_HINIT`, *float* `hinit`,
`IMSL_HMIN`, *float* `hmin`,
`IMSL_HMAX`, *float* `hmax`,
`IMSL_MAX_NUMBER_STEPS`, *int* `max_steps`,
`IMSL_MAX_NUMBER_FCN_EVALS`, *int* `max_fcn_evals`,
`IMSL_SCALE`, *float* `scale`,
`IMSL_NORM`, *int* `norm`,
`IMSL_FLOOR`, *float* `floor`,
`IMSL_NSTEP`, *int* `*nstep`,
`IMSL_NFCN`, *int* `*nfcn`,
`IMSL_NFCNJ`, *int* `*nfcnj`,
`IMSL_FCN_W_DATA`, *void* `fcn ()`, *void* `*data`,
`IMSL_JACOBIAN_W_DATA`, *void* `fcn ()`, *void* `*data`,
`0`)

Optional Arguments

`IMSL_JACOBIAN`, *void* `fcnj` (*int* `neq`, *float* `t`, *float* `*y`, *float* `yprime[]`, *float* `dypdy[]`)
User-supplied function to evaluate the Jacobian matrix where
float `yprime[]` (Input)
Array with `neq` components containing the vector $y' = f(t, y)$.

float `dypdy[]` (Output)

Array of size `neq` × `neq` containing the partial derivatives. Each derivative $\partial y'_i / \partial y_i$ is evaluated at the provided (t, y) values and is returned in array location `dypdy[(i – 1)*n + j – 1]`.
and `neq`, `t`, and `*y` are described in the "Required Arguments" section.

`IMSL_METHOD`, *int* `method` (Input)

Choose the class of integration methods.

1 Use implicit Adams method.

2 Use backward differentiation formula (BDF) methods.

Default: `method` = 2

`IMSL_MAXORD`, *int* `maxord` (Input)

Define the highest order formula to use of implicit Adams type or BDF type. The default is the value 12 for Adams formulas and is the value 5 for BDF formulas.

`IMSL_MITER`, *int* `miter` (Input)

Choose the method for solving the formula equations.

1 Use function iteration or successive substitution.

2 Use chord or modified Newton method and a user-supplied Jacobian matrix.

3 Same as 2 except Jacobian is approximated within the function by divided differences.

Default: `miter` = 3

`IMSL_TOL`, *float* `tol` (Input)

Tolerance for error control. An attempt is made to control the norm of the local error such that the global error is proportional to `tol`.

Default: `tol` = 0.001

`IMSL_HINIT`, *float* `hinit` (Input)

Initial value for the step size *h*. Steps are applied in the direction of integration.

Default: `hinit` = 0.001|`tend` – `t`|

`IMSL_HMIN`, *float* `hmin` (Input)

Minimum value for the step size *h*.

Default: `hmin` = 0.0

`IMSL_HMAX`, *float* `hmax` (Input)

Maximum value for the step size *h*.

Default: `hmax` = `imsl_amach(2)`

`IMSL_MAX_NUMBER_STEPS`, *int* `max_steps` (Input)

Maximum number of steps allowed.

Default: `max_steps` = 500

`IMSL_MAX_NUMBER_FCN_EVALS`, *int* `max_fcn_evals` (Input)

Maximum number of evaluations of y' allowed.

Default: `max_fcn_evals` = No enforced limit

IMSL_SCALE, *float* scale (Input)
A measure of the scale of the problem, such as an approximation to the Jacobian along the trajectory.
Default: scale = 1

IMSL_NORM, *int* norm (Input)
Switch determining the error norm. In the following, e_i is the absolute value of the error estimate for y_i.

0	minimum of the absolute error and the relative error, equals the maximum of e_i / (max ($\|y_i\|$, 1)) for i = 1, …, neq.
1	absolute error, equals $\max_i e_i$.
2	$\max_i (e_i / w_i)$where w_i = max ($\|y_i\|$, floor). The value of floor is reset using IMSL_FLOOR.

Default: norm = 0.

IMSL_FLOOR, *float* floor (Input)
This is used with IMSL_NORM. It provides a positive lower bound for the error norm option with value 2.
Default: floor = 1.0

IMSL_NSTEP, *int* *nstep (Output)
Returns the number of steps taken.

IMSL_NFCN, *int* *nfcn (Output)
Returns the number of evaluations of y' used.

IMSL_NFCNJ, *int* *nfcnj (Output)
Returns the number of Jacobian matrix evaluations used. This value will be nonzero only if the option IMSL_JACOBIAN is used.

IMSL_FCN_W_DATA, *void* fcn (*int* neq, *float* t, *float* *y, *float* *yprime, *void* *data), *void* *data, (Input)
User-supplied function to evaluate the right-hand side, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSL_JACOBIAN_W_DATA, *void* jacobian (*int* m, *int* n, *float* x[], *float* fjac[], *int* fjac_col_dim, *void* *data), *void* *data (Input)
User supplied function to compute the Jacobian, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_ode_adams_gear finds an approximation to the solution of a system of first-order differential equations of the form

$$\frac{dy}{dt} = y' = f(t, y)$$

with given initial conditions for y at the starting value for t. The function attempts to keep the global error proportional to a user-specified tolerance. The proportionality depends on the differential equation and the range of integration.

The code is based on using backward difference formulas not exceeding order five as outlined in Gear (1971) and implemented by Hindmarsh (1974). There is an optional use of the code that employs implicit Adams formulas. This use is intended for nonstiff problems with expensive functions $y' = f(t, y)$.

Examples

Example 1

This is a mildly stiff example problem (F2) from the test set of Enright and Pryce (1987):

$$\begin{aligned} y'_1 &= -y_1 - y_1 y_2 + k_1 y_2 \\ y'_2 &= -k_2 y_2 + k_3 (1 - y_2) y_1 \\ y_1(0) &= 1 \\ y_2(0) &= 0 \\ k_1 &= 294. \\ k_2 &= 3. \\ k_3 &= 0.01020408 \\ \texttt{tend} &= 240. \end{aligned}$$

The ODE solver is initialized by a call to `imsl_f_ode_adams_gear_mgr` with `IMSL_ODE_INITIALIZE`. This is the simplest use of the solver, so none of the default values are changed. The function `imsl_f_ode_adams_gear` is then called to integrate from $t = 0$ to $t = 240$.

```
#include <stdio.h>
#include <imsl.h>

void            fcn (int neq, float t, float y[], float yprime[]);

float           k1 = 294.0;     /* Model data */
float           k2 = 3.0;
float           k3 = 0.01020408;

main()
{
    int         neq = 2;            /* Number of ode's */
    float       t = 0.0;            /* Initial time */
    float       tend = 240.0;       /* Final time */
    float       y[2] = {1.0, 0.0};  /* Initial condition */
    void        *state;
                                    /* Initialize the ODE solver */
    imsl_f_ode_adams_gear_mgr(IMSL_ODE_INITIALIZE, &state, 0);
                                    /* Integrate from t=0 to tend=240 */
    imsl_f_ode_adams_gear (neq, &t, tend, y, state, fcn);
                                    /* Print the solution */
    printf("y[%f] = %f, %f\n", t, y[0], y[1]);
```

```
}

void fcn (int neq, float t, float y[], float yprime[])
{
    yprime[0] = -y[0] - y[0]*y[1] + k1*y[1];
    yprime[1] = -k2*y[1] + k3*(1.0-y[1])*y[0];
}
```

Output

```
y[240.000000] = 0.392391, 0.001334
```

Example 2

This problem is a stiff example (F5) from the test set of Enright and Pryce (1987). An initial step size of $h = 10^{-7}$ is suggested by these authors. It is necessary to provide for more evaluations of y' and for more steps than the default value allows. Both have been set to 4000.

$$
\begin{aligned}
y'_1 &= k_1\,(-k_2 y_1 y_2 + k_3 y_4 - k_4 y_1 y_3) \\
y'_2 &= -k_1 k_2 y_1 y_2 + k_5 y_4 \\
y'_3 &= k_1\,(-k_4 y_1 y_3 + k_6 y_4) \\
y'_4 &= k_1\,(k_2 y_1 y_2 - k_3 y_4 + k_4 y_1 y_3) \\
y_1(0) &= 3.365 \times 10^{-7} \\
y_2(0) &= 8.261 \times 10^{-3} \\
y_3(0) &= 1.641 \times 10^{-3} \\
y_4(0) &= 9.380 \times 10^{-6} \\
k_1 &= 10^{11} \\
k_2 &= 3. \\
k_3 &= 0.0012 \\
k_4 &= 9. \\
k_5 &= 2 \times 10^{7} \\
k_6 &= 0.001 \\
\texttt{tend} &= 100.
\end{aligned}
$$

The last call to `imsl_f_ode_adams_gear_mgr` with `IMSL_ODE_RESET` releases workspace.

```
#include <stdio.h>
#include <imsl.h>

void            fcn (int neq, float t, float y[], float yprime[]);

float           k1  = 1.e11;            /* Model data  */
float           k2  = 3.0;
float           k3  = 0.0012;
float           k4  = 9.0;
float           k5  = 2.e7;
float           k6  = 0.001;
```

```
main()
{
    int         neq = 4;                /* Number of ode's */
    float       t = 0.0;                /* Initial time */
    float       tend = 100.0;           /* Final time */
                                        /* Initial condition */
    float       y[4] = {3.365e-7, 8.261e-3, 1.642e-3, 9.380e-6};
    void        *state;
    int         *nfcn;
                                /* Initialize the ODE solver */
    imsl_f_ode_adams_gear_mgr(IMSL_ODE_INITIALIZE, &state,
                              IMSL_HINIT, 1.e-7,
                              IMSL_MAX_NUMBER_STEPS, 4000,
                              IMSL_MAX_NUMBER_FCN_EVALS, 4000,
                              IMSL_NFCN, &nfcn,
                              0);
                                /* Integrate from t=0 to tend=100 */
    imsl_f_ode_adams_gear (neq, &t, tend, y, state, fcn);
                                /* Release workspace and reset */
    imsl_f_ode_adams_gear_mgr(IMSL_ODE_RESET, &state, 0);
                                /* Print the solution */
    printf("y[%f] = %f, %f, %f, %f\n", t, y[0], y[1], y[2], y[3]);
                                /* Print the number of evaluations
                                   of yprime[] */
    printf("Number of yprime[] evaluations: %d\n", nfcn);
}

void fcn (int neq, float t, float y[], float yprime[])
{
    yprime[0] =  k1*(-k2*y[0]*y[1]+k3*y[3]-k4*y[0]*y[2]);
    yprime[1] = -k1*k2*y[0]*y[1] + k5*y[3];
    yprime[2] =  k1*(-k4*y[0]*y[2] + k6*y[3]);
    yprime[3] =  k1*(k2*y[0]*y[1] - k3*y[3] + k4*y[0]*y[2]);
}
```

Output

```
y[100.000000] = 0.000000, 0.003352, 0.005586, 0.000009
Number of yprime[] evaluations: 3630
```

Fatal Errors

IMSL_ODE_TOO_MANY_EVALS	Completion of the next step would make the number of function evaluations #, but only # are allowed.
IMSL_ODE_TOO_MANY_STEPS	Maximum number of steps allowed, # have been used. Try increasing the maximum number of steps allowed or increase the tolerance.

pde_method_of_lines

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.

Synopsis

#include `<imsl.h>`

void `imsl_f_pde_method_of_lines_mgr` (*int* `task`, *void* `**state`, ..., 0)

```
void imsl_f_pde_method_of_lines (int npdes, float *t, float tend,
      int nx, float xbreak[], float y[], void *state,
      void fcn_ut(), void fcn_bc())
```

The type *double* functions are `imsl_d_pde_method_of_lines_mgr` and `imsl_d_pde_method_of_lines`.

Required Arguments for imsl_f_pde_method_of_lines_mgr

int `task` (Input)
This function must be called with task set to `IMSL_PDE_INITIALIZE` to set up memory and default values prior to solving a problem and with task equal to `IMSL_PDE_RESET` to clean up after it has solved. These values for task are defined in the header file `imsl.h`.

void `**state` (Input/Output)
The current state of the PDE solution is held in a structure pointed to by `state`. It cannot be directly manipulated.

Required Arguments for imsl_f_pde_method_of_lines

int `npdes` (Input)
Number of differential equations.

float `*t` (Input/Output)
Independent variable. On input, t supplies the initial time, t_0. On output, t is set to the value to which the integration has been updated. Normally, this new value is `tend`.

float `tend` (Input)
Value of t = `tend` at which the solution is desired.

int `nx` (Input)
Number of mesh points or lines.

float `xbreak[]` (Input)
Array of length `nx` containing the breakpoints for the cubic Hermite splines used in the x discretization. The points in `xbreak` must be strictly increasing. The values `xbreak`[0] and `xbreak`[`nx` − 1] are the endpoints of the interval.

float `y[]` (Input/Output)
Array of size `npdes` by `nx` containing the solution. The array `y` contains the solution as y[`k`,`i`] = u_k(`x`, `tend`) at x = `xbreak`[`i`]. On input, y contains the

initial values. It must satisfy the boundary conditions. On output, `y` contains the computed solution.

void `*state` (Input/Output)
The current state of the PDE solution is held in a structure pointed to by state. It must be initialized by a call to `imsl_f_pde_method_of_lines_mgr`. It cannot be directly manipulated.

void `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`)
User-supplied function to evaluate u_t.

int `npdes` (Input)
Number of equations.

float `x` (Input)
Space variable, x.

float `t` (Input)
Time variable, t.

float `u[]` (Input)
Array of length `npdes` containing the dependent values, u.

float `ux[]` (Input)
Array of length `npdes` containing the first derivatives, u_x.

float `uxx[]` (Input)
Array of length `npdes` containing the second derivative, u_{xx}.

float `ut[]` (Output)
Array of length `npdes` containing the computed derivatives u_t.

void `fcn_bc`(*int* `npdes`, *float* `x`, *float* `t`, *float* `alpha[]`, *float* `beta[]`, *float* `gammap[]`)
User-supplied function to evaluate the boundary conditions. The boundary conditions accepted by `imsl_f_pde_method_of_lines` are

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k$$

Note: Users must supply the values α_k and β_k, which determine the values γ_k. Since γ_k can depend on t values of γ_k' also are required.

int `npdes` (Input)
Number of equations.

float `x` (Input)
Space variable, x.

float `t` (Input)
Time variable, t.

float `alpha[]` (Output)
Array of length `npdes` containing the α_k values.

float `beta[]` (Output)
Array of length `npdes` containing the β_k values.

float `gammap[]` (Output)
Array of length `npdes` containing the derivatives,

$$\frac{d\gamma_k}{dt} = \gamma_k'$$

Synopsis with Optional Arguments

#include `<imsl.h>`

void `imsl_f_pde_method_of_lines_mgr` (*int* `task`, *void* `**state`,
`IMSL_TOL`, *float* `tol`,
`IMSL_HINIT`, *float* `hinit`,
`IMSL_INITIAL_VALUE_DERIVATIVE`, *float* `initial_deriv[]`,
`IMSL_HTRIAL`, *float* `*htrial`,
`IMSL_FCN_UT_W_DATA`, *void* `fcn_ut` (), *void* `*data`,
`IMSL_FCN_BC_W_DATA`, *void* `fcn_bc` (), *void* `*data`,
0)

Optional Arguments

`IMSL_TOL`, *float* `tol` (Input)
Differential equation error tolerance. An attempt is made to control the local error in such a way that the global relative error is proportional to `tol`.
Default: `tol` = 100.0*`imsl_f_machine`(4)

`IMSL_HINIT`, *float* `hinit` (Input)
Initial step size in the *t* integration. This value must be nonnegative. If `hinit` is zero, an initial step size of 0.001|`tend` - t_0| will be arbitrarily used. The step will be applied in the direction of integration.
Default: `hinit` = 0.0

`IMSL_INITIAL_VALUE_DERIVATIVE`, *float* `initial_deriv[]` (Input/Output)
Supply the derivative values $u_x(x, t_0)$. This derivative information is input as

$$\texttt{initial\_deriv}(\texttt{k},\texttt{i}) = \frac{\partial u_k}{\partial x}\left(x, t(0)\right)$$

The array `initial_deriv` contains the derivative values as output:

$$\texttt{initial\_deriv}(\texttt{k},\texttt{i}) = \frac{\partial u_k}{\partial x}\left(x\texttt{tend}\right) \quad \text{at } x = x[i]$$

Default: Derivatives are computed using cubic spline interpolation

`IMSL_HTRIAL`, *float* `*htrial` (Output)
Return the current trial step size.

`IMSL_UT_FCN_W_DATA`, *void* `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`, *void* `*data`), *void* `*data` (Input)
User-supplied function to evaluate u_t, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSL_BC_FCN_W_DATA, *void* fcn_bc(*int* npdes, *float* x, *float* t, *float* alpha[], *float* beta[], *float* gammap[], *void* *data), *void* *data
(Input)
User-supplied function to evaluate the boundary conditions, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Let M = npdes, N = nx and x_i = xbreaK(I). The routine imsl_f_pde_method_of_lines uses the method of lines to solve the partial differential equation system

$$\frac{\partial u_k}{\partial t} = f_k\left(x, t, u_1, \ldots u_M, \frac{\partial u_1}{\partial x}, \ldots \frac{\partial u_M}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots \frac{\partial^2 u_M}{\partial x^2}\right)$$

with the initial conditions

$$u_k = u_k(x, t) \qquad \text{at } t = t_0$$

and the boundary conditions

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k \qquad \text{at } x = x_1 \text{ and at } x = x_N$$

for $k = 1, \ldots, M$.

Cubic Hermite polynomials are used in the x variable approximation so that the trial solution is expanded in the series

$$\hat{u}_k(x,t) = \sum_{i=1}^{N}\left(a_{i,k}(t)\phi_i(x) + b_{i,k}(t)\psi_i(x)\right)$$

where $\phi_i(x)$ and $\psi_i(x)$ are the standard basis functions for the cubic Hermite polynomials with the knots $x_1 < x_2 < \ldots < x_N$. These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\phi_i(x_l) = \delta_{il} \qquad \psi_i(x_l) = 0$$
$$\frac{d\phi_i}{dx}(x_l) = 0 \qquad \frac{d\psi_i}{dx}(x_l) = \delta_{il}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equation at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} - x_j\right)$$

$$p_{2j} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} + x_j\right)$$

for $j = 1, \ldots, N$. The collocation approximation to the differential equation is

$$\frac{da_{i,k}}{dt}\phi_i\left(p_j\right) + \frac{db_{i,k}}{dt}\psi_i\left(p_j\right) = f_k\left(p_j, t, \hat{u}_1\left(p_j\right), \ldots, \hat{u}_M\left(p_j\right), \ldots, \left(\hat{u}_1\right)_{xx}\left(p_j\right), \ldots, \left(\hat{u}_M\right)_{xx}\left(p_j\right)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N-1)$.

This is a system of $2M(N-1)$ ordinary differential equations in $2MN$ unknown coefficient functions, $a_{i,k}$ and $b_{i,k}$. This system can be written in the matrix–vector form as $A\, dc/dt = F(t, y)$ with $c(t_0) = c_0$ where c is a vector of coefficients of length $2MN$ and c_0 holds the initial values of the coefficients. The last $2M$ equations are obtained by differentiating the boundary conditions

$$\alpha_k \frac{da_k}{dt} + \beta_k \frac{db_k}{dt} = \frac{d\gamma_k}{dt}$$

for $k = 1, \ldots, M$.

The initial conditions $u_k(x, t_0)$ must satisfy the boundary conditions. Also, the $\gamma_k(t)$ must be continuous and have a smooth derivative, or the boundary conditions will not be properly imposed for $t > t_0$.

If $\alpha_k = \beta_k = 0$, it is assumed that no boundary condition is desired for the k-th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

If the number of partial differential equations is $M = 1$ and the number of breakpoints is $N = 4$, then

$$A = \begin{bmatrix} \alpha_1 & \beta_1 & & & & & & \\ \phi_1(p_1) & \psi_1(p_1) & \phi_2(p_1) & \psi_2(p_1) & & & & \\ \phi_1(p_2) & \psi_1(p_2) & \phi_2(p_2) & \psi_2(p_2) & & & & \\ & & \phi_3(p_3) & \psi_3(p_3) & \phi_4(p_3) & \psi_4(p_3) & & \\ & & \phi_3(p_4) & \psi_3(p_4) & \phi_4(p_4) & \psi_4(p_4) & & \\ & & & & \phi_5(p_5) & \psi_5(p_5) & \phi_6(p_5) & \psi_6(p_5) \\ & & & & \phi_5(p_6) & \psi_5(p_6) & \phi_6(p_6) & \psi_6(p_6) \\ & & & & & & \alpha_4 & \beta_4 \end{bmatrix}$$

The vector c is

$$c = [a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4]^T$$

and the right-side F is

$$F = \left[\gamma'(x_1), f(p_1), f(p_2), f(p_3), f(p_4), f(p_5), f(p_6), \gamma'(x_4)\right]^T$$

If $M > 1$, then each entry in the above matrix is replaced by an $M \times M$ diagonal matrix. The element α_1 is replaced by diag($\alpha_{1,1}$, …, $\alpha_{1,M}$). The elements α_N, β_1 and β_N are handled in the same manner. The $\phi_i(p_j)$ and $\psi_i(p_j)$ elements are replaced by $\phi_i(p_j)I_M$ and $\psi_i(p_j)I_M$ where I_M is the identity matrix of order M. See Madsen and Sincovec (1979) for further details about discretization errors and Jacobian matrix structure.

The input/output array `Y` contains the values of the $a_{k,i}$. The initial values of the $b_{k,i}$ are obtained by using the IMSL cubic spline routine `imsl_f_cub_spline_interp_e_cnd` (Chapter 3, "Interpolation and Approximation") to construct functions

$$\hat{u}_k(x, t_0)$$

such that

$$\hat{u}_k(x_i, t_0) = a_{ki}$$

The IMSL routine `imsl_f_cub_spline_value` , Chapter 3, "Interpolation and Approximation" is used to approximate the values

$$\frac{d\hat{u}_k}{dx}(x_i, t_0) \equiv b_{k,i}$$

There is an optional use of `imsl_f_pde_method_of_lines` that allows the user to provide the initial values of $b_{k,i}$.

The order of matrix A is $2MN$ and its maximum bandwidth is $6M - 1$. The band structure of the Jacobian of F with respect to c is the same as the band structure of A. This system is solved using a modified version of `imsl_f_ode_adams_gear`, 297. Some of the linear solvers were removed. Numerical Jacobians are used exclusively. The algorithm is unchanged. Gear's BDF method is used as the default because the system is typically stiff.

Four examples of PDEs are now presented that illustrate how users can interface their problems with IMSL PDE solving software. The examples are small and not indicative of the complexities that most practitioners will face in their applications. A set of seven sample application problems, some of them with more than one equation, is given in Sincovec and Madsen (1975). Two further examples are given in Madsen and Sincovec (1979).

Examples

Example 1

The normalized linear diffusion PDE, $u_t = u_{xx}$, $0 \le x \le 1$, $t > t_0$, is solved. The initial values are $t_0 = 0$, $u(x, t_0) = u_0 = 1$. There is a "zero-flux" boundary condition at $x = 1$, namely $u_x(1, t) = 0$, $(t > t_0)$. The boundary value of $u(0, t)$ is abruptly changed from u_0 to the value $u_1 = 0.1$. This transition is completed by $t = t_\delta = 0.09$.

Due to restrictions in the type of boundary conditions successfully processed by `imsl_f_pde_method_of_lines`, it is necessary to provide the derivative boundary value function γ' at $x = 0$ and at $x = 1$. The function γ at $x = 0$ makes a smooth transition from the value u_0 at $t = t_0$ to the value u_1 at $t = t_\delta$. The transition phase for γ' is computed

by evaluating a cubic interpolating polynomial. For this purpose, the function subprogram `imsl_f_cub_spline_value`, Chapter 3, Interpolation and Approximation" is used. The interpolation is performed as a first step in the user-supplied routine `fcn_bc`. The function and derivative values $\gamma(t_0) = u_0$, $\gamma'(t_0) = 0$, $\gamma(t_\delta) = u_1$, and $\gamma'(t_\delta) = 0$, are used as input to routine `imsl_f_cub_spline_interp_e_cnd`, to obtain the coefficients evaluated by `imsl_f_cub_spline_value`. Notice that $\gamma'(t) = 0$, $t > t_\delta$. The evaluation routine `imsl_f_cub_spline_value` will not yield this value so logic in the routine `fcn_bc` assigns $\gamma'(t) = 0$, $t > t_\delta$.

```
#include <imsl.h>
#include <math.h>

main()
{
        void            fcnut(int, float, float, float *, float *, float *,
                              float *);
        void            fcnbc(int, float, float, float *, float *,
                              float *);
        int             npdes = 1;
        int             nx = 8;
        int             i;
        int             j = 1;
        int             nstep = 10;
        float           t = 0.0;
        float           tend;
        float           xbreak[8];
        float           y[8];
        char            title[50];
        void           *state;

                        /* Set breakpoints and initial conditions */

        for (i = 0; i < nx; i++) {
                xbreak[i] = (float) i / (float) (nx - 1);
                y[i] = 1.0;
        }

                        /* Initialize the solver */

        imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                       0);

        while (j <= nstep) {
                tend = (float) j++ / (float) nstep;
                tend *= tend;

                        /* Solve the problem */

                imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                           state, fcnut, fcnbc);

                        /* Print results at current t=tend */

                sprintf(title, "solution at t = %4.2f\0", t);
                imsl_f_write_matrix(title, npdes, nx, y, 0);
        }
```

```
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
        float *ut)
{
                        /* Define the PDE */

        *ut = *uxx;
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
        float *gamp)
{
        static int      ndata;
        static int      first = 1;
        static float    delta = 0.09;
        static float    u0 = 1.0;
        static float    u1 = 0.1;
        static float    dfdata[2];
        static float    xdata[2];
        static float    fdata[2];
        static Imsl_f_ppoly *ppoly;

                        /* Compute interpolant first time only */

        if (first) {
                first = 0;
                ndata = 2;
                xdata[0] = 0.0;
                xdata[1] = delta;
                fdata[0] = u0;
                fdata[1] = u1;
                dfdata[0] = dfdata[1] = 0.0;
                ppoly = imsl_f_cub_spline_interp_e_cnd(ndata, xdata, fdata,
                                IMSL_LEFT, 1, dfdata[0],
                                IMSL_RIGHT, 1, dfdata[1],
                        0);
        }

                        /* Define boundary conditions */

        if (x == 0.0) {

                        /* These are for x = 0 */

                *alpha = 1.0;
                *beta = 0.0;
                *gamp = 0.0;

                        /* If in the boundary layer, compute
                           nonzero gamma prime */

                if (t <= delta)
                        *gamp = imsl_f_cub_spline_value(t, ppoly,
                                        IMSL_DERIV, 1,
                                        0);
        } else {
                        /* These are for x = 1 */
```

```
                *alpha = 0.0;
                *beta = 1.0;
                *gamp = 0.0;
        }
}
```

Output

```
                        solution at t = 0.01
      1          2          3          4          5          6
  0.969      0.997      1.000      1.000      1.000      1.000

      7          8
  1.000      1.000

                        solution at t = 0.04
      1          2          3          4          5          6
  0.625      0.871      0.962      0.991      0.998      1.000
        7          8
  1.000      1.000

                        solution at t = 0.09
      1          2          3          4          5          6
 0.1000     0.4602     0.7169     0.8671     0.9436     0.9781

      7          8
 0.9917     0.9951

                        solution at t = 0.16
      1          2          3          4          5          6
 0.1000     0.3130     0.5071     0.6681     0.7893     0.8708

      7          8
 0.9168     0.9315

                        solution at t = 0.25
      1          2          3          4          5          6
 0.1000     0.2567     0.4045     0.5354     0.6428     0.7224

      7          8
 0.7710     0.7874

                        solution at t = 0.36
      1          2          3          4          5          6
 0.1000     0.2176     0.3292     0.4292     0.5125     0.5751

      7          8
 0.6139     0.6270

                        solution at t = 0.49
      1          2          3          4          5          6
 0.1000     0.1852     0.2661     0.3386     0.3992     0.4448

      7          8
 0.4731     0.4827

                        solution at t = 0.64
      1          2          3          4          5          6
 0.1000     0.1588     0.2147     0.2648     0.3066     0.3381
```

```
      7           8
 0.3577      0.3643

                      solution at t = 0.81
      1           2           3           4           5           6
 0.1000      0.1387      0.1754      0.2083      0.2358      0.2565

      7           8
 0.2694      0.2738

                      solution at t = 1.00
      1           2           3           4           5           6
 0.1000      0.1242      0.1472      0.1678      0.1850      0.1980

      7           8
 0.2060      0.2087
```

Example 2

Here, Problem C is solved from Sincovec and Madsen (1975). The equation is of diffusion-convection type with discontinuous coefficients. This problem illustrates a simple method for programming the evaluation routine for the derivative, u_t. Note that the weak discontinuities at $x = 0.5$ are not evaluated in the expression for u_t. The problem is defined as

$$u_t = \partial u / \partial t = \partial / \partial x\left(D(x)\partial u / \partial x\right) - v(x)\partial u / \partial x$$

$$x \in [0, 1], t > 0$$

$$D(x) = \begin{cases} 5 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$v(x) = \begin{cases} 1000.0 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$u(x, 0) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ if } x > 0 \end{cases}$$

$$u(0, t) = 1, \quad u(1, t) = 0$$

```
#include <imsl.h>
#include <math.h>

main()
{
        void            fcnut(int, float, float, float *, float *, float *,
                              float *);
        void            fcnbc(int, float, float, float *, float *,
                              float *);
        int             npdes = 1;
        int             nx = 100;
        int             i;
        int             j = 1;
        int             nstep = 10;
        float           t = 0.0;
```

```
        float           tend;
        float           xbreak[100];
        float           y[100];
        float           tol, hinit;
        char            title[50];
        void           *state;

                        /* Set breakpoints and initial conditions */

        for (i = 0; i < nx; i++) {
                xbreak[i] = (float) i / (float) (nx - 1);
                y[i] = 0.0;
        }
        y[0] = 1.0;

                        /* Initialize the solver */

        tol = sqrt(imsl_f_machine(4));
        hinit = 0.01*tol;
        imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                       IMSL_TOL, tol,
                                       IMSL_HINIT, hinit,
                                       0);

        while (j <= nstep) {
                tend = (float) j++ / (float) nstep;

                        /* Solve the problem */

                imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                           state, fcnut, fcnbc);
        }
                        /* Print results at t=tend */

                sprintf(title, "solution at t = %4.2f\0", t);
                imsl_f_write_matrix(title, npdes, nx, y, 0);
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
        float *ut)
{
                        /* Define the PDE */

        float v;
        float d;


        if (x <= 0.5) {
                d = 5.0;
                v = 1000.0;
        }
        else
                d = v = 1.0;

        ut[0] = d*uxx[0] - v*ux[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
        float *gamp)
```

```
{
        *alpha = 1.0;
        *beta = 0.0;
        *gamp = 0.0;
}
```

Output

```
                        solution at t = 1.00
     1           2           3           4           5           6
 1.000       1.000       1.000       1.000       1.000       1.000

     7           8           9          10          11          12
 1.000       1.000       1.000       1.000       1.000       1.000

    13          14          15          16          17          18
 1.000       1.000       1.000       1.000       1.000       1.000

    19          20          21          22          23          24
 1.000       1.000       1.000       1.000       1.000       1.000

    25          26          27          28          29          30
 1.000       1.000       1.000       1.000       1.000       1.000

    31          32          33          34          35          36
 1.000       1.000       1.000       1.000       1.000       1.000

    37          38          39          40          41          42
 1.000       1.000       1.000       1.000       1.000       1.000

    43          44          45          46          47          48
 1.000       1.000       1.000       1.000       1.000       1.000

    49          50          51          52          53          54
 1.000       0.997       0.984       0.969       0.953       0.937

    55          56          57          58          59          60
 0.921       0.905       0.888       0.872       0.855       0.838

    61          62          63          64          65          66
 0.821       0.804       0.786       0.769       0.751       0.733

    67          68          69          70          71          72
 0.715       0.696       0.678       0.659       0.640       0.621

    73          74          75          76          77          78
 0.602       0.582       0.563       0.543       0.523       0.502

    79          80          81          82          83          84
 0.482       0.461       0.440       0.419       0.398       0.376

    85          86          87          88          89          90
 0.354       0.332       0.310       0.288       0.265       0.242

    91          92          93          94          95          96
 0.219       0.196       0.172       0.148       0.124       0.100

    97          98          99         100
 0.075       0.050       0.025       0.000
```

Example 3

In this example, using `imsl_f_pde_method_of_lines`, the linear normalized diffusion PDE $u_t = u_{xx}$ is solved but with an optional use that provides values of the derivatives, u_x, of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$, $n > 1$. The boundary conditions are "zero flux" conditions $u_x(0, t) = u_x(1, t) = 0$ for $t > 0$. Note that the initial data is compatible with these end conditions since the derivative function

$$u_x(x,0) = \frac{du(x,0)}{dx} = -(2n-1)\pi \sin\left[(2n-1)\pi x\right]$$

vanishes at $x = 0$ and $x = 1$.

This optional usage signals that the derivative of the initial data is passed by the user. The values $u(x, tend)$ and $u_x(x, tend)$ are output at the breakpoints with the optional usage.

```
#include <imsl.h>
#include <math.h>

main()
{
        void            fcnut(int, float, float, float *, float *, float *,
                              float *);
        void            fcnbc(int, float, float, float *, float *, float *);
        int             npdes = 1;
        int             nx = 10;
        int             i;
        int             j = 1;
        int             nstep = 10;
        float           t = 0.0;
        float           tend = 0.0;
        float           xbreak[10];
        float           y[10], deriv[10];
        float           tol, hinit;
        float           pi, arg;
        char            title1[50];
        char            title2[50];
        void           *state;

        pi = imsl_d_constant("pi", 0);
        arg = 9.0 * pi;

                        /* Set breakpoints and initial conditions */

        for (i = 0; i < nx; i++) {
                xbreak[i] = (float) i / (float) (nx - 1);
                y[i] = 1.0 + cos(arg * xbreak[i]);
                deriv[i] = -arg * sin(arg * xbreak[i]);
        }

                        /* Initialize the solver */

        tol = sqrt(imsl_f_machine(4));
```

```
        imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                       IMSL_TOL, tol,
                                       IMSL_INITIAL_VALUE_DERIVATIVE,
                                       deriv,
                                       0);

        while (j <= nstep) {
                j++;
                tend += 0.001;

                        /* Solve the problem */

                imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                           state, fcnut, fcnbc);

                        /* Print results at at every other t=tend */

                if (j % 2) {
                        sprintf(title1, "\nsolution at t = %5.3f\0", t);
                        sprintf(title2, "\nderivative at t = %5.3f\0", t);
                        imsl_f_write_matrix(title1, npdes, nx, y, 0);
                        imsl_f_write_matrix(title2, npdes, nx, deriv, 0);
                }
        }

}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
      float *ut)
{
                        /* Define the PDE */

        ut[0] = uxx[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
      float *gamp)
{
                        /* Define the boundary conditions */

        alpha[0] = 0.0;
        beta[0] = 1.0;
        gamp[0] = 0.0;
}
```

Output

```
                          solution at t = 0.002
         1           2           3           4           5           6
     1.233       0.767       1.233       0.767       1.233       0.767

         7           8           9          10
     1.233       0.767       1.233       0.767


                       derivative at t = 0.002
         1           2           3           4           5           6
 0.000e+00  -5.172e-07   1.911e-06   1.818e-06  -5.230e-07   2.408e-06
```

```
         7           8           9          10
-2.517e-06   3.194e-06  -3.608e-06   2.023e-06


                      solution at t = 0.004
         1           2           3           4           5           6
     1.053       0.947       1.053       0.947       1.053       0.947

         7           8           9          10
     1.053       0.947       1.053       0.947


                    derivative at t = 0.004
         1           2           3           4           5           6
 0.000e+00  -1.332e-06  -9.059e-06  -4.401e-06   5.006e-06  -2.134e-06

         7           8           9          10
-1.733e-06   4.625e-06   6.741e-07   2.023e-06



                      solution at t = 0.006
         1           2           3           4           5           6
     1.012       0.988       1.012       0.988       1.012       0.988

         7           8           9          10
     1.012       0.988       1.012       0.988


                    derivative at t = 0.006
         1           2           3           4           5           6
 0.000e+00  -1.408e-06  -1.018e-06  -6.572e-07  -8.213e-07  -1.151e-06

         7           8           9          10
 1.051e-06   1.257e-06  -2.920e-07   2.023e-06


                      solution at t = 0.008
         1           2           3           4           5           6
     1.003       0.997       1.003       0.997       1.003       0.997

         7           8           9          10
     1.003       0.997       1.003       0.997


                    derivative at t = 0.008
         1           2           3           4           5           6
 0.000e+00  -1.028e-06   4.270e-06   3.114e-06  -3.085e-06  -1.492e-06

         7           8           9          10
 2.126e-06  -1.280e-06  -1.541e-06   2.023e-06


                      solution at t = 0.010
         1           2           3           4           5           6
     1.001       0.999       1.001       0.999       1.001       0.999

         7           8           9          10
     1.001       0.999       1.001       0.999
```

```
                        derivative at t = 0.010
         1            2            3            4            5            6
 0.000e+00  -7.596e-07   2.819e-07   1.547e-07  -1.469e-06  -9.516e-07

         7            8            9           10
 2.889e-07   8.956e-08   5.992e-07   2.023e-06
```

Example 4

In this example, consider the linear normalized hyperbolic PDE, $u_{tt} = u_{xx}$, the "vibrating string" equation. This naturally leads to a system of first order PDEs. Define a new dependent variable $u_t = v$. Then, $v_t = u_{xx}$ is the second equation in the system. Take as initial data $u(x, 0) = \sin(\pi x)$ and $u_t(x, 0) = v(x, 0) = 0$. The ends of the string are fixed so $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$. The exact solution to this problem is $u(x, t) = \sin(\pi x) \cos(\pi t)$. Residuals are computed at the output values of t for $0 < t \leq 2$. Output is obtained at 200 steps in increments of 0.01.

Even though the sample code `imsl_f_pde_method_of_lines` gives satisfactory results for this PDE, users should be aware that for *nonlinear problems*, "shocks" can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pp 488-490, for an introductory discussion of shocks in hyperbolic systems.

```
#include <imsl.h>
#include <math.h>

main()
{
      void            fcnut(int, float, float, float *, float *, float *,
                            float *);
      void            fcnbc(int, float, float, float *, float *, float *);
      int             npdes = 2;
      int             nx = 10;
      int             i;
      int             j = 1;
      int             nstep = 200;
      float           t = 0.0;
      float           tend = 0.0;
      float           xbreak[20];
      float           y[20], deriv[20];
      float           tol, hinit;
      float           pi;
      float           error[10], erru;
      void           *state;

      pi = imsl_d_constant("pi", 0);

                      /* Set breakpoints and initial conditions */

      for (i = 0; i < nx; i++) {
              xbreak[i] = (float) i / (float) (nx - 1);
              y[i] = sin(pi * xbreak[i]);
              y[nx + i] = 0.0;
              deriv[i] = pi * cos(pi * xbreak[i]);
              deriv[nx + i] = 0.0;
```

```
      }
                     /* Initialize the solver */

      tol = sqrt(imsl_f_machine(4));
      imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                     IMSL_TOL, tol,
                                     IMSL_INITIAL_VALUE_DERIVATIVE,
                                     deriv,
                                     0);

      while (j <= nstep) {
              j++;
              tend += 0.01;
                     /* Solve the problem */

              imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                         state, fcnut, fcnbc);


                     /* Look at output at steps of 0.01
                        and compute errors */


              for (i = 0; i < nx; i++) {
                     error[i] = y[i] - sin(pi * xbreak[i]) *
                                cos(pi *tend);
                     erru = imsl_f_max(erru, fabs(error[i]));
                }
        }
      printf("Maximum error in u(x,t) = %e\n", erru);

}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
            float *ut)
{
                     /* Define the PDE */

        ut[0] = u[1];
        ut[1] = uxx[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
      float *gamp)
{
                    /* Define the boundary conditions */

        alpha[0] = 1.0;
        beta[0] = 0.0;
        gamp[0] = 0.0;
        alpha[1] = 1.0;
        beta[1] = 0.0;
        gamp[1] = 0.0;
}
```

Output

```
Maximum error in u(x,t) = 6.228203e-04
```

bvp_finite_difference

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.

Synopsis

#*include* `<imsl.h>`

float `*imsl_f_bvp_finite_difference` (*void* `fcneq()`, *void* `fcnjac()`, *void* `fcnbc()`, *int* `n`, *int* `nleft`, *int* `ncupbc`, *float* `tleft`, *float* `tright`, *int* `linear`, *float* `*nfinal`, *float* `*xfinal`, *float* `*yfinal`, …, 0)

The type *double* function is `imsl_d_bvp_finite_difference`.

Required Arguments

void `fcneq` (*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dydt[]`) (Input)
User supplied function to evaluate derivatives.

int `n` (Input)
Number of differential equations

float `t` (Input)
Independent variable, *t*.

float `y[]` (Input)
Array of size `n` containing the dependent variable values, *y*(t).

float `p` (Input)
Continuation parameter, *p*. See optional argument `IMSL_PROBLEM_EMBEDDED`.

float `dydt[]` (Output)
Array of size `n` containing the derivatives *y*′(*t*).

void `fcnjac`(*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dypdy[]`) (Input)
User supplied function to evaluate the Jacobian.

int `n` (Input)
Number of differential equations

float `t` (Input)
Independent variable, *t*.

float `y[]` (Input)
Array of size `n` containing the dependent variable values, *y*(t).

float `p` (Input)
Continuation parameter, *p*. See optional argument `IMSL_PROBLEM_EMBEDDED`.

float `dypdy[]` (Output)
`n` by `n` array containing the partial derivatives $a_{i,j} = \partial f_i / \partial y_j$ evaluated at (*t*, *y*). The values $a_{i,j}$ are returned in `dypdy`[(i-1)*n+(j-1)].

void `fcnbc`(*int* `n`, *float* `yleft[]`, *float* `yright[]`, *float* `p`, *float* `h[]`) (Input)
User supplied function to evaluate the boundary conditions.

int `n` (Input)
Number of differential equations.

float `yleft[]` (Input)
Array of size `n` containing the values of the dependent variable at the left endpoint.

float `yright[]` (Input)
Array of size `n` containing the values of the dependent variable at the right endpoint.

float `p` (Input)
Continuation parameter, *p*. See optional argument `IMSL_PROBLEM_EMBEDDED`.

float `h[]` (Output)
Array of size `n` containing the boundary condition residuals. The boundary conditions are defined by $h_i = 0$, for $i = 0, \ldots,$ `n`*-1*. The left endpoint conditions must be defined first, then, the conditions involving both endpoints, and finally the right endpoint conditions.

int `n` (Input)
Number of differential equations.

int `nleft` (Input)
Number of initial conditions. The value `nleft` must be greater than or equal to zero and less than `n`.

int `ncupbc` (Input)
Number of coupled boundary conditions. The value `nleft` + `ncupbc` must be greater than zero and less than or equal to `n`.

float `tleft` (Input)
The left endpoint.

float `tright` (Input)
The right endpoint.

int `linear` (Input)
Integer flag to indicate if the differential equations and the boundary conditions are linear. Set `linear` to one if the differential equations and the boundary conditions are linear, otherwise set `linear` to zero.

int *`nfinal` (Output)
Number of final grid points, including the endpoints.

float *`tfinal` (Output)
Array of size `mxgrid` containing the final grid points. Only the first `nfinal` points are significant. See optional argument `IMSL_MAX_SUBINTER` for definition of `mxgrid`.

float *yfinal (Output)

Array of size mxgrid by n containing the values of Y at the points in tfinal. See optional argument IMSL_MAX_SUBINTER for definition of mxgrid.

Synopsis with Optional Arugments

```
#include <imsl.h>

float *imsl_f_bvp_finite_difference (void fcneq(),void fcnjac(),
        void fcnbc(), int n, int nleft, int ncupbc, float tleft, float tright,
        int linear, float *nfinal, float *xfinal[], float *yfinal,
        IMSL_TOL, float tol,
        IMSL_HINIT, int ninit, float tinit[], float yinit[][],
        IMSL_PRINT, int iprint,
        IMSL_MAX_SUBINTER, int mxgrid,
        IMSL_PROBLEM_EMBEDDED, float pistep, void fcnpeq(),
        void fcnpbc(),
        IMSL_ERR_EST, float **errest,
        IMSL_ERR_EST_USER, float errest[],
        IMSL_FCN_W_DATA, void fcneq (),void *data,
        IMSL_JACOBIAN_W_DATA, void fcnjac (),void *data,
        IMSL_FCN_BC_W_DATA, void fcnbc (),void *data,
        IMSL_PROBLEM_EMBEDDED_W_DATA, float pistep,(),void *data,
        void fcnpeq(),void fcnpbc(),void *data,
        0)
```

Optional Arguments

IMSL_TOL, *float* tol (Input)

Relative error control parameter. The computations stop when

$$\left|E_{i,j}\right| / \max\left(y_{i,j}, 1.0\right) < tol \quad \text{for all } i = 0, n = 1, \text{ and } j = 0, ngrid - 1$$

Here $E_{i,j}$ is the estimated error on $y_{i,j}$

Default: tol = .001.

IMSL_HINIT, *int* ninit, *float* tinit[], *float* yinit[][], (Input)

Initial gridpoints. Number of initial grid points, including the endpoints, is given by ninit. tinit is an array of size ninit containing the initial grid points. yinit is an array size ninit by n containing an initial guess for the values of Y at the points in tinit.

Default: ninit =10, tinit[*] equally spaced in the interval [tleft, tright], and yinit[*][*] = 0.

IMSL_PRINT, *int* iprint (Input)

Parameter indicating the desired output level.

Iprint	**Action**
0	No output printed.
1	Intermediate output is printed.

Default: `iprint` = 0.

`IMSL_MAX_SUBINTER`, *int* `mxgrid` (Input)
Maximum number of grid points allowed.
Default: `mxgrid` = 100

`IMSL_PROBLEM_EMBEDDED`, *float* `pistep`, *void* `fcnpeq()`, *void* `fcnpbc()`
If this optional argument is supplied, then the routine `imsl_f_bvp_finite_difference` assumes that the user has embedded the problem into a one-parameter family of problems:

$$y' = y'(t, y, p)$$

$$h(\texttt{yleft}, \texttt{yright}, p) = 0$$

such that for $p = 0$ the problem is simple. For $p = 1$, the original problem is recovered. The routine `imsl_f_bvp_finite_difference` automatically attempts to increment from $p = 0$ to $p = 1$. The value `pistep` is the beginning increment used in this continuation. The increment will usually be changed by routine `imsl_f_bvp_finite_difference`, but an arbitrary minimum of 0.01 is imposed.

The argument `p` is the initial increment size for p. The functions `fcnpeq` and `fcnpbc` are user-supplied functions, and are defined:

void `fcnpeq`(*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dypdp[]`) (Input)
User supplied function to evaluate the derivative of y' with respect to the parameter p.

int `n` (Input)
Number of differential equations.

float `t` (Input)
Independent variable, t.

float `y[]` (Input)
Array of size `n` containing the dependent variable values.

float `p` (Input)
Continuation parameter, p.

float `dypdp[]` (Output)
Array of size `n` containing the derivative y' with respect to the parameter p at (t, y).

void `fcnpbc`(*int* `n`, *float* `yleft[]`, *float* `yright[]`, *float* `p`, *float* `h[]`)(Input)
User supplied function to evaluate the derivative of the boundary conditions with respect to the parameter p.

int `n` (Input)
Number of differential equations.

float `yleft[]` (Input)
Array of size `n` containing the values of the dependent variable at the left endpoint.

float `yright[]` (Input)
Array of size `n` containing the values of the dependent variable at the right endpoint.

float `p` (Input)
Continuation parameter, p.

float `h[]` (Output)
Array of size `n` containing the derivative of f_i with respect to p.

`IMSL_ERR_EST`, *float* `**errest` (Output)
Address of a pointer to an array of size `n` containing estimated error in `y`.

`IMSL_ERR_EST_USER`, *float* `errest[]` (Output)
User allocated array of size `n` containing estimated error in `y`.

`IMSL_FCN_W_DATA`, *void* `fcneq` (*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dydt[]`, *void* `*data`) ,*void* `*data`, (Input)
User-supplied function to evaluate derivatives, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_JACOBIAN_W_DATA`, *void* `fcnjac`(*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dypdy[]`, *void* `*data`) ,*void* `*data`, (Input)
User-supplied function to evaluate the Jacobian, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_FCN_BC_W_DATA`, *void* `fcnbc`(*int* `n`, *float* `yleft[]`, *float* `yright[]`, *float* `p`, *float* `h[]`, *void* `*data`) ,*void* `*data`, (Input)
User-supplied function to evaluate the boundary conditions , which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_PROBLEM_EMBEDDED_W_DATA`, *float* `pistep`, *void* `fcnpeq`(*void* `*data`), *void* `fcnpbc()`,*void* `*data`, (Input)
Same as optional argument `IMSL_PROBLEM_EMBEDDED`, except user-supplied functions also accept a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The routine `imsl_f_bvp_finite_difference` is based on the subprogram `PASVA3` by M. Lentini and V. Pereyra (see Pereyra 1978). The basic discretization is the trapezoidal rule over a nonuniform mesh. This mesh is chosen adaptively, to make the local error approximately the same size everywhere. Higher-order discretizations are obtained by deferred corrections. Global error estimates are produced to control the computation. The resulting nonlinear algebraic system is solved by Newton's method with step control. The linearized system of equations is solved by a special form of Gauss elimination that preserves the sparseness.

Example 1

This example solves the third-order linear equation

$$y''' - 2y'' + y' - y = \sin t$$

subject to the boundary conditions $y(0) = y(2\pi)$ and $y'(0) = y'(2\pi) = 1$. (Its solution is $y = \sin t$.) To use `imsl_f_bvp_finite_difference`, the problem is reduced to a system of first-order equations by defining $y_1 = y, y_2 = y'$ and $y_3 = y''$. The resulting system is

$$\begin{aligned} y_1' &= y_2 & y_2(0) - 1 &= 0 \\ y_2' &= y_3 & y_1(0) - y_1(2\pi) &= 0 \\ y_3' &= 2y_3 - y_2 + y_1 + \sin t & y_2(2\pi) - 1 &= 0 \end{aligned}$$

Note that there is one boundary condition at the left endpoint $t = 0$ and one boundary condition coupling the left and right endpoints. The final boundary condition is at the right endpoint. The total number of boundary conditions must be the same as the number of equations (in this case 3).

```
#include <math.h>
#include "imsl.h"

void fcneqn( int n, float t, float y[], float p, float dydt[]);
void fcnjac( int n, float t, float y[], float p, float dfdy[]);
void fcnbc( int n, float yleft[], float yright[], float p, float h[]);

#define MXGRID 100
#define N 3
void main()
{
  int n = N;
  int nleft = 1;
  int ncupbc = 1;
  float tleft = 0;
  float tright;
  int linear = 1;
  int nfinal;
  float tfinal[MXGRID];
  float yfinal[MXGRID][N];
  float errest[N];
```

```
  int i;

  tright = 2.0*imsl_f_constant("pi", 0);

  imsl_f_bvp_finite_difference( fcneqn, fcnjac, fcnbc,
                           n, nleft, ncupbc, tleft, tright,
                           linear, &nfinal, tfinal,
                           (float*)(&yfinal[0][0]),
                           IMSL_ERR_EST_USER, errest,
                           0);
  printf("           tfinal           y0            y1             y2 \n" );
  for( i=0; i<nfinal; i++ ) {
    printf( "%5d%15.6e%15.6e%15.6e%15.6e\n", i,
      tfinal[i], yfinal[i][0], yfinal[i][1], yfinal[i][2] );
  }
  printf("Error Estimates     ");
  printf("%15.6e%15.6e%15.6e\n",errest[0],errest[1],errest[2]);
  return;
}

void fcneqn( int n, float t, float y[], float p, float dydt[] )
{
  dydt[0] = y[1];
  dydt[1] = y[2];
  dydt[2] = 2*y[2] - y[1] + y[0] + sin(t);
}

void fcnjac( int n, float t, float y[], float p, float dfdy[] )
{
  dfdy[0*n+0] = 0;   /* df1/dy1 */
  dfdy[1*n+0] = 0;   /* df2/dy1 */
  dfdy[2*n+0] = 1;   /* df3/dy1 */
  dfdy[0*n+1] = 1;   /* df1/dy2 */
  dfdy[1*n+1] = 0;   /* df2/dy2 */
  dfdy[2*n+1] = -1;  /* df3/dy2 */
  dfdy[0*n+2] = 0;   /* df1/dy3 */
  dfdy[1*n+2] = 1;   /* df2/dy3 */
  dfdy[2*n+2] = 2;   /* df3/dy3 */
}

void fcnbc( int n, float yleft[], float yright[], float p, float h[] )
{
  h[0] = yleft[1] - 1;
  h[1] = yleft[0] - yright[0];
  h[2] = yright[1] - 1;
}
```

Output

```
           tfinal           y0            y1             y2
    0   0.000000e+00  -1.123446e-04   1.000000e+00   6.245916e-05
    1   3.490659e-01   3.419106e-01   9.397087e-01  -3.419581e-01
    2   6.981317e-01   6.426907e-01   7.660918e-01  -6.427230e-01
    3   1.396263e+00   9.847531e-01   1.737333e-01  -9.847453e-01
    4   2.094395e+00   8.660527e-01  -4.998748e-01  -8.660057e-01
    5   2.792527e+00   3.421828e-01  -9.395475e-01  -3.420647e-01
    6   3.490659e+00  -3.417236e-01  -9.396111e-01   3.418948e-01
    7   4.188790e+00  -8.656881e-01  -5.000588e-01   8.658734e-01
    8   4.886922e+00  -9.845795e-01   1.734572e-01   9.847519e-01
```

```
    9   5.585054e+00  -6.427722e-01   7.658259e-01   6.429526e-01
   10   5.934120e+00  -3.420819e-01   9.395434e-01   3.423984e-01
   11   6.283185e+00  -1.123446e-04   1.000000e+00   6.739637e-04
Error Estimates        2.840487e-04   1.792839e-04   5.587848e-04
```

Example 2

In this example, the following nonlinear problem is solved:

$$y'' - y^3 + (1 + \sin^2 t) \sin t = 0$$

with $y(0) = y(\pi) = 0$. Its solution is $y = \sin t$. As in Example 1, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$y_1' = y_2 \qquad y_1(0) = 0$$
$$y_2' = y_1^3 - \left(1 + \sin^2 t\right) \sin t \qquad y_1(\pi) = 0$$

In this problem, there is one boundary condition at the left endpoint and one at the right endpoint; there are no coupled boundary conditions.

```
#include <math.h>
#include "imsl.h"

void fcneqn(int n, float x, float y[], float p, float dydx[]);
void fcnjac(int n, float x, float y[], float p, float dfdy[]);
void fcnbc(int n, float yleft[], float yright[], float p, float h[]);

#define MXGRID 100
#define NINIT 12
#define N 2

void main()
{
  int n = N, nleft = 1, ncupbc = 0, linear = 0;
  int i, nfinal, ninit = NINIT;
  float tleft = 0, tright;
  float tinit[NINIT], yinit[N][NINIT];
  float tfinal[MXGRID], yfinal[N][MXGRID];
  float *errest, step;

  tright = imsl_f_constant("pi", 0);
  step = (tright-tleft) / (ninit-1);

  for( i=0; i<ninit; i++ ) {
    tinit[i] = tleft + i*step;
    yinit[i][0] = 0.4 * (tinit[i]-tleft) * (tright-tinit[i]);
    yinit[i][1] = 0.4 * (tright+tleft-2*tinit[i]);
  }
  imsl_f_bvp_finite_difference(fcneqn, fcnjac, fcnbc,
                           n, nleft, ncupbc, tleft, tright,
                           linear, &nfinal, tfinal,
```

```
                          (float*)(&yfinal[0][0]),
                          IMSL_HINIT, ninit, tinit, yinit,
                          IMSL_ERR_EST, &errest,
                          0);
  printf("              t             y0            y1\n" );
  for( i=0; i<nfinal; i++ ) {
    printf( "%5d%15.6e%15.6e%15.6e\n", i, tfinal[i], yfinal[i][0], yfinal[i][1]);
  }
  printf("Error Estimates     ");
  printf("%15.6e%15.6e\n",errest[0],errest[1]);
  return;
}

void fcneqn(int n, float t, float y[], float p, float dydt[])
{
  float sx = sin(t);
  dydt[0] = y[1];
  dydt[1] = y[0]*y[0]*y[0] - (sx*sx+1)*sx;
}

void fcnjac(int n, float t, float y[], float p, float dfdy[])
{
  dfdy[0*n+0] = 0;            /* df1/dy1 */
  dfdy[1*n+0] = 3*y[0]*y[0];  /* df2/dy1 */
  dfdy[0*n+1] = 1;            /* df1/dy2 */
  dfdy[1*n+1] = 0;            /* df2/dy2 */
}

void fcnbc(int n, float yleft[], float yright[], float p, float h[])
{
  h[0] = yleft[0];
  h[1] = yright[0];
}
```

Output

```
              t             y0            y1
    0   0.000000e+00   0.000000e+00   9.999277e-01
    1   2.855994e-01   2.817682e-01   9.594315e-01
    2   5.711987e-01   5.406458e-01   8.412407e-01
    3   8.567981e-01   7.557380e-01   6.548904e-01
    4   1.142397e+00   9.096186e-01   4.154530e-01
    5   1.427997e+00   9.898143e-01   1.423307e-01
    6   1.713596e+00   9.898143e-01  -1.423308e-01
    7   1.999195e+00   9.096185e-01  -4.154530e-01
    8   2.284795e+00   7.557380e-01  -6.548902e-01
    9   2.570394e+00   5.406460e-01  -8.412405e-01
   10   2.855994e+00   2.817682e-01  -9.594312e-01
   11   3.141593e+00   0.000000e+00  -9.999274e-01
Error Estimates        3.907291e-05   7.124317e-05
```

Example 3

In this example, the following nonlinear problem is solved:

$$y'' - y^3 = \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3} - \left(t - \frac{1}{2}\right)^8$$

with $y(0) = y(1) = \pi/2$. As in the previous examples, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$y_1' = y_2 \qquad y_1(0) = \pi/2$$

$$y_2' = y_1^3 - \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3} + \left(t - \frac{1}{2}\right)^8 \qquad y_1(1) = \pi/2$$

The problem is embedded in a family of problems by introducing the parameter p and by changing the second differential equation to

$$y_2' = py_1^3 + \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3} - \left(t - \frac{1}{2}\right)^8$$

At $p = 0$, the problem is linear; and at $p = 1$, the original problem is recovered. The derivatives $\partial y'/\partial p$ must now be specified in the subroutine `fcnpeq`. The derivatives $\partial f/\partial p$ are zero in `fcnpbc`.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>
void fcneqn(int n, float t, float y[], float p, float dydt[]);
void fcnjac(int n, float t, float y[], float p, float dfdy[]);
void fcnbc(int n, float yleft[], float yright[], float p, float h[]);
void fcnpeq(int n, float t, float y[], float p, float dfdp[]);
void fcnpbc(int n, float yleft[], float yright[], float p, float dhdp[]);

#define MXGRID 45
#define NINIT 12
#define N 2

void main()
{
  int n = 2;
  int nleft = 1;
  int ncupbc = 0;
  float tleft = 0;
  float tright = 1;
  float pistep = 0.1;
  int ninit = 5;
  float tinit[NINIT] = { 0.0, 0.4, 0.5, 0.6, 1.0 };
  float yinit[N][NINIT] = { 0.15749,   0.00215,
                            0.0,       0.00215,
                            0.15749,  -0.83995,
                           -0.05745,   0.0,
                            0.05745,   0.83995 };
```

```
  int linear = 0;
  int nfinal;
  float tfinal[MXGRID];
  float yfinal[MXGRID][N];
  float *errest;
  int i;

  imsl_f_bvp_finite_difference( fcneqn, fcnjac, fcnbc, n, nleft,
             ncupbc, tleft, tright,
             linear, &nfinal, tfinal, (float*)(&yfinal[0][0]),
             IMSL_MAX_SUBINTER, MXGRID,
             IMSL_PROBLEM_EMBEDDED, fcnpeq, fcnpbc, pistep,
             IMSL_HINIT, ninit, tinit, yinit,
             IMSL_ERR_EST, &errest,
             0 );
  printf("              t             y0            y1\n" );
  for( i=0; i<nfinal; i++ ) {
    printf("%5d%15.6e%15.6e%15.6e\n", i, tfinal[i], yfinal[i][0],
              yfinal[i][1]);
  }
  printf("Error Estimates     ");
  printf("%15.6e%15.6e\n",errest[0],errest[1]);
  return;
}


void fcneqn(int n, float t, float y[], float p, float dydt[])
{
  float z = t - 0.5;
  dydt[0] = y[1];
  dydt[1] = p*y[0]*y[0]*y[0] + 40./9.*pow(z*z,1./3.) - pow(z,8);
}
void fcnjac(int n, float t, float y[], float p, float dfdy[])
{
  dfdy[0*n+0] = 0;                   /* df0/dy0 */
  dfdy[0*n+1] = 1;                   /* df0/dy1 */
  dfdy[1*n+0] = 3.*(p)*(y[0]*y[0]);  /* df1/dy0 */
  dfdy[1*n+1] = 0;                   /* df1/dy1 */
}
void fcnbc(int n, float yleft[], float yright[], float p, float h[])
{
  float pi2 = imsl_f_constant("pi", 0)/2.0;
  h[0] = yleft[0] - pi2;
  h[1] = yright[0] - pi2;
}
void fcnpeq(int n, float t, float y[], float p, float dfdp[])
{
  dfdp[0] = 0;
  dfdp[1] = y[0]*y[0]*y[0];
}
void fcnpbc(int n, float yleft[], float yright[], float p, float dhdp[])
{
  dhdp[0] = 0;
  dhdp[1] = 0;
}
```

Output

```
            t             y0            y1
    0   0.000000e+00   1.570796e+00  -1.949336e+00
    1   4.444445e-02   1.490495e+00  -1.669566e+00
    2   8.888889e-02   1.421951e+00  -1.419465e+00
    3   1.333333e-01   1.363953e+00  -1.194307e+00
    4   2.000000e-01   1.294526e+00  -8.958461e-01
    5   2.666667e-01   1.243628e+00  -6.373191e-01
    6   3.333334e-01   1.208785e+00  -4.135206e-01
    7   4.000000e-01   1.187783e+00  -2.219351e-01
    8   4.250000e-01   1.183038e+00  -1.584200e-01
    9   4.500000e-01   1.179822e+00  -9.973146e-02
   10   4.625000e-01   1.178748e+00  -7.233893e-02
   11   4.750000e-01   1.178007e+00  -4.638249e-02
   12   4.812500e-01   1.177756e+00  -3.399763e-02
   13   4.875000e-01   1.177582e+00  -2.205548e-02
   14   4.937500e-01   1.177480e+00  -1.061177e-02
   15   5.000000e-01   1.177447e+00  -1.496867e-07
   16   5.062500e-01   1.177480e+00   1.061153e-02
   17   5.125000e-01   1.177582e+00   2.205518e-02
   18   5.187500e-01   1.177756e+00   3.399727e-02
   19   5.250000e-01   1.178007e+00   4.638219e-02
   20   5.375000e-01   1.178748e+00   7.233876e-02
   21   5.500000e-01   1.179822e+00   9.973124e-02
   22   5.750000e-01   1.183038e+00   1.584199e-01
   23   6.000000e-01   1.187783e+00   2.219350e-01
   24   6.666667e-01   1.208786e+00   4.135206e-01
   25   7.333333e-01   1.243628e+00   6.373190e-01
   26   8.000000e-01   1.294526e+00   8.958461e-01
   27   8.666667e-01   1.363953e+00   1.194307e+00
   28   9.111111e-01   1.421951e+00   1.419465e+00
   29   9.555556e-01   1.490495e+00   1.669566e+00
   30   1.000000e+00   1.570796e+00   1.949336e+00
Error Estimates        3.451270e-06   5.550027e-05
```

fast_poisson_2d

Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

Synopsis

#include `<imsl.h>`

float `*imsl_f_fast_poisson_2d` (*float* `rhs_pde()`, *float* `rhs_bc()`, *float* `coeff_u`, *int* `nx`, *int* `ny`, *float* `ax`, *float* `bx`, *float* `ay`, *float* `by`, *Imsl_bc_type* `bc_type[]`, ..., 0)

The type *double* function is `imsl_d_fast_poisson_2d`.

Required Arguments

float `rhs_pde` (*float* `x`, *float* `y`)

User-supplied function to evaluate the right-hand side of the partial differential equation at `x` and `y`.

float `rhs_bc`(*Imsl_pde_side* `side`, *float* `x`, *float* `y`)
User-supplied function to evaluate the right-hand side of the boundary conditions, on side `side`, at `x` and `y`. The value of `side` will be one of the following: `IMSL_RIGHT`, `IMSL_BOTTOM`, `IMSL_LEFT`, or `IMSL_TOP`.

float `coeff_u` (Input)
Value of the coefficient of *u* in the differential equation.

int `nx` (Input)
Number of grid lines in the *x*-direction. `nx` must be at least 4. See the description section for further restrictions on `nx`.

int `ny` (Input)
Number of grid lines in the *y*-direction. `ny` must be at least 4. See the "Description" section for further restrictions on `ny`.

float `ax` (Input)
The value of x along the left side of the domain.

float `bx` (Input)
The value of x along the right side of the domain.

float `ay` (Input)
The value of y along the bottom of the domain.

float `by` (Input)
The value of y along the top of the domain.

Imsl_bc_type `bc_type`[4] (Input)
Array of size 4 indicating the type of boundary condition on each side of the domain or that the solution is periodic. The sides are numbered as follows:

Side	Location
`IMSL_RIGHT_SIDE`(0)	x = `bx`
`IMSL_BOTTOM_SIDE`(1)	y = `ay`
`IMSL_LEFT_SIDE`(2)	x = `ax`
`IMSL_TOP_SIDE`(3)	y = `by`

The three possible boundary condition types are as follows:

Type	Condition
`IMSL_DIRICHLET`	Value of *u* is given.
`IMSL_NEUMANN`	Value of *du*/*dx* is given (on the right or left sides) or *du*/*dy* (on the bottom or top of the domain).
`IMSL_PERIODIC`	Periodic.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_fast_poisson_2d` (*float* `rhs_pde()`, *float* `rhs_bc()`, *float* `coeff_u`, *int* `nx`, *int* `ny`, *float* `ax`, *float* `bx`, *float* `ay`, *float* `by`,
Imsl_bc_type `bc_type[]`,
`IMSL_RETURN_USER`, *float* `u_user[]`,
`IMSL_ORDER`, *int* `order`,

IMSL_RHS_PDE_W_DATA, *float* rsh_pde (), *void* *data,
IMSL_RHS_BC_W_DATA, *float* rsh_bc (), *void* *data,
0)

Optional Arguments

IMSL_RETURN_USER, *float* u_user[] (Output)
User-supplied array of size nx by ny containing solution at the grid points.

IMSL_ORDER, *int* order (Input)
Order of accuracy of the finite-difference approximation. It can be either 2 or 4. Default: order = 4

IMSL_RSH_PDE_W_DATA, *float* rhs_pde (*float* x, *float* y, *void* *data), *void* *data, (Input)
User-supplied function to evaluate the right-hand side of the partial differential equation at x and y, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSL_RSH_BC_W_DATA, *float* rhs_bc(*Imsl_pde_side* side, *float* x, *float* y, *void* *data) , *void* *data, (Input)
User-supplied function to evaluate right-hand side of the boundary conditions, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

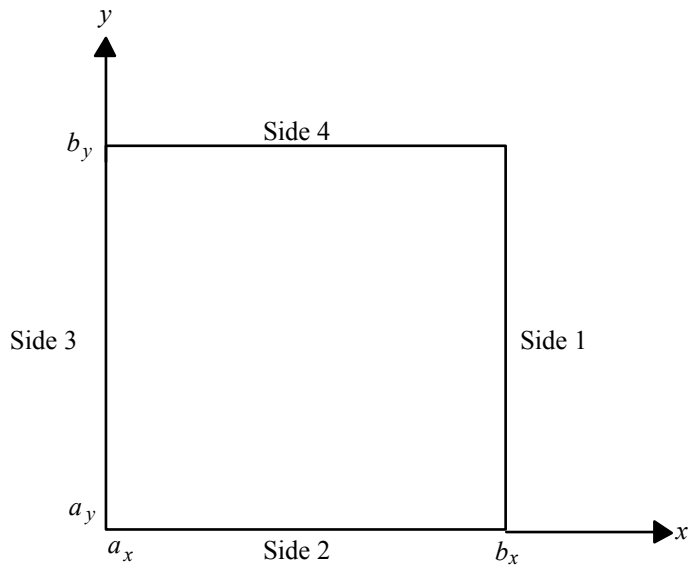
Description

Let c = coeff_u, a_x = ax, b_x = bx, a_y = ay, b_y = by, n_x = nx and n_y = ny.

imsl_f_fast_poisson_2d is based on the code HFFT2D by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = p$$

on the rectangular domain $(a_x, b_x) \times (a_y, b_y)$ with a user-specified combination of Dirichlet (solution prescribed), Neumann (first-derivative prescribed), or periodic boundary conditions. The sides are numbered clockwise, starting with the right side.



When $c = 0$ and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum ∞-norm is returned.

The solution is computed using either a second-or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies on the fact that $n_x - 1$ is highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If $n_x - 1$ is highly composite then the execution time of `imsl_f_fast_poisson_2d` is proportional to $n_x n_y \log_2 n_x$. If evaluations of $p(x, y)$ are inexpensive, then the difference in running time between `order` = 2 and `order` = 4 is small.

The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas hx = (`bx` − `ax`)/(`nx` − 1) and hy = (`by` − `ay`)/(`ny` − 1). The grid spacings in the x and y directions must be the same, i.e., `nx` and `ny` must be such that hx is equal to hy. Also, as noted above, `nx` and `ny` must be at least 4. To increase the speed of the fast Fourier transform, `nx` − 1 should be the product of small primes. Good choices are 17, 33, and 65.

If `-coeff_u` is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

Example

In this example, the equation

$$\frac{\partial^2 u}{\partial x_2} + \frac{\partial^2 u}{\partial y^2} + 3u = -2\sin(x + 2y) + 16e^{2x+3y}$$

with the boundary conditions

$$\frac{\partial u}{\partial y} = 2\cos(x+2y) + 3e^{2x+3y}$$

on the bottom side and

$$u = \sin(x+2y) + e^{2x+3y}$$

on the other three sides is solved. The domain is the rectangle [0, ¼] × [0, ½]. The output of `imsl_f_fast_poisson_2d` is a 17 × 33 table of values. The functions `imsl_f_spline_2d_value` are used to print a different table of values.

```
#include <imsl.h>
#include <math.h>

main()
{
        float           rhs_pde(float, float);
        float           rhs_bc(Imsl_pde_side, float, float);

        int             nx = 17;
        int             nxtabl = 5;
        int             ny = 33;
        int             nytabl = 5;

        int             i;
        int             j;
        Imsl_f_spline  *sp;
        Imsl_bc_type    bc_type[4];

        float           ax, ay, bx, by;
        float           x, y, xdata[17], ydata[33];
        float           coefu, *u;
        float           u_table;
        float           abs_error;

                        /* Set rectangle size */

        ax = 0.0;
        bx = 0.25;
        ay = 0.0;
        by = 0.50;

                        /* Set boundary conditions */

        bc_type[IMSL_RIGHT_SIDE] = IMSL_DIRICHLET_BC;
        bc_type[IMSL_BOTTOM_SIDE] = IMSL_NEUMANN_BC;
        bc_type[IMSL_LEFT_SIDE] = IMSL_DIRICHLET_BC;
        bc_type[IMSL_TOP_SIDE] = IMSL_DIRICHLET_BC;

                        /* Coefficient of u */
        coefu = 3.0;

                        /* Solve the PDE */

        u = imsl_f_fast_poisson_2d(rhs_pde, rhs_bc, coefu, nx, ny,
                                   ax, bx, ay, by, bc_type, 0);

                        /* Set up for interpolation */
```

```
        for (i = 0; i < nx; i++)
                xdata[i] = ax + (bx - ax) * (float) i / (float) (nx - 1);

        for (i = 0; i < ny; i++)
                ydata[i] = ay + (by - ay) * (float) i / (float) (ny - 1);

                        /* Compute interpolant */

        sp = imsl_f_spline_2d_interp(nx, xdata, ny, ydata, u, 0);

        printf("     x          y          u        error\n\n");
        for (i = 0; i < nxtabl; i++)
                for (j = 0; j < nytabl; j++) {
                        x = ax + (bx - ax) * (float) j / (float) (nxtabl -
                            1);
                        y = ay + (by - ay) * (float) i / (float) (nytabl -
                            1);
                        u_table = imsl_f_spline_2d_value(x, y, sp, 0);
                        abs_error = fabs(u_table - sin(x + 2.0 * y) -
                                        exp(2.0 * x + 3.0 * y));

                        /* Print computed answer and absolute on
                           nxtabl by nytabl grid */

                        printf("  %6.4f     %6.4f     %6.4f     %8.2e\n",
                               x, y, u_table, abs_error);
                }
}

float rhs_pde(float x, float y)
{
                        /* Define the right side of the PDE */

        return (-2.0 * sin(x + 2.0 * y) + 16.0 * exp(2.0 * x + 3.0 * y));
}

float rhs_bc(Imsl_pde_side side, float x, float y)
{
                        /* Define the boundary conditions */

        if (side == IMSL_BOTTOM_SIDE)
                return (2.0 * cos(x + 2.0 * y) + 3.0 * exp(2.0 * x + 3.0 *
                        y));
        else
                return (sin(x + 2.0 * y) + exp(2.0 * x + 3.0 * y));
}
```

Output

```
     x          y          u        error
  0.0000     0.0000     1.0000     0.00e+00
  0.0625     0.0000     1.1956     5.12e-06
  0.1250     0.0000     1.4087     7.19e-06
  0.1875     0.0000     1.6414     5.10e-06
  0.2500     0.0000     1.8961     8.67e-08
  0.0000     0.1250     1.7024     1.73e-07
  0.0625     0.1250     1.9562     6.39e-06
  0.1250     0.1250     2.2345     9.50e-06
```

```
0.1875     0.1250     2.5407     6.36e-06
0.2500     0.1250     2.8783     1.66e-07
0.0000     0.2500     2.5964     2.60e-07
0.0625     0.2500     2.9322     9.25e-06
0.1250     0.2500     3.3034     1.34e-05
0.1875     0.2500     3.7148     9.27e-06
0.2500     0.2500     4.1720     9.40e-08
0.0000     0.3750     3.7619     4.84e-07
0.0625     0.3750     4.2163     9.16e-06
0.1250     0.3750     4.7226     1.36e-05
0.1875     0.3750     5.2878     9.44e-06
0.2500     0.3750     5.9199     5.72e-07
0.0000     0.5000     5.3232     5.93e-07
0.0625     0.5000     5.9520     9.84e-07
0.1250     0.5000     6.6569     1.34e-06
0.1875     0.5000     7.4483     4.55e-07
0.2500     0.5000     8.3380     2.27e-06
```

Chapter 6: Transforms

Routines

6.1 Real Trigonometric FFTs

Real FFT ..... fft_real 341
Real FFT initialization ..... fft_real_init 345

6.2 Complex Exponential FFTs

Complex FFT ..... fft_complex 346
Complex FFT initialization ..... fft_complex_init 349

6.3 Real Sine and Cosine FFTs

Fourier cosine transform ..... fft_cosine 351
Fourier cosine transform initialization ..... fft_cosine_init 353
Fourier sine transform ..... fft_sine 355
Fourier sine transform initialization ..... fft_sine_init 357

6.4 Two-Dimensional FFTs

Complex two-dimensional FFT ..... fft_2d_complex 359

6.5 Convolution and Correlation

Real convolution/correlation ..... convolution 363
Complex convolution/correlation ..... convolution (complex) 370

6.6 Laplace Transform

Approximate inverse Laplace transform
of a complex function ..... inverse_laplace 376

Usage Notes

Fast Fourier Transforms

A fast Fourier transform (FFT) is simply a discrete Fourier transform that is computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately n^2 operations where n is the number of points in the transform, while the FFT (which computes the same values) takes approximately $n \log n$ operations. The algorithms in this chapter are modeled on the Cooley-Tukey (1965)

algorithm. Hence, these functions are most efficient for integers that are highly composite; that is, integers that are a product of small primes.

For the two functions `imsl_f_fft_real` (page 341) and `imsl_c_fft_complex` (page 346) , there is a corresponding initialization function. Use these functions *only* when repeatedly transforming sequences of the same length. In this situation, the initialization function computes the initial setup once; subsequently, the user calls the corresponding main function with the appropriate option. This may result in substantial computational savings. For more information on the use of these functions, consult the documentation under the appropriate function name.

In addition to the one-dimensional transformations described above, we also provide a complex two-dimensional FFT and its inverse.

Continuous Versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham 1974) as

$$\hat{f}(\omega) = (\Im f)(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \omega t} dt$$

We begin by making the following approximation:

$$\begin{aligned} \hat{f}(\omega) &\approx \int_{-T/2}^{T/2} f(t) e^{-2\pi i \omega t} dt \\ &= \int_{0}^{T} f(t - T/2) e^{-2\pi i \omega (t - T/2)} dt \\ &= e^{\pi i \omega T} \int_{0}^{T} f(t - T/2) e^{-2\pi i \omega t} dt \end{aligned}$$

If we approximate the last integral using the rectangle rule with spacing $h = T / n$, we have

$$\hat{f}(\omega) \approx e^{\pi i \omega T} h \sum_{k=0}^{n-1} e^{-2\pi i \omega k h} f(kh - T/2)$$

Finally, setting $\omega = j/T$ for $j = 0, \ldots, n - 1$ yields

$$\hat{f}(j/T) \approx e^{\pi i j} h \sum_{k=0}^{n-1} e^{-2\pi i j k / n} f(kh - T/2) = (-1)^j \sum_{k=0}^{n-1} e^{-2\pi i j k / n} f_k^h$$

where the vector $f^h = (f(-T/2), \ldots, f((n-1)h - T/2))$. Thus, after scaling the components by $(-1)^j h$, the discrete Fourier transform, as computed in `imsl_c_fft_complex` (with input f^h) is related to an approximation of the continuous Fourier transform by the above formula.

If the function f is expressed as a C function, then the continuous Fourier transform

$$\hat{f}$$

can be approximated using the IMSL function `imsl_f_int_fcn_fourier` (Chapter 4, "Quadrature").

fft_real

Computes the real discrete Fourier transform of a real sequence.

Synopsis

#include `<imsl.h>`

float `*imsl_f_fft_real` (*int* `n`, *float* `p[]`, …, 0)

The type *double* function is `imsl_d_fft_real`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed.

float `p[]` (Input)
Array with `n` components containing the periodic sequence.

Return Value

A pointer to the transformed sequence. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_fft_real` (*int* `n`, *float* `p[]`,
`IMSL_BACKWARD`,
`IMSL_PARAMS`, *float* `params[]`,
`IMSL_RETURN_USER`, *float* `q[]`,
0)

Optional Arguments

`IMSL_BACKWARD`
Compute the backward transform and return a pointer to the (backward) transformed sequence.

`IMSL_PARAMS`, *float* `params[]` (Input)
Pointer returned by a previous call to `imsl_f_fft_real_init`. If `imsl_f_fft_real` is used repeatedly with the same value of `n`, then it is more efficient to compute these parameters only once.

`IMSL_RETURN_USER`, *float* `q[]` (Output)
Store the result in the user-provided space pointed to by `q`. Therefore, no

storage is allocated for the solution, and `imsl_f_fft_real` returns `q`. The array `q` must be at least `n` long.

Description

The function `imsl_f_fft_real` computes the discrete Fourier transform of a real vector of size n. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when n is a product of small prime factors. If n satisfies this condition, then the computational effort is proportional to $n \log n$.

By default, `imsl_f_fft_real` computes the forward transform. If n is even, then the forward transform is

$$\begin{aligned} q_{2m-1} &= \sum_{k=0}^{n-1} p_k \cos\frac{2\pi km}{n} \qquad m = 1, \ldots, n/2 \\ q_{2m-2} &= -\sum_{k=0}^{n-1} p_k \sin\frac{2\pi km}{n} \qquad m = 1, \ldots, n/2 - 1 \\ q_0 &= \sum_{k=0}^{n-1} p_k \end{aligned}$$

If n is odd, q_m is defined as above for m from 1 to $(n - 1)/2$.

Let f be a real valued function of time. Suppose we sample f at n equally spaced time intervals of length Δ seconds starting at time t_0. That is, we have

$$p_i := f(t_0 + i\Delta) \quad i = 0, 1, \ldots, n - 1$$

We will assume that n is odd for the remainder of this discussion. The function `imsl_f_fft_real` treats this sequence as if it were periodic of period n. In particular, it assumes that $f(t_0) = f(t_0 + n\Delta)$. Hence, the period of the function is assumed to be $T = n\Delta$. We can invert the above transform for p as follows:

$$p_m = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1} \cos\frac{2\pi km}{n} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2} \sin\frac{2\pi km}{n} \right]$$

This formula is very revealing. It can be interpreted in the following manner. The coefficients q produced by `imsl_f_fft_real` determine an interpolating trigonometric polynomial to the data. That is, if we define

$$\begin{aligned} g(t) &= \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1} \cos\frac{2\pi k(t - t_0)}{n\Delta} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2} \sin\frac{2\pi k(t - t_0)}{n\Delta} \right] \\ &= \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1} \cos\frac{2\pi k(t - t_0)}{T} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2} \sin\frac{2\pi k(t - t_0)}{T} \right] \end{aligned}$$

then we have

$$f(t_0 + (i - 1)\,\Delta) = g(t_0 + (i - 1)\,\Delta$$

Now suppose we want to discover the dominant frequencies, forming the vector P of length $(n + 1)/2$ as follows:

$$P_0 := |q_0|$$
$$P_k := \sqrt{q_{2k-2}^2 + q_{2k-1}^2} \qquad k = 1, 2, \ldots, (n-1)/2$$

These numbers correspond to the energy in the spectrum of the signal. In particular, P_k corresponds to the energy level at frequency

$$\frac{k}{T} = \frac{k}{n\Delta} \quad k = 0, 1, \ldots, \frac{n-1}{2}$$

Furthermore, note that there are only $(n + 1)/2 \approx T/(2\Delta)$ resolvable frequencies when n observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal. Similar relations hold for the case when n is even.

If the optional argument `IMSL_BACKWARD` is specified, then the backward transform is computed. If n is even, then the backward transform is

$$q_m = p_0 + (-1)^m p_{n-1} + 2\sum_{k=0}^{n/2-1} p_{2k+1} \cos\frac{2\pi km}{n} - 2\sum_{k=0}^{n/2-2} p_{2k+2} \sin\frac{2\pi km}{n}$$

If n is odd,

$$q_m = p_0 + 2\sum_{k=0}^{(n-3)/2} p_{2k+1} \cos\frac{2\pi km}{n} - 2\sum_{k=0}^{(n-3)/2} p_{2k+2} \sin\frac{2\pi km}{n}$$

The backward Fourier transform is the unnormalized inverse of the forward Fourier transform.

The function `imsl_f_fft_real` is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Examples

Example 1

In this example, a pure cosine wave is used as a data vector, and its Fourier series is recovered. The Fourier series is a vector with all components zero except at the appropriate frequency where it has an n.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int         k, n = 7;
    float       two_pi = 2*imsl_f_constant("pi", 0);
    float       p[8], *q;
```

```
                                /* Fill q with a pure exponential signal */
    for (k = 0;  k < n;  k++)
        p[k] = cos(k*two_pi/n);

    q = imsl_f_fft_real (n, p, 0);

    printf("        index      p        q\n");
    for (k = 0;  k < n;  k++)
        printf("%11d%10.2f%10.2f\n", k, p[k], q[k]);
}
```

Output

```
index      p        q
  0      1.00      0.00
  1      0.62      3.50
  2     -0.22      0.00
  3     -0.90     -0.00
  4     -0.90     -0.00
  5     -0.22      0.00
  6      0.62     -0.00
```

Example 2

This example computes the Fourier transform of the vector x, where $x_j = (-1)^j$ for $j = 0$ to $n - 1$. The backward transform of this vector is now computed by using the optional argument IMSL_BACKWARD. Note that $s = nx$, that is, $s_j = (-1)^j n$, for $j = 0$ to $n - 1$.

```
#include <imsl.h>
#include <stdio.h>

main()
{
    int         k, n = 7;
    float       *q, *s, x[8];
                                /* Fill data vector */
    x[0] = 1.0;
    for (k = 1;  k<n;  k++)
        x[k] = -x[k-1];
                                /* Compute the forward transform of x */
    q = imsl_f_fft_real (n, x, 0);
                               /* Compute the backward transform of x */
    s = imsl_f_fft_real (n, q,
                         IMSL_BACKWARD,
                         0);
    printf("        index      x        q        s\n");
    for (k = 0;  k < n;  k++)
        printf("%11d%10.2f%10.2f%10.2f\n", k, x[k], q[k], s[k]);
}
```

Output

```
index      x        q        s
  0      1.00      1.00      7.00
  1     -1.00      1.00     -7.00
  2      1.00      0.48      7.00
  3     -1.00      1.00     -7.00
```

```
  4      1.00      1.25      7.00
  5     -1.00      1.00     -7.00
  6      1.00      4.38      7.00
```

fft_real_init

Computes the parameters for `imsl_f_fft_real`.

Synopsis

#include `<imsl.h>`

float `*imsl_f_fft_real_init` (*int* `n`)

The type *double* function is `imsl_d_fft_real_init`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed.

Return Value

A pointer to the parameter vector of length $2n + 15$ that can then be used by `imsl_f_fft_real` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Description

The function `imsl_f_fft_real_init` should be used when many calls are to be made to `imsl_f_fft_real` without changing the sequence length *n*. This function computes the parameters that are necessary for the real Fourier transform.

The function `imsl_f_fft_real_init` is based on the routine RFFTI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example computes three distinct real FFTs by calling `imsl_f_fft_real_ init` once and then calling `imsl_f_fft_real` three times.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int         k, j, n = 7;
    float       two_pi = 2*imsl_f_constant("pi", 0);
    float       p[8], *q, *work;
    work = imsl_f_fft_real_init (n);
    for (j = 0;  j < 3;  j++){
                                /* Fill p with a pure sinusoidal signal */
       for (k = 0;  k < n;  k++)
           p[k] = cos(k*two_pi*j/n);
```

```
        q = imsl_f_fft_real (n, p,
                             IMSL_PARAMS, work, 0);

        printf("        index      p        q\n");
        for (k = 0;  k < n;  k++)
            printf("%11d%10.2f%10.2f\n", k, p[k], q[k]);
    }
}
```

Output

```
index       p         q
  0      1.00      7.00
  1      1.00      0.00
  2      1.00      0.00
  3      1.00      0.00
  4      1.00      0.00
  5      1.00     -0.00
  6      1.00      0.00

 index       p         q
  0      1.00      0.00
  1      0.62      3.50
  2     -0.22      0.00
  3     -0.90     -0.00
  4     -0.90     -0.00
  5     -0.22      0.00
  6      0.62     -0.00

 index       p         q
  0      1.00     -0.00
  1     -0.22      0.00
  2     -0.90     -0.00
  3      0.62      3.50
  4      0.62     -0.00
  5     -0.90      0.00
  6     -0.22      0.00
```

fft_complex

Computes the complex discrete Fourier transform of a complex sequence.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_fft_complex` (*int* `n`, *f_complex* `p[]`, ..., 0)

The type *d_complex* function is `imsl_z_fft_complex`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed.

f_complex `p[]` (Input)
Array with `n` components containing the periodic sequence.

Return Value

If no optional arguments are used, `imsl_c_fft_complex` returns a pointer to the transformed sequence. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

f_complex *imsl_c_fft_complex (int n, f_complex p[],
        IMSL_BACKWARD,
        IMSL_PARAMS, float params[],
        IMSL_RETURN_USER, f_complex q[],
        0)
```

Optional Arguments

`IMSL_BACKWARD`
Compute the backward transform.

`IMSL_PARAMS`, *float* `params[]` (Input)
Pointer returned by a previous call to `imsl_c_fft_complex_init`. If `imsl_c_fft_complex` is used repeatedly with the same value of `n`, then it is more efficient to compute these parameters only once.

`IMSL_RETURN_USER`, *f_complex* `q[]` (Output)
Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_c_fft_complex` returns `q`. The array `q` must be of length at least `n`.

Description

The function `imsl_c_fft_complex` computes the discrete Fourier transform of a real vector of size n. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when n is a product of small prime factors. If n satisfies this condition, then the computational effort is proportional to $n \log n$.

By default, `imsl_c_fft_complex` computes the forward transform below.

$$q_j = \sum_{m=0}^{n-1} p_m e^{-2\pi imj/n}$$

Note that we can invert the Fourier transform as follows below.

$$p_m = \frac{1}{n} \sum_{j=0}^{n-1} q_j e^{2\pi ijm/n}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric interpolating polynomial to the data. The

function `imsl_c_fft_complex` is based on the complex FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the option `IMSL_BACKWARD` is selected, then the following computation is performed.

$$q_j = \sum_{m=0}^{n-1} p_m e^{2\pi imj/n}$$

Furthermore, the relation between the forward and backward transforms is that they are unnormalized inverses of each other. That is, the following code fragment begins with a vector p and concludes with a vector $p_2 = np$.

```
q  = imsl_c_fft_complex(n, p, 0);
p2 = imsl_c_fft_complex(n, q, IMSL_BACKWARD, 0);
```

Examples

Example 1

This example inputs a pure exponential data vector and recovers its Fourier series, which is a vector with all components zero except at the appropriate frequency where it has an n.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int            k, n = 7;
    float          two_pi = 2*imsl_f_constant("pi", 0);
    f_complex      p[8], *q, z;
                                  /* Fill p with a pure exponential signal */
    for (k = 0;  k < n;  k++) {
         z.re = 0.;
         z.im = k*two_pi/n;
         p[k] = imsl_c_exp(z);
     }
    q = imsl_c_fft_complex (n, p, 0);

    printf("        index    p.re      p.im      q.re      q.im\n");
    for (k = 0;  k < n;  k++)
        printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
                 q[k].re, q[k].im);
}
```

Output

```
        index    p.re      p.im      q.re      q.im
          0      1.00      0.00      0.00     -0.00
          1      0.62      0.78      7.00      0.00
          2     -0.22      0.97     -0.00     -0.00
          3     -0.90      0.43      0.00     -0.00
          4     -0.90     -0.43      0.00      0.00
```

```
          5     -0.22     -0.97     -0.00      0.00
          6      0.62     -0.78      0.00     -0.00
```

Example 2

The backward transform is used to recover the original sequence. Notice that the forward transform followed by the backward transform multiplies the entries in the original sequence by the length of the sequence.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int            k, n = 7;
    float          two_pi = 2*imsl_f_constant("pi", 0);
    f_complex      p[7], *q, *pp;
                                  /* Fill p with an increasing signal */
    for (k = 0;  k < n;  k++) {
         p[k].re = (float) k;
         p[k].im = 0.;
    }
    q  = imsl_c_fft_complex (n, p, 0);
    pp = imsl_c_fft_complex (n, q,
                             IMSL_BACKWARD,
                             0);
    printf("        index    p.re      p.im     pp.re     pp.im \n");
    for (k = 0;  k < n;  k++)
        printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
                 pp[k].re , pp[k].im);
}
```

Output

```
index    p.re      p.im     pp.re     pp.im
  0      0.00      0.00      0.00      0.00
  1      1.00      0.00      7.00      0.00
  2      2.00      0.00     14.00      0.00
  3      3.00      0.00     21.00      0.00
  4      4.00      0.00     28.00      0.00
  5      5.00      0.00     35.00      0.00
  6      6.00      0.00     42.00      0.00
```

fft_complex_init

Computes the parameters for `imsl_c_fft_complex`.

Synopsis

#include `<imsl.h>`

float `*imsl_c_fft_complex_init` (*int* `n`)

The type *double* function is `imsl_z_fft_complex_init`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed.

Return Value

A pointer to the parameter vector of type `float` and length $2n + 15$ which can then be used by `imsl_c_fft_complex` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Description

The routine `imsl_c_fft_complex_init` should be used when many calls are to be made to `imsl_c_fft_complex` without changing the sequence length *n*. This routine computes constants which are necessary for the real Fourier transform.

The function `imsl_c_fft_complex_init` is based on the routine CFFTI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example computes three distinct complex FFTs by calling `imsl_c_fft_complex_init` once, then calling `imsl_c_fft_complex` 3 times.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int            k, j, n = 7;
    float          two_pi = 2*imsl_f_constant("pi", 0), *work;
    f_complex      p[8], *q, z;
    work = imsl_c_fft_complex_init (n);
    for (j = 0;  j < 3;  j++){
                                /* Fill p with a pure exponential signal */
      for (k = 0;  k < n;  k++) {
           z.re = 0.;
           z.im = k*two_pi*j/n;
           p[k] = imsl_c_exp(z);
       }
    q = imsl_c_fft_complex (n, p,
                            IMSL_PARAMS, work, 0);

    printf("\n        index    p.re      p.im      q.re      q.im\n");
    for (k = 0;  k < n;  k++)
        printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
                 q[k].re, q[k].im);
    }
}
```

Output

```
index    p.re      p.im      q.re      q.im
  0      1.00      0.00      7.00      0.00
  1      1.00      0.00      0.00      0.00
  2      1.00      0.00      0.00      0.00
  3      1.00      0.00      0.00      0.00
  4      1.00      0.00      0.00      0.00
  5      1.00      0.00      0.00      0.00
  6      1.00      0.00      0.00      0.00

index    p.re      p.im      q.re      q.im
  0      1.00      0.00      0.00     -0.00
  1      0.62      0.78      7.00      0.00
  2     -0.22      0.97     -0.00     -0.00
  3     -0.90      0.43      0.00     -0.00
  4     -0.90     -0.43      0.00      0.00
  5     -0.22     -0.97     -0.00      0.00
  6      0.62     -0.78      0.00     -0.00

index    p.re      p.im      q.re      q.im
  0      1.00      0.00     -0.00     -0.00
  1     -0.22      0.97      0.00     -0.00
  2     -0.90     -0.43      7.00      0.00
  3      0.62     -0.78     -0.00     -0.00
  4      0.62      0.78      0.00     -0.00
  5     -0.90      0.43      0.00      0.00
  6     -0.22     -0.97     -0.00      0.00
```

fft_cosine

Computes the discrete Fourier cosine transformation of an even sequence.

Synopsis

#*include* `<imsl.h>`

float `*imsl_f_fft_cosine` (*int* `n`, *float* `p[]`, ..., 0)

The type *double* procedure is `imsl_d_fft_cosine`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

float `p[]` (Input)
Array of size `n` containing the sequence to be transformed.

Return Value

A pointer to the transformed sequence. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_fft_cosine (*int* n, *float* p[],
IMSL_RETURN_USER, *float* q[],
IMSL_PARAMS, *float* params[],
0)

Optional Arguments

IMSL_RETURN_USER, *float* q[] (Output)
Store the result in the user-provided space pointed to by q. Therefore, no storage is allocated for the solution, and imsl_f_fft_cosine returns q. The array must be of length n at least.

IMSL_PARAMS, *float* params[] (Input)
Pointer returned by a previous call to imsl_f_fft_cosine_init. If imsl_f_fft_cosine is used repeatedly with the same value of n, then it is more efficient to compute these parameters only once.
Default: Initializing parameters computed each time imsl_f_fft_cosine is entered

Description

The function imsl_f_fft_cosine computes the discrete Fourier cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $N - 1$ is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log*N*. Specifically, given an *N*-vector p, imsl_f_fft_cosine returns in q

$$q_m = 2\sum_{n=1}^{N-2} p_n \sin(\frac{mn\pi}{N-1}) + s_0 + s_{N-1}(-1)^m$$

Finally, note that the Fourier cosine transform is its own (unnormalized) inverse. The imsl_f_fft_cosine function is based on the sine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example inputs a pure cosine wave as a data vector and recovers its Fourier cosine series, which is a vector with all components zero, except n – 1 at the appropriate frequency.

```
#include <imsl.h>
#include <math.h>

main()
{
        int             n = 7;
        int             i;
        float           p[7];
        float          *q;
```

```
        float           pi;

        pi = imsl_f_constant("pi", 0);

                                /* Fill p with a pure cosine wave */

        for (i=0; i<n; i++)
                p[i] = cos((float)(i)*pi/(float)(n-1));

        q = imsl_f_fft_cosine (n, p, 0);

        printf ("      index\t   p\t   q\n");
        for (i=0; i<n; i++)
                printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);
}
```

Output

```
      index          p           q
        0         1.00        -0.00
        1         0.87         6.00
        2         0.50         0.00
        3        -0.00         0.00
        4        -0.50        -0.00
        5        -0.87        -0.00
        6        -1.00        -0.00
```

fft_cosine_init

Computes the parameters needed for `imsl_f_fft_cosine`.

Synopsis

#*include* `<imsl.h>`

float `*imsl_f_fft_cosine_init` (*int* `n`)

The type *double* procedure is `imsl_d_fft_cosine_init`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

Return Value

A pointer to parameter vector of length (3*`n` + 15) that can then be used by `imsl_f_fft_cosine` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Description

The function `imsl_f_fft_cosine_init` should be used when many calls must be made to `imsl_f_fft_cosine` without changing the sequence length `n`. The function `imsl_f_fft_cosine_init` is based on the routine COSTI in FFTPACK. The

package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example computes three distinct sine FFTs by calling `imsl_f_fft_cosine_init` once, then calling `imsl_f_fft_cosine` three times. The internal parameter initialization in `imsl_f_fft_cosine` is now skipped.

```
#include <imsl.h>
#include <math.h>

main()
{
        int             n = 7;
        int             i, k;
        float           p[7];
        float           q[7];
        float           pi;
        float          *params;

        pi = imsl_f_constant("pi", 0);

                                /* Compute parameters for transform of
                                   length n */

        params = imsl_f_fft_cosine_init (n);

                                /* Different frequencies of the same
                                   wave will be transformed */
        for (k=0; k<3; k++) {
                printf("\n");

                                /* Fill p with a pure cosine wave */

                for (i=0; i<n; i++)
                        p[i] = cos((float)((k+1)*i)*pi/(float)(n-1));

                                /* Compute the transform of p */

                imsl_f_fft_cosine (n, p,
                        IMSL_PARAMS, params,
                        IMSL_RETURN_USER, q,
                        0);

                printf ("      index\t   p\t   q\n");
                for (i=0; i<n; i++)
                        printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);

        }
}
```

Output

```
      index         p          q
        0         1.00       -0.00
        1         0.87        6.00
        2         0.50        0.00
```

```
  3        -0.00         0.00
  4        -0.50        -0.00
  5        -0.87        -0.00
  6        -1.00        -0.00

index          p           q
  0         1.00         0.00
  1         0.50        -0.00
  2        -0.50         6.00
  3        -1.00         0.00
  4        -0.50         0.00
  5         0.50         0.00
  6         1.00        -0.00

index          p           q
  0         1.00        -0.00
  1        -0.00         0.00
  2        -1.00        -0.00
  3         0.00         6.00
  4         1.00         0.00
  5        -0.00        -0.00
  6        -1.00         0.00
```

fft_sine

Computes the discrete Fourier sine transformation of an odd sequence.

Synopsis

#include `<imsl.h>`

float `*imsl_f_fft_sine` (*int* `n`, *float* `p[]`, …, 0)

The type *double* procedure is `imsl_d_fft_sine`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

float `p[]` (Input)
Array of size `n` containing the sequence to be transformed.

Return Value

A pointer to the transformed sequence. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_fft_sine` (*int* `n`, *float* `p[]`,
`IMSL_RETURN_USER`, *float* `q[]`,

IMSL_PARAMS, *float* params[],
0)

Optional Arguments

IMSL_RETURN_USER, *float* q[] (Output)
Store the result in the user-provided space pointed to by q. Therefore, no storage is allocated for the solution, and imsl_f_fft_sine returns q. The array must be of length at least n + 1.

IMSL_PARAMS, *float* params[] (Input)
Pointer returned by a previous call to imsl_f_fft_sine_init. If imsl_f_fft_sine is used repeatedly with the same value of n, then it is more efficient to compute these parameters only once.
Default: Initializing parameters computed each time imsl_f_fft_sine is entered

Description

The function imsl_f_fft_sine computes the discrete Fourier sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $N + 1$ is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to $N \log N$. Specifically, given an *N*-vector p, imsl_f_fft_sine returns in q

$$q_m = 2\sum_{n=0}^{N-1} p_n \sin\left(\frac{(m+1)(n+1)\pi}{N+1}\right)$$

Finally, note that the Fourier sine transform is its own (unnormalized) inverse. The function imsl_f_fft_sine is based on the sine FFT in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example inputs a pure sine wave as a data vector and recovers its Fourier sine series, which is a vector with all components zero, except n at the appropriate frequency.

```
#include <imsl.h>
#include <math.h>

main()
{
        int             n = 7;
        int             i;
        float           p[7];
        float           *q;
        float           pi;

        pi = imsl_f_constant("pi", 0);

                        /* fill p with a pure sine wave */
```

```
        for (i=0; i<n; i++)
                p[i] = sin((float)(i+1)*pi/(float)(n+1));

        q = imsl_f_fft_sine (n, p, 0);

        printf ("      index\t   p\t   q\n");
        for (i=0; i<n; i++)
                printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);
}
```

Output

```
      index      p        q
       0       0.38     8.00
       1       0.71     0.00
       2       0.92     0.00
       3       1.00     0.00
       4       0.92     0.00
       5       0.71     0.00
       6       0.38     0.00
```

fft_sine_init

Computes the parameters needed for `imsl_f_fft_sine`.

Synopsis

#include `<imsl.h>`

float `*imsl_f_fft_sine_init` (*int* `n`)

The type *double* procedure is `imsl_d_fft_sine_init`.

Required Arguments

int `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

Return Value

A pointer to parameter vector of length (*int*) (2.5`*n` + 15) that can then be used by `imsl_f_fft_sine` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Description

The function `imsl_f_fft_sine_init` should be used when many calls must be made to `imsl_f_fft_sine` without changing the sequence length `n`. The function `imsl_f_fft_sine_init` is based on the routine SINTI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

This example computes three distinct sine FFTs by calling `imsl_f_fft_sine_init` once, then calling `imsl_f_fft_sine` three times. The internal parameter initialization in `imsl_f_fft_sine` is now skipped.

```
#include <imsl.h>
#include <math.h>

main()
{
        int             n = 7;
        int             i, k;
        float           p[7];
        float           q[8];
        float           pi;
        float          *params;

        pi = imsl_f_constant("pi", 0);

                                /* Compute parameters for transform of
                                   length n */

        params = imsl_f_fft_sine_init (n);

                                /* Different frequencies of the same
                                   wave will be transformed */
        for (k=0; k<3; k++) {
                printf("\n");

                                /* Fill p with a pure sine wave */

                for (i=0; i<n; i++)
                        p[i] = sin((float)((k+1)*(i+1))*pi/(float)(n+1));

                                /* Compute the transform of p */

                imsl_f_fft_sine (n, p,
                        IMSL_PARAMS, params,
                        IMSL_RETURN_USER, q,
                        0);

                printf ("      index\t   p\t   q\n");
                for (i=0; i<n; i++)
                        printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);

        }
}
```

Output

```
      index         p           q
        0         0.38        8.00
        1         0.71        0.00
        2         0.92        0.00
        3         1.00        0.00
        4         0.92        0.00
        5         0.71        0.00
        6         0.38        0.00
```

```
index          p           q
  0         0.71        -0.00
  1         1.00         8.00
  2         0.71         0.00
  3        -0.00        -0.00
  4        -0.71         0.00
  5        -1.00        -0.00
  6        -0.71         0.00


index          p           q
  0         0.92         0.00
  1         0.71        -0.00
  2        -0.38         8.00
  3        -1.00         0.00
  4        -0.38         0.00
  5         0.71         0.00
  6         0.92         0.00
```

fft_2d_complex

Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_fft_2d_complex` (*int* `n`, *int* `m`, *f_complex* `p[]`, …, 0)

The type *d_complex* function is `imsl_z_fft_2d_complex`.

Required Arguments

int `n` (Input)
Number of rows in the two-dimensional transform.

int `m` (Input)
Number of columns in the two-dimensional transform.

f_complex `p[]` (Input)
Two-dimensional array of size $n \times m$ containing the sequence that is to be transformed.

Return Value

A pointer to the transformed array. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_fft_2d_complex` (*int* `n`, *int* `m`, *f_complex* `p[]`,
`IMSL_P_COL_DIM`, *int* `p_col_dim`,
`IMSL_BACKWARD`,
`IMSL_RETURN_USER`, *f_complex* `q[]`,
`IMSL_Q_COL_DIM`, *int* `q_col_dim`,
0)

Optional Arguments

`IMSL_P_COL_DIM`, *int* `p_col_dim` (Input)
The column dimension of `p`.
Default: `p_col_dim` = *m*

`IMSL_BACKWARD`
Compute the backward transform.

`IMSL_RETURN_USER`, *f_complex* `q[]` (Output)
Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_c_fft_2d_complex` returns `q`. The array must be of length at least $n \times m$.

`IMSL_Q_COL_DIM`, *int* `q_col_dim` (Input)
The column dimension of `q`.
Default: `q_col_dim` = *m*

Description

The function `imsl_c_fft_2d_complex` computes the discrete Fourier transform of a two-dimensional complex array of size $n \times m$. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when both n and m are a product of small prime factors. If n and m satisfy this condition, then the computational effort is proportional to $nm \log nm$.

By default, `imsl_c_fft_2d_complex` computes the forward transform below.

$$q_{jk} = \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} p_{st} e^{-2\pi ijs/n} e^{-2\pi ikt/m}$$

Note that we can invert the Fourier transform as follows.

$$p_{jk} = \frac{1}{nm} \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric interpolating polynomial to the data. The function `imsl_c_fft_2d_complex` is based on the complex FFT in FFTPACK,

which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the option IMSL_BACKWARD is selected, then the following computation is performed.

$$p_{jk} = \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

The relation between the forward and backward transforms is that they are unnormalized inverses of each other. That is, the following code fragment begins with a vector p and concludes with a vector $p_2 = nmp$.

```
q  = imsl_c_fft_2d_complex(n, m, p, 0);
p2 = imsl_c_fft_2d_complex(n, m, q, IMSL_BACKWARD, 0);
```

Examples

Example 1

This example computes the Fourier transform of the pure frequency input for a 5×4 array

$$p_{st} = e^{2\pi i 2s/5} \; e^{2\pi i t\, 3/4}$$

for $0 \le n \le 4$ and $0 \le m \le 3$. The result, $\hat{p} = q$, has all zeros except in the [2][3] position.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int            s, t, n = 5, m =4;
    float          two_pi = 2*imsl_f_constant("pi", 0);
    f_complex      p[5][4], *q, z, w;
                            /* Fill p with a pure exponential signal */
    for (s = 0;  s < n;  s++) {
         z.re = 0.;
         z.im = s*two_pi*2./n;
         for(t =0; t < m; t++){
         w.re = 0.;
         w.im = t*two_pi*3./m;
         p[s][t] = imsl_c_mul(imsl_c_exp(z),imsl_c_exp(w));
         }
     }
    q = imsl_c_fft_2d_complex (n, m, (f_complex*)p, 0);
                               /* Write the input */
    imsl_c_write_matrix ("The input matrix is ", 5, 4,  (f_complex*)p,
                                  IMSL_ROW_NUMBER_ZERO,
                                  IMSL_COL_NUMBER_ZERO, 0);
    imsl_c_write_matrix ("The output matrix is ", 5, 4, q,
                                  IMSL_ROW_NUMBER_ZERO,
```

```
                                    IMSL_COL_NUMBER_ZERO, 0);
}
```

Output

```
                          The input matrix is
                        0                       1                       2
0 (     1.000,     0.000)  (     0.000,    -1.000)  (    -1.000,    -0.000)
1 (    -0.809,     0.588)  (     0.588,     0.809)  (     0.809,    -0.588)
2 (     0.309,    -0.951)  (    -0.951,    -0.309)  (    -0.309,     0.951)
3 (     0.309,     0.951)  (     0.951,    -0.309)  (    -0.309,    -0.951)
4 (    -0.809,    -0.588)  (    -0.588,     0.809)  (     0.809,     0.588)

                        3
0 (    -0.000,     1.000)
1 (    -0.588,    -0.809)
2 (     0.951,     0.309)
3 (    -0.951,     0.309)
4 (     0.588,    -0.809)

                          The output matrix is
                        0                       1                       2
0 (        -0,        -0)  (         0,        -0)  (         0,        -0)
1 (         0,         0)  (         0,        -0)  (        -0,         0)
2 (        -0,        -0)  (         0,        -0)  (         0,        -0)
3 (         0,         0)  (         0,        -0)  (        -0,         0)
4 (        -0,        -0)  (         0,        -0)  (         0,        -0)

                        3
0 (         0,        -0)
1 (         0,        -0)
2 (        20,         0)
3 (        -0,        -0)
4 (        -0,        -0)
```

Example 2

This example uses the backward transform to recover the original sequence. Notice that the forward transform followed by the backward transform multiplies the entries in the original sequence by the product of the lengths of the two dimensions.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

main()
{
    int            s, t, n = 5, m =4;
    f_complex      p[5][4], *q, *p2;
                              /* Fill p with a pure exponential signal */
    for (s = 0;  s < n;  s++) {
         for(t =0; t < m; t++){
         p[s][t].re = s + 5*t;
         p[s][t].im = 0.;
         }
     }                        /* Forward transform */
    q = imsl_c_fft_2d_complex (n, m, (f_complex*)p, 0);
                              /* Backward transform */
    p2 = imsl_c_fft_2d_complex (n, m, q,
```

```
                                IMSL_BACKWARD, 0);
                              /* Write the input */
    imsl_c_write_matrix ("The input matrix is ", 5, 4, (f_complex*)p,
                                IMSL_ROW_NUMBER_ZERO,
                                IMSL_COL_NUMBER_ZERO, 0);
    imsl_c_write_matrix ("The output matrix is ", 5, 4, p2,
                                IMSL_ROW_NUMBER_ZERO,
                                IMSL_COL_NUMBER_ZERO, 0);
}
```

Output

```
                          The input matrix is
                           0                          1                          2
0 (          0,          0)  (          5,          0)  (         10,          0)
1 (          1,          0)  (          6,          0)  (         11,          0)
2 (          2,          0)  (          7,          0)  (         12,          0)
3 (          3,          0)  (          8,          0)  (         13,          0)
4 (          4,          0)  (          9,          0)  (         14,          0)

                           3
0 (         15,          0)
1 (         16,          0)
2 (         17,          0)
3 (         18,          0)
4 (         19,          0)

                          The output matrix is
                           0                          1                          2
0 (          0,          0)  (        100,          0)  (        200,          0)
1 (         20,          0)  (        120,          0)  (        220,          0)
2 (         40,          0)  (        140,          0)  (        240,          0)
3 (         60,          0)  (        160,          0)  (        260,          0)
4 (         80,          0)  (        180,          0)  (        280,          0)

                           3
0 (        300,          0)
1 (        320,          0)
2 (        340,          0)
3 (        360,          0)
4 (        380,          0)
```

convolution

Computes the convolution, and optionally, the correlation of two real vectors.

Synopsis

#*include* `<imsl.h>`

float `*imsl_f_convolution` (*int* `nx`, *float* `x[]`, *int* `ny`, *float* `y[]`, *int* `*nz`, …, 0)

The type *double* function is `imsl_d_convolution`.

Required Arguments

int `nx` (Input)
Length of the vector `x`.

float `x[]` (Input)
Real vector of length `nx`.

int `ny` (Input)
Length of the vector `y`.

float `y[]` (Input)
Real vector of length `ny`.

int `*nz` (Output)
Length of the output vector.

Return Value

A pointer to an array of length `nz` containing the convolution of `x` and `y`. To release this space, use `free`. If no zeros are computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
```

float `*imsl_f_convolution` (*int* `nx`, *float* `x[]`, *int* `ny`, *float* `y[]`, *int* `*nz`,
`IMSL_PERIODIC`,
`IMSL_CORRELATION`,
`IMSL_FIRST_CALL`,
`IMSL_CONTINUE_CALL`,
`IMSL_LAST_CALL`,
`IMSL_RETURN_USER`, *float* `z[]`,
`IMSL_Z_TRANS`, *float* `*zhat`,
0)

Optional Arguments

`IMSL_PERIODIC`
The input is periodic.

`IMSL_CORRELATION`
Return the correlation of `x` and `y`.

`IMSL_FIRST_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on the first call.

`IMSL_CONTINUE_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on intermediate calls.

`IMSL_LAST_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on the final call.

IMSL_RETURN_USER, *float* z[] (Output)
User-supplied array of length at least nz containing the convolution or correlation of x and y.

IMSL_Z_TRANS, *float* zhat[] (Output)
User-supplied array of length at least nz containing on output the discrete Fourier transform of z.

Description

The function imsl_f_convolution, by default, computes the discrete convolution of two sequences x and y. More precisely, let n_x be the length of x, and n_y denote the length of y. If a circular convolution is desired, the optional argument IMSL_PERIODIC must be selected. We set

$$n_z = \max\{n_y, n_x\},$$

and we pad out the shorter vector with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on x is interpreted as a positive number between 1 and n_z, modulo n_z.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where the following equation is true.

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of x and y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if option IMSL_PERIODIC is selected. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If option IMSL_PERIODIC is not selected, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq n_x + n_y - 1$. This will mean that both vectors will be padded with zeros.

We point out that no complex transforms of x or y are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Optionally, the function imsl_f_convolution computes the discrete correlation of two sequences x and y. More precisely, let n be the length of x and y. If a circular

correlation is desired, then option IMSL_PERIODIC must be selected. We set (on output)

$n_z = n$ if IMSL_PERIODIC is chosen

$(n_z = 2^{\alpha}3^{\beta}5^{\gamma} \geq 2n - 1)$ if IMSL_PERIODIC is not chosen

where α, β, and γ are nonnegtive integers yielding the smallest number of the type $2^{\alpha}3^{\beta}5^{\gamma}$ satisfying the inequality. Once n_z is determined, we pad out the vectors with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on x is interpreted as a positive number between one and n_z, modulo n_z. Note that this means that

$$z_{n_z - k}$$

contains the correlation of $x(k - 1)$ with y as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k - 1) = y(k)$ for all k, then we would expect

$$z_{n_z}$$

to be the largest component of z. The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}_j = \hat{x}_j \overline{y}_j$$

where the following equation is true.

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of x and the conjugate of the discrete Fourier transform of y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if IMSL_PERIODIC is selected. If n_z is the product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If IMSL_PERIODIC is not chosen, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeros.

We point out that no complex transforms of x or y are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Examples

Example 1

This example computes a nonperiodic convolution. The idea here is that you can compute a moving average of the type found in digital filtering using this function. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. We try to recover the values of an exponential function contaminated by noise. The large error for the last value has to do with the fact that the convolution is averaging the zeros in the "pad" rather than the function values. Notice that the signal size is 100, but only reports the errors at 10 points.

```
#include "imsl.h"
#include <math.h>

#define NFLTR  5
#define NY     100

        /* Define function */

#define F1(A)   exp(A)
main()
{
    int         i, k, nz;
    float       fltr[NFLTR], fltrer, origer, total1, total2, twopi,
                x, y[NY], *z, *noise;

        /* Set up the filter */
    for (i = 0; i < NFLTR; i++) fltr[i] = 0.2;

        /*
         * Set up y-vector for the nonperiodic casE.
         */

    twopi = 2.0*imsl_f_constant ("Pi", 0);
    imsl_random_seed_set(1234579);
    noise = imsl_f_random_uniform(NY, 0);

    for (i = 0; i < NY; i++) {
        x = (float)(i) / (NY - 1);
        y[i] = F1(x) + 0.5 *noise[i]  - 0.25;
    }
        /*
         * Call the convolution routine for the nonperiodic case.
         */

    z = imsl_f_convolution(NFLTR, fltr, NY, y, &nz, 0);
        /*
         * Call test routines to check z & zhat here. Print results
         */
    printf("\n Nonperiodic Case\n");
    printf("        x          F1(x)        Original Error");
    printf("   Filtered Error\n");

    total1 = 0.0;
    total2 = 0.0;
    for (i = 0; i < NY; i++) {
        if (i >= NY-2)
```

```
            k = i - NY + 2;
        else
            k = i + 2;
        x = (float)(i) / (float) (NY - 1);
        origer = fabs(y[i] - F1(x));
        fltrer = fabs(z[i+2] - F1(x));
        if ((i % 11) == 0) {
            printf(" %10.4f%13.4f%18.4f%18.4f\n",
                    x, F1(x), origer, fltrer);
        }
        total1 += origer;
        total2 += fltrer;
    }
    printf(" Average absolute error before filter:%10.5f\n",
            total1 /  (NY));
    printf(" Average absolute error after filter:%11.5f\n",
            total2 /  (NY));

}
```

Output

```
Nonperiodic Case
        x          F1(x)        Original Error   Filtered Error
     0.0000       1.0000            0.0811            0.3523
     0.1111       1.1175            0.0226            0.0754
     0.2222       1.2488            0.1526            0.0488
     0.3333       1.3956            0.0959            0.0161
     0.4444       1.5596            0.1747            0.0276
     0.5556       1.7429            0.1035            0.0250
     0.6667       1.9477            0.0402            0.0562
     0.7778       2.1766            0.0673            0.0835
     0.8889       2.4324            0.1044            0.0050
     1.0000       2.7183            0.0154            1.1255
Average absolute error before filter:   0.12481
Average absolute error after filter:    0.06785
```

Example 2

This example computes both a periodic correlation between two distinct signals x and y. There are 100 equally spaced points on the interval $[0, 2\pi]$ and $f_1(x) = \sin(x)$. Define x and y as follows:

$$x_i = f_1\left(\frac{2\pi i}{n-1}\right) \qquad i = 0, \ldots, n-1$$

$$y_i = f_1\left(\frac{2\pi i}{n-1} + \frac{\pi}{2}\right) \qquad i = 0, \ldots, n-1$$

Note that the maximum value of z (the correlation of x with) occurs at $i = 25$, which corresponds to the offset.

```
#include "imsl.h"
#include <math.h>

#define N    100
```

```
        /* Define function */

#define F1(A)   sin(A)

main()
{
    int         i, k, nz;
    float       pi, max,
                x[N], y[N], *z, xnorm, ynorm;

        /*
         * Set up y-vector for the nonperiodic case.
         */

    pi = imsl_f_constant ("Pi", 0);

    for (i = 0; i < N; i++) {
        x[i] = F1(2.0*pi*(float)(i) / (N-1));
        y[i] = F1(2.0*pi*(float)(i) / (N-1) + pi/2.0);
    }
        /*
         * Call the correlation function for the nonperiodic case.
         */

    z = imsl_f_convolution(N, x, N, y, &nz,
                IMSL_CORRELATION, IMSL_PERIODIC,0);

    xnorm = imsl_f_vector_norm (N, x, 0);
    ynorm = imsl_f_vector_norm (N, y, 0);
    for (i = 0; i < N; i++) {
        z[i] /= xnorm*ynorm;
    }

    max = z[0];
    k = 0;
    for (i = 1; i < N; i++) {
        if (max < z[i]) {
            max = z[i];
            k = i;
        }
    }

    printf("The element of Z with the largest normalized\n");
    printf("value is Z(%2d).\n", k);
    printf("The normalized value of Z(%2d) is %6.3f\n", k, z[k]);

}
```

Output

```
The element of Z with the largest normalized
value is Z(25).
The normalized value of Z(25) is  1.000
```

convolution (complex)

Computes the convolution, and optionally, the correlation of two complex vectors.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_convolution` (*int* `nx`, *f_complex* `x[]`, *int* `ny`, *f_complex* `y[]`, *int* `*nz`, ..., 0)

The type *double* function is `imsl_d_convolution`.

Required Arguments

int `nx` (Input)
Length of the vector *x*.

f_complex `x[]` (Input)
Real vector of length *nx*.

int `ny` (Input)
Length of the vector *y*.

f_complex `y[]` (Input)
Real vector of length *ny*.

int `*nz` (Output)
Length of the output vector.

Return Value

A pointer to an array of length `nz` containing the convolution of *x* and *y*. To release this space, use `free`. If no zeros are computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_convolution` (*int* `nx`, *f_complex* `x[]`, *int* `ny`, *f_complex* `y[]`, *int*`*nz`,
`IMSL_PERIODIC`,
`IMSL_CORRELATION`,
`IMSL_FIRST_CALL`,
`IMSL_CONTINUE_CALL`,
`IMSL_LAST_CALL`,
`IMSL_RETURN_USER`, *f_complex* `z[]`,
`IMSL_Z_TRANS`, *f_complex* `*zhat`,
0)

Optional Arguments

`IMSL_PERIODIC`
The input is periodic.

`IMSL_CORRELATION`
Return the correlation of x and y.

`IMSL_FIRST_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on the first call.

`IMSL_CONTINUE_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on intermediate calls.

`IMSL_LAST_CALL`
If the function is called multiple times with the same `nx` and `ny`, select this option on the final call.

`IMSL_RETURN_USER`, *f_complex* `z[]` (Output)
User-supplied array of length at least `nz` containing the convolution or correlation of `x` and `y`.

`IMSL_Z_TRANS`, *f_complex* `zhat[]` (Output)
User-supplied array of length at least `nz` containing on output the discrete Fourier transform of `z`.

Description

The function `imsl_c_convolution`, by default, computes the discrete convolution of two sequences x and y. More precisely, let n_x be the length of x, and n_y denote the length of y. If a circular convolution is desired, the optional argument `IMSL_PERIODIC` must be selected. We set

$$n_z = \max\{n_y, n_x\}$$

and we pad out the shorter vector with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on x is interpreted as a positive number between 1 and n_z, modulo n_z.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of `z` is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where the following equation is true.

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of x and y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if option `IMSL_PERIODIC` is selected. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If option `IMSL_PERIODIC` is not selected, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq n_x + n_y - 1$. This will mean that both vectors will be padded with zeros.

Optionally, the function `imsl_c_convolution` computes the discrete correlation of two sequences x and y. More precisely, let n be the length of x and y. If a circular correlation is desired, then option `IMSL_PERIODIC` must be selected.

We set (on output)

$n_z = n$ if `IMSL_PERIODIC` is chosen

$(n_z = 2^{\alpha}3^{\beta}5^{\gamma} \geq 2n - 1)$ if `IMSL_PERIODIC` is not chosen

where α, β, and γ are nonnegative integers yielding the smallest number of the type $2^{\alpha}3^{\beta}5^{\gamma}$ satisfying the inequality. Once n_z is determined, we pad out the vectors with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on x is interpreted as a positive number between one and n_z, modulo n_z. Note that this means that

$$z_{n_z - k}$$

contains the correlation of $x(k-1)$ with y as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k-1) = y(k)$ for all k, then we would expect

$$\Re z_{n_z}$$

to be the largest component of $\Re z$. The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication.

Thus, the Fourier transform of z is given by

$$\hat{z}_j = \hat{x}_j \overline{y}_j$$

where the following equation is true.

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of x and the conjugate of the discrete Fourier transform of y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if `IMSL_PERIODIC` is selected. If n_z is the product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If `IMSL_PERIODIC` is not chosen, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeros.

No complex transforms of x or y are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Examples

Example 1

This example computes a nonperiodic convolution. The purpose is to compute a moving average of the type found in digital filtering. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. We try to recover the values of an exponential function contaminated by noise. The large error for the last value has to do with the fact that the convolution is averaging the zeros in the "pad" rather than the function values. Notice that the signal size is 100, but only report the errors at ten points.

```
#include "imsl.h"
#include <math.h>

#define NFLTR  5
#define NY     100

#define F1(A)   (imsl_c_mul(imsl_cf_convert(exp(A),0.0), \
                           imsl_cf_convert(cos(A),sin(A)) ))

main()
{
    int         i, nz;
    f_complex   fltr[NFLTR], temp,
                y[NY], *z;
    float       x, twopi, total1, total2, *noise, origer, fltrer;

                /* Set up the filter */
    for (i = 0; i < NFLTR; i++) fltr[i] = imsl_cf_convert(0.2,0.0);

                /* Set up y-vector for the periodic case */

    twopi = 2.0*imsl_f_constant ("Pi", 0);
    imsl_random_seed_set(1234579);
    noise = imsl_f_random_uniform(2*NY, 0);
```

```
    for (i = 0; i < NY; i++) {
        x = (float)(i) / (NY - 1);
        temp = imsl_cf_convert(0.5*noise[i]-0.25, 0.5*noise[NY+i]-0.25);
        y[i] = imsl_c_add(F1(x), temp);
    }
                /* Call the convolution routine for the periodic case */
    z = imsl_c_convolution(NFLTR, fltr, NY, y, &nz, 0);


        /* Print results */
    printf(" Periodic Case\n");
    printf("        x          F1(x)        Original Error");
    printf("   Filtered Error\n");

    total1 = 0.0;
    total2 = 0.0;
    for (i = 0; i < NY; i++) {
        x = (float)(i) / (NY - 1);
        origer = imsl_c_abs(imsl_c_sub(y[i],F1(x)));
        fltrer = imsl_c_abs(imsl_c_sub(z[i+2],F1(x)));
        if ((i % 11) == 0)
            printf(" %10.4f   (%6.4f,%6.4f) %12.4f %15.4f\n",
                     x, (F1(x)).re, (F1(x)).im, origer, fltrer);

            total1 += origer;
            total2 += fltrer;
        }
        printf(" Average absolute error before filter:%10.5f\n",
                 total1 / (NY));
        printf(" Average absolute error after filter:%11.5f\n",
                 total2 / (NY));
}
```

Output

```
Periodic Case
        x          F1(x)         Original Error   Filtered Error
     0.0000   (1.0000,0.0000)       0.1684          0.3524
     0.1111   (1.1106,0.1239)       0.0582          0.0822
     0.2222   (1.2181,0.2752)       0.1991          0.1054
     0.3333   (1.3188,0.4566)       0.1487          0.1001
     0.4444   (1.4081,0.6706)       0.2381          0.1004
     0.5556   (1.4808,0.9192)       0.1037          0.0708
     0.6667   (1.5307,1.2044)       0.1312          0.0904
     0.7778   (1.5508,1.5273)       0.1695          0.0856
     0.8889   (1.5331,1.8885)       0.1851          0.0698
     1.0000   (1.4687,2.2874)       0.2130          1.0760
Average absolute error before filter:   0.19057
Average absolute error after filter:    0.10024
```

Example 2

This example computes both a periodic correlation between two distinct signals x and y. There are 100 equally spaced points on the interval $[0, 2\pi]$ and $f_1(x) = \cos(x) + i \sin(x)$. Define x and y as follows:

$$x_i = f_1\left(\frac{2\pi(i-1)}{n-1}\right) \qquad i = 1, \ldots, n$$

$$y_i = f_1\left(\frac{2\pi(i-1)}{n-1} + \frac{\pi}{2}\right) \qquad i = 1, \ldots, n$$

Note that the maximum value of z (the correlation of x with) occurs at $i = 25$, which corresponds to the offset.

```
#include "imsl.h"
#include <math.h>

#define N    100

                /*  Define function */

#define F1(A)   imsl_cf_convert(cos(A),sin(A))

main()
{
    int         i, k, nz;
    float       zreal[4*N], pi, max, xnorm, ynorm, sumx, sumy;
    f_complex   x[N], y[N], *z;

                /* Set up y-vector for the nonperiodic case */

    pi = imsl_f_constant ("Pi", 0);

    for (i = 0; i < N; i++) {
        x[i] = F1(2.0*pi*(float)(i) / (N-1));
        y[i] = F1(2.0*pi*(float)(i) / (N-1) + pi/2.0);
    }
                /* Call the correlation function for the
                   nonperidic case */

    z = imsl_c_convolution(N, x, N, y, &nz,
                IMSL_CORRELATION, IMSL_PERIODIC,0);

    sumx = sumy = 0.0;
    for (i = 0; i < N; i++) {
        sumx += imsl_c_abs(imsl_c_mul(x[i], x[i]));
        sumy += imsl_c_abs(imsl_c_mul(y[i], y[i]));
    }
    xnorm = sqrt((sumx));
    ynorm = sqrt((sumy));
    for (i = 0; i < N; i++) {
        zreal[i] = (z[i].re/(xnorm*ynorm));
    }

    max = zreal[0];
    k = 0;
    for (i = 1; i < N; i++) {
        if (max < zreal[i]) {
            max = zreal[i];
            k = i;
        }
    }
```

```
    printf("The element of Z with the largest normalized\n");
    printf("value is Z(%2d).\n", k);
    printf("The normalized value of Z(%2d) is %6.3f\n", k, zreal[k]);

}
```

Output

```
The element of Z with the largest normalized
value is Z(25).
The normalized value of Z(25) is  1.000
```

inverse_laplace

Computes the inverse Laplace transform of a complex function.

Synopsis

#include `<imsl.h>`

float `*imsl_f_inverse_laplace` (*f_complex* `fcn()`, *float* `sigma0`, *int* `n`, *float* `t[]`, ..., 0)

The type *double* procedure is `imsl_d_inverse_laplace`.

Required Arguments

f_complex `fcn`(*f_complex* `z`) (Input)
User-supplied function for which the inverse Laplace transform will be computed.

float `sigma0` (Input)
An estimate for the maximum of the real parts of the singularities of `fcn`. If unknown, set `sigma0` = 0.0.

int `n` (Input)
The number of points at which the inverse Laplace transform is desired.

float `t[]` (Input)
Array of size `n` containing the points at which the inverse Laplace transform is desired.

Return Value

A pointer to the array of length `n` whose *i*-th component contains the approximate value of the inverse Laplace transform at the point `t`[*i*]. To release this space, use `free`. If no solution was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_inverse_laplace` (*f_complex* `fcn()`, *float* `sigma0`, *int* `n`, *float* `t[]`,
`IMSL_RETURN_USER`, *float* `x[]`,
`IMSL_PSEUDO_ACCURACY`, *float* `pseudo_accuracy`,
`IMSL_FIRST_LAGUERRE_PARAMETER`, *float* `sigma`,
`IMSL_SECOND_LAGUERRE_PARAMETER`, *float* `bvalue`,
`IMSL_MAXIMUM_COEFFICIENTS`, *int* `mtop`,
`IMSL_ERROR_EST`, *float* `*error_est`,
`IMSL_DISCRETIZATION_ERROR_EST`, *float* `*disc_error_est`,
`IMSL_TRUNCATION_ERROR_EST`, *float* `*trunc_error_est`,
`IMSL_CONDITION_ERROR_EST`, *float* `*cond_error_est`,
`IMSL_DECAY_FUNCTION_COEFFICIENT`, *float* `*k`,
`IMSL_DECAY_FUNCTION_BASE`, *float* `*r`,
`IMSL_LOG_LARGEST_COEFFICIENTS`, *float* `*log_largest_coefs`,
`IMSL_LOG_SMALLEST_COEFFICIENTS`,
float `*log_smallest_coefs`,
`IMSL_UNDER_OVERFLOW_INDICATORS`,
Imsl_laplace_flow `*indicators`,
`IMSL_FCN_W_DATA`, *f_complex* `fcn` ( ), *void* `*data`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)
A user-allocated array of length `n` containing the approximate value of the inverse Laplace transform.

`IMSL_PSEUDO_ACCURACY`, *float* `pseudo_accuracy` (Input)
The required absolute uniform pseudo accuracy for the coefficients and inverse Laplace transform values.

Default: `pseudo_accuracy` = $\sqrt{\varepsilon}$, where ε is machine epsilon

`IMSL_FIRST_LAGUERRE_PARAMETER`, *float* `sigma` (Input)
The first parameter of the Laguerre expansion. If `sigma` is not greater than `sigma0`, it is reset to `sigma0` + 0.7.
Default: `sigma` = `sigma0` + 0.7

`IMSL_SECOND_LAGUERRE_PARAMETER`, *float* `bvalue` (Input)
The second parameter of the Laguerre expansion. If `bvalue` is less than 2.0*(`sigma` − `sigma0`), it is reset to 2.5*(`sigma` − `sigma0`).
Default: `bvalue` = 2.5*(`sigma` − `sigma0`)

`IMSL_MAXIMUM_COEFFICIENTS`, *int* `mtop` (Input)
An upper limit on the number of coefficients to be computed in the Laguerre expansion. Argument `mtop` must be a multiple of four.
Default: `mtop` = 1024

`IMSL_ERROR_EST`, *floa*`t *error_est` (Output)
Overall estimate of the pseudo error, `disc_error_est` +

`trunc_error_est` + `cond_error_est`. See the Description section for details.

`IMSL_DISCRETIZATION_ERROR_EST`, *float* `*disc_error_est` (Output)
Estimate of the pseudo discretization error.

`IMSL_TRUNCATION_ERROR_EST`, *float* `*trunc_error_est` (Output)
Estimate of the pseudo truncation error.

`IMSL_CONDITION_ERROR_EST`, *float* `*cond_error_est` (Output)
Estimate of the pseudo condition error on the basis of minimal noise levels in the function values.

`IMSL_DECAY_FUNCTION_COEFFICIENT`, *float* `*k` (Output)
The coefficient of the decay function. See the Description section for details.

`IMSL_DECAY_FUNCTION_BASE`, *float* `*r` (Output)
The base of the decay function. See the Description section for details.

`IMSL_LOG_LARGEST_COEFFICIENTS`, *float* `*log_largest_coefs` (Output)
The logarithm of the largest `coefficient` in the decay function. See the Description section for details.

`IMSL_LOG_SMALLEST_COEFFICIENTS`, *float* `*log_smallest_coefs` (Output)
The logarithm of the smallest nonzero `coefficient` in the decay function. See the Description section for details.

`IMSL_UNDER_OVERFLOW_INDICATORS`, *Imsl_laplace_flow* `**indicators` (Output)
The address of a pointer initialized by `imsl_f_inverse_laplace` to point to an array of length `n` containing the overflow/underflow indicators for the computed approximate inverse Laplace transform. For the *i*th point at which the transform is computed, indicators[i] signifies the following:

indicators [*i*]	Meaning
`IMSL_NORMAL_TERMINATION`	Normal termination.
`IMSL_TOO_LARGE`	The value of the inverse Laplace transform is too large to be representable. This component of the result is set to NaN.
`IMSL_TOO_SMALL`	The value of the inverse Laplace transform is found to be too small to be representable. This component of the result is set to 0.0.
`IMSL_TOO_LARGE_BEFORE_EXPANSION`	The value of the inverse Laplace transform is estimated to be too large, even before the series expansion, to be representable. This component of the result is set to NaN.

indicators [*i*]	Meaning
IMSL_TOO_SMALL_BEFORE_EXPANSON	The value of the inverse Laplace transform is estimated to be too small, even before the series expansion, to be representable. This component of the result is set to 0.0.

IMSL_FCN_W_DATA, *f_complex* fcn(*f_complex* z, *void* *data) ,*void* *data, (Input)
User supplied function for which the inverse Laplace transform will be computed, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function imsl_f_inverse_laplace computes the inverse Laplace transform of a complex-valued function. Recall that if *f* is a function that vanishes on the negative real axis, then the Laplace transform of *f* is defined by

$$L[f](s) = \int_0^\infty e^{-sx} f(x)\,dx$$

It is assumed that for some value of *s* the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on a modification of Weeks' method (see Weeks (1966)) due to Garbow et al. (1988). This method is suitable when *f* has continuous derivatives of all orders on [0, ∞). In particular, given a complex-valued function $F(s) = L[f](s)$, *f* can be expanded in a Laguerre series whose coefficients are determined by *F*. This is fully described in Garbow et al. (1988) and Lyness and Giunta (1986).

The algorithm attempts to return approximations $g(t)$ to $f(t)$ satisfying

$$\left|\frac{g(t) - f(t)}{e^{\sigma t}}\right| < \varepsilon$$

where ε = pseudo_accuracy and σ = sigma > sigma0. The expression on the left is called the *pseudo error*. An estimate of the pseudo error in available in error_est.

The first step in the method is to transform *F* to ϕ where

$$\phi(z) = \frac{b}{1-z} F\left(\frac{b}{1-z} - \frac{b}{2} + \sigma\right)$$

Then, if *f* is smooth, it is known that ϕ is analytic in the unit disc of the complex plane and hence has a Taylor series expansion

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s$$

which converges for all z whose absolute value is less than the radius of convergence R_c. This number is estimated in r, obtained through the optional argument IMSL_DECAY_FUNCTION_BASE. Using optional argument IMSL_DECAY_FUNCTION_COEFFICIENT, the smallest number K is estimated which satisfies

$$|a_s| < \frac{K}{R^s}$$

for all $R < R_c$.

The coefficients of the Taylor series for ϕ can be used to expand f in a Laguerre series

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt)$$

Examples

Example 1

This example computes the inverse Laplace transform of the function $(s-1)^{-2}$, and prints the computed approximation, true transform value, and difference at five points. The correct inverse transform is xe^x. From Abramowitz and Stegun (1964).

```
#include <imsl.h>
#include <math.h>

main()
{
        f_complex f(f_complex);
        int n = 5;
        float t[5];
        float true_inverse[5];
        float relative_diff[5];
        int i;
        float *inverse;

                        /* Initialize t and compute inverse */
        for (i=0; i<n; i++)
                t[i] = (float)i + 0.5;

        inverse = imsl_f_inverse_laplace(f, 1.5, n, t, 0);

                        /* Compute true inverse, relative difference */

        for (i=0; i<n; i++) {
                true_inverse[i] = t[i]*exp(t[i]);
                relative_diff[i] = fabs(inverse[i] - true_inverse[i])/
                                   true_inverse[i];
        }
```

```
        printf("\t   t\t\t  f_inv\t\t  true\t\t  diff\n");
        for (i=0; i<n; i++)
                printf ("\t%5.1f\t\t%7.3f\t\t%7.3f\t\t%6.1e\n", t[i],
                         inverse[i], true_inverse[i], relative_diff[i]);

}

f_complex f(f_complex s)
{
                        /* Return 1/(s-1)**2 */

        f_complex one = {1.0, 0.0};

        return (imsl_c_div(one,
                imsl_c_mul(imsl_c_sub(s, one), imsl_c_sub(s, one))));
}
```

Output

```
  t           f_inv          true          diff
 0.5          0.824         0.824        1.5e-05
 1.5          6.722         6.723        1.0e-05
 2.5         30.456        30.456        5.6e-07
 3.5        115.906       115.904        1.8e-05
 4.5        405.054       405.077        5.8e-05
```

Example 2

This example computes the inverse Laplace transform of the function $e^{-1/s}/s$, and prints the computed approximation, true transform value, and difference at five points. Additionally, the inverse is returned in user-supplied space, and a required accuracy for the inverse transform values is specified. The correct inverse transform is

$$J_0\left(2\sqrt{x}\right)$$

From Abramowitz and Stegun (1964).

```
#include <imsl.h>
#include <math.h>

main()
{
        f_complex f(f_complex);
        int n = 5;
        int i;
        float t[5];
        float true_inverse[5];
        float relative_diff[5];
        float inverse[5];
        Imsl_laplace_flow *indicators;

                                /* Initialize t and compute inverse */

        for (i=0; i<n; i++) t[i] = (float)i + 0.5;

        imsl_f_inverse_laplace(f, 0.0, n, t,
                IMSL_PSEUDO_ACCURACY, 1.0e-6,
```

```
                IMSL_UNDER_OVERFLOW_INDICATORS, &indicators,
                IMSL_RETURN_USER, inverse,
                0);
                                /* Compute true inverse, relative
                                   difference */

        for (i=0; i<n; i++) {
                true_inverse[i] = imsl_f_bessel_J0(2.0*sqrt(t[i]));
                relative_diff[i] = fabs((inverse[i] - true_inverse[i])/
                                   true_inverse[i]);
        }

                                /* Print results, noting if any results
                                   overflowed or underflowed */

        printf("\t   T\t\t  f_inv\t\t  true\t\t  diff\n");
        for (i=0; i<n; i++)
                if (indicators[i] == IMSL_NORMAL_TERMINATION)
                        printf ("\t%5.1f\t\t%7.3f\t\t%7.3f\t\t%6.1e\n",
                                t[i],
                               inverse[i], true_inverse[i],
                               relative_diff[i]);
                else
                        printf("Overflow or underflow noted.\n");
}

f_complex f(f_complex s)
{

                                /* Return (1/s)(exp(-1/s) */

        f_complex one = {1.0, 0.0};
        f_complex s_inverse;

        s_inverse = imsl_c_div(one, s);
        return (imsl_c_mul(s_inverse, imsl_c_exp(imsl_c_neg(s_inverse))));
}
```

Output

```
  T           f_inv          true          diff
 0.5          0.559          0.559        2.1e-07
 1.5         -0.023         -0.023        8.5e-06
 2.5         -0.310         -0.310        9.6e-08
 3.5         -0.401         -0.401        7.4e-08
 4.5         -0.370         -0.370        6.4e-07
```

Chapter 7: Nonlinear Equations

Routines

7.1 Zeros of a Polynomial

Real coefficients using Jenkins-Traub method .....zeros_poly 384

Complex coefficients using Jenkins-Traub method .....zeros_poly (complex) 386

7.2 Zeros of a Function

Real zeros of a real function..... zeros_fcn 388

7.3 Root of a System of Equations

Powell's hybrid method.....zeros_sys_eqn 393

Usage Notes

Zeros of a Polynomial

A polynomial function of degree n can be expressed as follows:

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where $a_n \neq 0$. The function `imsl_f_zeros_poly` finds zeros of a polynomial with real coefficients using the Jenkins-Traub method.

Zeros of a Function

The function `imsl_f_zeros_fcn` uses Müller's method to find the real zeros of a real-valued function.

Root of System of Equations

A system of equations can be stated as follows:

$$f_i(x) = 0, \text{ for } i = 1, 2, \ldots, n$$

where $x \in \mathbf{R}^n$, and $f_i : \mathbf{R}^n \rightarrow \mathbf{R}$.The function `imsl_f_zeros_sys_eqn` uses a modified hybrid method due to M.J.D. Powell to find the zero of a system of nonlinear equations.

zeros_poly

Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub, three-stage algorithm.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_f_zeros_poly` (*int* `ndeg`, *float* `coef[]`, ..., 0)

The type *d_complex* function is `imsl_d_zeros_poly`.

Required Arguments

int `ndeg` (Input)
Degree of the polynomial.

float `coef[]` (Input)
Array with `ndeg` + 1 components containing the coefficients of the polynomial in increasing order by degree. The polynomial is `coef`[n] z^n + `coef` [n − 1] z^{n-1} + … + `coef` [0], where n = `ndeg`.

Return Value

A pointer to the complex array of zeros of the polynomial. To release this space, use `free`. If no zeros are computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_f_zeros_poly` (*int* `ndeg`, *float* `coef[]`,
`IMSL_RETURN_USER`, *f_complex* `root[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *f_complex* `root[]` (Output)
Array with `ndeg` components containing the zeros of the polynomial.

Description

The function `imsl_f_zeros_poly` computes the n zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The function `imsl_f_zeros_poly` uses the Jenkins-Traub, three-stage algorithm (Jenkins and Traub 1970; Jenkins 1975). The zeros are computed one at a time for real zeros or two at a time for a complex conjugate pair. As the zeros are found, the real zero, or quadratic factor, is removed by polynomial deflation.

Examples

Example 1

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where z is a complex variable.

```
#include <imsl.h>

#define NDEG    3

main()
{
        f_complex       *zeros;
        static float    coeff[NDEG + 1] = {-2.0, 4.0, -3.0, 1.0};

        zeros = imsl_f_zeros_poly(NDEG, coeff, 0);

        imsl_c_write_matrix ("The complex zeros found are", 1, 3,
            zeros, 0);
}
```

Output

```
                    The complex zeros found are
                   1                         2                         3
(         1,         0)   (         1,         1)   (         1,        -1)
```

Example 2

The same problem is solved with the return option.

```
#include <imsl.h>

#define NDEG    3

main()
{
        f_complex       zeros[3];
        static float    coeff[NDEG + 1] = {-2.0, 4.0, -3.0, 1.0};

        imsl_f_zeros_poly(NDEG, coeff,
                          IMSL_RETURN_USER, zeros, 0);
```

```
        imsl_c_write_matrix ("The complex zeros found are", 1, 3,
            zeros, 0);
}
```

Output

```
                    The complex zeros found are
                   1                          2                          3
(         1,          0)   (          1,          1)   (          1,         -1)
```

Warning Errors

IMSL_ZERO_COEFF	The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.
IMSL_FEWER_ZEROS_FOUND	Fewer than ndeg zeros were found. The root vector will contain the value for machine infinity in the locations that do not contain zeros.

zeros_poly (complex)

Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub, three-stage algorithm.

Synopsis

#include <imsl.h>

f_complex *imsl_c_zeros_poly (*int* ndeg, *f_complex* coef[], ..., 0)

The type *d_complex* function is imsl_z_zeros_poly.

Required Arguments

int ndeg (Input)
Degree of the polynomial.

f_complex coef[] (Input)
Array with ndeg + 1 components containing the coefficients of the polynomial in increasing order by degree. The degree of the polynomial is

$$\texttt{coef}\,[n]\,z^{n} + \texttt{coef}\,[n-1]\,z^{n-1} + \ldots + \texttt{coef}\,[0]$$

where n = ndeg.

Return Value

A pointer to the complex array of zeros of the polynomial. To release this space, use free. If no zeros are computed, then NULL is returned.

Synopsis with Optional Arguments

#include <imsl.h>

f_complex *imsl_c_zeros_poly (*int* ndeg, *f_complex* coef[],
IMSL_RETURN_USER, *f_complex* root[],
0)

Optional Arguments

IMSL_RETURN_USER, *f_complex* root[] (Output)
Array with ndeg components containing the zeros of the polynomial.

Description

The function imsl_c_zeros_poly computes the n zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are complex and n is the degree of the polynomial.

The function imsl_c_zeros_poly uses the Jenkins-Traub, three-stage complex algorithm (Jenkins and Traub 1970, 1972). The zeros are computed one at a time in roughly increasing order of modulus. As each zero is found, the polynomial is deflated to one of lower degree.

Examples

Example 1

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - (3 + 6i) z^2 - (8 - 12i) z + 10$$

where z is a complex variable.

```
#include <imsl.h>

#define NDEG    3

main()
{
        f_complex       *zeros;
        f_complex coeff[NDEG + 1] = { {10.0, 0.0},
                                      {-8.0, 12.0},
                                      {-3.0, -6.0},
                                      { 1.0, 0.0} };

        zeros = imsl_c_zeros_poly(NDEG, coeff, 0);

        imsl_c_write_matrix ("The complex zeros found are", 1, 3,
               zeros, 0);
}
```

Output

```
                    The complex zeros found are
                   1                         2                         3
(         1,         1)  (         1,         2)  (         1,         3)
```

Example 2

The same problem is solved with the return option.

```
#include <imsl.h>

#define NDEG    3

main()
{
        f_complex       zeros[3];
        f_complex coeff[NDEG + 1] = { {10.0, 0.0},
                                      {-8.0, 12.0},
                                      {-3.0, -6.0},
                                      { 1.0, 0.0} };

        imsl_c_zeros_poly(NDEG, coeff, IMSL_RETURN_USER, zeros, 0);


        imsl_c_write_matrix ("The complex zeros found are", 1, 3,
            zeros, 0);
}
```

Output

```
                    The complex zeros found are
                   1                         2                         3
(         1,         1)  (         1,         2)  (         1,         3)
```

Warning Errors

IMSL_ZERO_COEFF	The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem.
IMSL_FEWER_ZEROS_FOUND	Fewer than ndeg zeros were found. The root vector will contain the value for machine infinity in the locations that do not contain zeros.

zeros_fcn

Finds the real zeros of a real function using Müller's method.

Synopsis

#include <imsl.h>

float *imsl_f_zeros_fcn (*float* fcn(), …, 0)

The type *double* function is imsl_d_zeros_fcn.

Required Arguments

float `fcn` (*float* `x`) (Input/Output)
User-supplied function to compute the value of the function of which the zeros will be found, where `x` is the point at which the function is evaluated.

Return Value

A pointer to the zeros *x* of the function. To release this space, use `free`. If no zeros can be found, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
float *imsl_f_zeros_fcn (float fcn(),
        IMSL_XGUESS, float xguess[],
        IMSL_NUM_ROOTS, int nroot,
        IMSL_ERR_ABS, float err_abs,
        IMSL_ERR_REL, float err_rel,
        IMSL_ETA, float eta,
        IMSL_EPS, float eps,
        IMSL_MAX_ITN, int max_itn,
        IMSL_RETURN_USER, float x[],
        IMSL_INFO, int **info,
        IMSL_INFO_USER, int info[],
        IMSL_FCN_W_DATA, float fcn (), void *data,
        0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `nroot` components containing the initial guesses for the zeros.
Default: `xguess` = 0

`IMSL_NUM_ROOTS`, *int* `nroot` (Input)
The number of zeros to be found by `imsl_f_zeros_fcn`.
Default: `nroot` = 1

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
First stopping criterion. A zero x_i is accepted if $|f(x_i)| <$ `err_abs`.
Default:

$$\text{err\_abs} = \sqrt{\varepsilon}$$

where e is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Second stopping criterion. A zero x_i is accepted if the relative change of two successive approximations to x_i is less than `err_rel`.
Default:

$$\text{err\_rel} = \sqrt{\varepsilon}$$

where ε is the machine precision

`IMSL_ETA`, *float* `eta` (Input)
Spread criteria for multiple zeros. If the zero x_i has been computed and $|x_i - x_j| <$ `eps`, where x_j is a previously computed zero, then the computation is restarted with a guess equal to $x_i +$ `eta`.
Default: `eta` = 0.01

`IMSL_EPS`, *float* `eps` (Input)
See `eta`.
Default:

$$\text{eps} = \sqrt{\varepsilon}$$

where ε is the machine precision

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
The maximum allowable number of iterations per zero.
Default: `max_itn` = 100

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `nroot` components containing the computed zeros.

`IMSL_INFO`, *int* `**info` (Output)
The address of a pointer `info` to an array of length `nroot` containing convergence information. On return, the necessary space is allocated by `imsl_f_zeros_fcn`. The value `info`[`j` − 1] is the number of iterations used in finding the *j*-th zero when convergence is achieved. If convergence is not obtained in `max_itn` iterations, `info`[`j` − 1] would be greater than `max_itn`.

`IMSL_INFO_USER`, *int* `info[]` (Output)
A user-allocated array with `nroot` components. On return, the value `info`[`j` − 1] is the number of iterations used in finding the *j*-th zero when convergence is achieved. If convergence is not obtained in `max_itn` iterations, `info`[`j` − 1] would be greater than `max_itn`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`) , *void* `*data` (Input)
User supplied function to compute the value of the function of which the zeros will be found, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_zeros_fcn` computes *n* real zeros of a real function *f*. Given a user-supplied function *f*(*x*) and an *n*-vector of initial guesses $x_1, x_2, \ldots, x_n$, the function uses Müller's method to locate *n* real zeros of *f*. The function has two convergence criteria: the first requires that

$$\left| f\left(x_i^{(m)}\right) \right|$$

be less than `err_abs`; the second requires that the relative change of any two successive approximations to an x_i be less than `err_rel`. Here,

$$x_i^{(m)}$$

is the m-th approximation to x_i. Let `err_abs` be denoted by ε_1 and `err_rel` be denoted by ε_2. The criteria may be stated mathematically as follows:

Criterion 1:

$$\left| f\left(x_i^{(m)}\right) \right| < \varepsilon_1$$

Criterion 2:

$$\left| \frac{x_i^{(m+1)} - x_i^{(m)}}{x_i^{(m)}} \right| < \varepsilon_2$$

"Convergence" is the satisfaction of either criterion.

Examples

Example 1

This example finds a real zero of the third-degree polynomial

$$f(x) = x^3 - 3x^2 + 3x - 1$$

```
#include <imsl.h>

float           fcn(float x);

main()
{
    float       *x;
                                /* Solve fcn(x)=0  for  x  */
    x = imsl_f_zeros_fcn (fcn, 0);
                                /* Print x */
    imsl_f_write_matrix ("x", 1, 1, x, 0);
}

float fcn(float x)
{
    return  x * x * x - 3.0 * x * x + 3.0 * x - 1.0;
}
```

Output

```
     x
         1
```

Example 2

This example finds three real zeros of the third-degree polynomial

$$f(x) = x^3 + 3x^2 - 4x - 6$$

with the three initial guesses (4.6, 0.0, –193.3).

```
#include <imsl.h>

float           fcn(float x);

main()
{
    float       xguess[ ] =  {4.6, 0.0, -193.3};
    int         nroot = 3;
    float       eps = 1.0e-5;
    float       err_abs = 1.0e-5;
    float       err_rel = 1.0e-5;
    float       eta = 1.0e-2;
    int         max_itn = 100;
    float       *x;
                                /* Solve fcn(x)=0  for  x  */
    x = imsl_f_zeros_fcn (fcn,
                          IMSL_XGUESS, xguess,
                          IMSL_ERR_REL, err_rel,
                          IMSL_ERR_ABS, err_abs,
                          IMSL_ETA, eta,
                          IMSL_EPS, eps,
                          IMSL_NUM_ROOTS, nroot,
                          IMSL_MAX_ITN, max_itn,
                          0);
                                /* Print x */
    imsl_f_write_matrix ("x", 1, 3, x, 0);
}

float fcn(float x)
{
    return  x * x * x + 3.0 * x * x - 4.0 * x - 6.0;
}
```

Output

```
                x
        1           2           3
    1.646      -1.000      -3.646
```

In the following plot, the initial guesses $x = 0.0$ and $x = 4.6$ are marked with hollow circles, and the solutions are marked with filled circles. The other initial guess $x = -193.3$ does not fit on this plot.

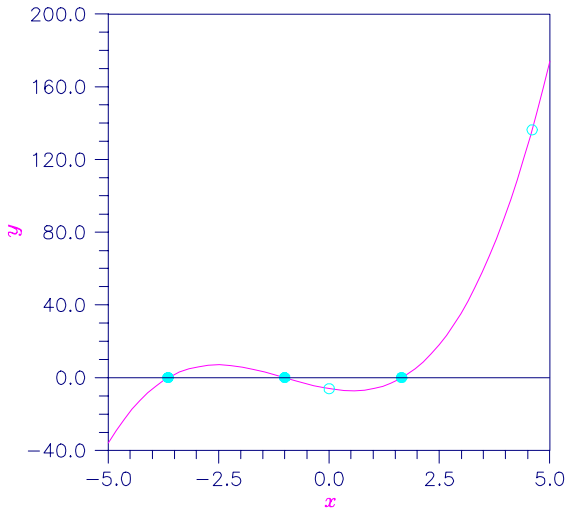


Figure 7-1 Plot of $x^3 + 3x^2 - 4x - 6$

Warning Errors

`IMSL_NO_CONVERGE_MAX_ITER`	Failure to converge within `max_itn` iterations for at least one of the `nroot` roots.

zeros_sys_eqn

Solves a system of n nonlinear equations $f(x) = 0$ using a modified Powell hybrid algorithm.

Synopsis

#include `<imsl.h>`

float `*imsl_f_zeros_sys_eqn` (*void* `fcn()`, *int* `n`, ..., 0)

The type *double* function is `imsl_d_zeros_sys_eqn`.

Required Arguments

void `fcn` (*int* `n`, *float* `x[]`, *float* `f[]`) (Input/Output)
User-supplied function to evaluate the system of equations to be solved, where `n` is the size of `x` and `f`, `x` is the point at which the functions are evaluated, and `f` contains the computed function values at the point `x`.

int `n` (Input)
The number of equations to be solved and the number of unknowns.

Return Value

A pointer to the vector *x* that is a solution of the system of equations. To release this space, use `free`. If no solution can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>
float *imsl_f_zeros_sys_eqn (void fcn(), int n,
        IMSL_XGUESS, float xguess[],
        IMSL_JACOBIAN, void jacobian(),
        IMSL_ERR_REL, float err_rel,
        IMSL_MAX_ITN, int max_itn,
        IMSL_RETURN_USER, float x[],
        IMSL_FNORM, float *fnorm,
        IMSL_FCN_W_DATA, void fcn ( ), void *data,
        IMSL_JACOBIAN_W_DATA, void jacobian(), void *data,
        0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with *n* components containing the initial estimate of the root.
Default: `xguess` = 0

`IMSL_JACOBIAN`, *void* `jacobian` (*int* `n`, *float* `x[]`, *float* `fjac[]`)
(Input/Output)
User-supplied function to evaluate the Jacobian, where `n` is the number of components in `x`, `x` is the point at which the Jacobian is evaluated, and `fjac` is the computed $n \times n$ Jacobian matrix at the point `x`. Note that each derivative $\partial f_i / \partial x_j$ should be returned in `fjac`[(`i`-1)*`n`+`j`-1].

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Stopping criterion. The root is accepted if the relative error between two successive approximations to this root is less than `err_rel`.
Default:

$$\text{err\_rel} = \sqrt{\varepsilon}$$

where ε is the machine precision

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
The maximum allowable number of iterations.
Default: `max_itn` = 200

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the best estimate of the root found by `f_zeros_sys_eqn`.

`IMSL_FNORM`, *float* `*fnorm` (Output)
Scalar with the value

$$f_1^2 + \ldots + f_n^2$$

at the point `x`.

`IMSL_FCN_W_DATA`, *void* `fcn` (*int* `n`, *float* `x[]`, *float* `f[]` , *void* `*data`) , *void* `*data` (Input)
User supplied function to evaluate the system of equations to be solved, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_JACOBIAN_W_DATA`, *void* `jacobian` (*int* `m`, *int* `n`, *float* `x[]`, *float* `fjac[]`, *int* `fjac_col_dim`, *void* `*data`), *void* `*data` (Input)
User supplied function to compute the Jacobian, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_zeros_sys_eqn` is based on the MINPACK subroutine HYBRDJ, which uses a modification of the hybrid algorithm due to M.J.D. Powell. This algorithm is a variation of Newton's method, which takes precautions to avoid undesirable large steps or increasing residuals. For further description, see Moré et al. (1980).

Examples

Example 1

The following 2×2 system of nonlinear equations

$$f_1(x) = x_1 + x_2 - 3$$
$$f_2(x) = x_1^2 + x_2^2 - 9$$

is solved.

```
#include <imsl.h>
#include <stdio.h>

#define N       2

void            fcn(int, float[], float[]);

void main()
{
    float       *x;

    x = imsl_f_zeros_sys_eqn(fcn, N, 0);
    imsl_f_write_matrix("The solution to the system is", 1, N, x, 0);
```

```
}


void fcn(int n, float x[], float f[])
{
    f[0] = x[0] + x[1] - 3.0;
    f[1] = x[0]*x[0] + x[1] * x[1] - 9.0;
}
```

Output

```
The solution to the system is
            1           2
            0           3
```

Example 2

The following 3 × 3 system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1 - 1} + (x_2 + x_3)^2 - 27$$
$$f_2(x) = e^{x_2 - 2} / x_1 + x_3^2 - 10$$
$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define N       3

void            fcn(int, float[], float[]);

void main()
{
    int         maxitn = 100;
    float       *x, err_rel = 0.0001, fnorm;
    float       xguess[N] = {4.0, 4.0, 4.0};

    x = imsl_f_zeros_sys_eqn(fcn, N,
                             IMSL_ERR_REL, err_rel,
                             IMSL_MAX_ITN, maxitn,
                             IMSL_XGUESS, xguess,
                             IMSL_FNORM, &fnorm,
                             0);
    imsl_f_write_matrix("The solution to the system is", 1, N, x, 0);
    printf("\nwith fnorm = %5.4f\n", fnorm);
}


void fcn(int n, float x[], float f[])
{
    f[0] = x[0] + exp(x[0] - 1.0) + (x[1] + x[2]) * (x[1] + x[2]) - 27.0;
    f[1] = exp(x[1] - 2.0) / x[0] + x[2] * x[2] - 10.0;
    f[2] = x[2] + sin(x[1] - 2.0) + x[1] * x[1] - 7.0;
}
```

Output

```
The solution to the system is
      1           2           3
      1           2           3

with fnorm = 0.0000
```

Warning Errors

IMSL_TOO_MANY_FCN_EVALS	The number of function evaluations has exceeded max_itn. A new initial guess may be tried.
IMSL_NO_BETTER_POINT	Argument err_rel is too small. No further improvement in the approximate solution is possible.
IMSL_NO_PROGRESS	The iteration has not made good progress. A new initial guess may be tried.

Appendix A: References

Abramowitz and Stegun

Abramowitz, Milton, and Irene A. Stegun (editors) (1964), *Handbook of Mathematical Functions with Formulas*, *Graphs*, *and Mathematical Tables*, National Bureau of Standards, Washington.

Ahrens and Dieter

Ahrens, J.H., and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing,* **12**, 223–246.

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.

Ashcraft

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse symmetric positive definite 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 (1979)

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.

Atkinson (1978)

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

Barnett

Barnett, A.R. (1981), An algorithm for regular and irregular Coulomb and Bessel functions of real order to machine accuracy, *Computer Physics Communication*, **21**, 297–314.

Barrett and Healy

Barrett, J.C., and M. J.R. Healy (1978), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **27**, 379–380.

Bays and Durham

Bays, Carter, and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59–64.

Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

Boisvert

Boisvert, Ronald (1984), A fourth order accurate fast direct method of the Helmholtz equation, *Elliptic Problem solvers II*, (edited by G. Birkhoff and A. Schoenstadt), Academic Press, Orlando, Florida, 35–44.

Bosten and Battiste

Bosten, Nancy E., and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156–157.

Brent

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.

Burgoyne

Burgoyne, F.D. (1963), Approximations to Kelvin functions, *Mathematics of Computation*, **83**, 295-298.

Carlson

Carlson, B.C. (1979), Computing elliptic integrals by duplication, *Numerische Mathematik*, **33**, 1–16.

Carlson and Notis

Carlson, B.C., and E.M. Notis (1981), Algorithms for incomplete elliptic integrals, *ACM Transactions on Mathematical Software*, **7**, 398–403.

Carlson and Foley

Carlson, R.E., and T.A. Foley (1991),The parameter R^2 in multiquadric interpolation, *Computer Mathematical Applications*, **21**, 29–42.

Cheng

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.

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.

Cooper

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190–192.

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.

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.

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., J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK User's Guide*, SIAM, Philadelphia.

Draper and Smith

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, 2nd. ed., John Wiley & Sons, New York.

DuCroz et al.

Du Croz, Jeremy, P. Mayes, and G. 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 et al.

Duff, I. S., A. M. Erisman, and J. K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

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.

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.

Farebrother and Berry

Farebrother, R.W., and G. Berry (1974), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **23**, 477.

Fisher

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *Annals of Eugenics*, **7**, 179– 188.

Fishman and Moore

Fishman, George S. and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$, *Journal of the American Statistical Association*, **77**, 129–136.

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.

Franke

Franke, R. (1982), Scattered data interpolation: Tests of some methods, *Mathematics of Computation*, **38**, 181–200.

Garbow et al.

Garbow, B.S., J.M. Boyle, K.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 on Mathematical Software*, **14**, 163–170.

Gautschi

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251–270.

Gautschi, Walter (1969), Complex error function, *Communications of the ACM*, **12**, 635. Gautschi, Walter (1970), Efficient computation of the complex error function, *SIAM Journal on Mathematical Analysis*, **7**, 187–198.

Gear

Gear, C.W. (1971), Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, New Jersey.

Gentleman

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448–454.

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 and Murray

Gill, Philip E., and Walter Murray (1976), *Minimization subject to bounds on the variables*, NPL Report NAC 92, National Physical Laboratory, England.

Gill et al.

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, G.H., and C.F. Van Loan (1989), *Matrix Computations*, Second Edition, The Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, 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 Redfish

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.

Hardy

Hardy, R.L. (1971), Multiquadric equations of topography and other irregular surfaces, *Journal of Geophysical Research*, **76**, 1905–1915.

Hart et al.

Hart, John F., E.W. Cheney, Charles L. Lawson, Hans J.Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, Jr., and Christoph Witzgall (1968), *Computer Approximations*, John Wiley & Sons, New York.

Healy

Healy, M.J.R. (1968), Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **17**, 195–197.

Herraman

Herraman, C. (1968), Sums of squares and products matrix, *Applied Statistics*, **17**, 289–292.

Higham

Higham, Nicholas J. (1988), FORTRAN Codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, *ACM Transactions on Mathematical Software*, **14**, 381-396.

Hill

Hill, G.W. (1970), Student's *t*-distribution, *Communications of the ACM*, **13**, 617–619.

Hindmarsh

Hindmarsh, A.C. (1974), *GEAR: Ordinary Differential Equation System Solver*, Lawrence Livermore National Laboratory Report UCID-30001, Revision 3, Lawrence Livermore National Laboratory, Livermore, Calif.

Hinkley

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67–69.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

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.

Irvine et al.

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129–151.

Jackson et al.

Jackson, K.R., W.H. Enright, and T.E. Hull (1978), A theoretical criterion for comparing Runge-Kutta formulas, *SIAM Journal of Numerical Analysis*, **15**, 618–641.

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, *Numerishe Mathematik*, **14**, 252–263.

Jenkins, M.A., and J.F. Traub (1972), Zeros of a complex polynomial, *Communications of the ACM*, **15**, 97– 99.

Jöhnk

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufalls-zahlen, *Metrika*, **8**, 5–15.

Kendall and Stuart

Kendall, Maurice G., and Alan Stuart (1973), *The Advanced Theory of Statistics*, Volume II, *Inference and Relationship*, Third Edition, Charles Griffin & Company, London, Chapter 30.

Kennedy and Gentle

Kennedy, William J., Jr., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kernighan and Richtie

Kernighan, Brian W., and Richtie, Dennis M. 1988, "The C Programming Language" Second Edition, **241**.

Kinnucan and Kuki

Kinnucan, P., and Kuki, H., (1968), *A single precision inverse error function subroutine*, Computation Center, University of Chicago.

Knuth

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume II: *Seminumerical Algorithms*, 2nd. ed., Addison-Wesley, Reading, Mass.

Learmonth and Lewis

Learmonth, G.P., and P.A.W. Lewis (1973), *Naval Postgraduate School Random Number Generator Package LLRANDOM, NPS55LW73061A*, Naval Postgraduate School, Monterey, California.

Lehmann

Lehmann, E.L. (1975), *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, San Francisco.

Levenberg

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

Leavenworth

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

Lentini and 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.

Lewis et al.

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/ 360, *IBM Systems Journal*, **8**, 136–146.

Liepman

Liepman, David S. (1964), Mathematical constants, in *Handbook of Mathematical Functions*, Dover Publications, New York.

Liu

Liu, J.W.H. (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, 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.

Lyness and Giunta

Lyness, J.N. and G. Giunta (1986), A modification of the Weeks Method for numerical inversion of the Laplace transform, *Mathematics of Computation*, **47**, 313–322.

Madsen and Sincovec

Madsen, N.K., and R.F. Sincovec (1979), Algorithm 540: PDECOL, General collocation software for partial differential equations, *ACM Transactions on Mathematical Software*, **5**, #3, 326–351.

Maindonald

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

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.H. Wilkinson (1971), Reduction of the Symmetric Eigenproblem $\mathbf{A}x = \lambda\mathbf{B}x$ and Related Problems to Standard Form, *Volume II, Linear Algebra Handbook*, Springer, New York.

Martin, R.S., and J.H. Wilkinson (1971), The Modified LR Algorithm for Complex Hessenberg Matrices, *Handbook, Volume II, Linear Algebra*, Springer, New York.

Mayle

Mayle, Jan, (1993), Fixed Income Securities Formulas for Price, Yield, and Accrued Interest, *SIA Standard Securities Calculation Methods*, Volume I, Third Edition, pages 17-35.

Michelli

Micchelli, C.A. (1986), Interpolation of scattered data: Distance matrices and conditionally positive definite functions, *Constructive Approximation*, **2**, 11–22.

Michelli 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.

Moré et al.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74, Argonne, Illinois.

Müller

Müller, 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.

Neter and Wasserman

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Illinois.

Neter et al.

Neter, John, William Wasserman, and Michael H. Kutner (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Illinois.

Østerby and Zlatev

Østerby, Ole, and Zahari Zlatev (1982), Direct Methods for Sparse Matrices, *Lecture Notes in Computer Science*, **157**, Springer-Verlag, New York.

Owen

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley Publishing Company, Reading, Mass.

Owen, D.B. (1965), A special case of the bivariate non-central *t* distribution, *Biometrika*, **52**, 437–446.

Parlett

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. Überhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

Powell

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, *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. (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 optimizations calculation*s, DAMTP Report NA17, University of Cambridge, England.

Powell, M.J.D. (1989), *TOLMIN: A fortran package for linearly constrained optimizations calculations*, DAMTP Report NA2, University of Cambridge, England.

Powell, M.J.D. (1983), *ZQPCVX a FORTRAN subroutine for convex quadratic programming*, DAMTP Report 1983/NA17, University of Cambridge, Cambridge, England.

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 minimum residual algorithm for solving nonsymmetric linear systems, *SIAM Journal of Scientific and Statistical Computing*, **7**, 856-869.

Sallas and Lionti

Sallas, William M., and Abby M. Lionti (1988), Some useful computing formulas for the nonfull rank linear model with linear equality restrictions, IMSL Technical Report 8805, IMSL, Houston.

Savage

Savage, I. Richard (1956), Contributions to the theory of rank order statistics—the two-sample case, *Annals of Mathematical Statistics*, **27**, 590–615.

Schmeiser

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, in *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154–160.

Schmeiser and Babu

Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917–926.

Schmeiser and Kachitvichyanukul

Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81–4, School of Industrial Engineering, Purdue University, West Lafayette, Indiana.

Schmeiser and Lal

Schmeiser, Bruce W., and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679–682.

Seidler and Carmichael

Seidler, Lee J. and Carmichael, D.R., (editors) (1980), *Accountants' Handbook*, Volume I, Sixth Edition, The Ronald Press Company, New York.

Shampine

Shampine, L.F. (1975), Discrete least squares polynomial fits, *Communications of the ACM*, **18**, 179–180.

Shampine and Gear

Shampine, L.F. and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1–17.

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, T.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185–187.

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.

Smith

Smith, P.W. (1990), On knots and nodes for spline interpolation, *Algorithms for Approximation II*, J.C. Mason and M.G. Cox, Eds., Chapman and Hall, New York.

Spellucci, Peter

Spellucci, P. (1998), An SQP method for general nonlinear programs using only equality constrained subproblems, *Math. Prog.,* **82**, 413-448, Physica Verlag, Heidelberg, Germany

Spellucci, P. (1998), A new technique for inconsistent problems in the SQP method. *Math. Meth. of Oper. Res.,***47**, 355-500, Physica Verlag, Heidelberg, Germany.

Stewart

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Strecok

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144–158.

Stroud and Secrest

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

Temme

Temme, N.M (1975), On the numerical evaluation of the modified Bessel Function of the third kind, *Journal of Computational Physics*, **19**, 324–337.

Tezuka

Tezuka, S. (1995), *Uniform Random Numbers: Theory and Practice*. Academic Publishers, Boston.

Thompson and Barnett

Thompson, I.J. and A.R. Barnett (1987), Modified Bessel functions $I_\nu(z)$ and $K_\nu(z)$ of real order and complex argument, *Computer Physics Communication*, **47**, 245–257.

Tukey

Tukey, John W. (1962), The future of data analysis, *Annals of Mathematical Statistics,* **33**, 1–67.

Velleman and Hoaglin

Velleman, Paul F., and David C. Hoaglin (1981), *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.

Walker

Walker, H.F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM Journal of Scientific and Statistical Computing*, **9**, 152-163.

Watkins

Watkins, David S., L. Elsner (1991), Convergence of algorithm of decomposition type for the eigenvalue problem, *Linear Algebra Applications*, **143**, pp. 29–47.

Weeks

Weeks, W.T. (1966), Numerical inversion of Laplace transforms using Laguerre functions, *J. ACM*, **13**, 419–429.

Appendix B: Alphabetical Summary of Routines

Function	Purpose Statement	Page
`accr_interest_maturity`	Evaluates the accrued interest for a security that pays at maturity.	580
`accr_interest_periodic`	Evaluates the accrued interest for a security that pays periodic interest.	582
`airy_Ai`	Evaluates the Airy function.	509
`airy_Ai_derivative`	Evaluates the derivative of the Airy function	511
`airy_Bi`	Evaluates the Airy function of the second kind.	510
`airy_Bi_derivative`	Evaluates the derivative of the Airy function of the second kind.	512
`bessel_exp_I0`	Evaluates the exponentially scale modified Bessel function of the first kind of order zero.	489
`bessel_exp_I1`	Evaluates the exponentially scaled modified Bessel function of the first kind of order one.	491
`bessel_exp_K0`	Evaluates the exponentially scaled modified Bessel function of the third kind of order zero.	495
`bessel_exp_K1`	Evaluates the exponentially scaled modified Bessel function of the third kind of order one.	497
`bessel_I0`	Evaluates the real modified Bessel function of the first kind of order zero $I0(x)$.	487
`bessel_I1`	Evaluates the real modified Bessel function of the first kind of order one $I1(x)$.	490
`bessel_Ix`	Evaluates a sequence of modified Bessel functions of the first kind with real order and complex arguments.	492
`bessel_J0`	Evaluates the real Bessel function of the first kind of order zero $J0(x)$.	478
`bessel_J1`	Evaluates the real Bessel function of the first kind of order one $J1(x)$.	480
`bessel_Jx`	Evaluates a sequence of Bessel functions of the first kind with real order and complex arguments.	481
`bessel_K0`	Evaluates the real modified Bessel function of the third kind of order zero $K0(x)$.	493

Function	Purpose Statement	Page
`bessel_K1`	Evaluates the real modified Bessel function of the third kind of order one $K1(x)$.	496
`bessel_Kx`	Evaluates a sequence of modified Bessel functions of the third kind with real order and complex arguments.	499
`bessel_Y0`	Evaluates the real Bessel function of the second kind of order zero $Y0(x)$.	482
`bessel_Y1`	Evaluates the real Bessel function of the second kind of order one $Y1(x)$.	484
`bessel_Yx`	Evaluates a sequence of Bessel functions of the second kind with real order and complex arguments.	485
`beta`	Evaluates the real beta function $\beta(x, y)$.	469
`beta_cdf`	Evaluates the beta probability distribution function	540
`beta_incomplete`	Evaluates the real incomplete beta function $Ix = \beta x(a, b)/\beta(a, b)$.	472
`beta_inverse_cdf`	Evaluates the inverse of the beta distribution function.	542
`binomial_cdf`	Evaluates the binomial distribution function.	536
`bivariate_normal_cdf`	Evaluates the bivariate normal distribution function.	543
`bond_equivalent_yield`	Evaluates the bond-equivalent for a Treasury yield.	584
`bounded_least_squares`	Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.	439
`bvp_finite_difference`	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.	321
`chi_squared_cdf`	Evaluates the chi-squared distribution function	524
`chi_squared_inverse_cdf`	Evaluates the inverse of the chi-squared distribution function.	526
`chi_squared_test`	Performs a chi-squared goodness-of-fit test	638
`constant`	Returns the value of various mathematical and physical constants.	719
`constrained_nlp`	Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method.	447
`convexity`	Evaluates the convexity for a security.	586
`convolution (complex)`	Computes the convolution, and optionally, the correlation of two complex vectors.	370
`convolution`	Computes the convolution, and optionally, the correlation of two real vectors.	363

Function	Purpose Statement	Page
coupon_days	Evaluates the number of days in the coupon period that contains the settlement date.	588
coupon_number	Evaluates the number of coupons payable between the settlement date and maturity date.	589
covariances	Computes the sample variance-covariance or correlation matrix.	646
ctime	Returns the number of CPU seconds used.	709
cub_spline_integral	Computes the integral of a cubic spline.	160
cub_spline_interp_e_cnd	Computes a cubic spline interpolant, specifying various endpoint conditions.	145
cub_spline_interp_shape	Computes a shape-preserving cubic spline.	152
cub_spline_smooth	Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.	205
cub_spline_value	Computes the value of a cubic spline or the value of one of its derivatives.	157
cumalative_interest	Evaluates the cumulative interest paid between two periods.	545
cumalative_principal	Evaluates the cumulative principal paid between two periods.	546
date_to_days	Evaluates the number of days from January 1, 1900, to the given date.	709
days_before_settlement	Evaluates the number of days from the beginning of the coupon period to the settlement date.	591
days_to_date	Gives the date corresponding to the number of days since January 1, 1900.	711
days_to_next_coupon	Evaluates the number of days from settlement date to the next coupon date.	592
depreciation_amordegrc	Evaluates the depreciation for each accounting period. Similar to `depreciation_amorlinc`.	594
depreciation_amorlinc	Evaluates the depreciation for each accounting period. Similar to `depreciation_amordegrc`.	596
depreciation_db	Evaluates the depreciation of an asset for a specified period using the fixed-declining balance method.	548
depreciation_ddb	Evaluates the depreciation of an asset for a specified period using the double-declining method.	550
depreciation_sln	Evaluates the straight line depreciation of an asset for one period.	551
depreciation_syd	Evaluates the sum-of-years digits depreciation of an asset for a specified period.	553

Function	Purpose Statement	Page
depreciation_vdb	Evaluates the depreciation of an asset for any given period, including partial periods, using the double-declining balance method.	554
discount_price	Evaluates the price per $100 face value of a discounted security.	597
discount_rate	Evaluates the discount rate for a security.	599
discount_yield	Evaluates the annual yield for a discounted security.	601
dollar_decimal	Converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number.	556
dollar_fraction	Converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fraction.	557
duration	Evaluates the annual duration of a security with periodic interest payment.	603
effective_rate	Evaluates the effective annual interest rate.	558
eig_gen (complex)	Computes the eigenexpansion of a complex matrix *A*.	120
eig_gen	Computes the eigenexpansion of a real matrix *A*	118
eig_herm (complex)	Computes the eigenexpansion of a complex Hermitian matrix *A*.	126
eig_sym	Computes the eigenexpansion of a real symmetric matrix *A*.	123
eig_symgen	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. *A* and *B* are real and symmetric. *B* is positive definite.	129
elliptic_integral_E	Evaluates the complete elliptic integral of the second kind $E(x)$.	501
elliptic_integral_K	Evaluates the complete elliptic integral of the kind $K(x)$.	500
elliptic_integral_RC	Evaluates an elementary integral from which inverse circular functions, logarithms, and inverse hyperbolic functions can be computed.	506
elliptic_integral_RD	Evaluates Carlson's elliptic integral of the second kind $R\mathrm{D}(x, y, z)$.	504
elliptic_integral_RF	Evaluates Carlson's elliptic integral of the first kind $R\mathrm{F}(x, y, z)$.	502
elliptic_integral_RJ	Evaluates Carlson's elliptic integral of the third kind $R\mathrm{J}(x, y, z, \rho)$.	505
erf	Evaluates the real error function $\mathrm{erf}(x)$.	460
erf_inverse	Evaluates the real inverse error function $\mathrm{erf}\text{-}1(x)$.	465
erfc	Evaluates the real complementary error function $\mathrm{erfc}(x)$.	461

Function	Purpose Statement	Page
erfc_inverse	Evaluates the real inverse complementary error function erfc-1(x).	467
erfce	Evaluates the exponentially scaled complementary error function.	463
erfe	Evaluates a scaled function related to erfc(z)	464
error_code	Gets the code corresponding to the error message from the last function called.	718
error_options	Sets various error handling options.	712
F_cdf	Evaluates the F distribution function.	528
F_inverse_cdf	Evaluates the inverse of the F distribution function.	530
fast_poisson_2d	Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.	332
faure_next_point	Evaluates a shuffled Faure sequence	687
fcn_derivative	Computes the first, second or third derivative of a user-supplied function.	286
fft_2d_complex	Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array.	359
fft_complex	Computes the complex discrete Fourier transform of a complex sequence.	346
fft_complex_init	Computes the parameters for `imsl_c_fft_complex`.	349
fft_cosine	Computes the discrete Fourier cosine transformation of an even sequence.	351
fft_cosine_init	Computes the parameters needed for `imsl_f_fft_cosine`.	353
fft_real	Computes the real discrete Fourier transform of a real sequence.	341
fft_real_init	Computes the parameters for `imsl_f_fft_real`	345
fft_sine	Computes the discrete Fourier sine transformation of an odd sequence.	355
fft_sine_init	Computes the parameters needed for `imsl_f_fft_sine`.	357
fresnel_integral_C	Evaluates the cosine Fresnel integral.	507
fresnel_integral_S	Evaluates the sine Fresnel integral.	508
future_value	Evaluates the future value of an investment.	559
future_value_schedule	Evaluates the future value of an initial principal after applying a series of compound interest rates.	561
gamma	Evaluates the real gamma function $\Gamma(x)$.	473
gamma_cdf	Evaluates the gamma distribution function	534

Function	Purpose Statement	Page
`gamma_incomplete`	Evaluates the incomplete gamma function $\gamma(a, x)$.	476
`gauss_quad_rule`	Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.	282
`geneig (complex)`	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B complex.	135
`geneig`	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B real.	132
`generate_test_band (complex)`	Generates test matrices of class $Ec(n, c)$.	784
`generate_test_band`	Generates test matrices of class $E(n, c)$.	782
`generate_test_coordinate (complex)`	Generates test matrices of class $D(n, c)$ and $E(n, c)$.	791
`generate_test_coordinate`	Generates test matrices of class $D(n, c)$ and $E(n, c)$.	786
`hypergeometric_cdf`	Evaluates the hypergeometric distribution function.	537
`int_fcn`	Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.	241
`int_fcn_2d`	Computes a two-dimensional iterated integral	272
`int_fcn_alg_log`	Integrates a function with algebraic-logarithmic singularities.	249
`int_fcn_cauchy`	Computes integrals of the form $\int_a^b \frac{f(x)}{x-c} dx$ in the Cauchy principal value sense.	265
`int_fcn_fourier`	Computes a Fourier sine or cosine transform.	261
`int_fcn_hyper_rect`	Integrates a function on a hyper-rectangle.	276
`int_fcn_inf`	Integrates a function over an infinite or semi-infinite interval.	253
`int_fcn_qmc`	Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.	279
`int_fcn_sing`	Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules.	237
`int_fcn_sing_pts`	Integrates a function with singularity points given	245
`int_fcn_smooth`	Integrates a smooth function using a nonadaptive rule.	268
`int_fcn_trig`	Integrates a function containing a sine or a cosine factor.	257
`interest_payment`	Evaluates the interest payment for a given period for an investment.	562
`interest_rate_annuity`	Evaluates the interest rate per period for an annuity.	563

Function	Purpose Statement	Page
`interest_rate_security`	Evaluates the interest rate for a fully invested security.	605
`internal_rate_of_return`	Evaluates the internal rate of return for a schedule of cash flows.	565
`internal_rate_schedule`	Evaluates the internal rate of return for a schedule of cash flows that is not necessarily periodic.	567
`inverse_laplace`	Computes the inverse Laplace transform of a complex function.	376
`kelvin_bei0`	Evaluates the Kelvin function of the first kind, bei, of order zero.	514
`kelvin_bei0_derivative`	Evaluates the derivative of the Kelvin function of the first kind, bei, of order zero.	518
`kelvin_ber0`	Evaluates the Kelvin function of the first kind, ber, of order zero.	513
`kelvin_ber0_derivative`	Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero.	517
`kelvin_kei0`	Evaluates the Kelvin function of the second kind, kei, of order zero.	516
`kelvin_kei0_derivative`	Evaluates the derivative of the Kelvin function of the second kind, kei, of order zero.	520
`kelvin_ker0`	Evaluates the Kelvin function of the second kind, der, of order zero.	515
`kelvin_ker0_derivative`	Evaluates the derivative of the Kelvin function of the second kind, ker, of order zero.	519
`lin_least_squares_gen`	Solves a linear least-squares problem $Ax = b$.	84
`lin_lsq_lin_constraints`	Solves a linear least squares problem with linear constraints.	92
`lin_prog`	Solves a linear programming problem using the revised simplex algorithm.	425
`lin_sol_def_cg`	Solves a real symmetric definite linear system using a conjugate gradient method.	78
`lin_sol_gen (complex)`	Solves a complex general system of linear equations $Ax = b$.	11
`lin_sol_gen`	Solves a real general system of linear equations $Ax = b$.	4
`lin_sol_gen_band (complex)`	Solves a complex general system of linear equations $Ax = b$.	31
`lin_sol_gen_band`	Solves a real geeral band system of linear equations $Ax=b$.	26
`lin_sol_gen_coordinate (complex)`	Solves a system of linear equations $Ax = b$, with sparse complex coefficient matrix A.	54
`lin_sol_gen_coordinate`	Solves a sparse system of linear equations $Ax = b$.	44

Function	Purpose Statement	Page
`lin_sol_gen_min_residual`	Solves a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method.	73
`lin_sol_nonnegdef`	Solves a real symmetric nonnegative definite system of linear equations $Ax = b$.	107
`lin_sol_posdef (complex)`	Solves a complex Hermitian positive definite system of linear equations $Ax = b$.	22
`lin_sol_posdef`	Solves a real symmetric positive definite system of linear equations $Ax = b$.	17
`lin_sol_posdef_band (complex)`	Solves a complex Hermitian positive definite system of linear equations $Ax = b$ in band symmetric storage mode.	39
`lin_sol_posdef_band`	Solves a real symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode.	35
`lin_sol_posdef_coordinate (complex)`	Solves a sparse Hermitian positive definite system of linear equations $Ax = b$.	68
`lin_sol_posdef_coordinate`	Solves a sparse real symmetric positive definite system of linear equations $Ax = b$.	62
`lin_svd_gen (complex)`	Computes the SVD, $A = USV$H, of a complex rectangular matrix A.	102
`lin_svd_gen`	Computes the SVD, $A = USV$T, of a real rectangular matrix A.	96
`log_beta`	Evaluates the logarithm of the real beta function ln $\beta(x, y)$.	471
`log_gamma`	Evaluates the logarithm of the absolute value of the gamma function log $\lvert\Gamma(x)\rvert$.	475
`machine (float)`	Returns information describing the computer's floating-point arithmetic.	725
`machine (integer)`	Returns integer information describing the computer's arithmetic.	723
`mat_add_band (complex)`	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.	764
`mat_add_band`	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.	760
`mat_add_coordinate (complex)`	Performs element-wise addition on two complex matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.	771
`mat_add_coordinate`	Performs element-wise addition of two real matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.	768
`mat_mul_rect (complex)`	Computes the transpose of a matrix, the conjugate-transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product.	738

Function	Purpose Statement	Page
mat_mul_rect	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product.	735
mat_mul_rect_band (complex)	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices of complex type and stored in band form.	746
mat_mul_rect_band	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices stored in band form.	742
mat_mul_rect_coordinate (complex)	Computes the transpose of a matrix, a matrix-vector product or a matrix-matrix product, all matrices stored in sparse coordinate form.	755
mat_mul_rect_coordinate	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices stored in sparse coordinate form.	751
matrix_norm	Computes various norms of a rectangular matrix.	775
matrix_norm_band	Computes various norms of a matrix stored in band storage mode.	777
matrix_norm_coordinate	Computes various norms of a matrix stored in coordinate format.	779
min_con_gen_lin	Minimizes a general objective function subject to linear equality/inequality constraints.	433
min_uncon	Finds the minimum point of a smooth function *f(x)* of a single variable using only function evaluations.	401
min_uncon_deriv	Finds the minimum point of a smooth function *f(x)* of a single variable using both function and first derivative evaluations.	405
min_uncon_multivar	Minimizes a function *f(x)* of *n* variables using a quasi-Newton method.	409
modified_duration	Evaluates the modified Macauley duration of a security.	607
modified_internal_rate	Evaluates the modified internal rate of return for a series of periodic cash flows.	569
net_present_value	Evaluates the net present value of an investment based on a series of periodic.	570
next_coupon_date	Evaluates the next coupon date after the settlement date.	608
nominal_rate	Evaluates the nominal annual interest rate.	571
nonlin_least_squares	Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm.	416
normal_cdf	Evaluates the standard normal (Gaussian) distribution function.	521
normal_inverse_cdf	Evaluates the inverse of the standard normal (Gaussian) distribution function.	523

Function	Purpose Statement	Page
`number_of_periods`	Evaluates the number of periods for an investment based on periodic and constant payment and a constant interest rate.	573
`ode_adams_gear`	Solves a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods.	297
`ode_runge_kutta`	Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.	291
`output_file`	Sets the output file or the error message output file.	704
`page`	Sets or retrieve the page width or length.	697
`payment`	Evaluates the periodic payment for an investment.	574
`pde_method_of_lines`	Solves a system of partial differential equations of the form $ut + f(x, t, u, ux, uxx)$ using the method of lines.	304
`poisson_cdf`	Evaluates the Poisson distribution function.	539
`poly_regression`	Performs a polynomial least-squares regression.	660
`present_value`	Evaluates the present value of an investment.	576
`present_value_schedule`	Evaluates the present value for a schedule of cash flows that is not necessarily periodic.	577
`previous_coupon_date`	Evaluates the previous coupon date before the settlement date.	610
`price`	Evaluates the price per $100 face value of a security that pays periodic interest.	612
`price_maturity`	Evaluates the price per $100 face value of a security that pays interest at maturity.	614
`principal_payment`	Evaluates the payment on the principal for a given period.	579
`quadratic_prog`	Solves a quadratic programming problem subject to linear equality or inequality constraints.	429
`radial_evaluate`	Evaluates a radial basis fit.	231
`radial_scattered_fit`	Computes an approximation to scattered data in R^n for $n \geq 2$ using radial basis functions.	225
`random_beta`	Generates pseudorandom numbers from a beta distribution.	684
`random_exponential`	Generates pseudorandom numbers from a standard exponential distribution.	685
`random_gamma`	Generates pseudorandom numbers from a standard gamma distribution.	682
`random_normal`	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.	679
`random_option`	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.	676

Function	Purpose Statement	Page
random_poisson	Generates pseudorandom numbers from a Poisson distribution.	680
random_seed_get	Retrieves the current value of the seed used in the IMSL random number generators.	674
random_seed_set	Initializes a random seed for use in the IMSL random number generators.	675
random_uniform	Generates pseudorandom numbers from a uniform (0, 1) distribution.	677
ranks	Computes the ranks, normal scores, or exponential scores for a vector of observations.	667
received_maturity	Evaluates the amount received for a fully invested security.	616
regression	Fits a multiple linear regression model using least squares.	651
scattered_2d_interp	Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.	220
simple_statistics	Computes basic univariate statistics.	629
smooth_1d_data	Smooth one-dimensional data by error detection	216
sort (integer)	Sorts an integer vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned.	730
sort	Sorts a vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned.	728
spline_2d_integral	Evaluates the integral of a tensor-product spline on a rectangular domain.	186
spline_2d_interp	Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data.	171
spline_2d_least_squares	Computes a two-dimensional, tensor-product spline approximant using least squares.	199
spline_2d_value	Computes the value of a tensor-product spline or the value of one of its partial derivatives.	182
spline_integral	Computes the integral of a spline.	180
spline_interp	Computes a spline interpolant.	161
spline_knots	Computes the knots for a spline interpolant.	167
spline_least_squares	Computes a least-squares spline approximation.	193
spline_lsq_constrained	Computes a least-squares constrained spline approximation.	209
spline_value	Computes the value of a spline or the value of one of its derivatives.	177
t_cdf	Evaluates the Student's *t* distribution function.	531

Function	Purpose Statement	Page
`t_inverse_cdf`	Evaluates the inverse of the Student's *t* distribution function.	533
`table_oneway`	Tallies observations into a one-way frequency table.	634
`treasury_bill_price`	Computes the price per $100 face value for a Treasury bill.	618
`treasury_bill_yield`	Computes the yield for a Treasury bill.	619
`user_fcn_least_squares`	Computes a least-squares fit using user-supplied functions.	189
`vector_norm`	Computes various norms of a vector or the difference of two vectors.	733
`version`	Returns integer information describing the version of the library, license number, operating system, and compiler.	708
`write_matrix`	Prints a rectangular matrix (or vector) stored in contiguous memory locations.	691
`write_options`	Sets or retrieve an option for printing a matrix.	698
`year_fraction`	Evaluates the year fraction that represents the number of whole days between two dates.	621
`yield_maturity`	Evaluates the annual yield of a security that pays interest at maturity.	622
`yield_periodic`	Evaluates the yield of a security that pays periodic interest.	624
`zeros_fcn`	Finds the real zeros of a real function using Müller's method.	388
`zeros_poly (complex)`	Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.	386
`zeros_poly`	Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.	384
`zeros_sys_eqn`	Solves a system of n nonlinear equations $f(x) = 0$ using a modified Powell hybrid algorithm.	393

Index

A

Adams-Gear method 297
Airy functions 509, 510, 511, 512
algebraic-logarithmic singularities 249
ANSI C ix
approximation 225
arithmetic 800

B

backward differentiation formulas 300
band matrices 760, 764
band storage mode 760, 764, 777
Bauer and Fike theorem 116
Bessel functions 478, 480, 481, 482, 484, 485, 487, 489, 490, 491, 492, 493, 495, 496, 497, 499
beta distributions 684
beta functions 469, 471, 472, 540, 542
binomial functions 536
bivariate functions 543
Blom scores 667
bond functions 580, 582, 584, 586, 588, 589, 591, 592, 594, 596, 597, 599, 601, 603, 605, 607, 608, 610, 612, 614, 616, 618, 619, 621, 622, 624
boundary conditions 321, 2

C

Cauchy principal 265
chi-squared functions 524, 526
chi-squared goodness-of-fit test 638
Cholesky factorization 17, 22, 35, 39, 107, 130
column pivoting 87
complex arithmetic xxiv, 800
complex general band system 31
complex Hermitian positive definite system 39
computer's arithmetic 723
computer's floating-point arithmetic 725
condition numbers 116
conjugate gradient method 78
constrained quadratic programming 447
Constrained_nlp
  nonlinear programming 447
convolution 363, 370
coordinate format 768, 771, 779
correlation 363, 370
correlation matrix 646
cosine factor 257
cosine Fresnel integrals 507
CPU time 709
cubic Hermite polynomials 304
cubic spline interpolant 217
cubic splines 145, 152, 157, 160, 205
current value of the seed 674

D

data types 800
dates and days 709, 711
decay rates 290
derivatives 286
differential equations 321, 2
  bvp_finite_difference 321
discrete Fourier cosine transformation 351, 353
discrete Fourier sine transformation 355, 357
distribution functions 521, 523, 524, 526, 528, 530, 531, 533, 534, 536, 537, 539, 540, 542, 543

E

eigenvalues 115, 116, 117, 118, 120, 123, 126, 129, 132, 135
eigenvectors 115, 116, 117, 118, 120, 123, 126, 129, 132, 135
elementary functions 800
elementary integrals 506
element-wise addition 768, 771
elliptic integrals 500, 501, 502, 504, 505

equality/inequality constraints 433
equilibrium 290
error detection 216
error functions 460, 461, 465, 467
  complementary
    exponentially scaled 463, 5
error handling xxiii, 712, 718
error messages 704
errors 797
Euler's constant 722
evaluation 157
even sequence 351
expected normal scores 667

F

factorization 2
fast Fourier transforms 339, 340, 341, 345, 346, 349, 359
Faure 689
Faure sequence 687
  faure_next_point 687
financial functions 545, 546, 548, 550, 551, 553, 554, 556, 557, 558, 559, 561, 562, 563, 565, 567, 569, 570, 571, 573, 574, 576, 577, 579
Fourier transform 261

G

gamma distributions 682
gamma functions 473, 475, 476, 534
Gauss quadrature 282
Gaussian elimination 7, 14
Gaussian functions 521, 523
Gauss-Kronrod rules 237, 241
generalized inverses 3, 99
GMRES method 73
Gray code 689

H

Harding, L.J. 7
Healy's algorithm 110
Helmholtz's equation 332
Hermitian matrices 126
HODIE finite-difference scheme 332
Householder' s method 86, 87, 99, 104
hypergeometric functions 537
hyper-rectangle 276, 279, 687

I

ill-conditioning 3
imsl.h include file x
infinite interval 253
initialize random seed 675
initial-value problems 289, 297
integration 180, 186, 237, 241, 245, 249, 253, 257, 261, 265, 268, 272, 276, 279, 282
interpolation 142, 145, 152, 161, 167, 171, 220
inverse matrix 11, 17, 22
inversions 2, 4

J

Jenkins-Traub algorithm 384, 386

K

Kelvin functions 513, 514, 515, 516, 517, 518, 519, 520

L

lack-of-fit test 660
least squares 142
least-squares approximation 209
least-squares fit 84, 139, 189, 193, 199, 216, 416, 660
least-squares solutions 3
Lebesque measure 688
Levenberg-Marquardt algorithm 416
linear constraints 92
linear equations 26, 31, 35, 44, 54, 62, 68
linear least squares 3
linear least-squares problem 92
linear system solution 2, 4, 107
loop unrolling and jamming 7
low-discrepancy 689
LU factorization 4, 11, 26, 31, 44, 54

M

mathematical constants 719
matrices xii, 2, 4, 7, 11, 14, 17, 22, 107, 691
  general xii
  Hermitian xiii
  multiplying 735
  rectangular xii

symmetric xii
matrix multiply 738
matrix transpose 742, 746, 751, 755
matrix-matrix product 742, 746, 751, 755
matrix-vector produce 755
matrix-vector product 742, 746, 751
matrix-vector products 735, 738
memory allocation xx
method of lines 304
minimization 399, 400, 401, 405, 409, 416, 425, 429, 433, 447, 2
Müller's method 388
multiple right-hand sides 3

N

non-ANSI C ix
nonlinear least squares 416
nonlinear programming problem 447, 2
norms of a vector 733
numerical ranking 667

O

odd sequence 355
one-way frequency table 634
order statistics 667
ordinary differential equations 289, 291, 297
output files 704
overflow xxiii

P

page size 697
partial differential equations 290, 304
partial pivoting 11, 13
Poisson distributions 680
Poisson functions 539
Poisson solver 332
polynomial functions 383
polynomials 140, 143
Powell hybrid algorithm 393
predator-prey model 294
printing 691, 697, 698
pseudorandom numbers 685

Q

QR factorizations 3, 84
quadratic programming 429
quadrature 235, 236, 237
quasi-Monte Carlo 279, 6
quasi-Newton method 409

R

radial-basis fit 231
radial-basis functions 225
random number generation 628, 629
random numbers 674, 675, 676, 677, 679, 680, 682, 684
rank deficiency 3
real general band system 26
real symmetric definite linear system 78
real symmetric positive definite system 35
rectangular matrix 775
regression 651, 660
restarted generalized minimum residual method 73
right-hand side data 4
Runge-Kutta-Verner method 291

S

Savage scores 667
scattered data 220, 225
select random number generator 676
semi-infinite interval 253
simplex algorithm 425
sine factor 257
sine Fresnel integrals 508
singular value decomposition 3
singularity 3
smoothed data 216
smoothing 205
sort 728, 730
sparse Hermitian positive definite system 68
sparse real symmetric positive definite system 62
sparse system 44
spline interpolant 161, 167, 171
splines 160
splines 140, 141, 143, 177, 180, 182, 186, 193, 199, 209
standard exponential distributions 685

statistics 629, 646, 651
Van der Waerden scores 667
stiff systems 290
storage modes xii
SVD factorization 96, 102
symbolic factorizations 62, 68

T

test matrices 782, 784, 786, 791
Thread Safe xi
  multithreaded application xi
  single-threaded application xi
  threads and error handling 799
time constants 290
Tukey scores 667

U

uncertainty 4
underflow xxiii
uniform mesh 332
univariate 249
univariate statistics 629

V

variable order 321, 2
vectors 691
Verner, J.H. 294
version 708

Z

zero of a system 393
zeros of a function 388