Visual Numerics™

IMSL®

C functions for mathematical applications

C/Math/Library™ 3.0

User's Guide

Click here to go to C/Stat/Library

Quick Tips on How to Use this Online Manual


Click to display only the page.


Click to display both bookmark and the page.


Double-click to jump to a topic when the bookmarks are displayed.


Click to jump to a topic when the bookmarks are displayed.


Click to display both thumbnails and the page.


Click and use to drag the page in vertical direction and to select items on the page.


Click and drag to page to magnify the view.


Click and drag to page to reduce the view.


Click and drag to the page to select text.


Click to go to the first page.


Click to go back to the previous page from which you jumped.


Click to go to the next page.


Click to go to the last page.


Click to go back to the previous view and page from which you jumped.


Click to return to the next view.


Click to view the page at 100% zoom.


Click to fit the entire page within the window.


Click to fit the page width inside the window.


Click to find part of a word, a complete word, or multiple words in a active document.

Printing an online file: Select **Print** from the **File** menu to print an online file. The dialog box that opens allows you to print full text, range of pages, or selection.

Important Note: The last blank page of each chapter (appearing in the hard copy documentation) has been deleted from the on-line documentation causing a skip in page numbering before the first page of the next chapter, for instance, Chapter 4 in the on-line documentation ends on page 279 and Chapter 5 begins on page 281.

Numbering Pages. When you refer to a page number in the PDF online documentation, be aware that the page number in the PDF online documentation will not match the page number in the original document. A PDF publication always starts on page 1, and supports only one page-numbering sequence per file.

Copying text. Click the  button and drag to select and copy text.

Viewing Multiple Online Manuals: Select **Open** from the **File** menu, and open the .PDF file you need. Select Cascade from the Window menu to view multiple files.

Resizing the Bookmark Area in Windows: Drag the double-headed arrow that appears on the area's border as you pass over it.

Resizing the Bookmark Area in UNIX: Click and drag the button  that appears on the area's border at the bottom of the vertical bar.

Jumping to Topics: Throughout the text of this manual, links to chapters and other sections appear in green color text to indicate that you can jump to them. To return to the page from which you jumped, click the return back icon  on the toolbar. ***Note: If you zoomed in or out after jumping to a topic, you will return to the previous zoom view(s) before returning to the page from which you jumped.***

Let's try it, click on the following green color text: **Chapter 1: Linear Systems**

If you clicked on the green color in the example above, Chapter 1: Linear Systems opened. To return to this page, click the  on the toolbar.

Visual Numerics, Inc.
Corporate Headquarters
1300 W Sam Houston Pkwy., Ste 150
Houston, Texas 77042-4548
USA

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

Visual Numerics International Ltd.
Centennial Court
Suite 1, North Wing
Easthampstead Road
BRACKNELL BERSHIRE
RG12 1YQ
UNITED KINGDOM, THE

PHONE: 011 01344 458700
FAX: 011 01344 453743
e-mail: info@vniuk.co.uk

Visual Numerics SARL
Tour Europe
33 Place des Corolles
F-92049 PARIS LA DEFENSE, Cedex
FRANCE

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

Visual Numerics S. A. de C.V.
Cerrada de Berna #3
Tercer Piso Col. Juarez
Mexico D. F. C. P. 06000
MEXICO
PHONE: +52-5-514-9730 or 9628
FAX: +52-5-514-4873

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 GOBANCHO CIYODA-KU
Tokyo, 113
JAPAN

PHONE: +81-3-5211-7760
FAX: +81-3-5211-7769
e-mail: vnijapan@po.iijnet.or.jp

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

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
Republic of South Korea

PHONE:011 82 2 3273 2632 or 2633
FAX: 011 82 2 3273 2634
e-mail: leevni@chollian.dacom.co.kr

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

COPYRIGHT NOTICE: Copyright 1990-1999, an unpublished work by Visual Numerics, Inc. All rights reserved.

VISUAL NUMERICS, INC., MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Visual Numerics, Inc., shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.

TRADEMARK NOTICE: IMSL, Visual Numerics, IMSL FORTRAN Numerical Libraries, IMSL Productivity Toolkit, IMSL Libraries Environment and Installation Assurance Test, C Productivity Tools, FORTRAN Productivity Tools, IMSL C/Math/Library, IMSL C/Stat/Library, IMSL Fortran 90 MP Library, and IMSL Exponent Graphics are registered trademarks or trademarks of Visual Numerics, Inc., in the U.S. and other countries. Sun, SunOS, and Solaris are registered trademarks or trademarks of Sun Microsystems, Inc. SPARC and SPARCompiler are registered trademarks or trademarks of SPARC International, Inc. Silicon Graphics is aregisterd trademark of Silicon Graphics, Inc. IBM, AIX, and RS/6000 are registered trademarks or trademarks of International Business Machines Corporation. HP is a trademark of Hewlett-Packard. Silicon Graphics and IRIX are registered trademarks or trademarks of Silicon Graphics, Inc. DEC and AXP are registered trademarks or trademarks of Digital Equipment Corporation. All other trademarks are the property of their respective owners.

Use of this document is governed by a Visual Numerics Software License Agreement. This document contains confidential and proprietary information constituting valuable trade secrets. No part of this document may be reproduced or transmitted in any form without the prior written consent of Visual Numerics.

RESTRICTED RIGHTS LEGEND: This documentation is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFAR 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR Supplement, when applicable. Contractor/Manufacturer is Visual Numerics, Inc., 1300 W Sam Houston Pkwy., Ste 150, Houston, Texas 77042.

IMSL Fortran and C
Application Development Tools

C/Stat/Library

Contents

Volume I

Introduction

Getting Started

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.

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).

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 that 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 Reference Material 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.

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.

Chapter 1: Linear Systems

Routines

1.5 Linear Least-squares with Full Matrices

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` 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`. 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`. 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` 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`**. 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` 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,
```

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>

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 $R^T R$ 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>

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;
        int         index;

        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;
        int         index;

        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 "Introduction" 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` 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` 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:

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

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>
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>
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, 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, 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>
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, 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 introduction 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` 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 $A^Tx = b$ is required, specify the option `IMSL_TRANSPOSE`. This keyword only alters the forward elimination and back substitution so that the operations $U^Ty = b$ and $L^Tx = y$ are performed to obtain the solution. So, with one call to produce the factorization, solutions to both $Ax = b$ and $A^Tx = b$ 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 $A^Tz = r$.

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>
```

```
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>
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, 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, 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 “Introduction” 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>
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_sparse_elem     *c;
        Imsl_f_numeric_factor   numeric_factor;
        float                  *b_1;
        float                  *b_2;
        float                  *x_1;
        float                  *x_2;
        int                     n;
        int                     ic;
        int                     nz;
        int                     i;
        int                     index;
        double                  time_1;
        double                  time_2;
        float                   error_1;
        float                   error_2;

        ic = 50;
        n = ic*ic;

                        /*  Generate two right hand sides */

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

                        /*  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 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* `*p`, *float* `*z`), *float* `*b`,
`IMSL_RETURN_USER`, *float* `x[]`,
`IMSL_MAX_ITER`, *int* `*maxit`,
`IMSL_REL_ERR`, *float* `tolerance`,
`IMSL_PRECOND`, *void* `precond` (*float* `*r`, *float* `*z`),
`IMSL_MAX_KRYLOV_SUBSPACE_DIM`, *int* `kdmax`,
`IMSL_HOUSEHOLDER_REORTHOG`,
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.

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

$$\kappa_m(v) = \text{span}(v, Av, \ldots, A^{m-1}v)$$

$v = r_0$ 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, 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, 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`,
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 = diag(A). The user-supplied vector `diagonal` should be set so that `diagonal[i]` = $A_{i,i}$.

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)

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

where

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

It is also known that

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

where the T_n are the symmetric, tridiagonal matrices

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

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, 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, &nz, 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

$$Q_k = l - \tau_k u_k u_k^T Q$$

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

$$x_1 = x_2 + x_3 \le 1$$

$$0 \le x_1 \le 0.5$$

$$0 \le x_2 \le 0.5$$

$$0 \le x_3 \le 0.5$$

```
#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 $s_{i,i} \leq |$`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 $s_{k,k} \leq |$`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)
The column dimension of the array containing the inverse of A.
Default: `inva_col_dim` = n

`IMSL_TOLERANCE`, *float* `tol` (Input)
Tolerance used in determining linear dependence.
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

$$\left| a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2 \right| \leq \varepsilon |a_{ii}|$$

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. $a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2 < -\varepsilon |a_{ii}|$
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

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. 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),

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

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

$$\tau = \max_{1 \leq j \leq n} \frac{\left\|A\tilde{x}_j - \tilde{\lambda}_j\tilde{x}_j\right\|_2}{n\varepsilon\|A\|_2\left\|\tilde{x}_j\right\|_2}$$

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 127), 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 121).

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 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_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

$$A = \begin{bmatrix} 1+i & 0.5+i & 5i \\ -10 & 2+i & 0 \\ 5+i & 1 & 0.5+3i \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0 & 0 \\ 3+3i & 3+3i & i \\ 4+2i & 0.5+i & 1+i \end{bmatrix}$$

```
#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_USER, 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

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` and `imsl_f_cub_spline_interp_shape`. 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` 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` 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`) and that compute the two-dimensional, tensor-product spline coefficients for a tensor-product, least-squares problem (`imsl_f_spline_2d_least_squares`). 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` and `imsl_f_spline_2d_integral`.

Scattered Data Interpolation

The IMSL C/Math/Library provides one function, `imsl_f_scattered_2d_interp`, 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` computes regressions with user-supplied functions. The function `imsl_f_spline_least_squares` 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`) mentioned above. The function `imsl_f_radial_scattered_fit` 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.

$$s_i(x) := \sum_{j_{d-1}=0}^{n_{d-1}-1} \cdots \sum_{j_0=0}^{n_0-1} c^i_{j_0,\ldots,j_{d-1}} B_{j_0,k_0,\mathbf{t}^0\ldots} B_{j_{d-1},k_{d-1},\mathbf{t}^{d-1}}$$

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]`	$= t_j$ $i = 0, \ldots, d-1 \quad j = 0, \ldots, n_i + k_i - 1$
`sp-> coef [i] [j]`	$= c_j$ $i = 0, \ldots, r-1 \quad j = j_0 + j_1 n_0 + \ldots + j_{d-1} n_0 \ldots n_{d-2}$

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

$$\xi^j_{M^j} \leq x_j < \xi^j_{M^{j+1}} \qquad j = 1, \ldots, d$$

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]`	$= \xi_j$ $i = 0, \ldots, d-1 \quad j = 0, \ldots, n_i - 1$
`ppoly->coef [i[ [j]`	$= c_j$ $i = 0, \ldots, r-1$ $j = 0, \ldots, k_0(n_0 - 1)\ldots k_{d-1}(n_{d-1} - 1)$

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.

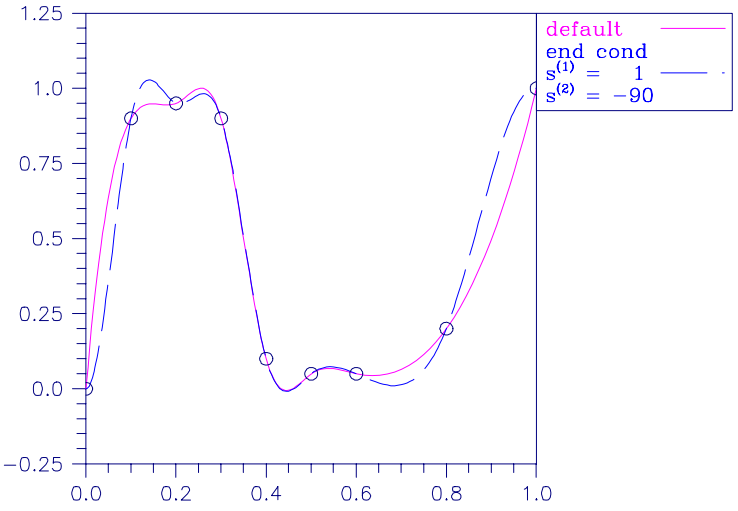


Figure 3-1 Two Interpolating Splines

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.

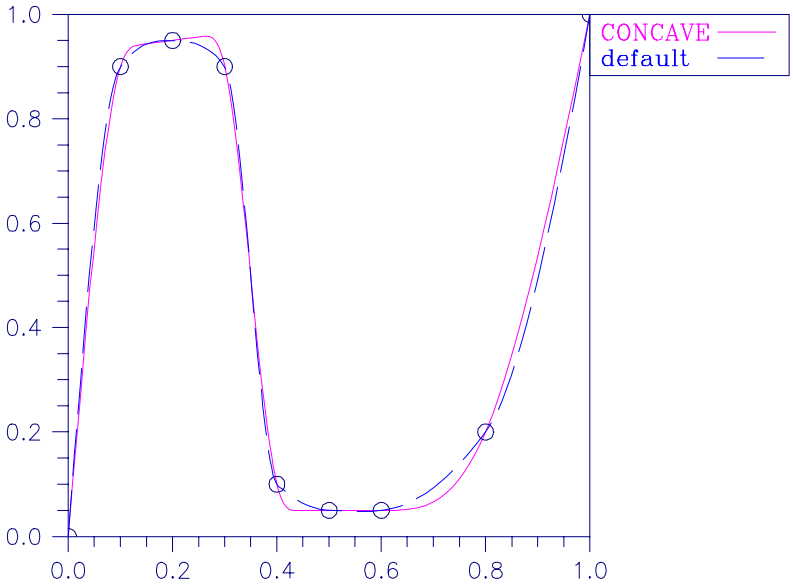


Figure 3-2 Two Shape-Preserving Splines

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 **t** and *x* are selected properly and the data points arise from the values of a smooth (say C^k) function *f*, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. The maximum absolute error satisfies

$$\left\| f - s \right\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} \le C \left\| f^{(k)} \right\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} |\mathbf{t}|^k$$

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.

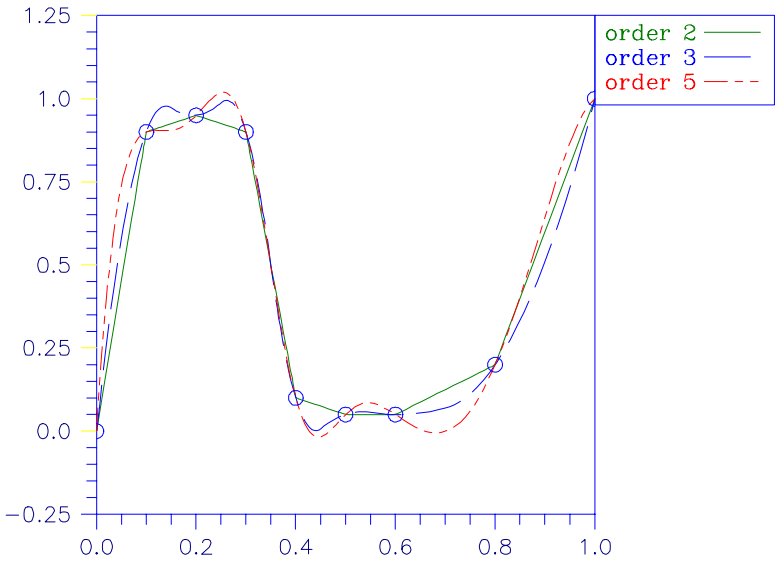


Figure 3-3 Three Spline Interpolants

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 **t** is the knot sequence.
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.

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

$$\mathbf{t}_i = x_0 \qquad \text{for } i = 0, \ldots, k-1$$
$$\mathbf{t}_i = \left(x_{i-\frac{k-1}{2}-1} + x_{i-1-\frac{k-2}{2}}\right)/2 \qquad \text{for } i = k, \ldots, n-1$$
$$\mathbf{t}_i = x_{n-1} \qquad \text{for } i = n, \ldots, n = k-1$$

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 159). 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_data` × `num_data` 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_xdata`

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 $\mathbf{t}_i \le \texttt{data}_i < \mathbf{t}_{i+order}$, where **t** is the knot sequence.
IMSL_DATA_TOO_LARGE	The data arrays xdata and ydata must satisfy $\texttt{data}_i \le \mathbf{t}_{num\_data}$, for $i = 1, \ldots,$ num_data.
IMSL_DATA_TOO_SMALL	The data arrays xdata and ydata must satisfy $\texttt{data}_i \ge \mathbf{t}_{order-1}$, 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

$$\int_a^b \int_c^d s(x,y)dydx$$

This function uses the (univariate integration) identity (22) in de Boor (1978, p. 151)

$$\int_{\mathbf{t}_0}^{x} \sum_{i=0}^{n-1} \alpha_i B_{i,k}(\tau) d\tau = \sum_{i=0}^{r-1} \left[\sum_{j=0}^{i} \alpha_j \frac{\mathbf{t}_{j+k} - \mathbf{t}_j}{k} \right] B_{i,k+1}(x)$$

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 (*float* fcn (*int* k, *float* x), *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[],
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

Description

The function `imsl_f_user_fcn_least_squares` computes a best least-squares approximation to given univariate data of the form

$$\left\{(x_i, f_i)\right\}_{i=0}^{n-1}$$

by M basis functions

$$\left\{F_j\right\}_{j=1}^{M}$$

(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       0.07
```

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

$$\mathbf{t}_0^g \le \ldots \le \mathbf{t}_{k-1}^g \le x_i \le \mathbf{t}_m^g \le \ldots \le \mathbf{t}_{m+k-1}^g \qquad i = 1, \ldots, M$$

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`.

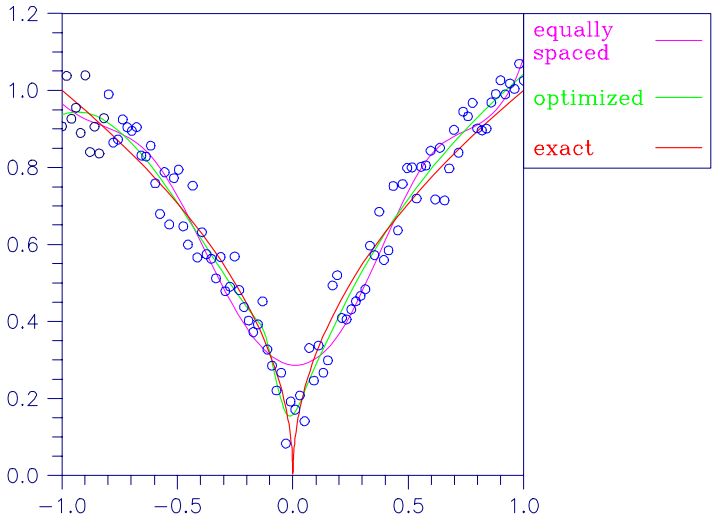


Figure 3-4 Two Fits to Noisy $\sqrt{|x|}$

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 $\texttt{data}_i \leq \mathbf{t}_{spline\_space\_dim}$, for $i = 1, \ldots, \texttt{num\_data}$.
IMSL_SPLINE_SMLST_ELEMNT	The data arrays `xdata` and `ydata` must satisfy $\texttt{data}_i \geq \mathbf{t}_{order-1}$, for $i = 1, \ldots, \texttt{num\_data}$.
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 $c \leq d$. Note that every `der` must be at least −1.

`constraints [i].type`	*i*-th constraint
1	$bl_i = f^{(d_i)}(x_i)$
2	$f^{(d_i)}(x_i) \leq bu_i$
3	$f^{(d_i)}(x_i) \geq bl_i$
4	$bl_i \leq f^{(d_i)}(x_i) \leq bu_i$
5	$bl_i = \int_c f(t)dt$
6	$\int_c f(t)dt \leq bu_i$
7	$\int_c f(t)dt \geq bl_i$
8	$bl_i \leq \int_c f(t)dt \leq bu_i$
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	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \leq bu_i$
11	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \geq bl_i$
12	$bl_i \leq f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \leq bu_i$

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 $0 \leq$ `nhard` $\leq$ `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	$= \int_{y_p}$
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

$$\frac{x}{2} + sin(\frac{x}{2})$$

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, rels, 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].itype = 3;
        constraint[i].ider = 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, rels, 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 `fdata[]` (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, `fdata` = f, `sdata` = s and `xdata` = x, the algorithm proceeds as follows. Although the user need not input an ordered `xdata` sequence, we will assume that x is increasing for simplicity. The algorithm first sorts the `xdata` values into an increasing sequence and then continues. A cubic spline interpolant is computed for each of the 6-point data sets (initially setting $s = f$)

$$(x_j, s_j) \qquad j = i-3, \ldots, i+3 \; j \neq i,$$

where $i = 4, \ldots, n-3$. 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

$$\left|pe_i\right| \leq sc\left(x_{i+3} - x_{i-3}\right)/6 \quad i = 4, \ldots, n-3$$

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, f, fdata[NDATA], sc, *sdata=NULL;
  float c, 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

$$s(x,y) = \sum_{m+n\leq 5} c_{mn}^T x^m y^n \qquad \forall x, y \in T$$

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_DELTA`, *float* `delta`,
`IMSL_WEIGHTS`, *float* `weights[]`,
`IMSL_QR`,
0)

Optional Arguments

`IMSL_CENTERS` (Input)
User-supplied centers. See the "Description" 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_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_QR`

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

$$x^0,\ldots,x^{n-1} \subset \mathbf{R}^d \quad f_0,\ldots,f_{n-1} \subset \mathbf{R}^1$$

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        j;
    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        j;
    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

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

$$\int_a^b \int_{g(x)}^{h(x)} f(x,y)dydx$$

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`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

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`.

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

$$\int_a^b \text{fcn}(x)dx$$

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,
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: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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`.

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

$$\int_a^b \text{fcn}(x)dx$$

is returned. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_int_fcn_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`,
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`.

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

$$\int_a^b \mathrm{fcn}(x)w(x)dx$$

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,
        0)
```

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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`.

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

$$\int_0^1 \left[(1+x)(1-x)\right]^{1/2} x\,ln\,(x)\,dx = \frac{3ln(2)-4}{9}$$

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

$$\int_0^1 \left[(1+x)(1-x)\right]^{1/2} x\,ln\,(x)\,dx = \frac{3ln(2)-4}{9}$$

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`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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`.

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`,
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

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

$$\int_0^1 ln\left(x\right) sin\left(10\pi x\right) dx$$

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,
        0)
```

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

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`.

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,
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.

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`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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.

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

$$\int_a^b \int_{gcn(x)}^{hcn(x)} \text{fcn}(x,y)\, dydx$$

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* `x`),
float `hcn` (*float* `x`),
`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`,
0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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`.

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

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1})\, dx_{n-1} \ldots dx_0$$

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`, 0)

Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $\text{err\_abs} = \sqrt{\varepsilon}$

where ε is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $\text{err\_rel} = \sqrt{\varepsilon}$

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.

Description

The function `imsl_f_int_fcn_hyper_rect` approximates the *n*-dimensional iterated integral

$$\int_{a_0}^{b} \ldots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1})\, dx_{n-1} \ldots dx_0$$

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.

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 2*k* 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`,
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` = $\sqrt[4]{\varepsilon}$

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

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

$$\frac{dy_i}{dt} = y_i' = f_i(t, y_1, \ldots, y_N) \qquad i = 1, \ldots, N$$

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

$$\frac{dy}{dt} = y' = f(t, y)$$

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 295, 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 **`fast_poisson_2d`**, page 311, 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_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()`)

Required Arguments for imsl_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_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_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,
        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.

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_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())`

Required Arguments for imsl_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_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_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,
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.

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()`)

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

$$\alpha_k u_k + \beta_k \frac{2\gamma_k}{2x} \equiv \gamma_k$$

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`,
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

$$\texttt{deriv[k,i]} = \frac{\partial u_k}{\partial u_x}(x, t_0)$$

The array `initial_deriv` contains the derivative values as output:

$$\texttt{deriv[k,i]} = \frac{\partial u_k}{\partial u_x}(x, \texttt{tend}) \qquad \text{at } x = x[i]$$

Default: Derivatives are computed using cubic spline interpolation

`IMSL_HTRIAL`, *float* `*htrial` (Output)
Return the current trial step size.

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

$$u_k = u_k(x, t) \quad \text{at } t = t_0$$

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

$$\hat{u}_k(x, t) = \sum_{i=1}^{M} \left(a_{i,k}(t)\phi_i(x) + b_{i,k}(t)\psi_i(x) \right)$$

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

$$\frac{da_{i,k}}{dt}\phi_i(p_j) + \frac{db_{i,k}}{dt}\psi_i(p_j) =$$
$$f_k\left(p_j, t, \hat{u}_1(p_j), \ldots, \hat{u}_M(p_j), \ldots, (\hat{u}_1)_{xx}(p_j), \ldots, (\hat{u}_M)_{xx}(p_j)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N - 1)$.

This is a system of $2M(N - 1)$ ordinary differential equations in $2M\,N$ 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 $2M\,N$ 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 $\text{diag}(\alpha_{1,1}, \ldots, \alpha_{1,M})$. 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`**, page 288. 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
```

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)

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 in 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,
0)
```

Optional Arguments

`IMSL_RETURN_USER`, *float* `u_user[]` (Output)
User-supplied array of size `nx*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

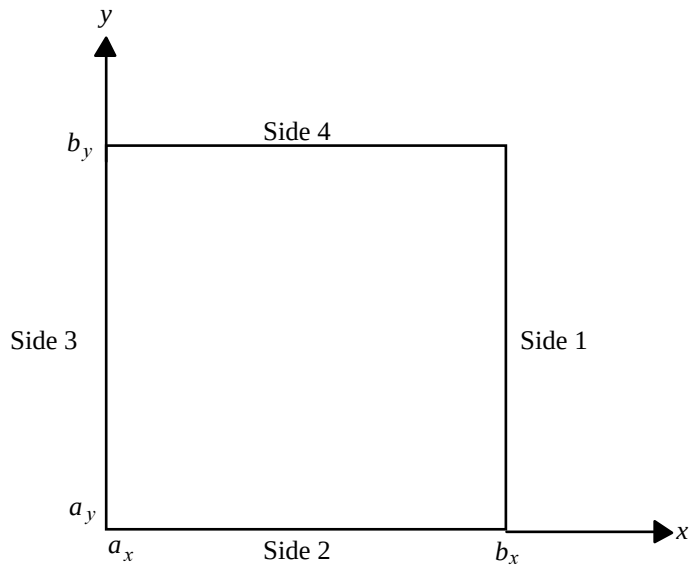
Description

Let c = `COEFU`, 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 `IORDER` = 2 and `IORDER` = 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

$$m = \sin\left(x + 2y) + e^{2x+3y}\right)$$

on the other three sides is solved. The domain is the rectangle [0, ¼] × [0, ½]. The output of `imsl_f_fast_poisson_3d` 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

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` and `imsl_c_fft_complex`, 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

$$q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos\frac{2\pi km}{n} \qquad m = 1, \ldots, n/2$$

$$q_{2m-2} = -\sum_{k=0}^{n-1} p_k \sin\frac{2\pi km}{n} \qquad m = 1, \ldots, n/2 - 1$$

$$q_0 = \sum_{k=0}^{n-1} p_k$$

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

$$g(t) = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi k(t-t_0)}{n\Delta} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi k(t-t_0)}{n\Delta}\right]$$
$$= \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi k(t-t_0)}{T} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi k(t-t_0)}{T}\right]$$

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`

$$q_m = 2\sum_{n=1}^{N-2} p_n \sin\left(\frac{mn\pi}{N-1}\right) + s_0 + s_{N-1}(-1)^m$$

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, p, 0);
                              /* Write the input */
    imsl_c_write_matrix ("The input matrix is ", 5, 4, 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, 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, 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

$$\hat{z}_j = \hat{x}_j \bar{y}_j$$

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, 2π] 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, k, 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_ERR_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,
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` + 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_ERR_EST`, *float* `*error_est` (Output)
Overall estimate of the pseudo error, `disc_error_est` + `trunc_error_est` + `cond_error_est`. Pseudo error = absolute error / exp(`sigma*bvalue`).

`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 for `acoef`, the coefficients of the Laguerre expansion.

`IMSL_DECAY_FUNCTION_BASE`, *float* `*r` (Output)
The base of the decay function for `acoef`. Here $|\texttt{acoef}[j]| \le k/r^{(j-1)}$ for some $j \ge m/2 - 1$, where m is the number of Laguerre coefficients actually computed.

`IMSL_LOG_LARGEST_COEFFICIENTS`, *float* `*log_largest_coefs` (Output)
The logarithm of the largest `acoef`.

`IMSL_LOG_SMALLEST_COEFFICIENTS`, *float* `*log_smallest_coefs` (Output)
The logarithm of the smallest nonzero `acoef`.

`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.
`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.

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-suppled 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 finv[5];
        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

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()
{
        int             i;
        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()
{
        int             i;
        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

$$\text{coef}[n]\, z^{n} + \text{coef}[n-1]\, z^{n-1} + \ldots + \text{coef}[0]$$

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()
{
        int             i;
        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()
{
        int             i;
        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[],
        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`.

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.

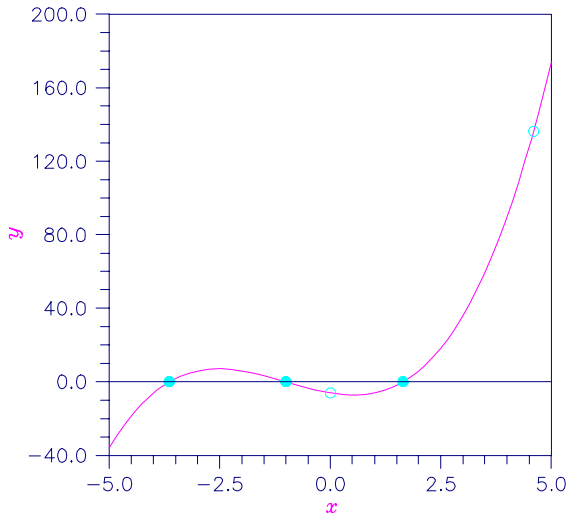


Figure 7-1 Plot of $x^3 + 3x^2 - 4x - 6$

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,
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 $\partial f_i / \partial x_i$ 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`.

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.

Chapter 8: Optimization

Routines

Usage Notes

Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$ is continuous and has derivatives of all orders required by the algorithms. The functions for unconstrained minimization are grouped into three categories: univariate functions, multivariate functions, and nonlinear least-squares functions.

For the univariate functions, it is assumed that the function is unimodal within the specified interval. For discussion on unimodality, see Brent (1973).

A quasi-Newton method is used for the multivariate function `imsl_f_min_uncon_multivar`. The default is to use a finite-difference approximation of the gradient of $f(x)$. Here, the gradient is defined to be the vector

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n} \right]$$

However, when the exact gradient can be easily provided, the keyword `IMSL_GRAD` should be used.

The nonlinear least-squares function uses a modified Levenberg-Marquardt algorithm. The most common application of the function is the nonlinear data-fitting problem where the user is trying to fit the data with a nonlinear model.

These functions are designed to find only a local minimum point. However, a function may have many local minima. Try different initial points and intervals to obtain a better local solution.

Double-precision arithmetic is recommended for the functions when the user provides only the function values.

Linearly Constrained Minimization

The linearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } A_1 x = b_1$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$, A_1 and A_2 are coefficient matrices, and b_1 and b_2 are vectors. If $f(x)$ is linear, then the problem is a linear programming problem. If $f(x)$ is quadratic, the problem is a quadratic programming problem.

The function `imsl_f_lin_prog` uses a revised simplex method to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

The function `imsl_f_quadratic_prog` is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then `imsl_f_quadratic_prog` modifies it to be positive definite. In this case, output should be interpreted with care because the problem has been changed slightly. Here, the Hessian of $f(x)$ is defined to be the $n \times n$ matrix

$$\nabla^2 f(x) = \left[\frac{\partial^2}{\partial x_i \partial x_j} f(x) \right]$$

Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$
$$\text{subject to } g_i(x) = 0 \qquad \text{for } i = 1, 2, \ldots, m_1$$
$$g_i(x) \geq 0 \qquad \text{for } i = m_1 + 1, \ldots, m$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$ and $g_i : \mathbf{R}^n \rightarrow \mathbf{R}$, for i = 1, 2, …, m.

The function `imsl_f_min_con_nonlin` uses a successive quadratic programming algorithm to solve this problem. A more complete discussion of this algorithm can be found in the documentation.

min_uncon

Find the minimum point of a smooth function $f(x)$ of a single variable using only function evaluations.

Synopsis

#include `<imsl.h>`

float `imsl_f_min_uncon` (*float* `fcn()`, *float* `a`, *float* `b`, …, 0)

The type *double* function is `imsl_d_min_uncon`.

Required Arguments

float `fcn`(*float* `x`) (Input/Output)
User-supplied function to compute the value of the function to be minimized where `x` is the point at which the function is evaluated, and `fcn` is the computed function value at the point `x`.

float `a` (Input)
The lower endpoint of the interval in which the minimum point of `fcn` is to be located.

float `b` (Input)
The upper endpoint of the interval in which the minimum point of `fcn` is to be located.

Return Value

The point at which a minimum value of `fcn` is found. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_min_uncon` (*float* `fcn()`, *float* `a`, *float* `b`,
`IMSL_XGUESS`, *float* `xguess`,
`IMSL_STEP`, *float* `step`,
`IMSL_ERR_ABS`, *float* `err_abs`,
`IMSL_MAX_FCN`, *int* `max_fcn`,
0)

Optional Arguments

`IMSL_XGUESS`, *float* `xguess` (Input)
An initial guess of the minimum point of `fcn`.
Default: `xguess` = (*a* + *b*)/2

`IMSL_STEP`, *float* `step` (Input)
An order of magnitude estimate of the required change in `x`.
Default: `step` = 1.0

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
The required absolute accuracy in the final value of `x`. On a normal return, there are points on either side of `x` within a distance `err_abs` at which `fcn` is no less than `fcn` at `x`.
Default: `err_abs` = 0.0001

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations allowed.
Default: `max_fcn` = 1000

Description

The function `imsl_f_min_uncon` uses a safeguarded quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the subroutine ZXLSF written by M.J.D. Powell at the University of Cambridge.

The function `imsl_f_min_uncon` finds the least value of a univariate function, *f*, which is specified by the function `fcn`. Other required data are two points *a* and *b* that define an interval for finding a minimum point from an initial estimate of the solution, x_0 where x_0 = `xguess`. The algorithm begins the search by moving from x_0 to $x = x_0 + s$ where s = `step` is an estimate of the required change in *x* and may be positive or negative. The first two function evaluations indicate the direction to the minimum point and the search strides out along this direction until a bracket on a minimum point is found or until *x* reaches one of the endpoints *a* or *b*. During this stage, the step length increases by a factor of between two and nine per function evaluation. The factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, we have three points,

$$x_1, x_2, x_3, \text{ with } x_1 < x_2 < x_3,\ f(x_1) \geq f(x_2), \text{ and } f(x_2) \leq f(x_3).$$

There are three main rules in the technique for choosing the new x from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter η, which depends on the closeness of f to a quadratic, and (iii) whether x_2 is near the center of the range between x_1 and x_3 or is relatively close to an end of this range. In outline, the new value of x is as near as possible to the predicted minimum point, subject to being at least ε from x_2, and subject to being in the longer interval between x_1 and x_2, or x_2 and x_3, when x_2 is particularly close to x_1 or x_3.

The algorithm is intended to provide fast convergence when f has a positive and continuous second derivative at the minimum. Also, the algorithim avoids gross inefficiencies in pathological cases, such as

$$f(x) = x + 1.001|x|$$

The algorithm can automatically make ε large in the pathological cases. In this case, it is usual for a new value of x to be at the midpoint of the longer interval that is adjacent to the least-calculated function value. The midpoint strategy is used frequently when changes to f are dominated by computer rounding errors, which will almost certainly happen if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the subroutine claims to have achieved the required accuracy if it decides that there is a local minimum point within distance δ of x, where δ = `err_abs`, even though the rounding errors in f may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high precision arithmetic is recommended.

Examples

Example 1

A minimum point of $f(x) = e^x - 5x$ is found.

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

float           fcn(float);

void main ()
{
    float       a = -100.0;
    float       b = 100.0;
    float       fx, x;

    x = imsl_f_min_uncon (fcn, a, b, 0);
    fx = fcn(x);

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


float fcn(float x)
{
```

```
    return exp(x) - 5.0*x;
}
```

Output

```
The solution is:    1.6094
The function evaluated at the solution is:   -3.0472
```

Example 2

A minimum point of $f(x) = x(x^3 - 1) + 10$ is found with an initial guess $x_0 = 3$.

```
#include <imsl.h>

float          fcn(float);

void main ()
{
    int         max_fcn =  50;
    float       a       = -10.0;
    float       b       =  10.0;
    float       xguess  =   3.0;
    float       step    =   0.1;
    float       err_abs =   0.001;
    float       fx, x;

    x = imsl_f_min_uncon (fcn, a, b,
                          IMSL_XGUESS, xguess,
                          IMSL_STEP, step,
                          IMSL_ERR_ABS, err_abs,
                          IMSL_MAX_FCN, max_fcn,
                          0);
    fx = fcn(x);

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

float fcn(float x)
{
    return x*(x*x*x-1.0) + 10.0;
}
```

Output

```
The solution is:    0.6298
The function evaluated at the solution is:    9.5275
```

Warning Errors

IMSL_MIN_AT_BOUND	The final value of x is at a bound.
IMSL_NO_MORE_PROGRESS	Computer rounding errors prevent further refinement of x.
IMSL_TOO_MANY_FCN_EVAL	Maximum number of function evaluations exceeded.

min_uncon_deriv

Finds the minimum point of a smooth function $f(x)$ of a single variable using both function and first derivative evaluations.

Synopsis

#include `<imsl.h>`

float `imsl_f_min_uncon_deriv` (*float* `fcn()`, *float* `grad()`, *float* `a`, *float* `b`, …, 0)

The type *double* function is `imsl_d_min_uncon_deriv`.

Required Arguments

float `fcn` (*float* `x`) (Input/Output)
User-supplied function to compute the value of the function to be minimized where `x` is the point at which the function is evaluated, and `fcn` is the computed function value at the point `x`.

float `grad` (*float* `x`) (Input/Output)
User-supplied function to compute the first derivative of the function where `x` is the point at which the derivative is evaluated, and `grad` is the computed value of the derivative at the point `x`.

float `a` (Input)
The lower endpoint of the interval in which the minimum point of `fcn` is to be located.

float `b` (Input)
The upper endpoint of the interval in which the minimum point of `fcn` is to be located.

Return Value

The point at which a minimum value of `fcn` is found. If no value can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_min_uncon_deriv` (*float* `fcn()`, *float* `grad()`, *float* `a`, *float* `b`,
`IMSL_XGUESS`, *float* `xguess`,
`IMSL_ERR_REL`, *float* `err_rel`,
`IMSL_GRAD_TOL`, *float* `grad_tol`,
`IMSL_MAX_FCN`, *int* `max_fcn`,
`IMSL_FVALUE`, *float* `*fvalue`,
`IMSL_GVALUE`, *float* `*gvalue`,
0)

Optional Arguments

`IMSL_XGUESS`, *float* `xguess` (Input)
An initial guess of the minimum point of `fcn`.
Default: `xguess` = $(a + b)/2$

`IMSL_ERR_REL`, *float* `err_rel` (Input)
The required relative accuracy in the final value of `x`. This is the first stopping criterion. On a normal return, the solution `x` is in an interval that contains a local minimum and is less than or equal to
max (1.0, $|x|$) * `err_rel`. When the given `err_rel` is less than zero,

$$\sqrt{\varepsilon}$$

is used as `err_rel` where ε is the machine precision.
Default:

$$\text{err\_ rel} = \sqrt{\varepsilon}$$

`IMSL_GRAD_TOL`, *float* `grad_tol` (Input)
The derivative tolerance used to decide if the current point is a local minimum. This is the second stopping criterion. `x` is returned as a solution when `grad` is less than or equal to `grad_tol`. `grad_tol` should be nonnegative; otherwise, zero would be used.
Default:

$$\text{grad\_ tol} = \sqrt{\varepsilon}$$

where ε is the machine precision

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations allowed.
Default: `max_fcn` = 1000

`IMSL_FVALUE`, *float* `*fvalue` (Output)
The function value at point `x`.

`IMSL_GVALUE`, *float* `*gvalue` (Output)
The derivative value at point `x`.

Description

The function `f_min_uncon_deriv` uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the function terminates with a solution. Otherwise, the point with least function value will be used as the starting point.

From the starting point, say x_c, the function value $f_c = f(x_c)$, the derivative value $g_c = g(x_c)$, and a new point x_n defined by $x_n = x_c - g_c$ are computed. The function $f_n = f(x_n)$, and the derivative $g_n = g(x_n)$ are then evaluated. If either

$f_n \geq f_c$ or g_n has the opposite sign of g_c, then there exists a minimum point between x_c and x_n, and an initial interval is obtained. Otherwise, since x_c is kept as the point that has lowest function value, an interchange between x_n and x_c is performed. The secant method is then used to get a new point

$$x_s = x_c - g_c\left(\frac{g_n - g_c}{x_n - x_c}\right)$$

Let $x_n = x_s$, and repeat this process until an interval containing a minimum is found or one of the convergence criteria is satisfied. The convergence criteria are as follows:

Criterion 1: $|x_c - x_n| \leq \varepsilon_c$

Criterion 2: $|g_c| \leq \varepsilon_g$

where $\varepsilon_c = \max\{1.0, |x_c|\}\,\varepsilon$, ε is an error tolerance, and ε_g is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. Function and derivative are then evaluated at that point, and accordingly a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval be reduced by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this function is repeated until one of the stopping criteria is met.

Examples

Example 1

In this example, a minimum point of $f(x) = e^x - 5x$ is found.

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

float           fcn(float);
float           deriv(float);

void main ()
{
    float       a = -10.0;
    float       b = 10.0;
    float       fx, gx, x;

    x = imsl_f_min_uncon_deriv (fcn, deriv, a, b, 0);
    fx = fcn(x);
    gx = deriv(x);

    printf ("The solution is:  %7.3f\n", x);
    printf ("The function evaluated at the solution is:  %9.3f\n", fx);
    printf ("The derivative evaluated at the solution is:  %7.3f\n", gx);
}


float fcn(float x)
{
      return exp(x) - 5.0*(x);
```

```
}


float deriv (float x)
{
     return exp(x) - 5.0;
}
```

Output

```
The solution is:    1.609
The function evaluated at the solution is:     -3.047
The derivative evaluated at the solution is:   -0.001
```

Example 2

A minimum point of $f(x) = x(x^3 - 1) + 10$ is found with an initial guess $x_0 = 3$.

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

float           fcn(float);
float           deriv(float);

void main ()
{
    int         max_fcn = 50;
    float       a = -10.0;
    float       b = 10.0;
    float       xguess = 3.0;
    float       fx, gx, x;

    x = imsl_f_min_uncon_deriv (fcn, deriv, a, b,
                                IMSL_XGUESS, xguess,
                                IMSL_MAX_FCN, max_fcn,
                                IMSL_FVALUE, &fx,
                                IMSL_GVALUE, &gx,
                                0);
     printf ("The solution is:  %7.3f\n", x);
     printf ("The function evaluated at the solution is:  %7.3f\n", fx);
      printf ("The derivative evaluated at the solution is:  %7.3f\n", gx);
}

float fcn(float x)
{
      return x*(x*x*x-1) + 10.0;
}

float deriv(float x)
{
    return  4.0*(x*x*x) - 1.0;
}
```

Output

```
The solution is:    0.630
The function evaluated at the solution is:    9.528
The derivative evaluated at the solution is:    0.000
```

Warning Errors

IMSL_MIN_AT_LOWERBOUND	The final value of x is at the lower bound.
IMSL_MIN_AT_UPPERBOUND	The final value of x is at the upper bound.
IMSL_TOO_MANY_FCN_EVAL	Maximum number of function evaluations exceeded.

min_uncon_multivar

Minimizes a function $f(x)$ of n variables using a quasi-Newton method.

Synopsis

#include <imsl.h>

float *imsl_f_min_uncon_multivar (*float* fcn(), *int* n, …, 0)

The type *double* function is imsl_d_min_uncon_multivar.

Required Arguments

float fcn (*int* n, *float* x[]) (Input/Output)
User-supplied function to evaluate the function to be minimized where n is the size of x, x is the point at which the function is evaluated, and fcn is the computed function value at the point x.

int n (Input)
Number of variables.

Return Value

A pointer to the minimum point x of the function. 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_min_uncon_multivar (*float* fcn(), *int* n,
IMSL_XGUESS, *float* xguess[],
IMSL_GRAD, *void* grad (*int* n, *float* x[], *float* g[]),
IMSL_XSCALE, *float* xscale[],
IMSL_FSCALE, *float* fscale,
IMSL_GRAD_TOL, *float* grad_tol,
IMSL_STEP_TOL, *float* step_tol,
IMSL_REL_FCN_TOL, *float* rfcn_tol,
IMSL_MAX_STEP, *float* max_step,
IMSL_GOOD_DIGIT, *int* ndigit,
IMSL_MAX_ITN, *int* max_itn,
IMSL_MAX_FCN, *int* max_fcn,

```
IMSL_MAX_GRAD, int max_grad,
IMSL_INIT_HESSIAN, int ihess,
IMSL_RETURN_USER, float x[],
IMSL_FVALUE, float *fvalue,
0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `n` components containing an initial guess of the computed solution.
Default: `xguess` = 0

`IMSL_GRAD`, *void* `grad` (*int* `n`, *float* `x[]`, *float* `g[]`) (Input/Output)
User-supplied function to compute the gradient at the point `x` where `n` is the size of `x`, `x` is the point at which the gradient is evaluated, and `g` is the computed gradient at the point `x`.

`IMSL_XSCALE`, *float* `xscale[]` (Input)
Array with `n` components containing the scaling vector for the variables. `xscale` is used mainly in scaling the gradient and the distance between two points. See keywords `IMSL_GRAD_TOL` and `IMSL_STEP_TOL` for more detail.
Default: `xscale[]` = 1.0

`IMSL_FSCALE`, *float* `fscale` (Input)
Scalar containing the function scaling. `fscale` is used mainly in scaling the gradient. See keyword `IMSL_GRAD_TOL` for more detail.
Default: `fscale` = 1.0

`IMSL_GRAD_TOL`, *float* `grad_tol` (Input)
Scaled gradient tolerance. The *i*-th component of the scaled gradient at `x` is calculated as

$$\frac{|g_i| * max(|x_i|, 1/s_i)}{max(|f(x)|, f_s)}$$

where $g = \nabla f(x)$, s = `xscale`, and f_s = `fscale`.
Default: grad_ tol = $\sqrt{\varepsilon}$, $\sqrt[3]{\varepsilon}$ in double where ε is the machine precision.

`IMSL_STEP_TOL`, *float* `step_tol` (Input)
Scaled step tolerance. The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{max(|x_i|, 1/s_i)}$$

where s = `xscale`.
Default: `step_tol` = $\varepsilon^{2/3}$

`IMSL_REL_FCN_TOL`, *float* `rfcn_tol` (Input)
Relative function tolerance.
Default: `rfcn_tol` = max (10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double

`IMSL_MAX_STEP`, *float* `max_step` (Input)
Maximum allowable step size.
Default: `max_step` = 1000max (ε_1, ε_2) where,

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$\varepsilon_2 = \|s\|_2$, s = `xscale`, and t = `xguess`.

`IMSL_GOOD_DIGIT`, *int* `ndigit` (Input)
Number of good digits in the function. The default is machine dependent.

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
Maximum number of iterations.
Default: `max_itn` = 100

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations.
Default: `max_fcn` = 400

`IMSL_MAX_GRAD`, *int* `max_grad` (Input)
Maximum number of gradient evaluations.
Default: `max_grad` = 400

`IMSL_INIT_HESSIAN`, *int* `ihess` (Input)
Hessian initialization parameter. If `ihess` is zero, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing

$$max\left(\left|f(t)\right|, f_s\right) * s_i^2$$

on the diagonal where t = `xguess`, f_s = `fscale`, and s = `xscale`.
Default: `ihess` = 0

`IMSL_RETURN_USER`, *float* `x[]` (Output)
User-supplied array with `n` components containing the computed solution.

`IMSL_FVALUE`, *float* `*fvalue` (Output)
Address to store the value of the function at the computed solution.

Description

The function `f_min_uncon_multivar` uses a quasi-Newton method to find the minimum of a function $f(x)$ of n variables. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point x_c, the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where B is a positive definite approximation of the Hessian, and g_c is the gradient evaluated at x_c. A line search is then used to find a new point

$$x_n = x_c + \lambda d, \lambda > 0$$

such that

$$f(x_n) \leq f(x_c) + \alpha g^T d, \quad \alpha \in (0, 0.5)$$

Finally, the optimality condition $||g(x)|| \leq \varepsilon$ is checked where ε is a gradient tolerance.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

In this implementation, the first stopping criterion for `imsl_f_min_uncon_multivar` occurs when the norm of the gradient is less than the given gradient tolerance `grad_tol`. The second stopping criterion for `imsl_f_min_uncon_multivar` occurs when the scaled distance between the last two steps is less than the step tolerance `step_tol`.

Since by default, a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended; the keyword `IMSL_GRAD` should be used to provide more accurate gradient evaluation.

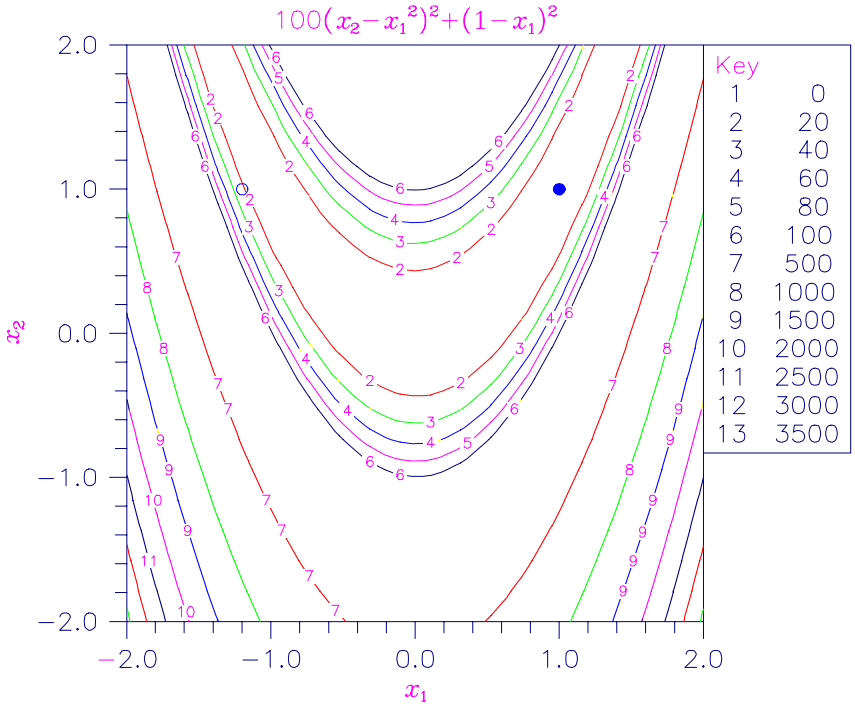


Figure 8-1 Plot of the Rosenbrock Function

Examples

Example 1

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2$$

is minimized. In the following plot, the solid circle marks the minimum.

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

void main()
{
        int             i, n=2;
        float           *result, fx;
        static float    rosbrk(int, float[]);
                                /* Minimize Rosenbrock function */

        result = imsl_f_min_uncon_multivar(rosbrk, n, 0);
        fx = rosbrk(n, result);

                                /* Print results */

        printf("  The solution is       ");
```

```
        for (i = 0; i < n; i++) printf("%8.3f", result[i]);
        printf("\n\n  The function value is %8.3f\n", fx);
}                                   /* end of main */


static float rosbrk(int n, float x[])
{
        float   f1, f2;

        f1 = x[1] - x[0]*x[0];
        f2 = 1.0 - x[0];

        return 100.0 * f1 * f1 + f2 * f2;
}                                   /* end of function */
```

Output

```
The solution is          1.000   1.000

The function value is    0.000
```

Example 2

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized with the initial guess $x = (-1.2, 1.0)$. The initial guess is marked with an open circle in the figure on page 391.

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

void main()
{
        int             i, n=2;
        float           *result, fx;
        static float    rosbrk(int, float[]);
        static void     rosgrd(int, float[], float[]);
        static float    xguess[2] = {-1.2e0, 1.0e0};
        static float    grad_tol = .0001;

/* Minimize Rosenbrock function using initial guesses of -1.2 and 1.0 */

        result = imsl_f_min_uncon_multivar(rosbrk, n, IMSL_XGUESS, xguess,
                                           IMSL_GRAD, rosgrd,
                                           IMSL_GRAD_TOL, grad_tol,
                                           IMSL_FVALUE, &fx, 0);

/* Print results */

        printf("  The solution is       ");
        for (i = 0; i < n; i++) printf("%8.3f", result[i]);
        printf("\n\n  The function value is %8.3f\n", fx);
}                                   /* End of main */
```

```
static float rosbrk(int n, float x[])
{
        float   f1, f2;

        f1 = x[1] - x[0]*x[0];
        f2 = 1.0e0 - x[0];

        return 100.0 * f1 * f1 + f2 * f2;
}                               /* End of function */

static void rosgrd(int n, float x[], float g[])
{

        g[0] = -400.0*(x[1]-x[0]*x[0])*x[0] - 2.0*(1.0-x[0]);
        g[1] = 200.0*(x[1]-x[0]*x[0]);

}                               /* End of function */
```

Output

```
  The solution is          1.000   1.000

  The function value is    0.000
```

Informational Errors

IMSL_STEP_TOLERANCE	Scaled step tolerance satisfied. The current point may be an approximate local solution, but it is also possible that the algorithm is making very slow progress and is not near a solution, or that `step_tol` is too big.

Warning Errors

IMSL_REL_FCN_TOLERANCE	Relative function convergence—Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance `rfcn_tol` = #.
IMSL_TOO_MANY_ITN	Maximum number of iterations exceeded.
IMSL_TOO_MANY_FCN_EVAL	Maximum number of function evaluations exceeded.
IMSL_TOO_MANY_GRAD_EVAL	Maximum number of gradient evaluations exceeded.
IMSL_UNBOUNDED	Five consecutive steps have been taken with the maximum step length.
IMSL_NO_FURTHER_PROGRESS	The last global step failed to locate a lower point than the current `x` value.

Fatal Errors

`IMSL_FALSE_CONVERGENCE`	False convergence—The iterates appear to be converging to a noncritical point. Possibly incorrect gradient information is used, or the function is discontinuous, or the other stopping tolerances are too tight.

nonlin_least_squares

Solve a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm.

Synopsis

#include `<imsl.h>`

float `*imsl_f_nonlin_least_squares` (*void* `fcn()`, *int* `m`, *int* `n`, …, 0)

The type *double* function is `imsl_d_nonlin_least_squares`.

Required Arguments

void `fcn` (*int* `m`, *int* `n`, *float* `x[]`, *float* `f[]`) (Input/Output)
User-supplied function to evaluate the function that defines the least-squares problem where `x` is a vector of length `n` at which point the function is evaluated, and `f` is a vector of length `m` containing the function values at point `x`.

int `m` (Input)
Number of functions.

int `n` (Input)
Number of variables where $n \leq m$.

Return Value

A pointer to the solution *x* of the nonlinear least-squares problem. 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_nonlin_least_squares` (*void* `fcn()`, *int* `m`, *int* `n`,
`IMSL_XGUESS`, *float* `xguess[]`,
`IMSL_JACOBIAN`, *void* `jacobian()`,
`IMSL_XSCALE`, *float* `xscale[]`,
`IMSL_FSCALE`, *float* `fscale[]`,
`IMSL_GRAD_TOL`, *float* `grad_tol`,
`IMSL_STEP_TOL`, *float* `step_tol`,
`IMSL_REL_FCN_TOL`, *float* `rfcn_tol`,

```
IMSL_ABS_FCN_TOL, float afcn_tol,
IMSL_MAX_STEP, float max_step,
IMSL_INIT_TRUST_REGION, float trust_region,
IMSL_GOOD_DIGIT, int ndigit,
IMSL_MAX_ITN, int max_itn,
IMSL_MAX_FCN, int max_fcn,
IMSL_MAX_JACOBIAN, int max_jacobian,
IMSL_INTERN_SCALE,
IMSL_TOLERANCE, float tolerance,
IMSL_RETURN_USER, float x[],
IMSL_FVEC, float **fvec,
IMSL_FVEC_USER, float fvec[],
IMSL_FJAC, float **fjac,
IMSL_FJAC_USER, float fjac[],
IMSL_FJAC_COL_DIM, int fjac_col_dim,
IMSL_RANK, int *rank,
IMSL_JTJ_INVERSE, float **jtj_inv,
IMSL_JTJ_INVERSE_USER, float jtj_inv[],
IMSL_JTJ_INV_COL_DIM, int jtj_inv_col_dim,
0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `n` components containing an initial guess.
Default: `xguess` = 0

`IMSL_JACOBIAN`, *void* `jacobian` (*int* `m`, *int* `n`, *float* `x[]`, *float* `fjac[]`, *int* `fjac_col_dim`)(Input)
User-supplied function to compute the Jacobian where `x` is a vector of length `n` at which point the Jacobian is evaluated, `fjac` is the computed $m \times n$ Jacobian at the point `x`, and `fjac_col_dim` is the column dimension of `fjac`. Note that each derivative $\partial f_i / \partial x_j$ should be returned in `fjac[(i-1)*fjac_col_dim+j-1]`

`IMSL_XSCALE`, *float* `xscale[]` (Input)
Array with `n` components containing the scaling vector for the variables. `xscale` is used mainly in scaling the gradient and the distance between two points. See keywords `IMSL_GRAD_TOL` and `IMSL_STEP_TOL` for more detail.
Default: `xscale[]` = 1

`IMSL_FSCALE`, *float* `fscale[]` (Input)
Array with `m` components containing the diagonal scaling matrix for the functions. The *i*-th component of `fscale` is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem.
Default: `fscale[]` = 1

`IMSL_GRAD_TOL`, *float* `grad_tol` (Input)

Scaled gradient tolerance. The *i*-th component of the scaled gradient at `x` is calculated as

$$\frac{|g_i| * max(|x_i|, 1/s_i)}{\frac{1}{2}\|F(x)\|_2^2}$$

where $g = \nabla F(x)$, s = `xscale`, and

$$\|F(x)\|_2^2 = \sum_{i=1}^{m} f_i(x)^2$$

Default:

$$\text{grad\_ tol} = \sqrt{\varepsilon}$$

$\sqrt[3]{\varepsilon}$ in double where ε is the machine precision

`IMSL_STEP_TOL`, *float* `step_tol` (Input)

Scaled step tolerance. The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_y|}{max(|x_i|, 1/s_i)}$$

where s = `xscale`.

Default: `step_tol` = $\varepsilon^{2/3}$ where ε is the machine precision.

`IMSL_REL_FCN_TOL`, *float* `rfcn_tol` (Input)

Relative function tolerance.

Default: `rfcn_tol` = max (10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double, where ε is the machine precision

`IMSL_ABS_FCN_TOL`, *float* `afcn_tol` (Input)

Absolute function tolerance.

Default: `afcn_tol` = max (10^{-20}, ε^2), max (10^{-40}, ε^2) in double, where ε is the machine precision.

`IMSL_MAX_STEP`, *float* `max_step` (Input)

Maximum allowable step size.

Default: `max_step` = 1000 max (ε_1, ε_2) where,

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}, \varepsilon_2 = \|s\|_2$$

s = `xscale`, and t = `xguess`

`IMSL_INIT_TRUST_REGION`, *float* `trust_region` (Input)

Size of initial trust region radius. The default is based on the initial scaled Cauchy step.

`IMSL_GOOD_DIGIT`, *int* `ndigit` (Input)
Number of good digits in the function.
Default: machine dependent

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
Maximum number of iterations.
Default: `max_itn` = 100

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations.
Default: `max_fcn` = 400

`IMSL_MAX_JACOBIAN`, *int* `max_jacobian` (Input)
Maximum number of Jacobian evaluations.
Default: `max_jacobian` = 400

`IMSL_INTERN_SCALE`
Internal variable scaling option. With this option, the values for xscale are set internally.

`IMSL_TOLERANCE`, *float* `tolerance` (Input)
The tolerance used in determining linear dependence for the computation of the inverse of $J^T J$. For `imsl_f_nonlin_least_squares`, if `IMSL_JACOBIAN` is specified, then `tolerance` = 100 × `imsl_d_machine`(4) is the default. Otherwise, the square root of `imsl_f_machine`(4) is the default. For `imsl_d_nonlin_least_ squares`, if `IMSL_JACOBIAN` is specified, then `tolerance` = 100 × `imsl_machine`(4) is the default. Otherwise, the square root of `imsl_d_machine`(4) is the default.
See **`imsl_f_machine`** (Chapter 12, “Utilities”).

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the computed solution.

`IMSL_FVEC`, *float* `**fvec` (Output)
The address of a pointer to a real array of length `m` containing the residuals at the approximate solution. On return, the necessary space is allocated by `imsl_f_nonlin_least_squares`. Typically, *float* `*fvec` is declared, and `&fvec` is used as an argument.

`IMSL_FVEC_USER`, *float* `fvec[]` (Output)
A user-allocated array of size `m` containing the residuals at the approximate solution.

`IMSL_FJAC`, *float* `**fjac` (Output)
The address of a pointer to an array of size $m \times n$ containing the Jacobian at the approximate solution. On return, the necessary space is allocated by `imsl_f_nonlin_least_squares`. Typically, *float* `*fjac` is declared, and `&fjac` is used as an argument.

`IMSL_FJAC_USER`, *float* `fjac[]` (Output)
A user-allocated array of size $m \times n$ containing the Jacobian at the approximate solution.

`IMSL_FJAC_COL_DIM`, *int* `fjac_col_dim` (Input)
The column dimension of `fjac`.
Default: `fjac_col_dim` = *n*

`IMSL_RANK`, *int* `*rank` (Output)
The rank of the Jacobian is returned in `*rank`.

`IMSL_JTJ_INVERSE`, *float* `**jtj_inv` (Output)
The address of a pointer to an array of size $n \times n$ containing the inverse matrix of $J^T J$ where the J is the final Jacobian. If $J^T J$ is singular, the inverse is a symmetric g_2 inverse of $J^T J$. (See **`imsl_f_lin_sol_nonnegdef`** in Chapter 1, "Linear Systems" for a discussion of generalized inverses and definition of the g_2 inverse.) On return, the necessary space is allocated by `imsl_f_nonlin_least_squares`.

`IMSL_JTJ_INVERSE_USER`, *float* `jtj_inv[]` (Output)
A user-allocated array of size $n \times n$ containing the inverse matrix of $J^T J$ where the J is the Jacobian at the solution.

`IMSL_JTJ_INV_COL_DIM`, *int* `jtj_inv_col_dim` (Input)
The column dimension of `jtj_inv`.
Default: `jtj_inv_col_dim` = *n*

Description

The function `imsl_f_nonlin_least_squares` is based on the MINPACK routine LMDER by Moré et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least-squares problems. The problem is stated as follows:

$$min \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

where $m \geq n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach,

$$\min_{x \in \mathbf{R}^n} \left\| F(x_c) + J(x_c)(x_n - x_c) \right\|_2$$
$$\text{subject to } \left\| x_n - x_c \right\|_2 \leq \delta_c$$

to get a new point x_n, which is computed as

$$x_n = x_c - (J(x_c)^T J(x_c) + \mu_c I)^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \geq \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$ and $\mu_c > 0$, otherwise. The value μ_c is defined by the function. The vector and matrix $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point x_c, respectively. This function is repeated until the stopping criteria are satisfied.

The first stopping criterion for `imsl_f_nonlin_least_squares` occurs when the norm of the function is less than the absolute function tolerance `fcn_tol`. The second stopping criterion occurs when the norm of the scaled gradient is less than the given

gradient tolerance `grad_tol`. The third stopping criterion for `imsl_f_nonlin_least_squares` occurs when the scaled distance between the last two steps is less than the step tolerance `step_tol`. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

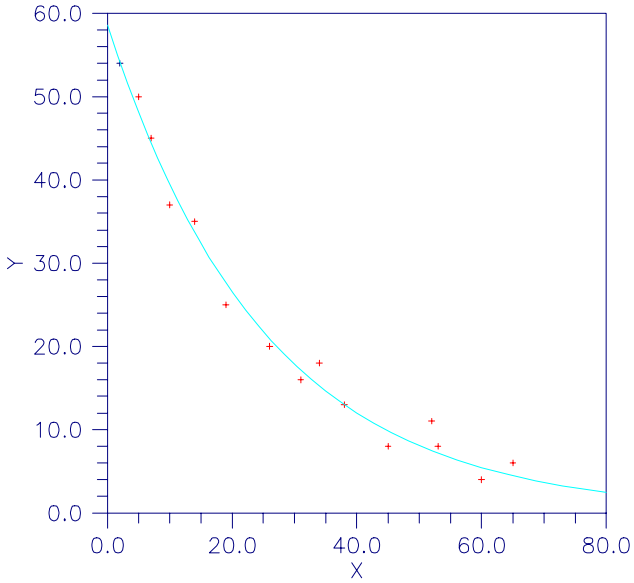


Figure 8-2 Plot of the Nonlinear Fit

Examples

Example 1

In this example, the nonlinear data-fitting problem found in Dennis and Schnabel (1983, p. 225),

$$min\frac{1}{2}\sum_{i=1}^{3} f_i(x)^2$$

where

$$f_i(x) = e^{t_i x} - y_i$$

is solved with the data t = (1, 2, 3) and y = (2, 4, 3).

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

void            fcn(int, int, float[], float[]);

void main()
{
        int             m=3, n=1;
        float           *result, fx[3];
```

```
        result = imsl_f_nonlin_least_squares(fcn, m, n, 0);
        fcn(m, n, result, fx);

/* Print results */

        imsl_f_write_matrix("The solution is", 1, 1, result, 0);
        imsl_f_write_matrix("The function values are", 1, 3, fx, 0);
}                                /* End of main */


void fcn(int m, int n, float x[], float f[])
{
     int   i;
     float y[3] = {2.0, 4.0, 3.0};
     float t[3] = {1.0, 2.0, 3.0};

     for (i=0; i<m; i++)
          f[i] =  exp(x[0]*t[i]) - y[i];

}                                /* End of function */
```

Output

```
 The solution is
      0.4401

      The function values are
         1           2           3
    -0.447      -1.589       0.744
```

Example 2

In this example, `imsl_f_nonlin_least_squares` is first invoked to fit the following nonlinear regression model discussed by Neter et al. (1983, pp. 475–478):

$$y_i = \theta_1 e^{\theta_2 x_i} + \varepsilon_i \quad i = 1, 2, \ldots, 15$$

where the ε_i's are independently distributed each normal with mean zero and variance σ^2. The estimate of σ^2 is then computed as

$$s^2 = \frac{\sum_{i=1}^{15} e_i^2}{15 - \text{rank}(J)}$$

where e_i is the i-th residual and J is the Jacobian. The estimated asymptotic variance-covariance matrix of $\hat{\theta}_1$ and $\hat{\theta}_2$ is computed as

$$\text{est. asy. var}\left(\hat{\theta}\right) = s^2\left(J^T J\right)^{-1}$$

Finally, the diagonal elements of this matrix are used together with `imsl_f_t_inverse_cdf` (Chapter 9) to compute 95% confidence intervals on θ_1 and θ_2.

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

void            exampl(int, int, float[], float[]);
```

```
void main()
{
        int             i, j, m=15, n=2, rank;
        float           a, *result, e[15], jtj_inv[4], s2, dfe;
        char            *fmt="%12.5e";
        static float    xguess[2] = {60.0, -0.03};
        static float    grad_tol = 1.0e-3;


        result = imsl_f_nonlin_least_squares(exampl, m, n,
                                        IMSL_XGUESS, xguess,
                                        IMSL_GRAD_TOL, grad_tol,
                                        IMSL_FVEC_USER, e,
                                        IMSL_RANK, &rank,
                                        IMSL_JTJ_INVERSE_USER, jtj_inv,
                                        0);
        dfe = (float) (m - rank);
        s2  = 0.0;
        for (i=0; i<m; i++)
            s2 += e[i] * e[i];
        s2 = s2 / dfe;
        j = n * n;
        for (i=0; i<j; i++)
             jtj_inv[i] = s2 * jtj_inv[i];
                                /* Print results */

        imsl_f_write_matrix (
                    "Estimated Asymptotic Variance-Covariance Matrix",
                    2, 2, jtj_inv, IMSL_WRITE_FORMAT, fmt, 0);
        printf(" \n           95%% Confidence Intervals   \n    ");
        printf("   Estimate  Lower Limit  Upper Limit \n ");
        for (i=0; i<n; i++) {
            j = i * (n+1);
            a = imsl_f_t_inverse_cdf (0.975, dfe) * sqrt(jtj_inv[j]);
            printf("  %10.3f %12.3f %12.3f \n", result[i],
                    result[i] - a, result[i] + a);
      }
}                               /* End of main */


void exampl(int m, int n, float x[], float f[])
{
     int   i;
     float y[15]     = { 54.0, 50.0, 45.0, 37.0, 35.0, 25.0, 20.0, 16.0,
                         18.0, 13.0,  8.0, 11.0,  8.0,  4.0,  6.0 };
     float xdata[15] = {  2.0,  5.0,  7.0, 10.0, 14.0, 19.0, 26.0, 31.0,
                         34.0, 38.0, 45.0, 52.0, 53.0, 60.0, 65.0 };


     for (i=0; i<m; i++)
          f[i] = y[i] - x[0]*exp(x[1]*xdata[i]);

}                               /* End of function */
```

Output

```
Estimated Asymptotic Variance-Covariance Matrix
                      1             2
         1   2.17524e+00  -1.80141e-03
```

```
   2  -1.80141e-03   2.97216e-06

    95% Confidence Intervals
Estimate  Lower Limit  Upper Limit
 58.608       55.422       61.795
 -0.040       -0.043       -0.036
```

Informational Errors

IMSL_STEP_TOLERANCE	Scaled step tolerance satisfied. The current point may be an approximate local solution, but it is also possible that the algorithm is making very slow progress and is not near a solution, or that `step_tol` is too big.

Warning Errors

IMSL_LITTLE_FCN_CHANGE	Both the actual and predicted relative reductions in the function are less than or equal to the relative function tolerance.
IMSL_TOO_MANY_ITN	Maximum number of iterations exceeded.
IMSL_TOO_MANY_FCN_EVAL	Maximum number of function evaluations exceeded.
IMSL_TOO_MANY_JACOBIAN_EVAL	Maximum number of Jacobian evaluations exceeded.
IMSL_UNBOUNDED	Five consecutive steps have been taken with the maximum step length.

Fatal Errors

IMSL_FALSE_CONVERGE	The iterates appear to be converging to a noncritical point.

lin_prog

Solves a linear programming problem using the revised simplex algorithm.

Synopsis

#include `<imsl.h>`

float `*imsl_f_lin_prog` (*int* `m`, *int* `n`, *float* `a[]`, *float* `b[]`,
float `c[]`, …, 0)

The type *double* function is `imsl_d_lin_prog`.

Required Arguments

int `m` (Input)
Number of constraints.

int n (Input)
Number of variables.

float a[] (Input)
Array of size $m \times n$ containing a matrix with coefficients of the m constraints.

float b[] (Input)
Array with m components containing the right-hand side of the constraints; if there are limits on both sides of the constraints, then b contains the lower limit of the constraints.

float c[] (Input)
Array with n components containing the coefficients of the objective function.

Return Value

A pointer to the solution *x* of the linear programming problem. 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_lin_prog (int m, int n, float a[], float b[], float c[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_UPPER_LIMIT, float bu[],
        IMSL_CONSTR_TYPE, int irtype[],
        IMSL_LOWER_BOUND, float xlb[],
        IMSL_UPPER_BOUND, float xub[],
        IMSL_MAX_ITN, int max_itn,
        IMSL_OBJ, float *obj,
        IMSL_RETURN_USER, float x[],
        IMSL_DUAL, float **y,
        IMSL_DUAL_USER, float y[],
        0)
```

Optional Arguments

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

IMSL_UPPER_LIMIT, *float* bu[] (Input)
Array with m components containing the upper limit of the constraints that have both the lower and the upper bounds. If no such constraint exists, then bu is not needed.

IMSL_CONSTR_TYPE, *int* irtype[] (Input)
Array with m components indicating the types of general constraints in the matrix a. Let $r_i = a_{i1}x_1 + \ldots + a_{in}x_n$. Then, the value of irtype(i) signifies the following:

irtype(i)	Constraint
0	$r_i = b_i$
1	$r_i \le bu_i$
2	$r_i \ge b_i$
3	$b_i \le r_i \le bu_i$

Default: `irtype` = 0

`IMSL_LOWER_BOUND`, *float* `xlb[]` (Input)
Array with `n` components containing the lower bound on the variables. If there is no lower bound on a variable, then 10^{30} should be set as the lower bound.
Default: `xlb` = 0

`IMSL_UPPER_BOUND`, *float* `xub[]` (Input)
Array with `n` components containing the upper bound on the variables. If there is no upper bound on a variable, then -10^{30} should be set as the upper bound.
Default: `xub` = ∞

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
Maximum number of iterations.
Default: `max_itn` = 10000

`IMSL_OBJ`, *float* `*obj` (Output)
Optimal value of the objective function.

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the primal solution.

`IMSL_DUAL`, *float* `**y` (Output)
The address of a pointer `y` to an array with `m` components containing the dual solution. On return, the necessary space is allocated by `imsl_f_lin_prog`. Typically, *float* `*y` is declared, and `&y` is used as an argument.

`IMSL_DUAL_USER`, *float* `y[]` (Output)
A user-allocated array of size `m`. On return, `y` contains the dual solution.

Description

The function `imsl_f_lin_prog` uses a revised simplex method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} c^T x \quad \text{subject to } b_l \le A_x \le b_u$$
$$x_l \le x \le x_u$$

where c is the objective coefficient vector, A is the coefficient matrix, and the vectors b_l, b_u, x_l, and x_u are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

Examples

Example 1

The linear programming problem in the standard form

$$\begin{aligned} & min\ f(x) = -x_1 - 3x_2 \\ \text{subject to } & x_1 + x_2 + x_3 = 1.5 \\ & x_1 + x_2 - x_4 = 0.5 \\ & x_1 + x_5 = 1.0 \\ & x_2 + x_6 = 1.0 \\ & x_i \geq 0, \text{ for } i = 1, \ldots, 6 \end{aligned}$$

is solved.

```
#include <imsl.h>

main()
{
    int         m = 4;
    int         n = 6;
    float       a[ ] = {1.0, 1.0, 1.0,  0.0, 0.0, 0.0,
                        1.0, 1.0, 0.0, -1.0, 0.0, 0.0,
                        1.0, 0.0, 0.0,  0.0, 1.0, 0.0,
                        0.0, 1.0, 0.0,  0.0, 0.0, 1.0};
    float       b[ ] = {1.5, 0.5, 1.0, 1.0};
    float       c[ ] = {-1.0, -3.0, 0.0, 0.0, 0.0, 0.0};
    float       *x;
                                /* Solve the LP problem  */

    x = imsl_f_lin_prog (m, n, a, b, c, 0);
                                /* Print x */
    imsl_f_write_matrix ("x", 1, 6, x, 0);
}
```

Output

```
                                x
        1           2           3           4           5           6
      0.5         1.0         0.0         1.0         0.5         0.0
```

Example 2

The linear programming problem in the previous example can be formulated as follows:

$$\begin{aligned} & \min f(x) = -x_1 - 3x_2 \\ \text{subject to } & 0.5 \leq x_1 + x_2 \leq 1.5 \\ & 0 \leq x_1 \leq 1.0 \\ & 0 \leq x_2 \leq 1.0 \end{aligned}$$

This problem can be solved more efficiently.

```
#include <imsl.h>

main()
{
    int         irtype[ ] = {3};
    int         m = 1;
    int         n = 2;
```

```
    float       xub[ ] = {1.0, 1.0};
    float       a[ ]   = {1.0, 1.0};
    float       b[ ]   = {0.5};
    float       bu[ ]  = {1.5};
    float       c[ ]   = {-1.0, -3.0};
    float       d[1];
    float       obj, *x;
                                /* Solve the LP problem  */

    x = imsl_f_lin_prog (m, n, a, b, c,
                         IMSL_UPPER_LIMIT, bu,
                         IMSL_CONSTR_TYPE, irtype,
                         IMSL_UPPER_BOUND, xub,
                         IMSL_DUAL_USER, d,
                         IMSL_OBJ, &obj,
                         0);
                                /* Print x */
    imsl_f_write_matrix ("x", 1, 2, x, 0);
                                /* Print d */
    imsl_f_write_matrix ("d", 1, 1, d, 0);
    printf("\n obj = %g \n", obj);
}
```

Output

```
           x
       1           2
     0.5         1.0

     d
        -1

 obj = -3.5
```

Warning Errors

IMSL_PROB_UNBOUNDED	The problem is unbounded.
IMSL_TOO_MANY_ITN	Maximum number of iterations exceeded.
IMSL_PROB_INFEASIBLE	The problem is infeasible.

Fatal Errors

IMSL_NUMERIC_DIFFICULTY	Numerical difficulty occurred. If float is currently being used, using double may help.
IMSL_BOUNDS_INCONSISTENT	The bounds are inconsistent.

quadratic_prog

Solves a quadratic programming problem subject to linear equality or inequality constraints.

Synopsis

#include <imsl.h>

float *imsl_f_quadratic_prog (*int* m, *int* n, *int* meq, *float* a[], *float* b[], *float* g[], *float* h[], ..., 0)

The type *double* function is `imsl_d_quadratic_prog`.

Required Arguments

int `m` (Input)
The number of linear constraints.

int `n` (Input)
The number of variables.

int `meq` (Input)
The number of linear equality constraints.

float `a[]` (Input)
Array of size $m \times n$ containing the equality constraints in the first `meq` rows, followed by the inequality constraints.

float `b[]` (Input)
Array with `m` components containing right-hand sides of the linear constraints.

float `g[]` (Input)
Array with `n` components containing the coefficients of the linear term of the objective function.

float `h[]` (Input)
Array of size $n \times n$ containing the Hessian matrix of the objective function. It must be symmetric positive definite. If `h` is not positive definite, the algorithm attempts to solve the QP problem with `h` replaced by `h + diag*` *I* such that `h + diag*` *I* is positive definite.

Return Value

A pointer to the solution *x* of the QP problem. 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_quadratic_prog (int m, int n, int meq, float a[], float b[],
        float g[], float h[],
        IMSL_A_COL_DIM, int a_col_dim,
        IMSL_H_COL_DIM, int h_col_dim,
        IMSL_RETURN_USER, float x[],
        IMSL_DUAL, float **y,
        IMSL_DUAL_USER, float y[],
        IMSL_ADD_TO_DIAG_H, float *diag,
        IMSL_OBJ, float *obj,
        0)
```

Optional Arguments

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
Leading dimension of *A* exactly as specified in the dimension statement of the

calling program.
Default: `a_col_dim` = *n*

`IMSL_H_COL_DIM`, *int* `h_col_dim` (Input)
Leading dimension of `h` exactly as specified in the dimension statement of the calling program.
Default: `n_col_dim` = *n*

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the solution.

`IMSL_DUAL`, *float* `**y` (Output)
The address of a pointer `y` to an array with `n` components containing the Lagrange multiplier estimates. On return, the necessary space is allocated by `imsl_f_quadratic_prog`. Typically, *float* `*y` is declared, and `&y` is used as an argument.

`IMSL_DUAL_USER`, *float* `y[]` (Output)
A user-allocated array with `n` components. On return, `y` contains the Lagrange multiplier estimates.

`IMSL_ADD_TO_DIAG_H`, *float* `*diag` (Output)
Scalar equal to the multiple of the identity matrix added to `h` to give a positive definite matrix.

`IMSL_OBJ`, *float* `*obj` (Output)
The optimal object function found.

Description

The function `imsl_f_quadratic_prog` is based on M.J.D. Powell's implementation of the Goldfarb and Idnani dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints (Goldfarb and Idnani 1983); i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} g^T x + \frac{1}{2} x^T H x$$
$$\text{subject to} \quad A_1 x = b_1$$
$$A_2 x \geq b_2$$

given the vectors b_1, b_2, and g, and the matrices H, A_1, and A_2. H is required to be positive definite. In this case, a unique x solves the problem or the constraints are inconsistent. If H is not positive definite, a positive definite perturbation of H is used in place of H. For more details, see Powell (1983, 1985).

If a perturbation of H, $H + \alpha I$, is used in the QP problem, then $H + \alpha I$ also should be used in the definition of the Lagrange multipliers.

Examples

Example 1

The quadratic programming problem

$$min\ f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1$$

$$\text{subject to} \quad x_1 + x_2 + x_3 + x_4 + x_5 = 5$$

$$x_3 - 2x_4 - 2x_5 = -3$$

is solved.

```
#include <imsl.h>

main()
{
    int         m = 2;
    int         n = 5;
    int         meq = 2;
    float       *x;
    float       h[ ] =  {2.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 2.0,-2.0, 0.0, 0.0,
                         0.0,-2.0, 2.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 2.0,-2.0,
                         0.0, 0.0, 0.0,-2.0, 2.0};
    float       a[ ] =  {1.0, 1.0, 1.0, 1.0, 1.0,
                         0.0, 0.0, 1.0,-2.0,-2.0};
    float       b[ ] =  {5.0, -3.0};
    float       g[ ] =  {-2.0, 0.0, 0.0, 0.0, 0.0};
                                /* Solve the QP problem  */
    x = imsl_f_quadratic_prog (m, n, meq, a, b, g, h, 0);
                                /* Print x */
    imsl_f_write_matrix ("x", 1, 5, x, 0);
}
```

Output

```
                    x
        1           2           3           4           5
        1           1           1           1           1
```

Example 2

Another quadratic programming problem

$$min\ f(x) = x_1^2 + x_2^2 + x_3^2 \qquad \text{subject to } x_1 + 2x_2 - x_3 = 4$$

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

is solved.

```
#include <imsl.h>

float    h[ ] = {2.0, 0.0, 0.0,
                 0.0, 2.0, 0.0,
                 0.0, 0.0, 2.0};
float    a[ ] = {1.0,  2.0, -1.0,
                 1.0, -1.0,  1.0};
float    b[ ] = {4.0, -2.0};
float    g[ ] = {0.0,  0.0, 0.0};
main()
```

```
{
    int           m = 2;
    int           n = 3;
    int           meq = 2;
    float         obj;
    float         d[2];
    float        *x;
                                    /* Solve the QP problem  */

    x = imsl_f_quadratic_prog (m, n, meq, a, b, g, h,
            IMSL_OBJ,           &obj,
            IMSL_DUAL_USER,   d,
            0);
                                    /* Print x */
    imsl_f_write_matrix ("x", 1, 3, x, 0);
                                    /* Print d */
    imsl_f_write_matrix ("d", 1, 2, d, 0);
    printf("\n obj = %g \n", obj);
}
```

Output

```
                x
        1           2           3
    0.286       1.429      -0.857

              d
        1           2
    1.143      -0.571

 obj = 2.85714
```

Warning Errors

IMSL_NO_MORE_PROGRESS	Due to the effect of computer rounding error, a change in the variables fail to improve the objective function value; usually the solution is close to optimum.

Fatal Errors

IMSL_SYSTEM_INCONSISTENT	The system of equations is inconsistent. There is no solution.

min_con_gen_lin

Minimizes a general objective function subject to linear equality/inequality constraints.

Synopsis

#include <imsl.h>

float *imsl_f_min_con_gen_lin (*void* fcn(), *int* nvar, *int* ncon, *int* neq, *float* a[], *float* b[], *float* xlb[], *float* xub[], ..., 0)

The type *double* function is `imsl_d_min_con_gen_lin`.

Required Arguments

void `fcn` (*int* `n`, *float* `x[]`, *float* `*f` (Input/Output)
User-supplied function to evaluate the function to be minimized. Argument `x` is a vector of length `n` at which point the function is evaluated, and `f` contains the function value at `x`.

int `nvar` (Input)
Number of variables.

int `ncon` (Input)
Number of linear constraints (excluding simple bounds).

int `neq` (Input)
Number of linear equality constraints.

float `a[]` (Input)
Array of size `ncon` × `nvar` containing the equality constraint gradients in the first `neq` rows followed by the inequality constraint gradients.

float `b[]` (Input)
Array of size `ncon` containing the right-hand sides of the linear constraints. Specifically, the constraints on the variables x_i, $i = 0$, `nvar` $- 1$, are $a_{k,0}x_0 + \ldots + a_{k,nvar-1}x_{nvar-1} = b_k$, $k = 0, \ldots,$ `neq` $- 1$ and $a_{k,0}x_0 + \ldots + a_{k,nvar-1}x_{nvar-1} \leq b_k$, $k =$ `neq`, …, `ncon` $- 1$. Note that the data that define the equality constraints come before the data of the inequalities.

float `xlb[]` (Input)
Array of length `nvar` containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set `xub`[i] = `xub`[i] to freeze the i-th variable. Specifically, these simple bounds are `xlb`[i] $\leq x_i$, for $i = 1, \ldots,$ `nvar`.

float `xub[]` (Input)
Array of length `nvar` containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. Specifically, these simple bounds are $x_i \leq$ `xub`[i], for $i = 1,$ `nvar`.

Return Value

A pointer to the solution x. 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_min_con_gen_lin` (*void* `fcn()`, *int* `nvar`, *int* `ncon`, *int* `a`,
float `b`, *float* `xlb[]`, *float* `xub[]`,
`IMSL_XGUESS`, *float* `xguess[]`,

```
IMSL_GRADIENT, void gradient(),
IMSL_MAX_FCN, int max_fcn,
IMSL_NUMBER_ACTIVE_CONSTRAINTS, int *nact,
IMSL_ACTIVE_CONSTRAINT, int **iact,
IMSL_ACTIVE_CONSTRAINT_USER, int *iact_user,
IMSL_LAGRANGE_MULTIPLIERS, float **lagrange,
IMSL_LAGRANGE_MULTIPLIERS_USER, float *lagrange_user,
IMSL_TOLERANCE, float tolerance,
IMSL_OBJ, float *obj,
IMSL_RETURN_USER, float x[],
0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `n` components containing an initial guess.
Default: `xguess` = 0

`IMSL_GRADIENT`, *void* `gradient` (*int* `n`, *float* `x[]`, *float* `g[]`) (Input)
User-supplied function to compute the gradient at the point `x`, where `x` is a vector of length `n`, and `g` is the vector of length `n` containing the values of the gradient of the objective function.

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations.
Default: `max_fcn` = 400

`IMSL_NUMBER_ACTIVE_CONSTRAINTS`, *int* `*nact` (Output)
Final number of active constraints.

`IMSL_ACTIVE_CONSTRAINT`, *int* `**iact` (Output)
The address of a pointer to an *int*, which on exit, points to an array containing the `nact` indices of the final active constraints.

`IMSL_ACTIVE_CONSTRAINT_USER`, *int* `*iact_user` (Output)
A user-supplied array of length at least `ncon` + 2`*nvar` containing the indices of the final active constraints in the first `nact` locations.

`IMSL_LAGRANGE_MULTIPLIERS`, *float* `**lagrange` (Output)
The address of a pointer, which on exit, points to an array containing the Lagrange multiplier estimates of the final active constraints in the first `nact` locations.

`IMSL_LAGRANGE_MULTIPLIERS_USER`, *float* `*lagrange_user` (Output)
A user-supplied array of length at least `nvar` containing the Lagrange multiplier estimates of the final active constraints in the first `nact` locations.

`IMSL_TOLERANCE`, *float* `tolerance` (Input)
The nonnegative tolerance on the first order conditions at the calculated solution.
Default: `tolerance` = $\sqrt{\varepsilon}$, where ε is machine epsilon

`IMSL_OBJ`, *float* `*obj` (Output)
The value of the objective function.

`IMSL_RETURN_USER`, *float* `x[]` (Output)
User-supplied array with `nvar` components containing the computed solution.

Description

The function `imsl_f_min_con_gen_lin` is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\min f(x)$$

subject to

$$A_1 x = b_1$$
$$A_2 x \le b_2$$
$$x_l \le x \le x_u$$

given the vectors b_1, b_2, x_l ,and x_u and the matrices A_1 and A_2.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise x^0, the initial guess, to satisfy

$$A_1 x = b_1$$

Next, x^0 is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible x^k, let J_k be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let I_k be the set of indices of active constraints. The following quadratic programming problem

$$min\ f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

subject to

$$a_j d = 0,\ j \in I_k$$

$$a_j d \le 0,\ j \in J_k$$

is solved to get (d^k, λ^k) where a_j is a row vector representing either a constraint in A_1 or A_2 or a bound constraint on x. In the latter case, the $a_j = e_i$ for the bound constraint $x_i \le (x_u)_i$ and $a_j = -e_i$ for the constraint $-x_i \le (x_l)_i$. Here, e_i is a vector with 1 as the i-th component, and zeros elsewhere. Variables λ^k are the Lagrange multipliers, and B^k is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction d^k is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f(x^k + \alpha^k d^k) \le f(x^k) + 0.1\ \alpha^k\ (d^k)^T\ \nabla f(x^k)$$

and

$$(d^K)^T \nabla f(x^k + \alpha^k d^k) \ge 0.7\ (d^k)^T \nabla f(x^K)$$

The main idea in forming the set J_k is that, if any of the equality constraints restricts the step-length α^k, then its index is not in J_k. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation B^K, is updated by the BFGS formula, if the condition

$$(d^K)^T \nabla f(x^k + \alpha^k d^k) - \nabla f(x^k) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\| \nabla f(x^k) - A^k \lambda^K \|_2 \leq \tau$$

is satisfied. Here τ is the supplied tolerance. For more details, see Powell (1988, 1989).

Since a finite difference method is used to approximate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, if the gradient can be easily provided, the option `IMSL_GRADIENT` should be used.

Example 1

In this example, the problem

$$\begin{gathered} \min f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1 \\ \text{subject to } x_1 + x_2 + x_3 + x_4 + x_5 = 5 \\ x_3 - 2x_4 - 2x_5 = -3 \\ 0 \leq x \leq 10 \end{gathered}$$

is solved.

```
#include <imsl.h>

main()
{
        void            fcn(int, float *, float *);
        int             neq = 2;
        int             ncon = 2;
        int             nvar = 5;
        float           obj;

        float           a[] = {1.0, 1.0, 1.0, 1.0, 1.0,
                               0.0, 0.0, 1.0, -2.0, -2.0};
        float           b[] = {5.0, -3.0};
        float           xlb[] = {0.0, 0.0, 0.0, 0.0, 0.0};
        float           xub[] = {10.0, 10.0, 10.0, 10.0, 10.0};
        float          *x;

        x = imsl_f_min_con_gen_lin(fcn, nvar, ncon, neq, a, b, xlb, xub,
                                   0);

        imsl_f_write_matrix("Solution", 1, nvar, x, 0);
}
```

```
void fcn(int n, float *x, float *f)
{
        *f = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] + x[3]*x[3] + x[4]*x[4]
             - 2.0*x[1]*x[2] - 2.0*x[3] * x[4] - 2.0*x[0];
}
```

Output

```
                    Solution
         1          2          3          4          5
         1          1          1          1          1
```

Example 2

In this example, the problem from Schittkowski (1987)

$$\min f(x) = -x_0 x_1 x_2$$

$$\text{subject to } -x_0 - 2x_1 - 2x_2 \le 0$$

$$x_0 + 2x_1 + 2x_2 \le 72$$

$$0 \le x_0 \le 20$$

$$0 \le x_1 \le 11$$

$$0 \le x_2 \le 42$$

is solved with an initial guess of $x_0 = 10$, $x_1 = 10$ and $x_2 = 10$.

```
#include <imsl.h>

main()
{
        void            fcn(int, float *, float *);
        void            grad(int, float *, float *);
        int             neq = 0;
        int             ncon = 2;
        int             nvar = 3;
        int             lda = 2;
        float           obj, x[3];
        float           a[] = {-1.0, -2.0, -2.0,
                               1.0, 2.0, 2.0};
        float           xlb[] = {0.0, 0.0, 0.0};
        float           xub[] = {20.0, 11.0, 42.0};
        float           xguess[] = {10.0, 10.0, 10.0};
        float           b[] = {0.0, 72.0};


        imsl_f_min_con_gen_lin(fcn, nvar, ncon, neq, a, b, xlb, xub,
                               IMSL_GRADIENT, grad,
                               IMSL_XGUESS, xguess,
                               IMSL_OBJ, &obj,
                               IMSL_RETURN_USER, x,
                               0);

        imsl_f_write_matrix("Solution", 1, nvar, x, 0);
        printf("Objective value = %f\n", obj);
}

void fcn(int n, float *x, float *f)
{
```

```
        *f = -x[0] * x[1] * x[2];
}

void grad(int n, float *x, float *g)
{
        g[0] = -x[1]*x[2];
        g[1] = -x[0]*x[2];
        g[2] = -x[0]*x[1];
}
```

Output

```
          Solution
 1           2           3
20          11          15

Objective value = -3300.000000
```

bounded_least_squares

Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.

Synopsis

#include `<imsl.h>`

float `*imsl_f_bounded_least_squares` (*void* `fcn()`, *int* `m`, *int* `n`, *int* `ibtype`, *float* `xlb[]`, *float* `xub[]`, ..., 0)

The type *double* function is `imsl_d_bounded_least_squares`.

Required Arguments

void `fcn` (*int* `m`, *int* `n`, *float* `x[]`, *float* `f[]`) (Input/Output)
User-supplied function to evaluate the function that defines the least-squares problem where `x` is a vector of length `n` at which point the function is evaluated, and `f` is a vector of length `m` containing the function values at point `x`.

int `m` (Input)
Number of functions.

int `n` (Input)
Number of variables where $n \leq m$.

int `ibtype` (Input)
Scalar indicating the types of bounds on the variables.

ibtype	Action
0	User will supply all the bounds.
1	All variables are nonnegative
2	All variables are nonpositive.

ibtype	Action
3	User supplies only the bounds on 1st variable, all other variables will have the same bounds

float `xlb[]` (Input, Output, or Input/Output)
Array with `n` components containing the lower bounds on the variables. (Input, if `ibtype` = 0; output, if `ibtype` = 1 or 2; Input/Output, if `ibtype` = 3)

If there is no lower bound on a variable, then the corresponding `xlb` value should be set to -10^6.

float `xub[]` (Input, Output, or Input/Output)
Array with `n` components containing the upper bounds on the variables. (Input, if `ibtype` = 0; output, if `ibtype` 1 or 2; Input/Output, if `ibtype` = 3)

If there is no upper bound on a variable, then the corresponding `xub` value should be set to 10^6.

Return Value

A pointer to the solution *x* of the nonlinear least-squares problem. 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_bounded_least_squares (void fcn(), int m, int n,
        int ibtype, float xlb[], float xub[],
        IMSL_XGUESS, float xguess[],
        IMSL_JACOBIAN, void jacobian(),
        IMSL_XSCALE, float xscale[],
        IMSL_FSCALE, float fscale[],
        IMSL_GRAD_TOL, float grad_tol,
        IMSL_STEP_TOL, float step_tol,
        IMSL_REL_FCN_TOL, float rfcn_tol,
        IMSL_ABS_FCN_TOL, float afcn_tol,
        IMSL_MAX_STEP, float max_step,
        IMSL_INIT_TRUST_REGION, float trust_region,
        IMSL_GOOD_DIGIT, int ndigit,
        IMSL_MAX_ITN, int max_itn,
        IMSL_MAX_FCN, int max_fcn,
        IMSL_MAX_JACOBIAN, int max_jacobian,
        IMSL_INTERN_SCALE,
        IMSL_RETURN_USER, float x[],
        IMSL_FVEC, float **fvec,
        IMSL_FVEC_USER, float fvec[],
        IMSL_FJAC, float **fjac,
        IMSL_FJAC_USER, float fjac[],
```

IMSL_FJAC_COL_DIM, *int* fjac_col_dim,
0)

Optional Arguments

IMSL_XGUESS, *float* xguess[] (Input)
Array with n components containing an initial guess.
Default: xguess = 0

IMSL_JACOBIAN, *void* jacobian (*int* m, *int* n, *float* x[], *float* fjac[], *int* fjac_col_dim) (Input)
User-supplied function to compute the Jacobian where x is a vector of length n at which point the Jacobian is evaluated, fjac is the computed $m \times n$ Jacobian at the point x, and fjac_col_dim is the column dimension of fjac. Note that each derivative f_i/x_j should be returned in fjac[(i−1)*fjac_col_dim+j −1].

IMSL_XSCALE, *float* xscale[] (Input)
Array with n components containing the scaling vector for the variables. Argument xscale is used mainly in scaling the gradient and the distance between two points. See keywords IMSL_GRAD_TOL and IMSL_STEP_TOL for more detail.
Default: xscale[] = 1

IMSL_FSCALE, *float* fscale[] (Input)
Array with m components containing the diagonal scaling matrix for the functions. The *i*-th component of fscale is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem.
Default: fscale[] = 1

IMSL_GRAD_TOL, *float* grad_tol (Input)
Scaled gradient tolerance. The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * max(|x_i|, 1/s_i)}{\frac{1}{2}||F(x)||_2^2}$$

where $g = \nabla F(x)$, s = xscale, and

$$\left\|F(x)\right\|_2^2 = \sum_{i=1}^{m} f_i(x)^2$$

Default: grad_tol = $\sqrt{\varepsilon}, \sqrt[3]{\varepsilon}$ in double where ε is the machine precision

IMSL_STEP_TOL, *float* step_tol (Input)
Scaled step tolerance. The *i*-th component of the scaled step between two points *x*, and *y*, is computed as

$$\frac{|x_i - y_y|}{max(|x_i|, 1/s_i)}$$

where s = `xscale`.
Default: `step_tol` = $\varepsilon^{2/3}$, where ε is the machine precision

`IMSL_REL_FCN_TOL`, *float* `rfcn_tol` (Input)
Relative function tolerance.
Default: `rfcn_tol` = max(10^{-10}, $\varepsilon^{2/3}$), max(10^{-20}, $\varepsilon^{2/3}$) in double, where ε is the machine precision

`IMSL_ABS_FCN_TOL`, *float* `afcn_tol` (Input)
Absolute function tolerance.
Default: `afcn_tol` = max(10^{-20}, ε^{2}), max(10^{-40}, ε^{2}) in double, where ε is the machine precision

`IMSL_MAX_STEP`, *float* `max_step` (Input)
Maximum allowable step size.
Default: `max_step` = 1000 max(ε_1, ε_2), where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}, \varepsilon_2 = ||s||_2$$

for s = `xscale` and t = `xguess`.

`IMSL_INIT_TRUST_REGION`, *float* `trust_region` (Input)
Size of initial trust region radius. The default is based on the initial scaled Cauchy step.

`IMSL_GOOD_DIGIT`, *int* `ndigit` (Input)
Number of good digits in the function.
Default: machine dependent

`IMSL_MAX_ITN`, *int* `max_itn` (Input)
Maximum number of iterations.
Default: `max_itn` = 100

`IMSL_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations.
Default: `max_fcn` = 400

`IMSL_MAX_JACOBIAN`, *int* `max_jacobian` (Input)
Maximum number of Jacobian evaluations.
Default: `max_jacobian` = 400

`IMSL_INTERN_SCALE`
Internal variable scaling option. With this option, the values for `xscale` are set internally.

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the computed solution.

`IMSL_FVEC`, *float* `**fvec` (Output)
The address of a pointer to a real array of length `m` containing the residuals at the approximate solution. On return, the necessary space is allocated by

`imsl_f_bounded_least_squares`. Typically, *float* `*fvec` is declared, and `&fvec` is used as an argument.

`IMSL_FVEC_USER`, *float* `fvec[]` (Output)
A user-allocated array of size `m` containing the residuals at the approximate solution.

`IMSL_FJAC`, *float* `**fjac` (Output)
The address of a pointer to an array of size $m \times n$ containing the Jacobian at the approximate solution. On return, the necessary space is allocated by `imsl_f_bounded_least_squares`. Typically, *float* `*fjac` is declared, and `&fjac` is used as an argument.

`IMSL_FJAC_USER`, *float* `fjac[]` (Output)
A user-allocated array of size $m \times n$ containing the Jacobian at the approximate solution.

`IMSL_FJAC_COL_DIM`, *int* `fjac_col_dim` (Input)
The column dimension of `fjac`.
Default: `fjac_col_dim` = n

Description

The function `imsl_f_bounded_least_squares` uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least-squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$min \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

$$\text{subject to } l \le x \le u$$

where $m \ge n$, $F : \mathbf{R}^n \rightarrow \mathbf{R}^m$, and $f_i(x)$ is the i-th component function of $F(x)$. From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a “free variable” if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where μ is the Levenberg-Marquardt parameter, $F = F(x)$, and J is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$||g(x_i)|| \le \varepsilon, \; l_i < x_i < u_i$$

$$g(x_i) < 0, \; x_i = u_i$$

$$g(x_i) > 0, \; x_i = l_i$$

where ε is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detail on the active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Jacobian for some single-precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high-precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, the option `IMSL_JACOBIAN` should be used.

Examples

Example 1

In this example, the nonlinear least-squares problem

$$\min \frac{1}{2}\sum_{i=0}^{1} f_i(x)^2$$
$$-2 \le x_0 \le 0.5$$
$$-1 \le x_1 \le 2$$

where

$$f_0(x) = 10(x_1 - x_0^2) \text{ and } f_1(x) = (1 - x_0)$$

is solved with an initial guess (–1.2, 1.0).

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

#define M       2
#define N       2
#define LDFJAC  2

main()
{
        void    rosbck(int, int, float *, float *);
        int  ibtype = 0;
        float   xlb[N] = {-2.0, -1.0};
        float   xub[N] = {0.5, 2.0};
        float  *x;

        x = imsl_f_bounded_least_squares (rosbck, M, N, ibtype, xlb,
                                          xub, 0);

        printf("x[0] = %f\n", x[0]);
        printf("x[1] = %f\n", x[1]);
}
```

```
void rosbck (int m, int n, float *x, float *f)
{
        f[0] = 10.0*(x[1] - x[0]*x[0]);
        f[1] = 1.0 - x[0];
}
```

Output

```
x[0] = 0.500000
x[1] = 0.250000
```

Example 2

This example solves the nonlinear least-squares problem

$$\min \frac{1}{2}\sum_{i=0}^{1} f_i(x)^2$$
$$-2 \le x_0 \le 0.5$$
$$-1 \le x_1 \le 2$$

where

$$f_0(x) = 10(x_1 - x_0^2) \text{ and } f_1(x) = (1 - x_0)$$

This time, an initial guess (–1.2, 1.0) is supplied, as well as the analytic Jacobian. The residual at the approximate solution is returned.

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

#define M       2
#define N       2
#define LDFJAC  2

main()
{
        void    rosbck(int, int, float *, float *);
        void    jacobian(int, int, float *, float *, int);
        int     ibtype = 0;
        float   xlb[N] = {-2.0, -1.0};
        float   xub[N] = {0.5, 2.0};
        float   xguess[N] = {-1.2, 1.0};
        float  *fvec;
        float  *x;

        x = imsl_f_bounded_least_squares (rosbck, M, N, ibtype, xlb, xub,
                                          IMSL_JACOBIAN, jacobian,
                                          IMSL_XGUESS, xguess,
                                          IMSL_FVEC, &fvec,
                                          0);

        printf("x[0] = %f\n", x[0]);
        printf("x[1] = %f\n\n", x[1]);
        printf("fvec[0] = %f\n", fvec[0]);
        printf("fvec[1] = %f\n\n", fvec[1]);
}
```

```
void rosbck (int m, int n, float *x, float *f)
{
        f[0] = 10.0*(x[1] - x[0]*x[0]);
        f[1] = 1.0 - x[0];
}

void jacobian (int m, int n, float *x, float *fjac, int fjac_col_dim)
{
        fjac[0] = -20.0*x[0];
        fjac[1] = 10.0;
        fjac[2] = -1.0;
        fjac[3] = 0.0;
}
```

Output

```
x[0] = 0.500000
x[1] = 0.250000

fvec[0] = 0.000000
fvec[1] = 0.500000
```

min_con_nonlin

Solves a general nonlinear programming problem using the successive quadratic programming algorithm.

Synopsis

#include `<imsl.h>`

float `*imsl_f_min_con_nonlin` (*void* `fcn()`, *int* `m`, *int* `meq`, *int* `n`, *int* `ibtype`, *float* `xlb[]`, *float* `xub[]`, …, 0)

The type *double* function is `imsl_d_min_con_nonlin`.

Required Arguments

void `fcn` (*int* `m`, *int* `meq`, *int* `n`, *float* `x[]`, *int* `active[]`, *float* `*f`, *float* `g[]`)
User-supplied function to evaluate the functions at a given point where

int `m` (Input)
Total number of constraints.

int `meq` (Input)
Number of equality constraints.

int `n` (Input)
Number of variables.

float `x[]` (Input)
Array with `n` components at which point the function is evaluated.

int `active[]` (Input)
Array with `mmax` components indicating the active constraints, where `mmax` is the maximum of (1, *m*).

float `*f` (Output)
The computed function value at the point `x`.

float `g[]` (Output)
Array with `mmax` components containing the values of the constraints at point `x` where `mmax` is the maximum of (1, *m*).

int `m` (Input)
Total number of constraints.

int `meq` (Input)
Number of equality constraints.

int `n` (Input)
Number of variables.

int `ibtype` (Input)
Scalar indicating the types of bounds on variables.

`ibtype`	Action
0	User will supply all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on first variable, all other variables will have the same bounds.

float `xlb[]` (Input, Output, or Input/Output)
Array with `n` components containing the lower bounds on the variables. (Input, if `ibtype` = 0; output, if `ibtype` = 1 or 2; Input/Output, if `ibtype` = 3)

If there is no lower bound on a variable, then the corresponding `xlb` value should be set to -10^6.

float `xub[]` (Input, Output, or Input/Output)
Array with `n` components containing the upper bounds on the variables. (Input, if `ibtype` = 0; output, if `ibtype` 1 or 2; Input/Output, if `ibtype` = 3)

If there is no upper bound on a variable, then the corresponding `xub` value should be set to 10^6.

Return Value

A pointer to the solution *x* of the nonlinear programming problem. 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_min_con_nonlin (*void* fcn(), *int* m, *int* meq, *int* n,
int ibtype, *float* xlb, *float* xub,
IMSL_XGUESS, *float* xguess[],
IMSL_GRADIENT, *void* grad(),
IMSL_ERR_REL, *float* err_rel,
IMSL_XSCALE, *float* xscale[],
IMSL_PRINT, *int* iprint,
IMSL_RETURN_USER, *float* x[],
IMSL_ITMAX, *int* itmax,
IMSL_OBJ, *float* *obj,
0)

Optional Arguments

IMSL_XGUESS, *float* xguess[] (Input)
Array with n components containing an initial guess of the computed solution.
Default: xguess = 0

IMSL_GRADIENT, *void* grad (*int* m, *int* meq, *int* mmax, *int* n, *float* x[], *int* active[], *float* f, *float* g[], *float* df[], *float* dg[])
User-supplied function to evaluate the gradients at a given point where

int m (Input)
Total number of constraints.

int meq (Input)
Number of equality constraints.

int mmax (Input)
Maximum of (1, *m*).

int n (Input)
Number of variables.

float x[] (Input)
Array with n components at which point the function is evaluated.

int active[] (Input)
Array with mmax components indicating the active constraints.

float f (Input)
The computed function value at the point x.

float g[] (Input)
Array with mmax components containing the values of the constraints at point x.

float df[] (Output)
Array with n components containing the values of the gradient of the objective function.

float `dg[]` (Output)
Array of size *mmax* × *n* containing the values of the gradients for the active constraints.

`IMSL_ERR_REL`, *float* `err_rel` (Input)
The final accuracy.
Default: err_ rel = $\sqrt{\varepsilon}$

where ε is the machine precision.

`IMSL_XSCALE`, *float* `xscale[]` (Input)
Array with `n` components containing the reciprocal magnitude of each variable. The argument `xscale` is used to choose the finite-difference stepsize, `h`. The *i*-th component of `h` is computed as

$$\sqrt{\varepsilon} * max\left(\left|x_i\right|, 1/s_i\right) * \text{sign}\left(x_i\right)$$

where ε is the machine precision, s = `xscale`, and sign (x_i) = 1 if $x_i \geq 0$; otherwise, sign $(x_i) = -1$.
Default: `xscale[]` = 1

`IMSL_PRINT`, *int* `iprint` (Input)
Parameter indicating the desired output level.

`iprint`	**Action**
0	No output printed.
1	Only a final convergence analysis is given.
2	One line of intermediate results is printed for each iteration.
3	Detailed information is printed for each iteration.

Default: `iprint` = 0

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `n` components containing the computed solution.

`IMSL_ITMAX`, *int* `itmax` (Input)
Maximum number of iterations allowed.
Default: `itmax` = 200

`IMSL_OBJ`, *float* `*obj` (Output)
Scalar containing the value of the objective function at the computed solution.

Description

The function `f_min_con_nonlin` is based on subroutine NLPQL developed by Schittkowski (1986). It uses a successive quadratic programming method to solve the general nonlinear programming problem. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

$$\text{subject to } g_j(x) = 0, \text{ for } j = 1, \ldots, m_e$$

$$g_j(x) \geq 0, \text{ for } j = m_e + 1, \ldots, m$$

$$(x_l \leq x \leq x_u)$$

where all problem functions are assumed to be continuously differentiable. The method, based on the iterative formulation and solution of quadratic programming (QP) subproblems, obtains these subproblems by using a quadratic approximation of the Lagrangian and by linearizing the constraints. That is,

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} d^T B_k d + \nabla f(x_k)^T d$$

$$\text{subject to } \nabla g_j(x_k)^T d + g_j(x_k) = 0, \text{ for } j = 1, \ldots, m_e$$

$$\nabla g_j(x_k)^T d + g_j(x_k) \geq 0, \text{ for } j = m_e + 1, \ldots, m$$

$$x_l - x_k \leq d \leq x_u - x_k$$

where B_k is a positive definite approximation of the Hessian, and x_k is the current iterate. Let d_k be the solution of the subproblem. A line search is used to find a new point x_{k+1},

$$x_{k+1} = x_k + \lambda d_k \qquad \lambda \in (0, 1]$$

such that a "merit function" will have a lower function value at the new point. Here, the augmented Lagrange function (Schittkowski 1986) is used as the merit function.

When optimality is not achieved, B_k is updated according to the modified BFGS formula (Powell 1978). Note that this algorithm may generate infeasible points during the solution process. Therefore, if feasibility must be maintained for intermediate points, this function may not be suitable. For more theoretical and practical details, see Stoer (1985), Schittkowski (1980, 1983, 1986) and Gill et al. (1985).

Examples

Example 1

The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

$$\text{subject to} \qquad g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2 / 4 - x_2^2 + 1 \geq 0$$

is solved.

```
#include <imsl.h>

#define M       2
#define ME      1
#define N       2
```

```
void main()
{
        int             ibtype = 0;
        float           *x;
        void         fcn(int, int, int, float[], int[], float*, float[]);
        static float    xlb[N] = {-1.0e6, -1.0e6};
        static float    xub[N] = {1.0e6, 1.0e6};


        x = imsl_f_min_con_nonlin(fcn, M, ME, N, ibtype, xlb, xub, 0);

        imsl_f_write_matrix ("The solution is", 1, N, x, 0);
}

                           /* Himmelblau problem 1 */
void fcn(int m, int me, int n, float x[], int active[], float *f,
         float g[])
{
        float           tmp1, tmp2;

        tmp1 = x[0] - 2.0e0;
        tmp2 = x[1] - 1.0e0;
        *f = tmp1 * tmp1 + tmp2 * tmp2;

        if (active[0])
                g[0] = x[0] - 2.0e0 * x[1] + 1.0e0;
        if (active[1])
                g[1] = -(x[0]*x[0]) / 4.0e0 - x[1]*x[1] + 1.0e0;
        return;
}
```

Output

```
The solution is
     1           2
0.8229      0.9115
```

Example 2

The previous example is solved with an initial guess (2.0, 2.0).

```
#include <imsl.h>

#define M       2
#define ME      1
#define N       2

void main()
{
        int             ibtype = 0;
        float           *x;
        void            fcn(int, int, int, float[], int[], float*,
                            float[]);
        void            grad(int, int, int, int, float[], int[], float,
                             float[], float[], float[]);
        static float    xguess[N] = {2.0e0, 2.0e0};
        static float    xlb[N] = {-1.0e6, -1.0e6};
        static float    xub[N] = {1.0e6, 1.0e6};
```

```
        x = imsl_f_min_con_nonlin(fcn, M, ME, N, ibtype, xlb, xub,
                                  IMSL_XGUESS, xguess,
                                  IMSL_GRADIENT, grad,
                                  0);
        imsl_f_write_matrix ("The solution is", 1, N, x, 0);
}

                               /* Himmelblau problem 1 */
void fcn(int m, int me, int n, float x[], int active[],
         float *f, float g[])
{
        float           tmp1, tmp2;

        tmp1 = x[0] - 2.0e0;
        tmp2 = x[1] - 1.0e0;
        *f = tmp1 * tmp1 + tmp2 * tmp2;

        if (active[0])
                g[0] = x[0] - 2.0e0 * x[1] + 1.0e0;
        if (active[1])
                g[1] = -(x[0]*x[0]) / 4.0e0 - x[1]*x[1] + 1.0e0;
        return;
}

void grad(int m, int me, int mmax, int n, float x[], int active[],
          float f, float g[], float df[], float dg[])
{

        df[0] = 2.0 * (x[0] - 2.0);
        df[1] = 2.0 * (x[1] - 1.0);

        if (active[0]) {
            dg[0] = 1.0;
            dg[1] = -2.0;
       }
        if (active[1]) {
            dg[2] = -0.5 * x[0];
            dg[3] = -2.0 * x[1];
       }
        return;
}
```

Output

```
The solution is
     1          2
0.8229      0.9114
```

Warning Errors

IMSL_TOO_MANY_ITN	Maximum number of iterations exceeded.

Fatal Errors

IMSL_UPHILL_DIRECTION	Search direction uphill.
IMSL_TOO_MANY_LINESEARCH	Line search took more than five function calls.
IMSL_NO_PROGRESS_MADE	Search direction is close to zero.

`IMSL_QP_INCONSISTENT`	The constraints for the QP subproblem are inconsistent.

Chapter 9: Special Functions

Routines

9.1 Error and Gamma Functions

Error Functions

Gamma Functions

9.2 Bessel Functions

erf

Evaluates the real error function erf(*x*).

Synopsis

#include `<imsl.h>`

float `imsl_f_erf` (*float* `x`)

The type *double* procedure is `imsl_d_erf`.

Required Arguments

float `x` (Input)
Point at which the error function is to be evaluated.

Return Value

The value of the error function erf(*x*).

Description

The error function erf(*x*) is defined to be

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

All values of *x* are legal.

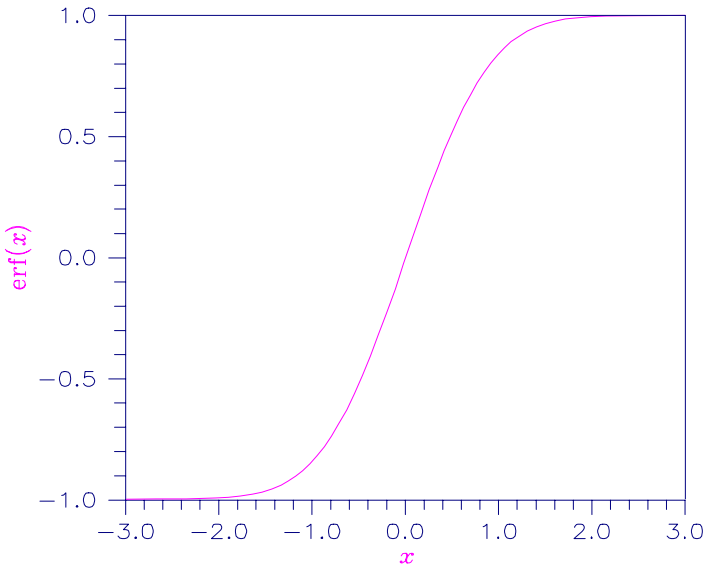


Figure 9-1 Plot of erf(x)

Example

Evaluate the error function at $x = 1/2$.

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       ans;

    ans = imsl_f_erf(x);
    printf("erf(%f) = %f\n", x, ans);
}
```

Output

```
erf(0.500000) = 0.520500
```

erfc

Evaluates the real complementary error function erfc(x).

Synopsis

#include `<imsl.h>`

float `imsl_f_erfc` (*float* `x`)

The type *double* procedure is `imsl_d_erfc`.

Required Arguments

float `x` (Input)
Point at which the complementary error function is to be evaluated.

Return Value

The value of the complementary error function erfc(x).

Description

The complementary error function erfc(x) is defined to be

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

The argument x must not be so large that the result underflows. Approximately, x should be less than

$$\left[-ln\left(\sqrt{\pi} s\right) \right]^{1/2}$$

where s is the smallest representable floating-point number.

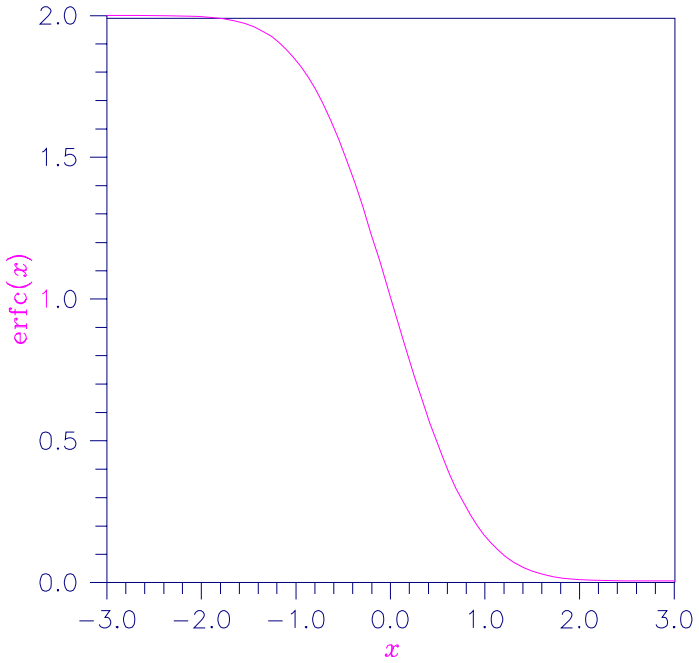


Figure 9-2 Plot of erfc(x)

Example

Evaluate the error function at $x = 1/2$.

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       ans;

    ans = imsl_f_erfc(x);
    printf("erfc(%f) = %f\n", x, ans);
}
```

Output

```
erfc(0.500000) = 0.479500
```

Alert Errors

IMSL_LARGE_ARG_UNDERFLOW	The argument x is so large that the result underflows.

erf_inverse

Evaluates the real inverse error function $\text{erf}^{-1}(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_erf_inverse` (*float* `x`)

The type *double* procedure is `imsl_d_erf_inverse`.

Required Arguments

float `x` (Input)
Point at which the inverse error function is to be evaluated. It must be between –1 and 1.

Return Value

The value of the inverse error function $\text{erf}^{-1}(x)$.

Description

The inverse error function $\text{erf}^{-1}(x)$ is such that $x = \text{erf}(y)$, where

$$\text{erf}(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-t^2} dt$$

The inverse error function is defined only for $-1 < x < 1$.

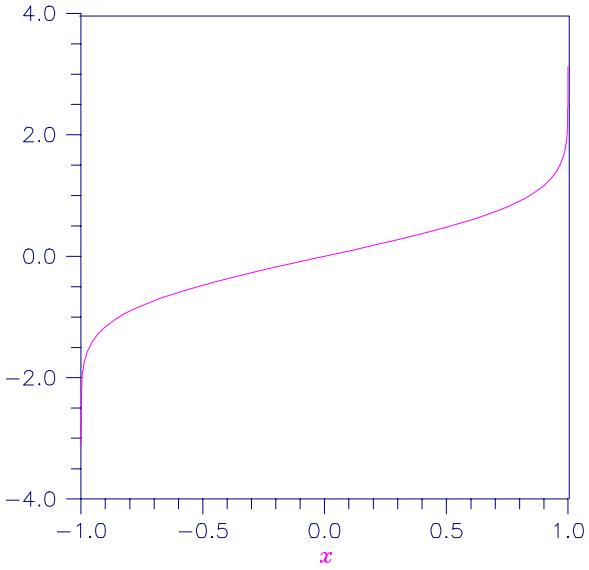


Figure 9-3 Plot of $\text{erf}^{-1}(x)$

Example

Evaluate the inverse error function at $x = 1/2$.

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       ans;

    ans = imsl_f_erf_inverse(x);
    printf("inverse erf(%f) = %f\n", x, ans);
}
```

Output

```
inverse erf(0.500000) = 0.476936
```

Warning Errors

IMSL_LARGE_ABS_ARG_WARN	The answer is less accurate than half precision because $\|x\|$ is too large.

Fatal Errors

IMSL_REAL_OUT_OF_RANGE	The inverse error function is defined only for $-1 < x < 1$.

erfc_inverse

Evaluates the real inverse complementary error function $\text{erfc}^{-1}(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_erfc_inverse` (*float* `x`)

The type *double* procedure is `imsl_d_erfc_inverse`.

Required Arguments

float `x` (Input)
Point at which the inverse complementary error function is to be evaluated. The argument x must be in the range $0 < x < 2$.

Return Value

The value of the inverse complementary error function.

Description

The inverse complementary error function $y = \text{erfc}^{-1}(x)$ is such that $x = \text{erfc}(y)$ where

$$\text{erfc}(y) = \frac{2}{\sqrt{\pi}} \int_y^{\infty} e^{-t^2} dt$$

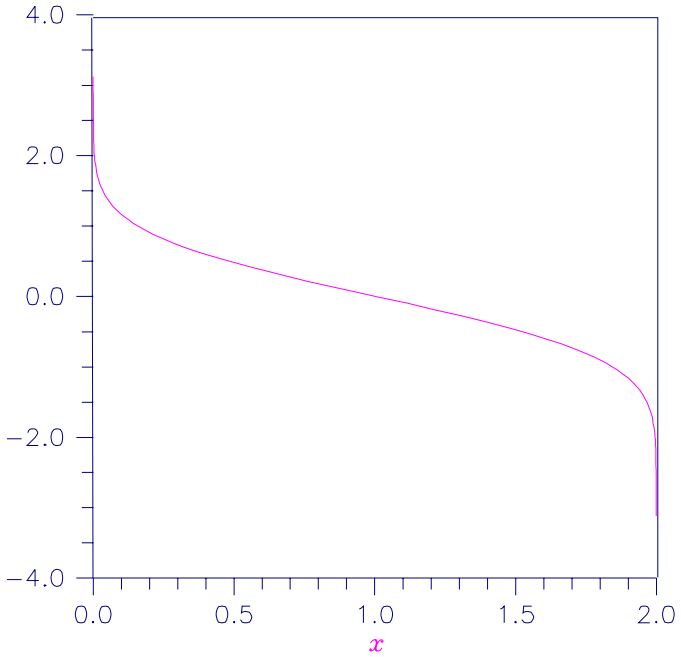


Figure 9-4 Plot of $\text{erfc}^{-1}(x)$

Example

Evaluate the inverse complementary error function at $x = 1/2$.

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       ans;

    ans = imsl_f_erfc_inverse(x);
    printf("inverse erfc(%f) = %f\n", x, ans);
}
```

Output

```
inverse erfc(0.500000) = 0.476936
```

Alert Errors

`IMSL_LARGE_ARG_UNDERFLOW` The argument x must not be so large that the result underflows. Very approximately, x should be less than

$$2 - \sqrt{\varepsilon / (4\pi)}$$

where ε is the machine precision.

Warning Errors

IMSL_LARGE_ARG_WARN	$\|x\|$ should be less than $1/\sqrt{\varepsilon}$ where ε is the machine precision, to prevent the answer from being less accurate than half precision.

Fatal Errors

IMSL_ERF_ALGORITHM	The algorithm failed to converge.
IMSL_SMALL_ARG_OVERFLOW	The computation of $e^{x^2}\operatorname{erfc} x$ must not overflow.
IMSL_REAL_OUT_OF_RANGE	The function is defined only for $0 < x < 2$.

beta

Evaluates the real beta function $\beta(x, y)$.

Synopsis

#include <imsl.h>

float imsl_f_beta (*float* x, *float* y)

The type *double* procedure is imsl_d_beta.

Required Arguments

float x (Input)
Point at which the beta function is to be evaluated. It must be positive.

float y (Input)
Point at which the beta function is to be evaluated. It must be positive.

Return Value

The value of the beta function $\beta(x, y)$. If no result can be computed, NaN is returned.

Description

The beta function, $\beta(x, y)$, is defined to be

$$\beta(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

The beta function requires that $x > 0$ and $y > 0$. It underflows for large arguments.

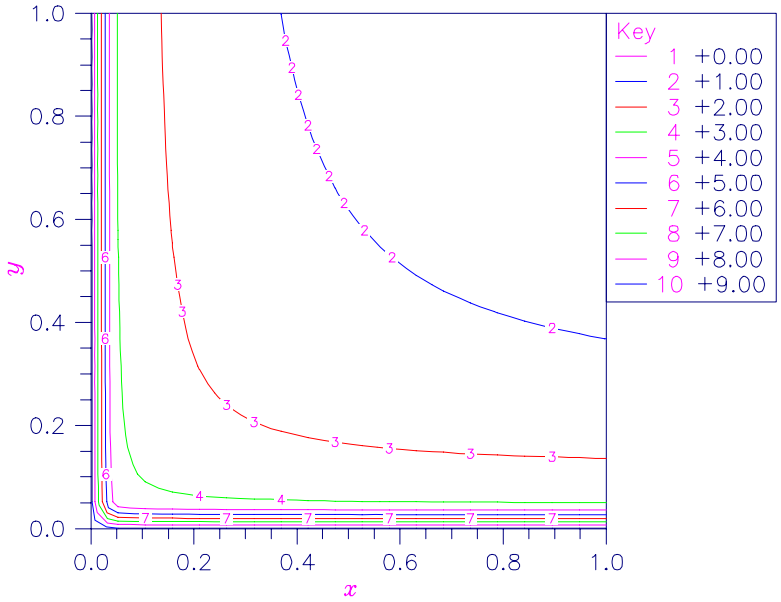


Figure 9-5 Plot of $\beta(x,y)$

Example

Evaluate the beta function β (0.5, 0.2).

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       y = 0.2;
    float       ans;

    ans = imsl_f_beta(x, y);
    printf("beta(%f,%f) = %f\n", x, y, ans);
}
```

Output

```
beta(0.500000,0.200000) = 6.268653
```

Alert Errors

IMSL_BETA_UNDERFLOW	The arguments must not be so large that the result underflows.

Fatal Errors

IMSL_ZERO_ARG_OVERFLOW	One of the arguments is so close to zero that the result overflows.

log_beta

Evaluates the logarithm of the real beta function ln $\beta(x, y)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_log_beta` (*float* `x`, *float* `y`)

The type *double* procedure is `imsl_d_log_beta`.

Required Arguments

float `x` (Input)
Point at which the logarithm of the beta function is to be evaluated. It must be positive.

float `y` (Input)
Point at which the logarithm of the beta function is to be evaluated. It must be positive.

Return Value

The value of the logarithm of the beta function $\beta(x, y)$.

Description

The beta function, $\beta(x, y)$, is defined to be

$$\beta(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

and `imsl_f_log_beta` returns ln $\beta(x, y)$.

The logarithm of the beta function requires that $x > 0$ and $y > 0$. It can overflow for very large arguments.

Example

Evaluate the log of the beta function ln $\beta(0.5, 0.2)$.

```
#include <imsl.h>

main()
{
    float       x = 0.5;
    float       y = 0.2;
    float       ans;

    ans = imsl_f_log_beta(x, y);
    printf("log beta(%f,%f) = %f\n", x, y, ans);
}
```

Output

```
log beta(0.500000,0.200000) = 1.835562
```

Warning Errors

IMSL_X_IS_TOO_CLOSE_TO_NEG_1	The result is accurate to less than one precision because the expression $-x/(x + y)$ is too close to −1.

beta_incomplete

Evaluates the real incomplete beta function $I_x = \beta_x(a,b)/\beta(a,b)$.

Synopsis

#include <imsl.h>

float imsl_f_beta_incomplete (*float* x, *float* a, *float* b)

The type *double* procedure is imsl_d_beta_incomplete.

Required Arguments

float x (Input)
Point at which the incomplete beta function is to be evaluated.

float a (Input)
Point at which the incomplete beta function is to be evaluated.

float b (Input)
Point at which the incomplete beta function is to be evaluated.

Return Value

The value of the incomplete beta function.

Description

The incomplete beta function is defined to be

$$I_x(a,b) = \frac{\beta_x(a,b)}{\beta(a,b)} = \frac{1}{\beta(a,b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

The incomplete beta function requires that $0 \le x \le 1$, $a > 0$, and $b > 0$. It underflows for sufficiently small x and large a. This underflow is not reported as an error. Instead, the value zero is returned.

gamma

Evaluates the real gamma function $\Gamma(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_gamma` (*float* `x`)

The type *double* procedure is `imsl_d_gamma`.

Required Arguments

float `x` (Input)
Point at which the gamma function is to be evaluated.

Return Value

The value of the gamma function $\Gamma(x)$.

Description

The gamma function, $\Gamma(x)$, is defined to be

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

For $x < 0$, the above definition is extended by analytic continuation.

The gamma function is not defined for integers less than or equal to zero. It underflows for $x << 0$ and overflows for large x. It also overflows for values near negative integers.

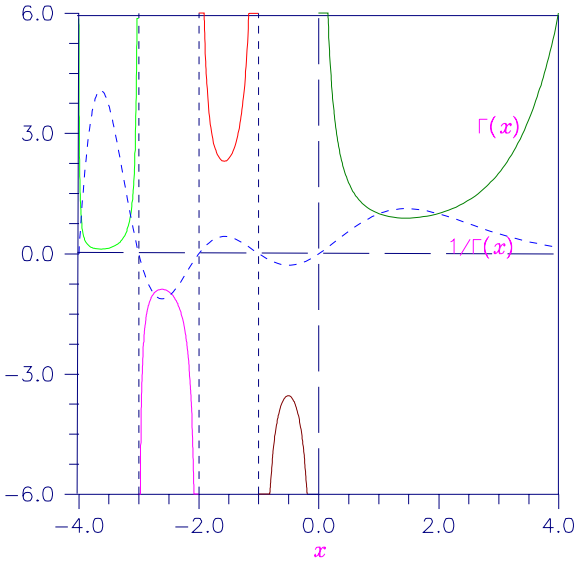


Figure 9-6 Plot of Γ(x) and 1/Γ(x)

Example

In this example, $\Gamma(1.5)$ is computed and printed.

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

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_gamma(x);
    printf("Gamma(%f) = %f\n", x, ans);
}
```

Output

```
Gamma(1.500000) = 0.886227
```

Alert Errors

IMSL_SMALL_ARG_UNDERFLOW	The argument x must be large enough that $\Gamma(x)$ does not underflow. The underflow limit occurs first for arguments close to large negative half integers. Even though other arguments away from these half integers may yield machine-representable values of $\Gamma(x)$, such arguments are considered illegal. Users who need such values should use the $\log\Gamma(x)$ function imsl_f_log_gamma.

Warning Errors

IMSL_NEAR_NEG_INT_WARN	The result is accurate to less than one-half precision because x is too close to a negative integer.

Fatal Errors

IMSL_ZERO_ARG_OVERFLOW	The argument for the gamma function is too close to zero.
IMSL_NEAR_NEG_INT_FATAL	The argument for the function is too close to a negative integer.
IMSL_LARGE_ARG_OVERFLOW	The function overflows because x is too large.
IMSL_CANNOT_FIND_XMIN	The algorithm used to find $x_{\min}$ failed. This error should never occur.
IMSL_CANNOT_FIND_XMAX	The algorithm used to find $x_{\max}$ failed. This error should never occur.

log_gamma

Evaluates the logarithm of the absolute value of the gamma function $\log|\Gamma(x)|$.

Synopsis

#include `<imsl.h>`

float `imsl_f_log_gamma` (*float* `x`)

The type *double* procedure is `imsl_d_log_gamma`.

Required Arguments

float `x` (Input)
Point at which the logarithm of the absolute value of the gamma function is to be evaluated.

Return Value

The value of the logarithm of gamma function, $\log|\Gamma(x)|$.

Description

The logarithm of the absolute value of the gamma function $\log|\Gamma(x)|$ is computed.

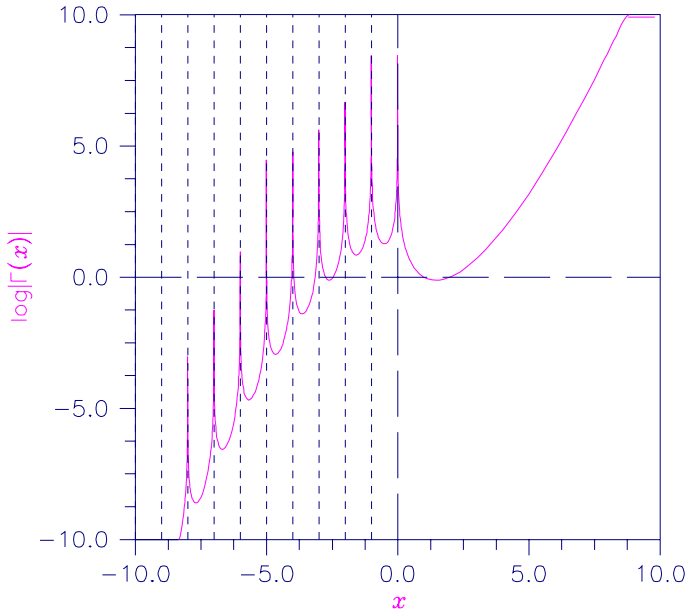


Figure 9-7 Plot of log |Γ(x)|

Example

In this example, $\log |\Gamma(3.5)|$ is computed and printed.

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

main()
{
    float       x = 3.5;
    float       ans;

    ans = imsl_f_log_gamma(x);
    printf("log gamma(%f) = %f\n", x, ans);
}
```

Output

```
log gamma(3.500000) = 1.200974
```

Warning Errors

IMSL_NEAR_NEG_INT_WARN	The result is accurate to less than one-half precision because x is too close to a negative integer.

Fatal Errors

IMSL_NEGATIVE_INTEGER	The argument for the function cannot be a negative integer.
IMSL_NEAR_NEG_INT_FATAL	The argument for the function is too close to a negative integer.
IMSL_LARGE_ABS_ARG_OVERFLOW	$\|x\|$ must not be so large that the result overflows.

gamma_incomplete

Evaluates the incomplete gamma function $\gamma(a, x)$.

Synopsis

#include <imsl.h>

float imsl_f_gamma_incomplete (*float* a, *float* x)

The type *double* procedure is imsl_d_gamma_incomplete.

Required Arguments

float a (Input)
Parameter of the incomplete gamma function is to be evaluated. It must be positive.

float x (Input)
Point at which the incomplete gamma function is to be evaluated. It must be nonnegative.

Return Value

The value of the incomplete gamma function $\gamma(a, x)$.

Description

The incomplete gamma function, $\gamma(a, x)$, is defined to be

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt \qquad \text{for } x > 0$$

The incomplete gamma function is defined only for $a > 0$. Although $\gamma(a, x)$ is well defined for $x > -\infty$, this algorithm does not calculate $\gamma(a, x)$ for negative x. For large a and sufficiently large x, $\gamma(a, x)$ may overflow. $\gamma(a, x)$ is bounded by $\Gamma(a)$, and users may find this bound a useful guide in determining legal values for a.

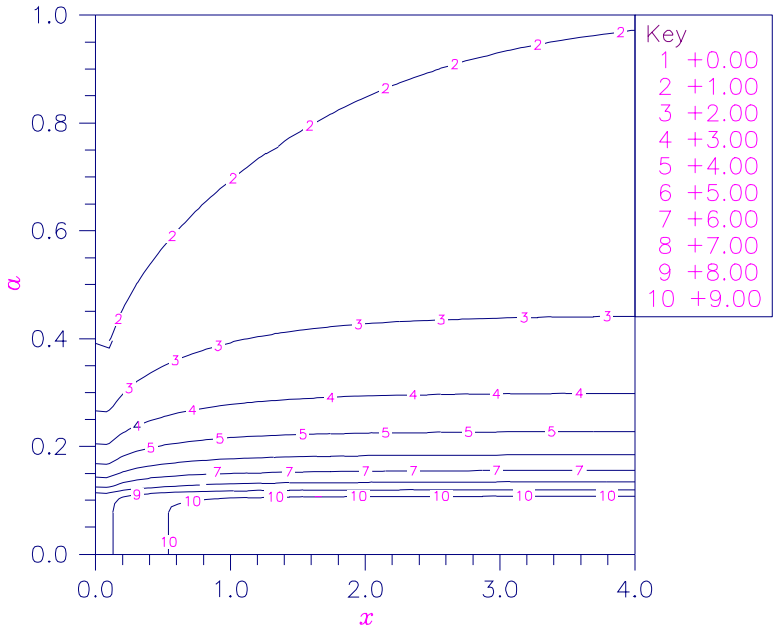


Figure 9-8 Plot of γ(a, x)

Example

Evaluate the incomplete gamma function at $a = 1$ and $x = 3$.

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

main()
{
    float       x = 3.0;
    float       a = 1.0;
    float       ans;

    ans = imsl_f_gamma_incomplete(a, x);
    printf("incomplete gamma(%f,%f) = %f\n", a, x, ans);
}
```

Output

```
incomplete gamma(1.000000,3.000000) = 0.950213
```

Fatal Errors

`IMSL_NO_CONV_200_TS_TERMS`	The function did not converge in 200 terms of Taylor series.
`IMSL_NO_CONV_200_CF_TERMS`	The function did not converge in 200 terms of the continued fraction.

bessel_J0

Evaluates the real Bessel function of the first kind of order zero $J_0(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_J0` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_J0`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,sin\,\theta)\,d\theta$$

If no solution can be computed, NaN is returned.

Description

Because the Bessel function $J_0(x)$ is oscillatory, its computation becomes inaccurate as $|x|$ increases.

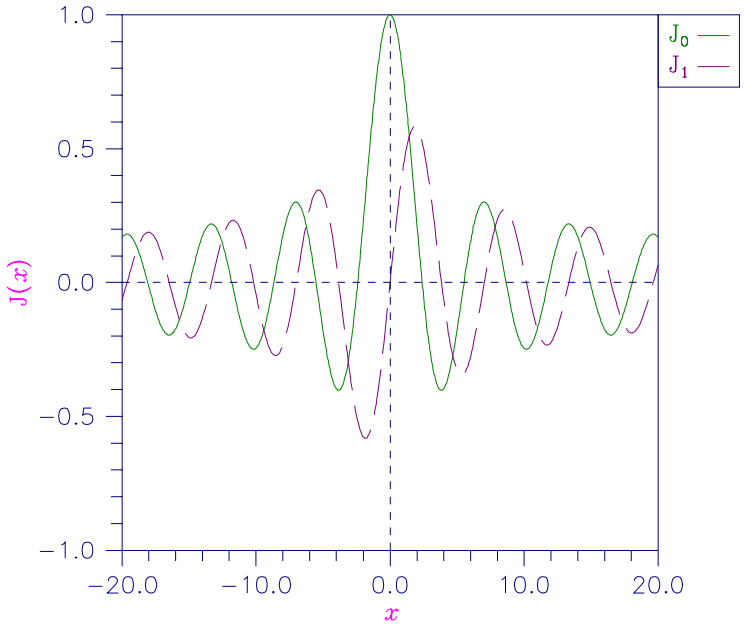


Figure 9-9 Plot of J0 (x) and J1 (x)

Example

The Bessel function $J_0(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_J0(x);
    printf("J0(%f) = %f\n", x, ans);
}
```

Output

```
J0(1.500000) = 0.511828
```

Warning Errors

IMSL_LARGE_ABS_ARG_WARN	$\|x\|$ should be less than $1/\sqrt{\varepsilon}$ where ε is the machine precision, to prevent the answer from being less accurate than half precision.

Fatal Errors

`IMSL_LARGE_ABS_ARG_FATAL`	$\|x\|$ should be less than $1/\varepsilon$ where ε is the machine precision for the answer to have any precision.

bessel_J1

Evaluates the real Bessel function of the first kind of order one $J_1(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_J1` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_J1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$J_1(x) = \frac{1}{\pi}\int_0^{\pi} cos(\theta - x\,sin\,\theta)\,d\theta$$

If no solution can be computed, NaN is returned.

Description

Because the Bessel function $J_1(x)$ is oscillatory, its computation becomes inaccurate as $|x|$ increases.

Example

The Bessel function $J_1(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_J1(x);
    printf("J1(%f) = %f\n", x, ans);
}
```

Output

```
J1(1.500000) = 0.557937
```

Alert Errors

IMSL_SMALL_ABS_ARG_UNDERFLOW	To prevent $J_1(x)$ from underflowing, either x must be zero, or $\|x\| > 2s$ where s is the smallest representable positive number.

Warning Errors

IMSL_LARGE_ABS_ARG_WARN	$\|x\|$ should be less than $1/\sqrt{\varepsilon}$ where ε is the machine precision to prevent the answer from being less accurate than half precision.

Fatal Errors

IMSL_LARGE_ABS_ARG_FATAL	$\|x\|$ should be less than $1/\varepsilon$ where ε is the machine precision for the answer to have any precision.

bessel_Jx

Evaluates a sequence of Bessel functions of the first kind with real order and complex arguments.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_bessel_Jx` (*float* `xnu`, *f_complex* `z`, *int* `n`, …, 0)

The type *d_complex* function is `imsl_z_bessel_Jx`.

Required Arguments

float `xnu` (Input)
The lowest order desired. The argument `xnu` must be greater than −1/2.

f_complex `z` (Input)
Argument for which the sequence of Bessel functions is to be evaluated.

int `n` (Input)
Number of elements in the sequence.

Return Value

A pointer to the `n` values of the function through the series. Element *i* contains the value of the Bessel function of order `xnu` + *i* for $i = 0, \ldots, n - 1$.

Synopsis with Optional Arguments

f_complex `*imsl_c_bessel_Jx` (*float* `xnu`, *f_complex* `z`, *int* `n`
`IMSL_RETURN_USER`, *f_complex* `bessel[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *f_complex* `bessel[]` (Output)
Store the sequence of Bessel functions in the user-provided array `bessel[]`.

Description

The Bessel function $J_\nu(z)$ is defined to be

$$J_\nu(z) = \frac{1}{\pi}\int_0^\pi cos(z\,sin\,\theta - \nu\theta)\,d\theta - \frac{sin(\nu\pi)}{\pi}\int_0^\infty e^{z\,sinh\,t - \nu t}\,dt$$

for $|arg\ z| < \frac{\pi}{2}$

This function is based on the code BESSCC of Barnett (1981) and Thompson and Barnett (1987). This code computes $J_\nu(z)$ from the modified Bessel function $I_\nu(z)$, using the following relation, with $\rho = e^{i\pi/2}$:

$$Y_\nu(z) = \begin{cases} \rho I_\nu(z/\rho) & \text{for } -\pi/2 < arg\ z \le \pi \\ \rho^3 I_\nu(\rho^3 z) & \text{for } -\pi < arg\ z \le \pi/2 \end{cases}$$

Example

In this example, $J_{0.3+\nu-1}$ $(1.2 + 0.5i)$, $\nu = 1, \ldots, 4$ is computed and printed.

```
#include <imsl.h>

main()
{
    int         n = 4;
    int         i;
    float       xnu = 0.3;
    static f_complex    z = {1.2, 0.5};
    f_complex  *sequence;

    sequence = imsl_c_bessel_Jx(xnu, z, n, 0);

    for (i = 0; i < n; i++)
    printf("I sub %4.2f ((%4.2f,%4.2f)) = (%5.3f,%5.3f)\n",
        xnu+i, z.re, z.im, sequence[i].re, sequence[i].im);
}
```

Output

```
I sub 0.30 ((1.20,0.50)) = (0.774,-0.107)
I sub 1.30 ((1.20,0.50)) = (0.400,0.159)
I sub 2.30 ((1.20,0.50)) = (0.087,0.092)
I sub 3.30 ((1.20,0.50)) = (0.008,0.024)
```

bessel_Y0

Evaluates the real Bessel function of the second kind of order zero $Y_0(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_Y0` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_Y0`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$Y_0(x) = \frac{1}{\pi}\int_0^{\pi} sin(x\,sin\,\theta)\,d\theta$$

If no solution can be computed, NaN is returned.

Description

This function is sometimes called the Neumann function, $N_0(x)$, or Weber's function.

Since $Y_0(x)$ is complex for negative x and is undefined at $x = 0$, `imsl_f_bessel_Y0` is defined only for $x > 0$. Because the Bessel function $Y_0(x)$ is oscillatory, its computation becomes inaccurate as x increases.

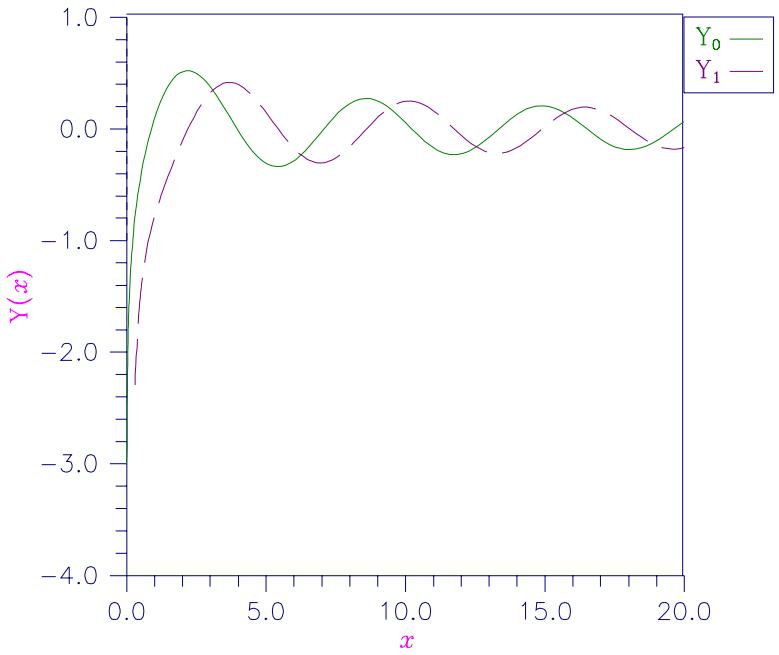


Figure 9-10 Plot of Y0(x) and Y1(x)

Example

The Bessel function $Y_0(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_Y0(x);
    printf("Y0(%f) = %f\n", x, ans);
}
```

Output

```
Y0(1.500000) = 0.382449
```

Warning Errors

IMSL_LARGE_ABS_ARG_WARN	$\|x\|$ should be less than $1/\sqrt{\varepsilon}$ where ε is the machine precision to prevent the answer from being less accurate than half precision.

Fatal Errors

IMSL_LARGE_ABS_ARG_FATAL	$\|x\|$ should be less than $1/\varepsilon$ where ε is the machine precision for the answer to have any precision.

bessel_Y1

Evaluates the real Bessel function of the second kind of order one $Y_1(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_Y1` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_Y1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$Y_1(x) = -\frac{1}{\pi}\int_0^{\pi} sin(\theta - x\,sin\,\theta)\,d\theta$$

If no solution can be computed, then NaN is returned.

Description

This function is sometimes called the Neumann function, $N_1(x)$, or Weber's function.

Since $Y_1(x)$ is complex for negative x and is undefined at $x = 0$, `imsl_f_bessel_Y1` is defined only for $x > 0$. Because the Bessel function $Y_1(x)$ is oscillatory, its computation becomes inaccurate as x increases.

Example

The Bessel function $Y_1(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_Y1(x);
    printf("Y1(%f) = %f\n", x, ans);
}
```

Output

```
Y1(1.500000) = -0.412309
```

Warning Errors

IMSL_LARGE_ABS_ARG_WARN	$\|x\|$ should be less than $1/\sqrt{\varepsilon}$ where ε is the machine precision to prevent the answer from being less accurate than half precision.

Fatal Errors

IMSL_SMALL_ARG_OVERFLOW	The argument x must be large enough ($x > \max(1/b, s)$ where s is the smallest repesentable positive number and b is the largest repesentable number) that $Y_1(x)$ does not overflow.
IMSL_LARGE_ABS_ARG_FATAL	$\|x\|$ should be less than $1/\varepsilon$ where ε is the machine precision for the answer to have any precision.

bessel_Yx

Evaluates a sequence of Bessel functions of the second kind with real order and complex arguments.

Synopsis

#include <imsl.h>

f_complex *imsl_c_bessel_Yx (*float* xnu, *f_complex* z, *int* n, …, 0)

The type *d_complex* function is imsl_z_bessel_Yx.

Required Arguments

float xnu (Input)
The lowest order desired. The argument xnu must be greater than –1/2.

f_complex z (Input)
Argument for which the sequence of Bessel functions is to be evaluated.

int n (Input)
Number of elements in the sequence.

Return Value

A pointer to the n values of the function through the series. Element *i* contains the value of the Bessel function of order xnu + *i* for $i = 0, \ldots, n - 1$.

Synopsis with Optional Arguments

f_complex *imsl_c_bessel_Yx (*float* xnu, *f_complex* z, *int* n,
IMSL_RETURN_USER, *f_complex* bessel[],
0)

Optional Arguments

`IMSL_RETURN_USER, f_complex bessel[]` (Output)
Store the sequence of Bessel functions in the user-provided array `bessel[]`.

Description

The Bessel function $Y_\nu(z)$ is defined to be

$$Y_\nu(z) = \frac{1}{\pi}\int_0^\pi sin(z\,sin\,\theta - \nu\theta)\,d\theta - \frac{sin(\nu\pi)}{\pi}\int_0^\infty \left[e^{\nu t} + e^{-\nu t}\,cos(\nu t)\right] e^{z\,sinh\,t}\,dt$$

$$\text{for } |arg\,z| < \frac{\pi}{2}$$

This function is based on the code BESSCC of Barnett (1981) and Thompson and Barnett (1987). This code computes $Y_\nu(z)$ from the modified Bessel functions $I_\nu(z)$ and $K_\nu(z)$, using the following relation:

$$Y_\nu(z) = e^{(\nu+1)\pi i/2} I_\nu(z) - \frac{2}{\pi} e^{-\nu\pi i/2} K_\nu(z) \qquad \text{for } -\pi < arg\,z \le \frac{\pi}{2}$$

Example

In this example, $Y_{0.3+\nu-1}$ (1.2 + 0.5i), ν = 1, …, 4 is computed and printed.

```
#include <imsl.h>

main()
{
    int         n = 4;
    int         i;
    float       xnu = 0.3;
    static f_complex    z = {1.2, 0.5};
    f_complex  *sequence;

    sequence = imsl_c_bessel_Yx(xnu, z, n, 0);

    for (i = 0; i < n; i++)
    printf("Y sub %4.2f ((%4.2f,%4.2f)) = (%5.3f,%5.3f)\n",
        xnu+i, z.re, z.im, sequence[i].re, sequence[i].im);
}
```

Output

```
Y sub 0.30 ((1.20,0.50)) = (-0.013,0.380)
Y sub 1.30 ((1.20,0.50)) = (-0.716,0.338)
Y sub 2.30 ((1.20,0.50)) = (-1.048,0.795)
Y sub 3.30 ((1.20,0.50)) = (-1.625,3.684)
```

bessel_I0

Evaluates the real modified Bessel function of the first kind of order zero $I_0(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_I0` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_I0`.

Required Arguments

float `x` (Input)
Point at which the modified Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$I_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,cos\,\theta)\,d\theta$$

If no solution can be computed, NaN is returned.

Description

For large $|x|$, `imsl_f_bessel_I0` will overflow.

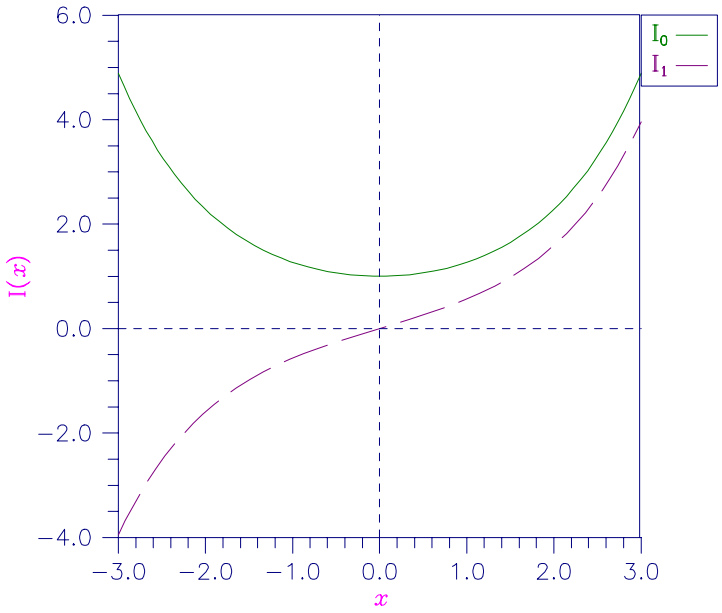


Figure 9-11 Plot of I0(x) and I1(x)

Example

The Bessel function $I_0(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_I0(x);
    printf("I0(%f) = %f\n", x, ans);
}
```

Output

```
I0(1.500000) = 1.646723
```

Fatal Errors

`IMSL_LARGE_ABS_ARG_FATAL`	The absolute value of *x* must not be so large that $e^{\lvert x \rvert}$ overflows.

bessel_exp_I0

Evaluates the exponentially scaled modified Bessel function of the first kind of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_exp_I0` (*float* `x`)

The type *double* function is `imsl_d_bessel_exp_I0`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the scaled Bessel function $e^{-\lvert x \rvert} I_0(x)$. If no solution can be computed, NaN is returned.

Description

The Bessel function is $I_0(x)$ is defined to be

$$I_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,cos\,\theta)\,d\theta$$

Example

The expression $e^{-4.5}I_0$ (4.5) is computed directly by calling `imsl_f_bessel_exp_I0` and indirectly by calling `imsl_f_bessel_I0`. The absolute difference is printed. For large `x`, the internal scaling provided by `imsl_f_bessel_exp_I0` avoids overflow that may occur in `imsl_f_bessel_I0`.

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

main()
{
        float   x = 4.5;
        float   ans;
        float   error;

        ans = imsl_f_bessel_exp_I0 (x);
        printf("(e**(-4.5))I0(4.5) = %f\n\n", ans);

        error = fabs(ans - (exp(-x)*imsl_f_bessel_I0(x)));
        printf ("Error = %e\n", error);
}
```

Output

```
(e**(-4.5))I0(4.5) = 0.194198

Error = 4.898845e-09
```

bessel_I1

Evaluates the real modified Bessel function of the first kind of order one $I_1(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_I1` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_I1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the Bessel function

$$I_1(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\cos\theta}\cos\theta\, d\theta$$

If no solution can be computed, NaN is returned.

Description

For large $|x|$, `imsl_f_bessel_I1` will overflow. It will underflow near zero.

Example

The Bessel function $I_1(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_I1(x);
    printf("I1(%f) = %f\n", x, ans);
}
```

Output

```
I1(1.500000) = 0.981666
```

Alert Errors

`IMSL_SMALL_ABS_ARG_UNDERFLOW`	The argument should not be so close to zero that $I_1(x) \approx x/2$ underflows.

Fatal Errors

`IMSL_LARGE_ABS_ARG_FATAL`	The absolute value of x must not be so large that $e^{\lvert x \rvert}$ overflows.

bessel_exp_I1

Evaluates the exponentially scaled modified Bessel function of the first kind of order one.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_exp_I1` (*float* `x`)

The type *double* function is `imsl_d_bessel_exp_I1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the scaled Bessel function $e^{-|x|} I_1(x)$. If no solution can be computed, NaN is returned.

Description

The function `imsl_f_bessel_I1` underflows if $|x| / 2$ underflows. The Bessel function $I_1(x)$ is defined to be

$$I_1(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\cos\theta}\cos\theta\, d\theta$$

Example

The expression $e^{-4.5} I_0(4.5)$ is computed directly by calling `imsl_f_bessel_exp_I1` and indirectly by calling `imsl_f_bessel_I1`. The absolute difference is printed. For large `x`, the internal scaling provided by `imsl_f_bessel_exp_I1` avoids overflow that may occur in `imsl_f_bessel_I1`.

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

main()
{
        float   x = 4.5;
        float   ans;
        float   error;

        ans = imsl_f_bessel_exp_I1 (x);
        printf("(e**(-4.5))I1(4.5) = %f\n\n", ans);

        error = fabs(ans - (exp(-x)*imsl_f_bessel_I1(x)));
        printf ("Error = %e\n", error);
}
```

Output

```
(e**(-4.5))I1(4.5) = 0.170959

Error = 1.469216e-09
```

bessel_Ix

Evaluates a sequence of modified Bessel functions of the first kind with real order and complex arguments.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_bessel_Ix` (*float* `xnu`, *f_complex* `z`, *int* `n`, …, 0)

The type *d_complex* function is `imsl_z_bessel_Ix`.

Required Arguments

float `xnu` (Input)
The lowest order desired. Argument `xnu` must be greater than –1/2.

f_complex `z` (Input)
Argument for which the sequence of Bessel functions is to be evaluated.

int `n` (Input)
Number of elements in the sequence.

Return Value

A pointer to the `n` values of the function through the series. Element *i* contains the value of the Bessel function of order `xnu` + *i* for *i* = 0, …, *n* – 1.

Synopsis with Optional Arguments

f_complex `*imsl_c_bessel_Ix` (*float* `xnu`, *f_complex* `z`, *int* `n`,
`IMSL_RETURN_USER`, *f_complex* `bessel[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *f_complex* `bessel[]` (Output)
Store the sequence of Bessel functions in the user-provided array `bessel[]`.

Description

The Bessel function $I_\nu(z)$ is defined to be

$$I_\nu(z) = e^{-\nu\pi i/2} J_\nu\left(ze^{\pi i/2}\right) \quad \text{for } -\pi < arg\, z \le \frac{\pi}{2}$$

For large arguments, `z`, Temme's (1975) algorithm is used to find $I_\nu(z)$. The $I_\nu(z)$ values are recurred upward (if this is stable). This involves evaluating a continued fraction. If this evaluation fails to converge, the answer may not be accurate.

For moderate and small arguments, Miller's method is used.

Example

In this example, $J_{0.3+\nu-1}$ (1.2 + 0.5*i*), ν = 1, …, 4 is computed and printed.

```
#include <imsl.h>

main()
{
    int         n = 4;
    int         i;
    float       xnu = 0.3;
    static f_complex    z = {1.2, 0.5};
    f_complex  *sequence;

    sequence = imsl_c_bessel_Ix(xnu, z, n, 0);
```

```
    for (i = 0; i < n; i++)
    printf("I sub %4.2f ((%4.2f,%4.2f)) = (%5.3f,%5.3f)\n",
        xnu+i, z.re, z.im, sequence[i].re, sequence[i].im);
}
```

Output

```
I sub 0.30 ((1.20,0.50)) = (1.163,0.396)
I sub 1.30 ((1.20,0.50)) = (0.447,0.332)
I sub 2.30 ((1.20,0.50)) = (0.082,0.127)
I sub 3.30 ((1.20,0.50)) = (0.006,0.029)
```

bessel_K0

Evaluates the real modified Bessel function of the third kind of order zero $K_0(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_K0` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_K0`.

Required Arguments

float `x` (Input)
Point at which the modified Bessel function is to be evaluated. It must be positive.

Return Value

The value of the modified Bessel function

$$K_0(x) = \int_0^\infty cos(x \sin t)\, dt$$

If no solution can be computed, then NaN is returned.

Description

Since $K_0(x)$ is complex for negative x and is undefined at $x = 0$, `imsl_f_bessel_K0` is defined only for $x > 0$. For large x, `imsl_f_bessel_K0` will underflow.

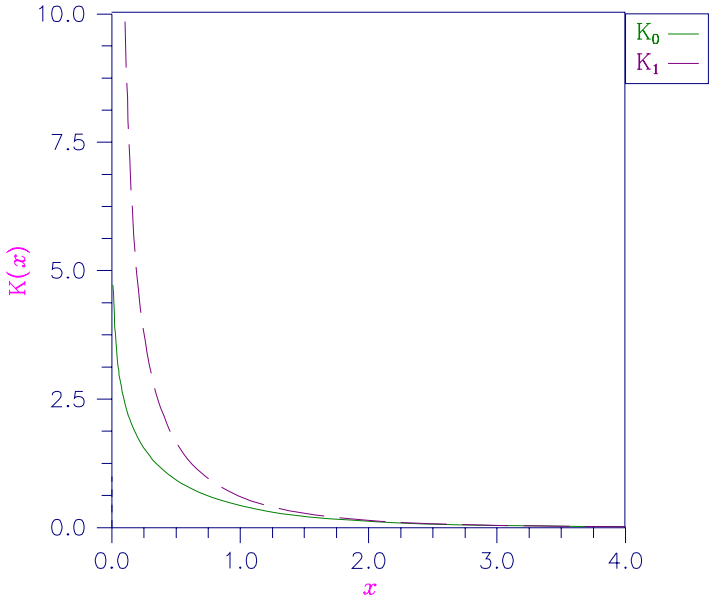


Figure 9-12 Plot of $K_0(x)$ and $K_1(x)$

Example

The Bessel function $K_0(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_K0(x);
    printf("K0(%f) = %f\n", x, ans);
}
```

Output

```
K0(1.500000) = 0.213806
```

Alert Errors

`IMSL_LARGE_ARG_UNDERFLOW`	The argument x must not be so large that the result (approximately equal to $\sqrt{\pi/(2x)}e^{-x}$ underflows.

bessel_exp_K0

Evaluates the exponentially scaled modified Bessel function of the third kind of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_exp_K0` (*float* `x`)

The type *double* function is `imsl_d_bessel_exp_K0`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the scaled Bessel function $e^{x}K_0(x)$. If no solution can be computed, NaN is returned.

Description

The argument must be greater than zero for the result to be defined. The Bessel function $K_0(x)$ is defined to be

$$K_0(x) = \int_0^{\infty} cos(x \sin t)\, dt$$

Example

The expression

$$\sqrt{e}K_0(0.5)$$

is computed directly by calling `imsl_f_bessel_exp_K0` and indirectly by calling `imsl_f_bessel_K0`. The absolute difference is printed. For large `x`, the internal scaling provided by `imsl_f_bessel_exp_K0` avoids underflow that may occur in `imsl_f_bessel_K0`.

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

main()
{
        float   x = 0.5;
        float   ans;
        float   error;

        ans = imsl_f_bessel_exp_K0 (x);
        printf("(e**0.5)K0(0.5) = %f\n\n", ans);

        error = fabs(ans - (exp(x)*imsl_f_bessel_K0(x)));
        printf ("Error = %e\n", error);
}
```

Output

```
(e**0.5)K0(0.5) = 1.524109

Error = 2.028498e-08
```

bessel_K1

Evaluates the real modified Bessel function of the third kind of order one $K_1(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_K1` (*float* `x`)

The type *double* procedure is `imsl_d_bessel_K1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated. It must be positive.

Return Value

The value of the Bessel function

$$K_1(x) = \int_0^\infty sin(x \sin t) \sin t \; dt$$

If no solution can be computed, NaN is returned.

Description

Since $K_1(x)$ is complex for negative x and is undefined at $x = 0$, `imsl_f_bessel_K1` is defined only for $x > 0$. For large x, `imsl_f_bessel_K1` will underflow.
See Figure 9-12 for a graph of $K_1(x)$.

Example

The Bessel function $K_1(1.5)$ is evaluated.

```
#include <imsl.h>

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsl_f_bessel_K1(x);
    printf("K1(%f) = %f\n", x, ans);
}
```

Output

```
K1(1.500000) = 0.277388
```

Alert Errors

IMSL_LARGE_ARG_UNDERFLOW	The argument x must not be so large that the result, approximately equal to, $\sqrt{\pi / (2x)}e^{-x}$ underflows.

Fatal Errors

IMSL_SMALL_ARG_OVERFLOW	The argument x must be large enough ($x > \max(1/b, s)$ where s is the smallest representable positive number and b is the largest repesentable number) that $K_1(x)$ does not overflow.

bessel_exp_K1

Evaluates the exponentially scaled modified Bessel function of the third kind of order one.

Synopsis

#include `<imsl.h>`

float `imsl_f_bessel_exp_K1` (*float* `x`)

The type *double* function is `imsl_d_bessel_exp_K1`.

Required Arguments

float `x` (Input)
Point at which the Bessel function is to be evaluated.

Return Value

The value of the scaled Bessel function $e^x K_1(x)$. If no solution can be computed, NaN is returned.

Description

The result

$$\text{imsl\_ f\_ bessel\_ exp\_ K1} = e^x K_1(x) \approx \frac{1}{x}$$

overflows if x is too close to zero. The definition of the Bessel function

$$K_1(x) = \int_0^\infty sin(x \sin t) \sin t \; dt$$

Example

The expression

$$\sqrt{e} K_1(0.5)$$

is computed directly by calling `imsl_f_bessel_exp_K1` and indirectly by calling `imsl_f_bessel_K1`. The absolute difference is printed. For large `x`, the internal scaling provided by `imsl_f_bessel_exp_K1` avoids underflow that may occur in `imsl_f_bessel_K1`.

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

main()
{
        float   x = 0.5;
        float   ans;
        float   error;

        ans = imsl_f_bessel_exp_K1 (x);
        printf("(e**0.5)K1(0.5) = %f\n\n", ans);

        error = fabs(ans - (exp(x)*imsl_f_bessel_K1(x)));
        printf ("Error = %e\n", error);
}
```

Output

```
(e**0.5)K1(0.5) = 2.731010

Error = 5.890406e-08
```

bessel_Kx

Evaluates a sequence of modified Bessel functions of the third kind with real order and complex arguments.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_bessel_Kx` (*float* `xnu`, *f_complex* `z`, *int* `n`, …, 0)

The type *d_complex* function is `imsl_z_bessel_Jx`.

Required Arguments

float `xnu` (Input)
The lowest order desired. The argument `xnu` must be greater than –1/2.

f_complex `z` (Input)
Argument for which the sequence of Bessel functions is to be evaluated.

int `n` (Input)
Number of elements in the sequence.

Return Value

A pointer to the `n` values of the function through the series. Element *i* contains the value of the Bessel function of order `xnu` + *i* for *i* = 0, …, *n* – 1.

Synopsis with Optional Arguments

f_complex `*imsl_c_bessel_Kx` (*float* `xnu`, *f_complex* `z`,
int `IMSL_RETURN_USER`, *f_complex* `bessel[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *f_complex* `bessel[]` (Output)
Store the sequence of Bessel functions in the user-provided array `bessel[]`.

Description

The Bessel function $K_\nu(z)$ is defined to be

$$K_\nu(z) = \frac{\pi}{2} e^{\nu\pi i/2}\left[iJ_\nu(iz) - Y_\nu(iz)\right] \quad \text{for } -\pi < arg\, z \le \frac{\pi}{2}$$

This function is based on the code BESSCC of Barnett (1981) and Thompson and Barnett (1987).

For moderate or large arguments, *z*, Temme's (1975) algorithm is used to find $K_\nu(z)$. This involves evaluating a continued fraction. If this evaluation fails to converge, the answer may not be accurate. For small *z*, a Neumann series is used to compute $K_\nu(z)$. Upward recurrence of the $K_\nu(z)$ is always stable.

Example

In this example, $K_{0.3+\nu-1}(1.2 + 0.5i)$, $\nu = 1, \ldots, 4$ is computed and printed.

```
#include <imsl.h>

main()
{
    int         n = 4;
    int         i;
    float       xnu = 0.3;
    static f_complex    z = {1.2, 0.5};
    f_complex  *sequence;

    sequence = imsl_c_bessel_Kx(xnu, z, n, 0);

    for (i = 0; i < n; i++)
    printf("K sub %4.2f ((%4.2f,%4.2f)) = (%5.3f,%5.3f)\n",
        xnu+i, z.re, z.im, sequence[i].re, sequence[i].im);
}
```

Output

```
K sub 0.30 ((1.20,0.50)) = (0.246,-0.200)
K sub 1.30 ((1.20,0.50)) = (0.336,-0.362)
K sub 2.30 ((1.20,0.50)) = (0.587,-1.126)
K sub 3.30 ((1.20,0.50)) = (0.719,-4.839)
```

elliptic_integral_K

Evaluates the complete elliptic integral of the kind $K(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_K` (*float* `x`)

The type *double* function is `imsl_d_elliptic_integral_K`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The complete elliptic integral $K(x)$.

Description

The complete elliptic integral of the first kind is defined to be

$$K(x) = \int_0^{\pi/2} \frac{d\theta}{\left[1 - x \sin^2\theta\right]^{1/2}} \text{ for } 0 \le x < 1$$

The argument x must satisfy $0 \le x < 1$; otherwise, `imsl_f_elliptic_integral_K` returns `imsl_f_machine`(2), the largest representable floating-point number.

The function $K(x)$ is computed using the routine **`imsl_f_elliptic_integral_RF`** (page 473) and the relation $K(x) = R_F(0, 1 - x, 1)$.

Example

The integral $K(0)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.0;
        float   ans;

        x = imsl_f_elliptic_integral_K (x);

        printf ("K(0.0) = %f\n", x);
}
```

Output

```
K(0.0) = 1.570796
```

elliptic_integral_E

Evaluates the complete elliptic integral of the second kind $E(x)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_E` (*float* `x`)

The type *double* function is `imsl_d_elliptic_integral_E`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The complete elliptic integral $E(x)$.

Description

The complete elliptic integral of the second kind is defined to be

$$E(x) = \int_0^{\pi/2} \left[1 - x \sin^2\theta\right]^{1/2} d\theta \text{ for } 0 \le x < 1$$

The argument x must satisfy $0 \le x < 1$; otherwise, `imsl_f_elliptic_integral_E` returns `imsl_f_machine`(2), the largest representable floating-point number.

The function $E(x)$ is computed using the routine `imsl_f_elliptic_integral_RF` (page 473) and **`imsl_f_elliptic_integral_RD`** (page 474). The computation is done using the relation

$$E(x) = R_F(0,1-x,1) - \frac{x}{3} R_D(0,1-x,1)$$

Example

The integral $E(0.33)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.33;
        float   ans;

        x = imsl_f_elliptic_integral_E (x);

        printf ("E(0.33) = %f\n", x);
}
```

Output

```
E(0.33) = 1.431832
```

elliptic_integral_RF

Evaluates Carlson's elliptic integral of the first kind $R_F(x, y, z)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_RF` (*float* `x`, *float* `y`, *float* `z`)

The type *double* function is `imsl_d_elliptic_integral_RF`.

Required Arguments

float `x` (Input)
First variable of the incomplete elliptic integral. It must be nonnegative.

float `y` (Input)
Second variable of the incomplete elliptic integral. It must be nonnegative.

float `z` (Input)
Third variable of the incomplete elliptic integral. It must be nonnegative.

Return Value

The complete elliptic integral $R_F(x, y, z)$

Description

Carlson's elliptic integral of the first kind is defined to be

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)(t+z)\right]^{1/2}}$$

The arguments must be nonnegative and less than or equal to $b/5$. In addition, $x + y$, $x + z$, and $y + z$ must be greater than or equal to $5s$. Should any of these conditions fail, `imsl_f_elliptic_integral_RF` is set to b. Here, b = `imsl_f_machine`(2) is the largest and s = `imsl_f_machine`(1) is the smallest representable number.

The function `imsl_f_elliptic_integral_RF` is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral $R_F(0, 1, 2)$ is computed.

```
#include <imsl.h>

main()
{
        float   x = 0.0;
        float   y = 1.0;
        float   z = 2.0;
        float   ans;

        x = imsl_f_elliptic_integral_RF (x, y, z);

        printf ("RF(0, 1, 2) = %f\n", x);
}
```

Output

```
RF(0, 1, 2) = 1.311029
```

elliptic_integral_RD

Evaluates Carlson's elliptic integral of the second kind $R_D(x, y, z)$.

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_RD` (*float* `x`, *float* `y`, *float* `z`)

The type *double* function is `imsl_d_elliptic_integral_RD`.

Required Arguments

float `x` (Input)
First variable of the incomplete elliptic integral. It must be nonnegative.

float `y` (Input)
Second variable of the incomplete elliptic integral. It must be nonnegative.

float `z` (Input)
Third variable of the incomplete elliptic integral. It must be positive.

Return Value

The complete elliptic integral $R_D(x, y, z)$

Description

Carlson's elliptic integral of the first kind is define to be

$$R_D(x,y,z) = \frac{3}{2}\int_0^{\infty} \frac{dt}{\left[(t+x)(t+y)(t+z)^3\right]^{1/2}}$$

The arguments must be nonnegative and less than or equal to $0.69(-ln\varepsilon)^{1/9}s^{-2/3}$ where ε = `imsl_f_machine`(4) is the machine precision, s = `imsl_f_machine`(1) is the smallest representable positive number. Furthermore, $x + y$ and z must be greater than `max`$\{3s^{2/3}, 3/b^{2/3}\}$, where b = `imsl_f_machine`(2) is the largest floating point number. If any of these conditions are false, then `imsl_f_elliptic_integral_RD` returns b.

The function `imsl_f_elliptic_integral_RD` is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral $R_D(0, 2, 1)$ is computed.

```
#include <imsl.h>

main()
{
        float   x = 0.0;
        float   y = 2.0;
        float   z = 1.0;
        float   ans;

        x = imsl_f_elliptic_integral_RD (x, y, z);

        printf ("RD(0, 2, 1) = %f\n", x);
}
```

Output

```
RD(0, 2, 1) = 1.797210
```

elliptic_integral_RJ

Evaluates Carlson's elliptic integral of the third kind $R_J(x, y, z, \rho)$

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_RJ` (*float* `x`, *float* `y`, *float* `z`, *float* `rho`)

The type *double* function is `imsl_d_elliptic_integral_RJ`.

Required Arguments

float `x` (Input)
First variable of the incomplete elliptic integral. It must be nonnegative.

float `y` (Input)
Second variable of the incomplete elliptic integral. It must be nonnegative.

float `z` (Input)
Third variable of the incomplete elliptic integral. It must be positive.

float `rho` (Input)
Fourth variable of the incomplete elliptic integral. It must be positive.

Return Value

The complete elliptic integral $R_J(x, y, z, \rho)$

Description

Carlson's elliptic integral of the third kind is defined to be

$$R_J(x,y,z,\rho) = \frac{3}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)(t+z)(t+\rho)^2\right]^{1/2}}$$

The arguments must be nonnegative. In addition, $x + y$, $x + z$, $y + z$ and ρ must be greater than or equal to $(5s)^{1/3}$ and less than or equal to $0.3(b/5)^{1/3}$, where s = `imsl_f_machine`(1) is the smallest representable floating-point number. Should any of these conditions fail, `imsl_f_elliptic_integral_RJ` is set to b = `imsl_f_machine`(2), the largest floating-point number.

The function `imsl_f_elliptic_integral_RJ` is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral R_J (2, 3, 4, 5) is computed.

```
#include <imsl.h>

main()
{
        float   x = 2.0;
        float   y = 3.0;
        float   z = 4.0;
        float   rho = 5.0;
        float   ans;

        x = imsl_f_elliptic_integral_RJ (x, y, z, rho);

        printf ("RJ(2, 3, 4, 5) = %f\n", x);
}
```

Output

```
RJ(2, 3, 4, 5) = 0.142976
```

elliptic_integral_RC

Evaluates an elementary integral from which inverse circular functions, logarithms and inverse hyperbolic functions can be computed.

Synopsis

#include `<imsl.h>`

float `imsl_f_elliptic_integral_RC` (*float* `x`, *float* `y`)

The type *double* function is `imsl_d_elliptic_integral_RC`.

Required Arguments

float `x` (Input)
First variable of the incomplete elliptic integral. It must be nonnegative and must satisfy the conditions given below.

float `y` (Input)
Second variable of the incomplete elliptic integral. It must be positive and must satisfy the conditions given below.

Return Value

The elliptic integral R_C (x, y).

Description

Carlson's elliptic integral of the third kind is defined to be

$$R_C(x,y) = \frac{1}{2}\int_0^{\infty} \frac{dt}{\left[(t+x)(t+y)^2\right]^{1/2}}$$

The argument x must be nonnegative, y must be positive, and $x + y$ must be less than or equal to $b/5$ and greater than or equal to $5s$. If any of these conditions are false, the `imsl_f_elliptic_integral_RC` is set to b. Here,
b = `imsl_f_machine`(2) is the largest and s = `imsl_f_machine`(1) is the smallest representable floating-point number.

The function `imsl_f_elliptic_integral_RC` is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral R_C (2.25, 2) is computed.

```
#include <imsl.h>

main()
{
        float   x = 2.25;
        float   y = 2.0;
        float   ans;

        x = imsl_f_elliptic_integral_RC (x, y);

        printf ("RC(2.25, 2.0) = %f\n", x);
}
```

Output

```
RC(2.25, 2.0) = 0.693147
```

fresnel_integral_C

Evaluates the cosine Fresnel integral.

Synopsis

#include `<imsl.h>`

float `imsl_f_fresnel_integral_C` (*float* `x`)

The type *double* function is `imsl_d_fresnel_integral_C`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The cosine Fresnel integral.

Description

The cosine Fresnel integral is defined to be

$$C(x) = \int_0^x cos(\frac{\pi}{2} t^2)dt$$

Example

The Fresnel integral $C(1.75)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 1.75;
        float   ans;

        x = imsl_f_fresnel_integral_C (x);

        printf ("C(1.75) = %f\n", x);
}
```

Output

```
C(1.75) = 0.321935
```

fresnel_integral_S

Evaluates the sine Fresnel integral.

Synopsis

#include `<imsl.h>`

float `imsl_f_fresnel_integral_S` (*float* `x`)

The type *double* function is `imsl_d_fresnel_integral_S`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The sine Fresnel integral.

Description

The sine Fresnel integral is defined to be

$$S(x) = \int_0^x sin(\frac{\pi}{2} t^2)dt$$

Example

The Fresnel integral $S(1.75)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 1.75;
        float   ans;

        x = imsl_f_fresnel_integral_S (x);

        printf ("S(1.75) = %f\n", x);
}
```

Output

```
S(1.75) = 0.499385
```

airy_Ai

Evaluates the Airy function.

Synopsis

#include `<imsl.h>`

float `imsl_f_airy_Ai` (*float* x)

The type *double* function is `imsl_d_airy_Ai`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Airy function evaluated at *x*, Ai(*x*).

Description

The airy function Ai(*x*) is defined to be

$$Ai(x) = \frac{1}{\pi}\int_0^{\infty} cos(xt + \frac{1}{3}t^3)dt = \sqrt{\frac{x}{3\pi^2}}K_{1/3}(\frac{2}{3}x^{3/2})$$

The Bessel function $K_\nu(x)$ is defined on page 470.

If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε = `imsl_f_machine`(4) is the machine precision.

Finally, x should be less than x_{max} so the answer does not underflow. Very approximately, $x_{max} = \{-1.5\ln s\}^{2/3}$, where s = `imsl_f_machine`(1), the smallest representable positive number.

Example

In this example, Ai(−4.9) is evaluated.

```
#include <imsl.h>

main()
{
        float   x = -4.9;
        float   ans;

        x = imsl_f_airy_Ai (x);

        printf ("Ai(-4.9) = %f\n", x);
}
```

Output

```
Ai(-4.9) = 0.374536
```

airy_Bi

Evaluates the Airy function of the second kind.

Synopsis

#include `<imsl.h>`

float `imsl_f_airy_Bi` (*float* `x`)

The type double function is `imsl_d_airy_Bi`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Airy function of the second kind evaluated at x, Bi(x).

Description

The airy function Bi(x) is defined to be

$$\text{Bi}(x) = \frac{1}{\pi}\int_0^\infty exp(xt - \frac{1}{3}t^3)dt + \frac{1}{\pi}\int_0^\infty sin(xt + \frac{1}{3}t^3)dt$$

It can also be expressed in terms of modified Bessel functions of the first kind, $I_v(x)$, and Bessel functions of the first kind $J_v(x)$ (see **bessel_Ix** and **bessel_Jx**:

$$\text{Bi}(x) = \sqrt{\frac{x}{3}}\left[I_{-1/3}(\frac{2}{3}x^{3/2}) + I_{1/3}(\frac{2}{3}x^{3/2})\right] \text{ for } x > 0$$

and

$$\text{Bi}(x) = \sqrt{\frac{-x}{3}}\left[J_{-1/3}(\frac{2}{3}|x|^{3/2}) - J_{1/3}(\frac{2}{3}|x|^{3/2})\right] \text{ for } x < 0$$

Let ε = `imsl_f_machine`(4), the machine precision. If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. In addition, x should not be so large that $exp[(2/3)x^{3/2}]$ overflows.

Example

In this example, Bi(–4.9) is evaluated.

```
#include <imsl.h>

main()
{
        float   x = -4.9;
        float   ans;

        x = imsl_f_airy_Bi (x);

        printf ("Bi(-4.9) = %f\n", x);
}
```

Output

```
Bi(-4.9) = -0.057747
```

airy_Ai_derivative

Evaluates the derivative of the Airy function.

Synopsis

#include `<imsl.h>`

float `imsl_f_airy_Ai_derivative` (*float* `x`)

The type *double* function is `imsl_d_airy_Ai_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Airy function.

Description

The airy function Ai′(x) is defined to be the derivative of the Airy function, Ai(x) (see page 480). If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε = `imsl_f_machine`(4) is the machine precision. Finally, x should be less than x_{max} so that the answer does not underflow. Very approximately, $x_{max} = \{-1.51\ln s\}$, where s = `imsl_f_machine`(1), the smallest representable positive number.

Example

In this example, Ai′(−4.9) is evaluated.

```
#include <imsl.h>

main()
{
        float   x = -4.9;
        float   ans;

        x = imsl_f_airy_Ai_derivative (x);

        printf ("Ai'(-4.9) = %f\n", x);
}
```

Output

```
Ai'(-4.9) = 0.146958
```

airy_Bi_derivative

Evaluates the derivative of the Airy function of the second kind.

Synopsis

#include `<imsl.h>`

float `imsl_f_airy_Bi_derivative` (*float* `x`)

The type *double* function is `imsl_d_airy_Bi_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Airy function of the second kind.

Description

The airy function Bi′(*x*) is defined to be the derivative of the Airy function of the second kind, Bi(*x*) (see page 481). If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε = `imsl_f_machine`(4) is the machine precision. In addition, *x* should not be so large that $exp[(2/3)x^{3/2}]$ overflows.

Example

In this example, Bi′(−4.9) is evaluated.

```
#include <imsl.h>

main()
{
        float   x = -4.9;
        float   ans;

        x = imsl_f_airy_Bi_derivative (x);

        printf ("Bi'(-4.9) = %f\n", x);
}
```

Output

```
Bi'(-4.9) = 0.827219
```

kelvin_ber0

Evaluates the Kelvin function of the first kind, `ber`, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_ber0` (*float* `x`)

The type double function is `imsl_d_kelvin_ber0`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Kelvin function of the first kind, `ber`, of order zero evaluated at *x*.

Description

The Kelvin function $\texttt{ber}_0(x)$ is defined to be $\Re J_0(xe^{3\pi i/4})$. The Bessel function $J_0(x)$ is defined

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,sin\,\theta)d\theta$$

The function `imsl_f_kelvin_ber0` is based on the work of Burgoyne (1963).

Example

In this example, $\texttt{ber}_0$ (0.4) is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.4;
        float   ans;

        x = imsl_f_kelvin_ber0 (x);

        printf ("ber0(0.4) = %f\n", x);
}
```

Output

```
ber0(0.4) = 0.999600
```

kelvin_bei0

Evaluates the Kelvin function of the first kind, bei, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_bei0` (*float* `x`)

The type *double* function is `imsl_d_kelvin_bei0`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Kelvin function of the first kind, bei, of order zero evaluated at *x*.

Description

The Kelvin function $\text{bie}_0(x)$ is defined to be $\Im J_0(xe^{3\pi i/4})$. The Bessel function $J_0(x)$ is defined

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,sin\,\theta)d\theta$$

The function `imsl_f_kelvin_bei0` is based on the work of Burgoyne (1963).

In `imsl_f_kelvin_bei0`, *x* must be less than 119.

Example

In this example, $\text{bei}_0(0.4)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.4;
        float   ans;

        x = imsl_f_kelvin_bei0 (x);

        printf ("bei0(0.4) = %f\n", x);
}
```

Output

```
bei0(0.4) = 0.039998
```

kelvin_ker0

Evaluates the Kelvin function of the second kind, ker, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_ker0` (*float* `x`)

The type *double* function is `imsl_d_kelvin_ker0`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Kelvin function of the second kind, ker, of order zero evaluated at *x*.

Description

The modified Kelvin function $\ker_0(x)$ is defined to be $\Re K_0(xe^{\pi i/4})$. The Bessel function $K_0(x)$ is defined

$$K_0(x) = \int_0^\infty \cos(x \sin t)\, dt$$

The function `imsl_f_kelvin_ker0` is based on the work of Burgoyne (1963).

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, then zero is returned.

Example

In this example, $\ker_0(0.4)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.4;
        float   ans;

        x = imsl_f_kelvin_ker0 (x);

        printf ("ker0(0.4) = %f\n", x);
}
```

Output

```
ker0(0.4) = 1.062624
```

kelvin_kei0

Evaluates the Kelvin function of the second kind, kei, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_kei0` (*float* `x`)

The type *double* function is `imsl_d_kelvin_kei0`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The Kelvin function of the second kind, kei, of order zero evaluated at *x*.

Description

The modified Kelvin function $\texttt{kei}_0(x)$ is defined to be $\Im K_0(xe^{\pi i/4})$. The Bessel function $K_0(x)$ is defined

$$K_0(x) = \int_0^\infty cos(x \sin t)\, dt$$

The function `imsl_f_kelvin_kei0` is based on the work of Burgoyne (1963).

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, zero is returned.

Example

In this example, $\texttt{kei}_0(0.4)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.4;
        float   ans;

        x = imsl_f_kelvin_kei0 (x);

        printf ("kei0(0.4) = %f\n", x);
}
```

Output

```
kei0(0.4) = -0.703800
```

kelvin_ber0_derivative

Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_ber0_derivative` (*float* `x`)

The type *double* function is `imsl_d_kelvin_ber0_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Kelvin function of the first kind, ber, of order zero evaluated at *x*.

Description

The function $\mathrm{ber}_0'(x)$ is defined to be

$$\frac{d}{dx}\mathrm{ber}_0(x)$$

The function `imsl_f_kelvin_ber0_derivative` is based on the work of Burgoyne (1963).

If $|x| > 119$, NaN is returned.

Example

In this example, $\mathrm{ber}_0'(0.6)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.6;
        float   ans;

        x = imsl_f_kelvin_ber0_derivative (x);

        printf ("ber0'(0.6) = %f\n", x);
}
```

Output

```
ber0'(0.6) = -0.013498
```

kelvin_bei0_derivative

Evaluates the derivative of the Kelvin function of the first kind, bei, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_bei0_derivative` (*float* `x`)

The type *double* function is `imsl_d_kelvin_bei0_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Kelvin function of the first kind, bei, of order zero evaluated at *x*.

Description

The function $\text{bei}_0'(x)$ is defined to be

$$\frac{d}{dx}\text{bei}_0(x)$$

The function `imsl_f_kelvin_bei0_derivative` is based on the work of Burgoyne (1963).

If $|x| > 119$, NaN is returned.

Example

In this example, $\text{bei}_0'(0.6)$ is evaluated.

```
#include <imsl.h>
main()
{
        float   x = 0.6;
        float   ans;

        x = imsl_f_kelvin_bei0_derivative (x);

        printf ("bei0'(0.6) = %f\n", x);
}
```

Output

```
bei0'(0.6) = 0.299798
```

kelvin_ker0_derivative

Evaluates the derivative of the Kelvin function of the second kind, ker, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_ker0_derivative` (*float* `x`)

The type *double* function is `imsl_d_kelvin_ker0_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Kelvin function of the second kind, ker, of order zero evaluated at *x*.

Description

The function $\texttt{ker}_0'(x)$ is defined to be

$$\frac{d}{dx}\ker_0(x)$$

The function `imsl_f_kelvin_ker0_derivative` is based on the work of Burgoyne (1963).

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, zero is returned.

Example

In this example, $\texttt{ker}_0'(0.6)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.6;
        float   ans;

        x = imsl_f_kelvin_ker0_derivative (x);

        printf ("ker0'(0.6) = %f\n", x);
}
```

Output

```
ker0'(0.6) = -1.456538
```

kelvin_kei0_derivative

Evaluates the derivative of the Kelvin function of the second kind, kei, of order zero.

Synopsis

#include `<imsl.h>`

float `imsl_f_kelvin_kei0_derivative` (*float* `x`)

The type *double* function is `imsl_d_kelvin_kei0_derivative`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

The derivative of the Kelvin function of the second kind, `kei`, of order zero evaluated at *x*.

Description

The function $\texttt{kei}_0'(x)$ is defined to be

$$\frac{d}{dx}\text{kei}_0(x)$$

The function `imsl_f_kelvin_kei0_derivative` is based on the work of Burgoyne (1963).

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, zero is returned.

Example

In this example, $\texttt{kei}_0'(0.6)$ is evaluated.

```
#include <imsl.h>

main()
{
        float   x = 0.6;
        float   ans;

        x = imsl_f_kelvin_kei0_derivative (x);

        printf ("kei0'(0.6) = %f\n", x);
}
```

Output

```
kei0'(0.6) = 0.348164
```

normal_cdf

Evaluates the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_normal_cdf` (*float* `x`)

The type *double* function is `imsl_d_normal_cdf`.

Required Arguments

float `x` (Input)
Point at which the normal distribution function is to be evaluated.

Return Value

The probability that a normal random variable takes a value less than or equal to *x*.

Description

The function `imsl_f_normal_cdf` evaluates the distribution function, Φ, of a standard normal (Gaussian) random variable; that is,

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The standard normal distribution (for which `imsl_f_normal_cdf` is the distribution function) has mean of 0 and variance of 1. The probability that a normal random variable with mean μ and variance σ^2 is less than *y* is given by `imsl_f_normal_cdf` evaluated at $(y - \mu)/\sigma$.

Φ(*x*) is evaluated by use of the complementary error function, `imsl_f_erfc`. The relationship is:

$$\Phi(x) = \text{erfc}\left(-x / \sqrt{2.0}\right) / 2$$

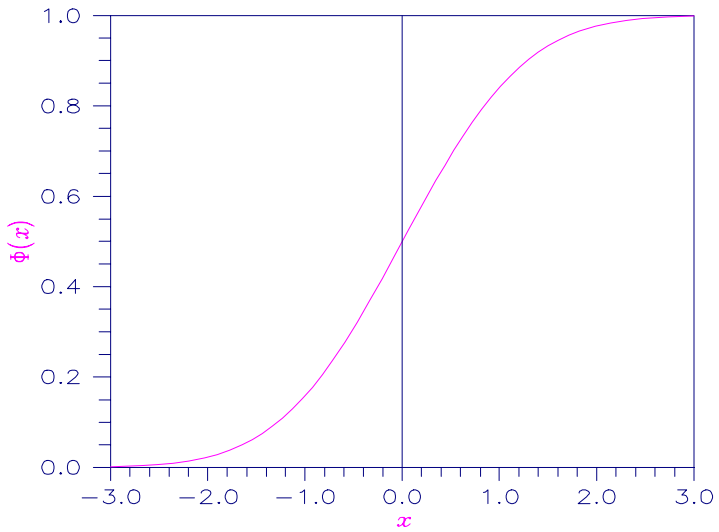


Figure 9-13 Plot of Φ(x)

Example

Suppose *X* is a normal random variable with mean 100 and variance 225. This example finds the probability that *X* is less than 90 and the probability that *X* is between 105 and 110.

```
#include <imsl.h>

main()
{
    float       p, x1, x2;
```

```
    x1  = (90.0-100.0)/15.0;
    p   = imsl_f_normal_cdf(x1);
    printf("The probability that X is less than 90 is %6.4f\n\n", p);

    x1 = (105.0-100.0)/15.0;
    x2 = (110.0-100.0)/15.0;
    p  = imsl_f_normal_cdf(x2) - imsl_f_normal_cdf(x1);
     printf("The probability that X is between 105 and 110 is %6.4f\n", p);
}
```

Output

```
The probability that X is less than 90 is 0.2525

The probability that X is between 105 and 110 is 0.1169
```

normal_inverse_cdf

Evaluates the inverse of the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_normal_inverse_cdf` (*float* `p`)

The type *double* procedure is `imsl_d_normal_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the normal distribution function is to be evaluated. The argument `p` must be in the open interval (0.0, 1.0).

Return Value

The inverse of the normal distribution function evaluated at `p`. The probability that a standard normal random variable takes a value less than or equal to `imsl_f_normal_inverse_cdf` is `p`.

Description

The function `imsl_f_normal_inverse_cdf` evaluates the inverse of the distribution function, Φ, of a standard normal (Gaussian) random variable; that is, `imsl_f_normal_inverse_cdf`(p) = Φ^{-1} (p) where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x. The standard normal distribution has a mean of 0 and a variance of 1.

The function `imsl_f_normal_inverse_cdf(`*p*`)` is evaluated by use of minimax rational-function approximations for the inverse of the error function. General descriptions of these approximations are given in Hart et al. (1968) and Strecok (1968). The rational functions used in `imsl_f_normal_inverse_cdf` are described by Kinnucan and Kuki (1968).

Example

This example computes the point such that the probability is 0.9 that a standard normal random variable is less than or equal to this point.

```
#include <imsl.h>

main()
{
    float       x;
    float       p = 0.9;

    x = imsl_f_normal_inverse_cdf(p);
    printf("The 90th percentile of a standard normal is %6.4f.\n", x);
}
```

Output

```
The 90th percentile of a standard normal is 1.2816.
```

chi_squared_cdf

Evaluates the chi-squared distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_chi_squared_cdf` (*float* `chi_squared`, *float* `df`)

The type *double* function is `imsl_d_chi_squared_cdf`.

Required Arguments

float `chi_squared` (Input)
Argument for which the chi-squared distribution function is to be evaluated.

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. The argument `df` must be greater than or equal to 0.5.

Return Value

The probability that a chi-squared random variable takes a value less than or equal to `chi_squared`.

Description

The function `imsl_f_chi_squared_cdf` evaluates the distribution function, *F*, of a chi-squared random variable `x = chi_squared` with `ν = df`. Then,

$$F(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

For $\nu > 65$, `imsl_f_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) to the normal distribution, and function `imsl_f_normal_cdf` is used to evaluate the normal distribution function.

For $\nu \leq 65$, `imsl_f_chi_squared_cdf` uses series expansions to evaluate the distribution function. If $x < max(\nu/2, 26)$, `imsl_f_chi_squared_cdf` uses the series 6.5.29 in Abramowitz and Stegun (1964); otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.

Example

Suppose *X* is a chi-squared random variable with 2 degrees of freedom. This example finds the probability that *X* is less than 0.15 and the probability that *X* is greater than 3.0.

```
#include <imsl.h>

void main()
{
    float       chi_squared = 0.15;
    float       df = 2.0;
    float       p;

    p    = imsl_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared",
           "with 2 df is less than 0.15 is", p);

    chi_squared = 3.0;
    p    = 1.0 - imsl_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared",
           "with 2 df is greater than 3.0 is", p);
}
```

Output

```
The probability that chi-squared with 2 df is less than 0.15 is 0.0723
The probability that chi-squared with 2 df is greater than 3.0 is 0.2231
```

Informational Errors

`IMSL_ARG_LESS_THAN_ZERO`	The input argument, `chi_squared`, is less than zero.

Alert Errors

`IMSL_NORMAL_UNDERFLOW`	Using the normal distribution for large degrees of freedom, underflow would have occurred.

chi_squared_inverse_cdf

Evaluates the inverse of the chi-squared distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_chi_squared_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsl_d_chi_squared_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the chi-squared distribution function is to be evaluated. The argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. The argument `df` must be greater than or equal to 0.5.

Return Value

The inverse of the chi-squared distribution function evaluated at `p`. The probability that a chi-squared random variable takes a value less than or equal to `imsl_f_chi_squared_inverse_cdf` is `p`.

Description

The function `imsl_f_chi_squared_inverse_cdf` evaluates the inverse distribution function of a chi-squared random variable with ν = `df` and with probability p. That is, it determines x = `imsl_f_chi_squared_inverse_cdf(p,df)` such that

$$p = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is p.

For $\nu < 40$, `imsl_f_chi_squared_inverse_cdf` uses bisection (if $\nu \le 2$ or $p > 0.98$) or regula falsi to find the point at which the chi-squared distribution function is equal to p. The distribution function is evaluated using function `imsl_f_chi_squared_cdf`.

For $40 \le \nu < 100$, a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.18) to the normal distribution is used. The function

`imsl_f_normal_cdf` is used to evaluate the inverse of the normal distribution function. For $\nu \geq 100$, the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.17) is used.

Example

In this example, the 99-th percentage point is calculated for a chi-squared random variable with two degrees of freedom. The same calculation is made for a similar variable with 64 degrees of freedom.

```
#include <imsl.h>

void main ()
{
    float       df, x;
    float       p = 0.99;

    df = 2.0;
    x  = imsl_f_chi_squared_inverse_cdf(p, df);
    printf("For p = .99 with  2 df, x = %7.3f.\n", x);

    df = 64.0;
    x  = imsl_f_chi_squared_inverse_cdf(p,df);
    printf("For p = .99 with 64 df, x = %7.3f.\n", x);
}
```

Output

```
For p = .99 with  2 df, x =   9.210.
For p = .99 with 64 df, x =  93.217.
```

Warning Errors

`IMSL_UNABLE_TO_BRACKET_VALUE`	The bounds that enclose *p* could not be found. An approximation for `imsl_f_chi_squared_inverse_cdf` is returned.
`IMSL_CHI_2_INV_CDF_CONVERGENCE`	The value of the inverse chi-squared could not be found within a specified number of iterations. An approximation for `imsl_f_chi_squared_inverse_cdf` is returned.

F_cdf

Evaluates the *F* distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_F_cdf` (*float* `f`, *float* `df_denominator`, *float* `df_numerator`)

The type *double* function is `imsl_d_F_cdf`.

Required Arguments

float `f` (Input)
Point at which the *F* distribution function is to be evaluated.

float `df_numerator` (Input)
The numerator degrees of freedom. The argument `df_numerator` must be positive.

float `df_denominator` (Input)
The denominator degrees of freedom. The argument `df_denominator` must be positive.

Return Value

The probability that an *F* random variable takes a value less than or equal to the input point, `f`.

Description

The function `imsl_f_F_cdf` evaluates the distribution function of a Snedecor's *F* random variable with `df_numerator` and `df_denominator`. The function is evaluated by making a transformation to a beta random variable and then by evaluating the incomplete beta function. If X is an *F* variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1 X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$.

The function `imsl_f_F_cdf` also uses a relationship between *F* random variables that can be expressed as follows:

$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$where F_F is the distribution function for an *F* random variable.

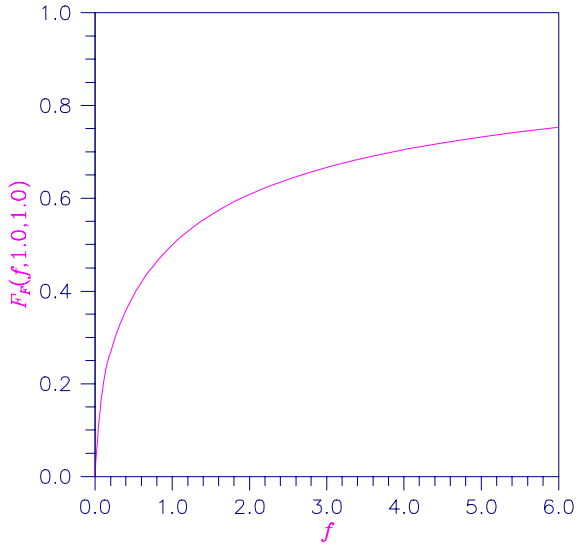


Figure 9-14 Plot of F_F (f, 1.0, 1.0)

Example

This example finds the probability that an *F* random variable with one numerator and one denominator degree of freedom is greater than 648.

```
#include <imsl.h>

main()
{
    float       p;
    float       F = 648.0;
    float       df_numerator = 1.0;
    float       df_denominator = 1.0;

    p = 1.0 - imsl_f_F_cdf(F,df_numerator, df_denominator);
    printf("%s %s %6.4f.\n", "The probability that an F(1,1) variate",
           "is greater than 648 is", p);
}
```

Output

```
The probability that an F(1,1) variate is greater than 648 is 0.0250.
```

F_inverse_cdf

Evaluates the inverse of the *F* distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_F_inverse_cdf` (*float* `p`, *float* `df_numerator`, *float* `df_denominator`)

The type *double* procedure is `imsl_d_F_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the *F* distribution function is to be evaluated. The argument `p` must be in the open interval (0.0, 1.0).

float `df_numerator` (Input)
Numerator degrees of freedom. Argument `df_numerator` must be positive.

float `df_denominator` (Input)
Denominator degrees of freedom. Argument `df_denominator` must be positive.

Return Value

The value of the inverse of the *F* distribution function evaluated at `p`. The probability that an *F* random variable takes a value less than or equal to `imsl_f_F_inverse_cdf` is `p`.

Description

The function `imsl_f_F_inverse_cdf` evaluates the inverse distribution function of a Snedecor's *F* random variable with ν_1 = `df_numerator` numerator degrees of freedom and ν_2 = `df_denominator` denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable and then by evaluating the inverse of an incomplete beta function. If X is an *F* variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1, X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. If $P \leq 0.5$, `imsl_f_F_inverse_cdf` uses this relationship directly; otherwise, it also uses a relationship between *F* random variables that can be expressed as follows:

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

Example

In this example, the 99-th percentage point is calculated for an *F* random variable with seven degrees of freedom. The same calculation is made for a similar variable with one degree of freedom.

```
#include <imsl.h>

main()
{
    float        df_denominator = 1.0;
    float        df_numerator = 7.0;
    float        f;
    float        p = 0.99;

    f = imsl_f_F_inverse_cdf(p, df_numerator, df_denominator);

    printf("The F(7,1) 0.01 critical value is %6.3f\n", f);
}
```

Output

```
The F(7,1) 0.01 critical value is 5928.370
```

Fatal Errors

`IMSL_F_INVERSE_OVERFLOW`	Function `imsl_f_F_inverse_cdf` is set to machine infinity since overflow would occur upon modifying the inverse value for the *F* distribution with the result obtained from the inverse beta distribution.

t_cdf

Evaluates the Student's *t* distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_t_cdf` (*float* `t`, *float* `df`)

The type *double* function is `imsl_d_t_cdf`.

Required Arguments

float `t` (Input)
Argument for which the Student's *t* distribution function is to be evaluated.

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The probability that a Student's *t* random variable takes a value less than or equal to the input `t`.

Description

The function `imsl_f_t_cdf` evaluates the distribution function of a Student's *t* random variable with ν_1 = `df` degrees of freedom. If the square of *t* is greater than or equal to ν, the relationship of a *t* to an *F* random variable (and subsequently, to a beta random variable) is exploited, and percentage points from a beta distribution are used. Otherwise, the method described by Hill (1970) is used. If ν is not an integer, if ν is greater than 19, or if ν is greater than 200, a Cornish-Fisher expansion is used to evaluate the distribution function. If ν is less than 20 and $|t|$ is less than 2.0, a trigonometric series (see Abramowitz and Stegun 1964, equations 26.7.3 and 26.7.4, with some rearrangement) is used. For the remaining cases, a series given by Hill (1970) that converges well for large values of *t* is used.

Example

This example finds the probability that a *t* random variable with six degrees of freedom is greater in absolute value than 2.447. The fact that *t* is symmetric about zero is used.

```
#include <imsl.h>

main ()
{
    float       p;
    float       t = 2.447;
    float       df = 6.0;

    p  = 2.0*imsl_f_t_cdf(-t,df);
    printf("Pr(|t(6)| > 2.447) = %6.4f\n", p);
}
```

Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

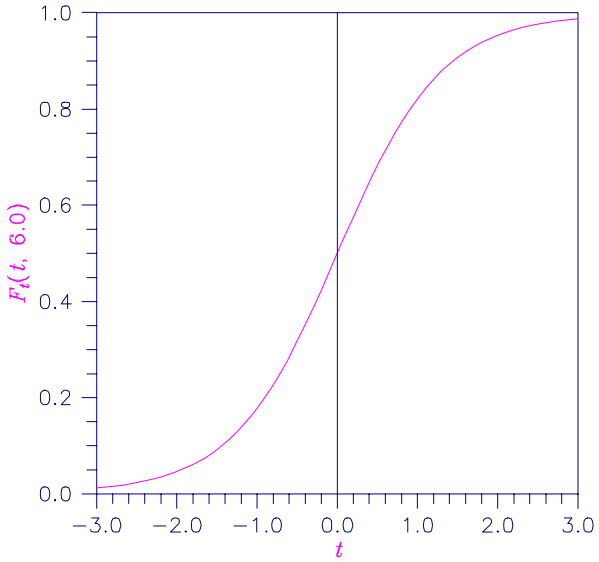


Figure 9-15 Plot of F_t(t, 6.0)

t_inverse_cdf

Evaluates the inverse of the Student's *t* distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_t_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsl_d_t_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the Student's *t* distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The inverse of the Student's *t* distribution function evaluated at `p`. The probability that a Student's *t* random variable takes a value less than or equal to `imsl_f_t_inverse_cdf` is `p`.

Description

The function `imsl_f_t_inverse_cdf` evaluates the inverse distribution function of a Student's *t* random variable with $\nu = df$ degrees of freedom. If ν equals 1 or 2, the inverse can be obtained in closed form. If ν is between 1 and 2, the relationship of a *t* to a beta random variable is exploited, and the inverse of the beta distribution is used to evaluate the inverse; otherwise, the algorithm of Hill (1970) is used. For small values of ν greater than 2, Hill's algorithm inverts an integrated expansion in $1/(1 + t^2/\nu)$ of the *t* density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

Example

This example finds the 0.05 critical value for a two-sided *t* test with six degrees of freedom.

```
#include <imsl.h>

void main()
{
    float       df = 6.0;
    float       p = 0.975;
    float       t;

    t  = imsl_f_t_inverse_cdf(p,df);

    printf("The two-sided t(6) 0.05 critical value is %6.3f\n", t);
}
```

Output

```
The two-sided t(6) 0.05 critical value is  2.447
```

Informational Errors

`IMSL_OVERFLOW`	Function `imsl_f_t_inverse_cdf` is set to machine infinity since overflow would occur upon modifying the inverse value for the *F* distribution with the result obtained from the inverse beta distribution.

gamma_cdf

Evaluates the gamma distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_gamma_cdf` (*float* `x`, *float* `a`)

The type *double* procedure is `imsl_d_gamma_cdf`.

Required Arguments

float `x` (Input)
Argument for which the gamma distribution function is to be evaluated.

float `a` (Input)
The shape parameter of the gamma distribution. This parameter must be positive.

Return Value

The probability that a gamma random variable takes a value less than or equal to `x`.

Description

The function `imsl_f_gamma_cdf` evaluates the distribution function, *F*, of a gamma random variable with shape parameter *a*, that is,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. (The gamma function is the integral from zero to infinity of the same integrand as above). The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The gamma distribution is often defined as a two-parameter distribution with a scale parameter *b* (which must be positive) or even as a three-parameter distribution in which the third parameter *c* is a location parameter.

In the most general case, the probability density function over (c, ∞) is

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}$$

If *T* is such a random variable with parameters *a*, *b*, and *c*, the probability that $T \le t_0$ can be obtained from `imsl_f_gamma_cdf` by setting $x = (t_0 - c)/b$.

If *x* is less than *a* or if *x* is less than or equal to 1.0, `imsl_f_gamma_cdf` uses a series expansion. Otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun 1964.)

Example

Let *X* be a gamma random variable with a shape parameter of four. (In this case, it has an *Erlang distribution* since the shape parameter is an integer.) This example finds the probability that *X* is less than 0.5 and the probability that *X* is between 0.5 and 1.0.

```
#include <imsl.h>

main()
{
    float       p, x;
```

```
    float       a = 4.0;

    x = 0.5;
    p = imsl_f_gamma_cdf(x,a);
    printf("The probability that X is less than 0.5 is %6.4f\n", p);

    x = 1.0;
    p = imsl_f_gamma_cdf(x,a) - p;
    printf("The probability that X is between 0.5 and 1.0 is %6.4f\n", p);
}
```

Output

```
The probability that X is less than 0.5 is 0.0018
The probability that X is between 0.5 and 1.0 is 0.0172
```

Informational Errors

IMSL_LESS_THAN_ZERO	The input argument, x, is less than zero.

Fatal Errors

IMSL_X_AND_A_TOO_LARGE	The function overflows because *x* and *a* are too large.

binomial_cdf

Evaluates the binomial distribution function.

Synopsis

#include <imsl.h>

float imsl_f_binomial_cdf (*int* k, *int* n, *float* p)

The type *double* procedure is imsl_d_binomial_cdf.

Required Arguments

int k (Input)
Argument for which the binomial distribution function is to be evaluated.

int n (Input)
Number of Bernoulli trials.

float p (Input)
Probability of success on each trial.

Return Value

The probability that *k* or fewer successes occur in *n* independent Bernoulli trials, each of which has a probability *p* of success.

Description

The function `imsl_f_binomial_cdf` evaluates the distribution function of a binomial random variable with parameters n and p. It does this by summing probabilities of the random variable taking on the specific values in its range. These probabilities are computed by the recursive relationship

$$Pr(X = j) = \frac{(n+1-j)p}{j(1-p)} Pr(X = j-1)$$

To avoid the possibility of underflow, the probabilities are computed forward from zero if k is not greater than $n \times p$; otherwise, they are computed backward from n. The smallest positive machine number, ε, is used as the starting value for summing the probabilities, which are rescaled by $(1 - p)^n \varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done.

For the special case of p is zero, `imsl_f_binomial_cdf` is set to 1; and for the case p is 1, `imsl_f_binomial_cdf` is set to 1 if $k = n$ and is set to zero otherwise.

Example

Suppose X is a binomial random variable with an $n = 5$ and a $p = 0.95$. This example finds the probability that X is less than or equal to three.

```
#include <imsl.h>

void main()
{
    int         k = 3;
    int         n = 5;
    float       p = 0.95;
    float       pr;

    pr = imsl_f_binomial_cdf(k,n,p);
    printf("Pr(x <= 3) = %6.4f\n", pr);
}
```

Output

```
Pr(x <= 3) = 0.0226
```

Informational Errors

`IMSL_LESS_THAN_ZERO`	The input argument, k, is less than zero.
`IMSL_GREATER_THAN_N`	The input argument, k, is greater than the number of Bernoulli trials, n.

hypergeometric_cdf

Evaluates the hypergeometric distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_hypergeometric_cdf` (*int* `k`, *int* `n`, *int* `m`, *int* `l`)

The type *double* procedure is `imsl_d_hypergeometric_cdf`.

Required Arguments

int `k` (Input)
Argument for which the hypergeometric distribution function is to be evaluated.

int `n` (Input)
Sample size `n` must be greater than or equal to `k`.

int `m` (Input)
Number of defectives in the lot.

int `l` (Input)
Lot size `l` must be greater than or equal to `n` and `m`.

Return Value

The probability that *k* or fewer defectives occur in a sample of size *n* drawn from a lot of size *l* that contains *m* defectives.

Description

The function `imsl_f_hypergeometric_cdf` evaluates the distribution function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *x* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type. The probability function is

$$Pr(x = j) = \frac{\binom{m}{j}\binom{l-m}{n-j}}{\binom{l}{n}} \qquad \text{for } j = i, i+1, \ldots, min(n,m)$$

where $i = max\ (0, n - l + m)$.

If *k* is greater than or equal to *i* and less than or equal to min (*n*, *m*), `imsl_f_hypergeometric_cdf` sums the terms in this expression for *j* going from *i* up to *k*. Otherwise, 0 or 1 is returned, as appropriate.

To avoid rounding in the accumulation, `imsl_f_hypergeometric_cdf` performs the summation differently, depending on whether *k* is greater than the mode of the

distribution, which is the greatest integer in $(m + 1)(n + 1)/(l + 2)$.

Example

Suppose X is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. This example evaluates the distribution function at 7.

```
#include <imsl.h>

void main()
{
    int         k = 7;
    int         l = 1000;
    int         m = 70;
    int         n = 100;
    float       p;

    p = imsl_f_hypergeometric_cdf(k,n,m,l);
    printf("\nPr (x <= 7) = %6.4f", p);
}
```

Output

```
Pr (x <= 7) = 0.599
```

Informational Errors

`IMSL_LESS_THAN_ZERO`	The input argument, *k*, is less than zero.
`IMSL_K_GREATER_THAN_N`	The input argument, *k*, is greater than the sample size.

Fatal Errors

`IMSL_LOT_SIZE_TOO_SMALL`	Lot size must be greater than or equal to *n* and *m*.

poisson_cdf

Evaluates the Poisson distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_poisson_cdf` (*int* `k`, *float* `theta`)

The type *double* function is `imsl_d_poisson_cdf`.

Required Arguments

int `k` (Input)
Argument for which the Poisson distribution function is to be evaluated.

float `theta` (Input)
Mean of the Poisson distribution. Argument `theta` must be positive.

Return Value

The probability that a Poisson random variable takes a value less than or equal to k.

Description

The function `imsl_f_poisson_cdf` evaluates the distribution function of a Poisson random variable with parameter `theta`. The mean of the Poisson random variable, `theta`, must be positive. The probability function (with θ = `theta`) is

$$f(x) = e^{-\theta}\,\theta^{x}/x!, \text{ for } x = 0, 1, 2, \ldots$$

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. The function `imsl_f_poisson_cdf` uses the recursive relationship

$$f(x+1) = f(x)\theta/(x+1), \text{ for } x = 0, 1, 2, \ldots, k-1$$

with $f(0) = e^{-\theta}$.

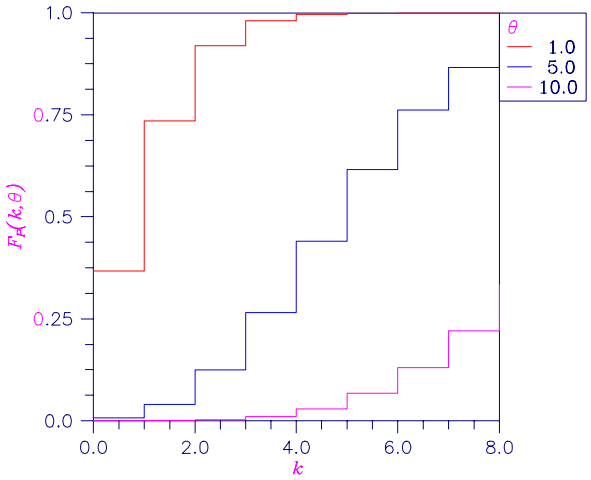


Figure 9-16 Plot of $F_p(k, \theta)$

Example

Suppose X is a Poisson random variable with $\theta = 10$. This example evaluates the probability that $X \leq 7$.

```
#include <imsl.h>

void main()
{
    int         k = 7;
    float       theta = 10.0;
    float       p;

    p = imsl_f_poisson_cdf(k, theta);
```

```
    printf("Pr(x <= 7) = %6.4f\n", p);
}
```

Output

```
Pr(x <= 7) = 0.2202
```

Informational Errors

`IMSL_LESS_THAN_ZERO`	The input argument, *k*, is less than zero.

beta_cdf

Evaluates the beta probability distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_beta_cdf` (*float* `x`, *float* `pin`, *float* `qin`)

The type *double* function is `imsl_d_beta_cdf`.

Required Arguments

float `x` (Input)
Argument for which the beta probability distribution function is to be evaluated.

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

The probability that a beta random variable takes on a value less than or equal to *x*.

Description

Function `imsl_f_beta_cdf` evaluates the distribution function of a beta random variable with parameters `pin` and `qin`. This function is sometimes called the incomplete beta ratio and with p = `pin` and q = `qin`, is denoted by $I_x(p, q)$. It is given by

$$I_x(p,q) = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function by $I_x(p, q)$ is the probability that the random variable takes a value less than or equal to *x*.

The integral in the expression above is called the incomplete beta function and is denoted by $\beta_x(p, q)$. The constant in the expression is the reciprocal of the beta function (the incomplete function evaluated at one) and is denoted by $\beta(p, q)$.

Function `beta_cdf` uses the method of Bosten and Battiste (1974).

Example

Suppose X is a beta random variable with parameters 12 and 12. (X has a symmetric distribution.) This example finds the probability that X is less than 0.6 and the probability that X is between 0.5 and 0.6. (Since X is a symmetric beta random variable, the probability that it is less than 0.5 is 0.5.)

```
#include <imsl.h>

main()
{
    float           p, pin, qin, x;

    pin = 12.0;
    qin = 12.0;
    x = 0.6;
    p = imsl_f_beta_cdf(x, pin, qin);
    printf(" The probability that X is less than 0.6 is %6.4f\n",
        p);
    x = 0.5;
    p -= imsl_f_beta_cdf(x, pin, qin);
    printf(" The probability that X is between 0.5 and 0.6 is %6.4f\n",
        p);
}
```

Output

```
 The probability that X is less than 0.6 is 0.8364
 The probability that X is between 0.5 and 0.6 is 0.3364
```

beta_inverse_cdf

Evaluates the inverse of the beta distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_beta_inverse_cdf` (*float* `p`, *float* `pin`, *float* `qin`)

The type *double* function is `imsl_d_beta_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the beta distribution function is to be evaluated. Argument `p` must be in the open interval (0.0 ,1.0).

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

Function `imsl_f_beta_inverse_cdf` evaluates the inverse distribution function of a beta random variable with parameters `pin` and `qin`.

Description

With P = `p`, p = `pin`, and q = `qin`, function imsl_f_`beta_inverse_cdf` returns x such that

$$P = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is P.

Example

Suppose X is a beta random variable with parameters 12 and 12. (X has a symmetric distribution.) This example finds the value x such that the probability that $X \leq x$ is 0.9.

```
#include <imsl.h>

main()
{
    float           p, pin, qin, x;


    pin = 12.0;
    qin = 12.0;
    p = 0.9;
    x = imsl_f_beta_inverse_cdf(p, pin, qin);
    printf(" X is less than %6.4f with probability 0.9.\n",
        x);
}
```

Output

```
 X is less than 0.6299 with probability 0.9.
```

bivariate_normal_cdf

Evaluates the bivariate normal distribution function.

Synopsis

#include `<imsl.h>`

float `imsl_f_bivariate_normal_cdf` (*float* `x`, *float* `y`, *float* `rho`)

The type *double* function is `imsl_d_bivariate_normal_cdf`.

Required Arguments

float `x` (Input)
The *x*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `y` (Input)
The *y*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `rho` (Input)
Correlation coefficient.

Return Value

The probability that a bivariate normal random variable with correlation `rho` takes a value less than or equal to *x* and less than or equal to *y*.

Example

Suppose (*X*, *Y*) is a bivariate normal random variable with mean (0, 0) and variance-covariance matrix

$$\begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}$$

This example finds the probability that *X* is less than −2.0 and *Y* is less than 0.0.

```
#include <imsl.h>

main()
{
    float           p, rho, x, y;

    x = -2.0;
    y = 0.0;
    rho = 0.9;
    p = imsl_f_bivariate_normal_cdf(x, y, rho);
    printf(" The probability that X is less than -2.0"
               " and Y is less than 0.0 is %6.4f\n", p);

}
```

Output

```
The probability that X is less than -2.0 and Y is less than 0.0 is 0.0228
```

Chapter 10: Statistics and Random Number Generation

Routines

Usage Notes

Statistics

The functions in this section can be used to compute some common univariate summary statistics, perform a one-sample goodness-of-fit test, produce measures of correlation, perform multiple and polynomial regression analysis, and compute ranks (or a transformation of the ranks, such as normal or exponential scores). The user is referred to the individual functions for additional information.

Overview of Random Number Generation

"Random Numbers" describes functions for the generation of random numbers and of random samples and permutations. These functions are useful for applications in Monte Carlo or simulation studies. Before using any of the random number generators, the generator must be initialized by selecting a *seed* or starting value. This can be done by calling the function `imsl_random_seed_set`. If the user does not select a seed, one is generated using the system clock. A seed needs to be selected only once in a program, unless two or more separate streams of random numbers are maintained. There are other utility functions in this chapter for selecting the form of the basic generator, for restarting simulations, and for maintaining separate simulation streams.

In the following discussions, the phrases "random numbers," "random deviates," "deviates," and "variates" are used interchangeably. The phrase "pseudorandom" is sometimes used to emphasize that the numbers generated are really not "random," since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they *simulate* the realizations of independent and identically distributed random variables.

The Basic Uniform Generator

The random number generators in this chapter use a multiplicative congruential method. The form of the generator is

$$x_i = cx_{i-1} \bmod (2^{31} - 1).$$

Each x_i is then scaled into the unit interval (0,1). If the multiplier, c, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator will have a maximal period of $2^{31} - 2$. There are several other considerations, however. See Knuth (1981) for a good general discussion. The possible values for c in the IMSL generators are 16807, 397204094, and 950706376. The selection is made by the function `imsl_random_ option`. The choice of 16807 will result in the fastest execution time, but other evidence suggests that the performance of 950706376 is best among these three choices (Fishman and Moore 1982). If no selection is made explicitly, the functions use the multiplier 16807, which has been in use for some time (Lewis et al. 1969).

The generation of uniform (0,1) numbers is done by the function `imsl_f_random_uniform`. This function is *portable* in the sense that, given the same seed, it produces the same sequence in all computer/compiler environments.

Shuffled Generators

The user also can select a shuffled version of these generators using `imsl_random_option`. The shuffled generators use a scheme due to Learmonth and Lewis (1973). In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from

1 to 128. The *j*-th entry in the table is then delivered as the random number, and x_i, after being scaled into the unit interval, is inserted into the *j*-th position in the table. This scheme is similar to that of Bays and Durham (1976), and their analysis is applicable to this scheme as well.

Setting the Seed

The seed of the generator can be set in `imsl_random_seed_set` and can be retrieved by `imsl_random_seed_get`. Prior to invoking any generator in this section, the user can call `imsl_random_seed_set` to initialize the seed, which is an integer variable with a value between 1 and 2147483647. If it is not initialized by `imsl_random_seed_set`, a random seed is obtained from the system clock. Once it is initialized, the seed need not be set again.

If the user wishes to restart a simulation, `imsl_random_seed_get` can be used to obtain the final seed value of one run to be used as the starting value in a subsequent run. Also, if two simultaneous random number streams are desired in one run, `imsl_random_seed_set` and `imsl_random_seed_get` can be used before and after the invocations of the generators in each stream.

simple_statistics

Computes basic univariate statistics.

Synopsis

#include `<imsl.h>`

float `*imsl_f_simple_statistics` (*int* `n_observations`, *int* `_variables`, *float* `x[]` ,…, 0)

The type *double* procedure is `imsl_d_simple_statistics`.

Required Arguments

int `n_observations` (Input)
The number of observations.

int `n_variables` (Input)
The number of variables.

float `x[]` (Input)
Array of size `n_observations` × `n_variables` containing the data matrix.

Return Value

A pointer to a matrix containing some simple statistics for each of the columns in `x`. If `MEDIAN` and `MEDIAN_AND_SCALE` are not used as optional arguments, the size of the matrix is 14 by `n_variables`. The columns of this matrix correspond to the columns of `x` and the rows contain the following statistics:

Row	Statistic
0	the mean
1	the variance
2	the standard deviation
3	the coefficient of skewness
4	the coefficient of excess (kurtosis)
5	the minimum value
6	the maximum value
7	the range
8	the coefficient of variation (when defined) If the coefficient of variation is not defined, zero is returned.
9	the number of observations (the counts)
10	a lower confidence limit for the mean (assuming normality) The default is a 95 percent confidence interval.
11	an upper confidence limit for the mean (assuming normality)
12	a lower confidence limit for the variance (assuming normality) The default is a 95 percent confidence interval.
13	an upper confidence limit for the variance (assuming normality)

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_simple_statistics (int n_observations, int n_variables,
        float x[],
        IMSL_CONFIDENCE_MEANS, float confidence_means,
        IMSL_CONFIDENCE_VARIANCES, float confidence_variances,
        IMSL_X_COL_DIM, int x_col_dim,
        IMSL_STAT_COL_DIM, int stat_col_dim,
        IMSL_MEDIAN,
        IMSL_MEDIAN_AND_SCALE,
        IMSL_RETURN_USER, float simple_statistics[],
        0)
```

Optional Arguments

`IMSL_CONFIDENCE_MEANS`, *float* `confidence_means` (Input)

The confidence level for a two-sided interval estimate of the means (assuming normality) in percent. Argument `confidence_means` must be between 0.0 and 100.0 and is often 90.0, 95.0, or 99.0. For a one-sided confidence interval with confidence level *c*, set `confidence_means` = 100.0 – 2(100 – *c*). If

IMSL_CONFIDENCE_MEANS is not specified, a 95 percent confidence interval is computed.

IMSL_CONFIDENCE_VARIANCES, *float* confidence_variances (Input)
The confidence level for a two-sided interval estimate of the variances (assuming normality) in percent. The confidence intervals are symmetric in probability (rather than in length). For a one-sided confidence interval with confidence level *c*, set confidence_means = 100.0 – 2(100 – *c*). If IMSL_CONFIDENCE_VARIANCES is not specified, a 95 percent confidence interval is computed.

IMSL_X_COL_DIM, *int* x_col_dim (Input)
The column dimension of array x.
Default: x_col_dim = n_variables

IMSL_STAT_COL_DIM, *int* stat_col_dim (Input)
The column dimension of the returned value array, or if IMSL_RETURN_USER is specified, the column dimension of array simple_statistics.
Default: stat_col_dim = n_variables

IMSL_MEDIAN, *or*
IMSL_MEDIAN_AND_SCALE
Exactly one of these optional arguments can be specified in order to indicate the additional simple robust statistics to be computed. If IMSL_MEDIAN is specified, the medians are computed and stored in one additional row (row number 14) in the returned matrix of simple statistics. If IMSL_MEDIAN_AND_SCALE is specified, the medians, the medians of the absolute deviations from the medians, and a simple robust estimate of scale are computed, then stored in three additional rows (rows 14, 15, and 16) in the returned matrix of simple statistics.

IMSL_RETURN_USER, *float* simple_statistics[] (Output)
Store the matrix of statistics in the user-provided array simple_statistics. If neither IMSL_MEDIAN nor IMSL_MEDIAN_AND_SCALE is specified, the matrix is 14 by n_variables. If IMSL_MEDIAN is specified, the matrix is 15 by n_variables. If IMSL_MEDIAN_AND_SCALE is specified, the matrix is 17 by n_variables.

Description

For the data in each column of *x*, imsl_f_simple_statistics computes the sample mean, variance, minimum, maximum, and other basic statistics. It also computes confidence intervals for the mean and variance (under the hypothesis that the sample is from a normal population).

The definitions of some of the statistics are given below in terms of a single variable *x* of which the *i*-th datum is x_i.

Mean

$$\bar{x} = \frac{\sum x_i}{n}$$

Variance

$$s^2 = \frac{\sum \left(x_i - \bar{x}\right)^2}{n-1}$$

Skewness

$$\frac{\sum \left(x_i - \bar{x}\right)^3 / n}{[\sum \left(x_i - \bar{x}\right)^2 / n]^{3/2}}$$

Excess or Kurtosis

$$\frac{\sum \left(x_i - \bar{x}\right)^4 / n}{[\sum \left(x_i - \bar{x}\right)^2 / n]^2} - 3$$

Minimum

$$x_{min} = min\left(x_i\right)$$

Maximum

$$x_{max} = max\left(x_i\right)$$

Range

$$x_{max} - x_{min}$$

Coefficient of Variation

$$s / \bar{x} \text{ for } \bar{x} \neq 0$$

Median

$$\text{median } \left\{x_i\right\} = \begin{cases} \text{middle } x_i \text{ after sorting if } n \text{ is odd} \\ \text{average of middle two } x_i\text{'s if } n \text{ is even} \end{cases}$$

Median Absolute Deviation

$$\text{MAD} = \text{median}\left\{\left|x_i - \text{median}\left\{x_j\right\}\right|\right\}$$

Simple Robust Estimate of Scale

$$\text{MAD} / \Phi^{-1}(3/4)$$

where $\Phi^{-1}(3/4) \approx 0.6745$ is the inverse of the standard normal distribution function evaluated at 3/4. This standardizes MAD in order to make the scale estimate consistent at the normal distribution for estimating the standard deviation (Huber 1981, pp. 107–108).

Example

This example uses data from Draper and Smith (1981). There are five variables and 13 observations.

```
#include <imsl.h>

#define N_VARIABLES            5
#define N_OBSERVATIONS         13

main()
{
    float       *simple_statistics;
    float       x[] = {7., 26.,  6., 60.,  78.5,
                       1., 29., 15., 52.,  74.3,
                      11., 56.,  8., 20., 104.3,
                      11., 31.,  8., 47.,  87.6,
                       7., 52.,  6., 33.,  95.9,
                      11., 55.,  9., 22., 109.2,
                       3., 71., 17.,  6., 102.7,
                       1., 31., 22., 44.,  72.5,
                       2., 54., 18., 22.,  93.1,
                      21., 47.,  4., 26., 115.9,
                       1., 40., 23., 34.,  83.8,
                      11., 66.,  9., 12., 113.3,
                      10., 68.,  8., 12., 109.4};
    char        *row_labels[] = {"means", "variances", "std. dev",
                                 "skewness", "kurtosis", "minima",
                                 "maxima", "ranges", "C.V.", "counts",
                                 "lower mean", "upper mean",
                                 "lower var", "upper var"};

    simple_statistics = imsl_f_simple_statistics(N_OBSERVATIONS,
                                                 N_VARIABLES, x, 0);

    imsl_f_write_matrix("* * * Statistics * * *\n", 14, N_VARIABLES,
                        simple_statistics,
                        IMSL_ROW_LABELS,  row_labels,
                        IMSL_WRITE_FORMAT, "%7.3f",
                        0);
}
```

Output

```
    * * * Statistics * * *

                1        2        3        4        5
means         7.462   48.154   11.769   30.000   95.423
variances    34.603  242.141   41.026  280.167  226.314
std. dev      5.882   15.561    6.405   16.738   15.044
skewness      0.688   -0.047    0.611    0.330   -0.195
kurtosis      0.075   -1.323   -1.079   -1.014   -1.342
minima        1.000   26.000    4.000    6.000   72.500
maxima       21.000   71.000   23.000   60.000  115.900
ranges       20.000   45.000   19.000   54.000   43.400
C.V.          0.788    0.323    0.544    0.558    0.158
counts       13.000   13.000   13.000   13.000   13.000
lower mean    3.907   38.750    7.899   19.885   86.332
upper mean   11.016   57.557   15.640   40.115  104.514
lower var    17.793  124.512   21.096  144.065  116.373
upper var    94.289  659.817  111.792  763.434  616.688
```

table_oneway

Tallies observations into a one-way frequency table.

Synopsis

#include `<imsl.h>`

float `*imsl_f_table_oneway` (*int* `n_observations`, *float* `x[]`,
int `_intervals`, …, 0)

The type *double* function is `imsl_d_table_oneway`.

Required Arguments

int `n_observations` (Input)

Number of observations.

float `x[]` (Input)

Array of length `n_observations` containing the observations.

int `n_intervals` (Input)

Number of intervals (bins).

Return Value

Pointer to an array of length `n_intervals` containing the counts.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_table_oneway` (*int* `n_observations`, *float* `x[]`,
int `n_intervals`,
`IMSL_DATA_BOUNDS`, *float* `*minimum`, *float* `*maximum`,

```
        IMSL_KNOWN_BOUNDS, float lower_bound, float upper_bound,
        IMSL_CUTPOINTS, float cutpoints[],
        IMSL_CLASS_MARKS, float class_marks[],
        IMSL_RETURN_USER, float table_oneway[],
        0)
```

Optional Arguments

`IMSL_DATA_BOUNDS`, *float* `*minimum`, *float* `*maximum` (Output)
or
`IMSL_KNOWN_BOUNDS`, *float* `lower_bound`, *float* `upper_bound` (Input)
or
`IMSL_CUTPOINTS`, *float* `cutpoints[]` (Input)
or
`IMSL_CLASS_MARKS`, *float* `class_marks[]` (Input)

None, or exactly one, of these four optional arguments can be specified in order to define the intervals or bins for the one-way table. If none is specified, or if `IMSL_DATA_BOUNDS` is specified, `n_intervals`, intervals of equal length, are used with the initial interval starting with the minimum value in `x` and the last interval ending with the maximum value in `x`. The initial interval is closed on the left and right. The remaining intervals are open on the left and closed on the right. When `IMSL_DATA_BOUNDS` is explicitly specified, the minimum and maximum values in `x` are output in `minimum` and `maximum`. With this option, each interval is of (`maximum`–`minimum`)/ `n_intervals` length. If `IMSL_KNOWN_BOUNDS` is specified, two semi-infinite intervals are used as the initial and last interval. The initial interval is closed on the right and includes `lower_bound` as its right endpoint. The last interval is open on the left and includes all values greater than `upper_ bound`. The remaining `n_intervals` – 2 intervals are each of length

$$\frac{\text{upper\_ bound - lower\_ bound}}{\text{n\_ intervals} - 2}$$

and are open on the left and closed on the right. Argument `n_intervals` must be greater than or equal to three for this option. If `IMSL_CLASS_MARKS` is specified, equally spaced class marks in ascending order must be provided in the array `class_marks` of length `n_intervals`. The class marks are the midpoints of each of the `n_intervals`, and each interval is taken to have length `class_marks`[1] – `class_marks`[0]. The argument `n_intervals` must be greater than or equal to two for this option. If `IMSL_ CUTPOINTS` is specified, cutpoints (boundaries) must be provided in the array `cutpoints` of length `n_intervals` – 1. This option allows unequal interval lengths. The initial interval is closed on the right and includes the initial cutpoint as its right endpoint. The last interval is open on the left and includes all values greater than the last cutpoint. The remaining `n_intervals` – 2 intervals are open on the left and closed on the right. The argument `n_interval` must be greater than or equal to three for this option.

IMSL_RETURN_USER, *float* table[] (Output)

Counts are stored in the user-supplied array table of length n_intervals.

Examples

Example 1

The data for this example is from Hinkley (1977) and Velleman and Hoaglin (1981). They are the measurements (in inches) of precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
#include <imsl.h>
main()
{
    int     n_intervals=10;
    int     n_observations=30;
    float   *table;
    float   x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
                   2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
                   0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
                  1.89, 0.90, 2.05};
    table = imsl_f_table_oneway (n_observations, x, n_intervals, 0);
    imsl_f_write_matrix("counts", 1, n_intervals, table, 0);
  }
```

Output

```
                              counts
         1           2           3           4           5           6
         4           8           5           5           3           1

         7           8           9          10
         3           0           0           1
```

Example 2

This example selects IMSL_KNOWN_BOUNDS and sets lower_bound = 0.5 and upper_bound = 4.5 so that the eight interior intervals each have width (4.5 – 0.5)/(10 – 2) = 0.5. The 10 intervals are (–∞, 0.5], (0.5, 1.0], …, (4.0, .5], and (4.5, ∞].

```
#include <imsl.h>
main()
{
    int     n_observations=30;
    int     n_intervals=10;
    float   *table;
    float   lower_bound=0.5, upper_bound=4.5;
    float   x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
                   2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
                   0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
                   1.89, 0.90, 2.05};
    table = imsl_f_table_oneway (n_observations, x, n_intervals,
                                 IMSL_KNOWN_BOUNDS, lower_bound,
                                 upper_bound, 0);
    imsl_f_write_matrix("counts", 1, n_intervals, table, 0);
 }
```

Output

```
                          counts
        1          2          3          4          5          6
        2          7          6          6          4          2

        7          8          9         10
        2          0          0          1
```

Example 3

This example inputs 10 class marks 0.25, 0.75, 1.25, …, 4.75. This defines the class intervals (0.0, 0.5], (0.5, 1.0], …, (4.0, 4.5], (4.5, 5.0]. Note that unlike the previous example, the initial and last intervals are the same length as the remaining intervals.

```
#include <imsl.h>
main()
{
    int        n_intervals=10;
    int        n_observations=30;
    double     *table;
    double     x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                      3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                      1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                      4.75, 2.48, 0.96, 1.89, 0.90, 2.05};
    double     class_marks[] = {0.25, 0.75, 1.25, 1.75, 2.25, 2.75,
                                3.25, 3.75, 4.25, 4.75};
    table = imsl_d_table_oneway (n_observations, x, n_intervals,
                                 IMSL_CLASS_MARKS, class_marks,
                                 0);
    imsl_d_write_matrix("counts", 1, n_intervals, table, 0);
}
```

Output

```
                          counts
        1          2          3          4          5          6
        2          7          6          6          4          2

        7          8          9         10
        2          0          0          1
```

Example 4

This example inputs nine cutpoints 0.5, 1.0, 1.5, 2.0, …, 4.5 to define the same 10 intervals as in Example 3. Here again, the initial and last intervals are semi-infinite intervals.

```
#include <imsl.h>
main()
{
    int        n_intervals=10;
    int        n_observations=30;
    double     *table;
    double     x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                      3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                      1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                      4.75, 2.48, 0.96, 1.89, 0.90, 2.05};
```

```
    double     cutpoints[] = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0,
                              4.5};
    table = imsl_d_table_oneway (n_observations, x, n_intervals,
                                 IMSL_CUTPOINTS, cutpoints,
                                 0);
    imsl_d_write_matrix("counts", 1, n_intervals, table, 0);
}
```

Output

```
                counts
1          2          3          4          5          6
2          7          6          6          4          2

7          8          9         10
2          0          0          1
```

chi_squared_test

Performs a chi-squared goodness-of-fit test.

Synopsis

#include `<imsl.h>`

float `imsl_f_chi_squared_test` (*float* `user_proc_cdf()`,
int `n_observations`, *int* `n_categories`, *float* `x[]`, …, 0)

The type *double* function is `imsl_d_chi_squared_test`.

Required Arguments

float `user_proc_cdf` (*float* `y`) (Input)
User-supplied function that returns the hypothesized, cumulative distribution function at the point *y*.

int `n_observations` (Input)
The number of data elements input in `x`.

int `n_categories` (Input)
The number of cells into which the observations are to be tallied.

float `x[]` (Input)
Array with `n_observations` components containing the vector of data elements for this test.

Return Value

The *p*-value for the goodness-of-fit chi-squared statistic.

Synopsis with Optional Arguments

#include `<imsl.h>`

float imsl_f_chi_squared_test (*float* *user_proc_cdf(), *int*
n_observations, *int* n_categories, *float* x[],
IMSL_N_PARAMETERS_ESTIMATED, *int* n_parameters,
IMSL_CUTPOINTS, *float* **p_cutpoints,
IMSL_CUTPOINTS_USER, *float* cutpoints[],
IMSL_CUTPOINTS_EQUAL,
IMSL_CHI_SQUARED, *float* *chi_squared,
IMSL_DEGREES_OF_FREEDOM, *float* *df,
IMSL_FREQUENCIES, *float* frequencies[],
IMSL_BOUNDS, *float* lower_bound, *float* upper_bound,
IMSL_CELL_COUNTS, *float* **p_cell_counts,
IMSL_CELL_COUNTS_USER, *float* cell_counts[],
IMSL_CELL_EXPECTED, *float* **p_cell_expected,
IMSL_CELL_EXPECTED_USER, *float* cell_expected[],
IMSL_CELL_CHI_SQUARED, *float* **p_cell_chi_squared,
IMSL_CELL_CHI_SQUARED_USER, *float* cell_chi_squared[],
0)

Optional Arguments

IMSL_N_PARAMETERS_ESTIMATED, *int* n_parameters (Input)
The number of parameters estimated in computing the cumulative distribution function.

IMSL_CUTPOINTS, *float* **p_cutpoints (Output)
The address of a pointer to the cutpoints array. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_cutpoints is declared; &p_cutpoints is used as an argument to this function; and free(p_cutpoints) is used to free this array.

IMSL_CUTPOINTS_USER, *float* cutpoints[] (Input or Output)
Array with n_categories – 1 components containing the vector of cutpoints defining the cell intervals. The intervals defined by the cutpoints are such that the lower endpoint is not included, and the upper endpoint is included in any interval. If IMSL_CUTPOINTS_EQUAL is specified, equal probability cutpoints are computed and returned in cutpoints.

IMSL_CUTPOINTS_EQUAL
If IMSL_CUTPOINTS_USER is specified, then equal probability cutpoints can still be used if, in addition, the IMSL_CUTPOINTS_EQUAL option is specified. If IMSL_CUTPOINTS_USER is not specified, equal probability cutpoints are used by default.

IMSL_CHI_SQUARED, *float* *chi_squared (Output)
If specified, the chi-squared test statistic is returned in *chi_squared.

IMSL_DEGREES_OF_FREEDOM, *float* *df (Output)
If specified, the degrees of freedom for the chi-squared goodness-of-fit test is returned in *df.

IMSL_FREQUENCIES, *float* frequencies[] (Input)
Array with n_observations components containing the vector frequencies for the observations stored in x.

IMSL_BOUNDS, *float* lower_bound, *float* upper_bound (Input)
If IMSL_BOUNDS is specified, then lower_bound is the lower bound of the range of the distribution, and upper_bound is the upper bound of this range. If lower_bound = upper_bound, a range on the whole real line is used (the default). If the lower and upper endpoints are different, points outside the range of these bounds are ignored. Distributions conditional on a range can be specified when IMSL_BOUNDS is used. By convention, lower_bound is excluded from the first interval, but upper_bound is included in the last interval.

IMSL_CELL_COUNTS, *float* **p_cell_counts (Output)
The address of a pointer to an array containing the cell counts. The cell counts are the observed frequencies in each of the n_categories cells. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_cell_counts is declared; &p_cell_counts is used as an argument to this function; and free(p_cell_counts) is used to free this array.

IMSL_CELL_COUNTS_USER, *float* cell_counts[] (Output)
If specified, the n_categories cell counts are returned in the array cell_counts provided by the user.

IMSL_CELL_EXPECTED, *float* **p_cell_expected (Output)
The address of a pointer to the cell expected values. The expected value of a cell is the expected count in the cell given that the hypothesized distribution is correct. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_cell_expected is declared; &p_cell_expected is used as an argument to this function; and free(p_cell_expected) is used to free this array.

IMSL_CELL_EXPECTED_USER, *float* cell_expected[] (Output)
If specified, the n_categories cell expected values are returned in the array cell_expected provided by the user.

IMSL_CELL_CHI_SQUARED, *float* **p_cell_chi_squared (Output)
The address of a pointer to an array of length n_categories containing the cell contributions to chi-squared. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_cell_chi_squared is declared; &p_cell_chi_squared is used as an argument to this function; and free(p_cell_chi_squared) is used to free this array.

IMSL_CELL_CHI_SQUARED_USER, *float* cell_chi_squared[] (Output)
If specified, the cell contributions to chi-squared are returned in the array cell_chi_squared provided by the user.

Description

The function `imsl_f_chi_squared_test` performs a chi-squared goodness-of-fit test that a random sample of observations is distributed according to a specified theoretical cumulative distribution. The theoretical distribution, which may be continuous, discrete, or a mixture of discrete and continuous distributions, is specified via the user-defined function `user_proc_cdf`. Because the user is allowed to give a range for the observations, a test conditional upon the specified range is performed.

Argument `n_categories` gives the number of intervals into which the observations are to be divided. By default, equiprobable intervals are computed by `imsl_f_chi_squared_test`, but intervals that are not equiprobable can be specified (through the use of optional argument `IMSL_CUTPOINTS`).

Regardless of the method used to obtain the cutpoints, the intervals are such that the lower endpoint is not included in the interval, while the upper endpoint is always included. If the cumulative distribution function has discrete elements, then user-provided cutpoints should always be used since `imsl_f_chi_squared_test` cannot determine the discrete elements in discrete distributions.

By default, the lower and upper endpoints of the first and last intervals are $-\infty$ and $+\infty$, respectively. If `IMSL_BOUNDS` is specified, the endpoints are defined by the user via the two arguments `lower_bound` and `upper_bound`.

A tally of counts is maintained for the observations in x as follows. If the cutpoints are specified by the user, the tally is made in the interval to which x_i belongs using the endpoints specified by the user. If the cutpoints are determined by `imsl_f_chi_squared_test`, then the cumulative probability at x_i, $F(x_i)$, is computed via the function `user_proc_cdf`. The tally for x_i is made in interval number

$$\lfloor mF(x_i)+1 \rfloor \text{ where } m = \text{ n\_ categories and } \lfloor \cdot \rfloor$$

is the function that takes the greatest integer that is no larger than the argument of the function. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred to reduce the total computing time.

If the expected count in any cell is less than 1, then a rule of thumb is that the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.

Programming Notes

The user must supply a function `user_proc_cdf` with calling sequence `user_proc_cdf(y)`, that returns the value of the cumulative distribution function at any point `y` in the (optionally) specified range. Many of the cumulative distribution functions in Chapter 9, “Special Functions,” can be used for `user_proc_cdf`, either directly, if the calling sequence is correct, or indirectly, if, for example, the sample means and standard deviations are to be used in computing the theoretical cumulative distribution function.

Examples

Example 1

This example illustrates the use of `imsl_f_chi_squared_test` on a randomly generated sample from the normal distribution. One-thousand randomly generated observations are tallied into 10 equiprobable intervals. The null hypothesis that the sample is from a normal distribution is specified by use of the `imsl_f_normal_cdf` as the hypothesized distribution function. In this example, the null hypothesis is not rejected.

```
#include <imsl.h>

#define SEED                    123457
#define N_CATEGORIES                10
#define N_OBSERVATIONS            1000

main()
{
    float       *x, p_value;

    imsl_random_seed_set(SEED);
                                /* Generate Normal deviates */
    x = imsl_f_random_normal (N_OBSERVATIONS, 0);
                                /* Perform chi squared test */
    p_value = imsl_f_chi_squared_test (imsl_f_normal_cdf, N_OBSERVATIONS,
                                       N_CATEGORIES, x, 0);
                                /* Print results */
    printf ("p value %7.4f\n", p_value);
}
```

Output

```
p value  0.1546
```

Example 2

In this example, some optional arguments are used for the data in the initial example.

```
#include <imsl.h>

#define SEED                    123457
#define N_CATEGORIES                10
#define N_OBSERVATIONS            1000

main()
{
    float     *cell_counts, *cutpoints, *cell_chi_squared;
    float     chi_squared_statistics[3], *x;
    char      *stat_row_labels[] = {"chi-squared", "degrees of freedom",
                                    "p-value"};
    imsl_random_seed_set(SEED);
                                /* Generate Normal deviates */
    x = imsl_f_random_normal (N_OBSERVATIONS, 0);
                                /* Perform chi squared test */
    chi_squared_statistics[2] =
        imsl_f_chi_squared_test (imsl_f_normal_cdf,
                N_OBSERVATIONS, N_CATEGORIES, x,
```

```
                IMSL_CUTPOINTS, &cutpoints,
                IMSL_CELL_COUNTS, &cell_counts,
                IMSL_CELL_CHI_SQUARED, &cell_chi_squared,
                IMSL_CHI_SQUARED, &chi_squared_statistics[0],
                IMSL_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
                0);
                            /* Print results */
    imsl_f_write_matrix ("\nChi Squared Statistics\n", 3, 1,
                         chi_squared_statistics,
                         IMSL_ROW_LABELS, stat_row_labels,
                         0);
    imsl_f_write_matrix ("Cut Points", 1, N_CATEGORIES-1, cutpoints, 0);
    imsl_f_write_matrix ("Cell Counts", 1, N_CATEGORIES, cell_counts,
                  0);
    imsl_f_write_matrix ("Cell Contributions to Chi-Squared", 1,
                  N_CATEGORIES, cell_chi_squared,
                  0);
}
```

Output

```
    Chi Squared Statistics

chi-squared             13.18
degrees of freedom       9.00
p-value                  0.15

                            Cut Points
         1           2           3           4           5           6
    -1.282      -0.842      -0.524      -0.253      -0.000       0.253

         7           8           9
     0.524       0.842       1.282

                            Cell Counts
         1           2           3           4           5           6
       106         109          89          92          83          87

         7           8           9          10
       110         104         121          99

                  Cell Contributions to Chi-Squared
         1           2           3           4           5           6
      0.36        0.81        1.21        0.64        2.89        1.69

         7           8           9          10
      1.00        0.16        4.41        0.01
```

Example 3

In this example, a discrete Poisson random sample of size 1000 with parameter $\theta = 5.0$ is generated via function **imsl_f_random_poisson** (page 570). In the call to `imsl_f_chi_squared_test`, function **imsl_f_poisson_cdf** (see page 510) is used as function `user_proc_cdf`.

```
#include <imsl.h>

#define SEED               123457
```

```
#define N_CATEGORIES            10
#define N_PARAMETERS_ESTIMATED   0
#define N_NUMBERS             1000
#define THETA                  5.0

float           user_proc_cdf(float);

main()
{
    int        i, *poisson;
    float      cell_statistics[3][N_CATEGORIES];
    float      chi_squared_statistics[3], x[N_NUMBERS];
    float      cutpoints[]       = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5,
                                    7.5, 8.5, 9.5};
    char      *cell_row_labels[] = {"count", "expected count",
                                    "cell chi-squared"};
    char      *cell_col_labels[] = {"Poisson value", "0", "1", "2",
                                    "3", "4", "5", "6", "7", "8", "9"};
    char      *stat_row_labels[] = {"chi-squared", "degrees of freedom",
                                    "p-value"};

    imsl_random_seed_set(SEED);
                                /* Generate the data */
    poisson = imsl_random_poisson(N_NUMBERS, THETA, 0);
                               /* Copy data to a floating point vector*/
    for (i = 0; i < N_NUMBERS; i++)
         x[i] = poisson[i];

    chi_squared_statistics[2] =
      imsl_f_chi_squared_test(user_proc_cdf, N_NUMBERS, N_CATEGORIES, x,
                IMSL_CUTPOINTS_USER,        cutpoints,
                IMSL_CELL_COUNTS_USER,      &cell_statistics[0][0],
                IMSL_CELL_EXPECTED_USER,    &cell_statistics[1][0],
                IMSL_CELL_CHI_SQUARED_USER, &cell_statistics[2][0],
                IMSL_CHI_SQUARED,           &chi_squared_statistics[0],
                IMSL_DEGREES_OF_FREEDOM,    &chi_squared_statistics[1],
                0);
                                /* Print results */
    imsl_f_write_matrix("\nChi-squared statistics\n", 3, 1,
                        &chi_squared_statistics[0],
                        IMSL_ROW_LABELS,     stat_row_labels,
                        0);
    imsl_f_write_matrix("\nCell Statistics\n", 3, N_CATEGORIES,
                        &cell_statistics[0][0],
                        IMSL_ROW_LABELS,     cell_row_labels,
                        IMSL_COL_LABELS,     cell_col_labels,
                        0);
}


float user_proc_cdf(float k)
{
    float           cdf_v;

    cdf_v = imsl_f_poisson_cdf ((int) k, THETA);
    return cdf_v;
}
```

Output

```
    Chi-squared statistics

chi-squared              10.48
degrees of freedom        9.00
p-value                   0.31


                            Cell Statistics

Poisson value         0          1          2          3          4
count              41.0       94.0      138.0      158.0      150.0
expected count     40.4       84.2      140.4      175.5      175.5
cell chi-squared    0.0        1.1        0.0        1.7        3.7

Poisson value         5          6          7          8          9
count             159.0      116.0       75.0       37.0       32.0
expected count    146.2      104.4       65.3       36.3       31.8
cell chi-squared    1.1        1.3        1.4        0.0        0.0
```

Warning Errors

IMSL_EXPECTED_VAL_LESS_THAN_1	An expected value is less than 1.
IMSL_EXPECTED_VAL_LESS_THAN_5	An expected value is less than 5.

Fatal Errors

IMSL_ALL_OBSERVATIONS_MISSING	All observations contain missing values.
IMSL_INCORRECT_CDF_1	The function `user_proc_cdf` is not a cumulative distribution function. The value at the lower bound must be nonnegative, and the value at the upper bound must not be greater than one.
IMSL_INCORRECT_CDF_2	The function `user_proc_cdf` is not a cumulative distribution function. The probability of the range of the distribution is not positive.
IMSL_INCORRECT_CDF_3	The function `user_proc_cdf` is not a cumulative distribution function. Its evaluation at an element in `x` is inconsistent with either the evaluation at the lower or upper bound.
IMSL_INCORRECT_CDF_4	The function `user_proc_cdf` is not a cumulative distribution function. Its evaluation at a cutpoint is inconsistent with either the evaluation at the lower or upper bound.

IMSL_INCORRECT_CDF_5	An error has occurred when inverting the cumulative distribution function. This function must be continuous and defined over the whole real line.

covariances

Computes the sample variance-covariance or correlation matrix.

Synopsis

#include <imsl.h>

float *imsl_f_covariances (*int* n_observations, *int* n_variables, *float* x[], …, 0)

The type *double* function is imsl_d_covariances.

Required Arguments

int n_observations (Input)
The number of observations.

int n_variables (Input)
The number of variables.

float x[] (Input)
Array of size n_observations × n_variables containing the matrix of data.

Return Value

If no optional arguments are used, imsl_f_covariances returns a pointer to an n_variables × n_variables matrix containing the sample variance-covariance matrix of the observations. The rows and columns of this matrix correspond to the columns of x.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_covariances (*int* n_observations, *int* n_variables, *float* x[],
IMSL_X_COL_DIM, *int* x_col_dim,
IMSL_VARIANCE_COVARIANCE_MATRIX,
IMSL_CORRECTED_SSCP_MATRIX,
IMSL_CORRELATION_MATRIX,
IMSL_STDEV_CORRELATION_MATRIX,
IMSL_MEANS, *float* **p_means,
IMSL_MEANS_USER, *float* means[],
IMSL_COVARIANCE_COL_DIM, *int* covariance_col_dim,
IMSL_RETURN_USER, *float* covariance[],
0)

Optional Arguments

`IMSL_X_COL_DIM`, *int* `x_col_dim` (Input)
The column dimension of array `x`.
Default: `x_col_dim` = `n_variables`

`IMSL_VARIANCE_COVARIANCE_MATRIX`, *or*
`IMSL_CORRECTED_SSCP_MATRIX`, *or*
`IMSL_CORRELATION_MATRIX`, *or*
`IMSL_STDEV_CORRELATION_MATRIX`
Exactly one of these options can be used to specify the type of matrix to be computed.

Keyword	Type of Matrix
`IMSL_VARIANCE_COVARIANCE_MATRIX`	variance-covariance matrix (default)
`IMSL_CORRECTED_SSCP_MATRIX`	corrected sums of squares and crossproducts matrix
`IMSL_CORRELATION_MATRIX`	correlation matrix
`IMSL_STDEV_CORRELATION_MATRIX`	correlation matrix except for the diagonal elements which are the standard deviations

`IMSL_MEANS`, *float* `**p_means` (Output)
The address of a pointer to the array containing the means of the variables in `x`. The components of the array correspond to the columns of `x`. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*p_means` is declared; `&p_means` is used as an argument to this function; and `free`(`p_means`) is used to free this array.

`IMSL_MEANS_USER`, *float* `means[]` (Output)
Calculate the `n_variables` means and store them in the memory provided by the user. The elements of `means` correspond to the columns of `x`.

`IMSL_COVARIANCE_COL_DIM`, *int* `covariance_col_dim` (Input)
The column dimension of array `covariance`, if `IMSL_RETURN_USER` is specified, or the column dimension of the return value otherwise.
Default: `covariance_col_dim` = `n_variables`

`IMSL_RETURN_USER`, *float* `covariance[]` (Output)
If specified, the output is stored in the array `covariance` of size `n_variables` × `n_variables` provided by the user.

Description

The function `imsl_f_covariances` computes estimates of correlations, covariances, or sums of squares and crossproducts for a data matrix `x`. The means, (corrected) sums

of squares, and (corrected) sums of crossproducts are computed using the method of provisional means. Let

$$\bar{x}_{ki}$$

denote the mean based on i observations for the k-th variable, and let c_{jki} denote the sum of crossproducts (or sum of squares if $j = k$) based on i observations. Then, the method of provisional means finds new means and sums of crossproducts as follows:

The means and crossproducts are initialized as:

$$\begin{aligned} \bar{x}_{k0} &= 0.0 \qquad k = 1, \ldots, p \\ c_{jk0} &= 0.0 \qquad j, k = 1, \ldots, p \end{aligned}$$

where p denotes the number of variables. Letting $x_{k,i+1}$ denote the k-th variable on observation $i + 1$, each new observation leads to the following updates for

$$\bar{x}_{ki}$$

and c_{jki} using update constant r_{i+1}:

$$\begin{aligned} r_{i+1} &= \frac{1}{i+1} \\ \bar{x}_{k,i+1} &= \bar{x}_{ki} + \left(x_{k,i+1} - \bar{x}_{ki}\right) r_{i+1} \\ c_{jk,i+1} &= c_{jki} + \left(x_{j,i+1} - \bar{x}_{ji}\right)\left(x_{k,i+1} - \bar{x}_{ki}\right)\left(1 - r_{i+1}\right) \end{aligned}$$

Usage Notes

The function `imsl_f_covariances` uses the following definition of a sample mean:

$$\bar{x}_k = \frac{\sum_{i=1}^{n} x_{ki}}{n}$$

where n is the number of observations. The following formula defines the sample covariance, $s_{j\,k}$, between variables j and k:

$$s_{jk} = \frac{\sum_{i=1}^{n} \left(x_{ji} - \bar{x}_j\right)\left(x_{ki} - \bar{x}_k\right)}{n-1}$$

The sample correlation between variables j and k, r_{jk}, is defined as follows:

$$r_{jk} = \frac{s_{jk}}{\sqrt{s_{jj} s_{kk}}}$$

Examples

Example 1

The first example illustrates the use of `imsl_f_covariances` for the first 50 observations in the Fisher iris data (Fisher 1936). Note in this example that the first variable is constant over the first 50 observations.

```
#include <imsl.h>

#define N_VARIABLES      5
#define N_OBSERVATIONS  50


main()
{
    float       *covariances, *means;
    float       x[] = {1.0, 5.1, 3.5, 1.4, .2,  1.0, 4.9, 3.0, 1.4, .2,
                       1.0, 4.7, 3.2, 1.3, .2,  1.0, 4.6, 3.1, 1.5, .2,
                       1.0, 5.0, 3.6, 1.4, .2,  1.0, 5.4, 3.9, 1.7, .4,
                       1.0, 4.6, 3.4, 1.4, .3,  1.0, 5.0, 3.4, 1.5, .2,
                       1.0, 4.4, 2.9, 1.4, .2,  1.0, 4.9, 3.1, 1.5, .1,
                       1.0, 5.4, 3.7, 1.5, .2,  1.0, 4.8, 3.4, 1.6, .2,
                       1.0, 4.8, 3.0, 1.4, .1,  1.0, 4.3, 3.0, 1.1, .1,
                       1.0, 5.8, 4.0, 1.2, .2,  1.0, 5.7, 4.4, 1.5, .4,
                       1.0, 5.4, 3.9, 1.3, .4,  1.0, 5.1, 3.5, 1.4, .3,
                       1.0, 5.7, 3.8, 1.7, .3,  1.0, 5.1, 3.8, 1.5, .3,
                       1.0, 5.4, 3.4, 1.7, .2,  1.0, 5.1, 3.7, 1.5, .4,
                       1.0, 4.6, 3.6, 1.0, .2,  1.0, 5.1, 3.3, 1.7, .5,
                       1.0, 4.8, 3.4, 1.9, .2,  1.0, 5.0, 3.0, 1.6, .2,
                       1.0, 5.0, 3.4, 1.6, .4,  1.0, 5.2, 3.5, 1.5, .2,
                       1.0, 5.2, 3.4, 1.4, .2,  1.0, 4.7, 3.2, 1.6, .2,
                       1.0, 4.8, 3.1, 1.6, .2,  1.0, 5.4, 3.4, 1.5, .4,
                       1.0, 5.2, 4.1, 1.5, .1,  1.0, 5.5, 4.2, 1.4, .2,
                       1.0, 4.9, 3.1, 1.5, .2,  1.0, 5.0, 3.2, 1.2, .2,
                       1.0, 5.5, 3.5, 1.3, .2,  1.0, 4.9, 3.6, 1.4, .1,
                       1.0, 4.4, 3.0, 1.3, .2,  1.0, 5.1, 3.4, 1.5, .2,
                       1.0, 5.0, 3.5, 1.3, .3,  1.0, 4.5, 2.3, 1.3, .3,
                       1.0, 4.4, 3.2, 1.3, .2,  1.0, 5.0, 3.5, 1.6, .6,
                       1.0, 5.1, 3.8, 1.9, .4,  1.0, 4.8, 3.0, 1.4, .3,
                       1.0, 5.1, 3.8, 1.6, .2,  1.0, 4.6, 3.2, 1.4, .2,
                       1.0, 5.3, 3.7, 1.5, .2,  1.0, 5.0, 3.3, 1.4, .2};

    covariances = imsl_f_covariances (N_OBSERVATIONS, N_VARIABLES, x, 0);
    imsl_f_write_matrix ("The default case: variances/covariances",
                         N_VARIABLES, N_VARIABLES, covariances,
                         IMSL_PRINT_UPPER,
                         0);
}
```

Output

```
            The default case: variances/covariances
             1          2          3          4          5
1      0.0000     0.0000     0.0000     0.0000     0.0000
2                 0.1242     0.0992     0.0164     0.0103
3                            0.1437     0.0117     0.0093
4                                       0.0302     0.0061
5                                                  0.0111
```

Example 2

This example illustrates the use of some optional arguments in `imsl_f_covariances`. Once again, the first 50 observations in the Fisher iris data are used.

```
#include <imsl.h>

#define N_VARIABLES      5
#define N_OBSERVATIONS  50

main()
{
    char        *title;
    float       *means, *correlations;
    float       x[] = {1.0, 5.1, 3.5, 1.4, .2,  1.0, 4.9, 3.0, 1.4, .2,
                       1.0, 4.7, 3.2, 1.3, .2,  1.0, 4.6, 3.1, 1.5, .2,
                       1.0, 5.0, 3.6, 1.4, .2,  1.0, 5.4, 3.9, 1.7, .4,
                       1.0, 4.6, 3.4, 1.4, .3,  1.0, 5.0, 3.4, 1.5, .2,
                       1.0, 4.4, 2.9, 1.4, .2,  1.0, 4.9, 3.1, 1.5, .1,
                       1.0, 5.4, 3.7, 1.5, .2,  1.0, 4.8, 3.4, 1.6, .2,
                       1.0, 4.8, 3.0, 1.4, .1,  1.0, 4.3, 3.0, 1.1, .1,
                       1.0, 5.8, 4.0, 1.2, .2,  1.0, 5.7, 4.4, 1.5, .4,
                       1.0, 5.4, 3.9, 1.3, .4,  1.0, 5.1, 3.5, 1.4, .3,
                       1.0, 5.7, 3.8, 1.7, .3,  1.0, 5.1, 3.8, 1.5, .3,
                       1.0, 5.4, 3.4, 1.7, .2,  1.0, 5.1, 3.7, 1.5, .4,
                       1.0, 4.6, 3.6, 1.0, .2,  1.0, 5.1, 3.3, 1.7, .5,
                       1.0, 4.8, 3.4, 1.9, .2,  1.0, 5.0, 3.0, 1.6, .2,
                       1.0, 5.0, 3.4, 1.6, .4,  1.0, 5.2, 3.5, 1.5, .2,
                       1.0, 5.2, 3.4, 1.4, .2,  1.0, 4.7, 3.2, 1.6, .2,
                       1.0, 4.8, 3.1, 1.6, .2,  1.0, 5.4, 3.4, 1.5, .4,
                       1.0, 5.2, 4.1, 1.5, .1,  1.0, 5.5, 4.2, 1.4, .2,
                       1.0, 4.9, 3.1, 1.5, .2,  1.0, 5.0, 3.2, 1.2, .2,
                       1.0, 5.5, 3.5, 1.3, .2,  1.0, 4.9, 3.6, 1.4, .1,
                       1.0, 4.4, 3.0, 1.3, .2,  1.0, 5.1, 3.4, 1.5, .2,
                       1.0, 5.0, 3.5, 1.3, .3,  1.0, 4.5, 2.3, 1.3, .3,
                       1.0, 4.4, 3.2, 1.3, .2,  1.0, 5.0, 3.5, 1.6, .6,
                       1.0, 5.1, 3.8, 1.9, .4,  1.0, 4.8, 3.0, 1.4, .3,
                       1.0, 5.1, 3.8, 1.6, .2,  1.0, 4.6, 3.2, 1.4, .2,
                       1.0, 5.3, 3.7, 1.5, .2,  1.0, 5.0, 3.3, 1.4, .2};

    correlations = imsl_f_covariances (N_OBSERVATIONS,
                      N_VARIABLES-1, x+1,
                      IMSL_STDEV_CORRELATION_MATRIX,
                      IMSL_X_COL_DIM, N_VARIABLES,
                      IMSL_MEANS, &means,
                      0);
    imsl_f_write_matrix ("Means\n", 1, N_VARIABLES-1, means, 0);
    title = "Correlations with Standard Deviations on the Diagonal\n";
    imsl_f_write_matrix (title, N_VARIABLES-1, N_VARIABLES-1,
                         correlations, IMSL_PRINT_UPPER,
                         0);
}
```

Output

```
           Means

    1          2          3          4
5.006      3.428      1.462      0.246
```

```
Correlations with Standard Deviations on the Diagonal

            1           2           3           4
1      0.3525      0.7425      0.2672      0.2781
2                  0.3791      0.1777      0.2328
3                              0.1737      0.3316
4                                          0.1054
```

Warning Errors

`IMSL_CONSTANT_VARIABLE`	Correlations are requested, but the observations on one or more variables are constant. The corresponding correlations are set to NaN.

regression

Fits a multiple linear regression model using least squares.

Synopsis

#include `<imsl.h>`

float `*imsl_f_regression` (*int* `n_observations`, *int* `n_independent`, *float* `x[]`, *float* `y[]`, …, 0)

The type *double* function is `imsl_d_regression`.

Required Arguments

int `n_observations` (Input)
The number of observations.

int `n_independent` (Input)
The number of independent (explanatory) variables.

float `x[]` (Input)
Array of size `n_observations` × `n_independent` containing the matrix of independent (explanatory) variables.

float `y[]` (Input)
Array of length `n_observations` containing the dependent (response) variable.

Return Value

If the optional argument `IMSL_NO_INTERCEPT` is not used, `imsl_f_regression` returns a pointer to an array of length `n_independent` + 1 containing a least-squares solution for the regression coefficients. The estimated intercept is the initial component of the array.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_regression (*int* n_observations, *int* n_independent,
float x[], *float* y[],
IMSL_X_COL_DIM, *int* x_col_dim,
IMSL_NO_INTERCEPT,
IMSL_TOLERANCE, *float* tolerance,
IMSL_RANK, *int* *rank,
IMSL_COEF_COVARIANCES, *float* **p_coef_covariances,
IMSL_COEF_COVARIANCES_USER, *float* coef_covariances[],
IMSL_COV_COL_DIM, *int* cov_col_dim,
IMSL_X_MEAN, *float* **p_x_mean,
IMSL_X_MEAN_USER, *float* x_mean[],
IMSL_RESIDUAL, *float* **p_residual,
IMSL_RESIDUAL_USER, *float* residual[],
IMSL_ANOVA_TABLE, *float* **p_anova_table,
IMSL_ANOVA_TABLE_USER, *float* anova_table[],
IMSL_RETURN_USER, *float* coefficients[],
0)

Optional Arguments

IMSL_X_COL_DIM, *int* x_col_dim (Input)
The column dimension of x.
Default: x_col_dim = n_independent

IMSL_NO_INTERCEPT
By default, the fitted value for observation *i* is

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \ldots + \hat{\beta}_k x_k$$

where *k* = n_independent. If IMSL_NO_INTERCEPT is specified, the intercept term

$$\hat{\beta}_0$$

is omitted from the model.

IMSL_TOLERANCE, *float* tolerance (Input)
The tolerance used in determining linear dependence. For imsl_f_regression, tolerance = 100 × imsl_f_machine(4) is the default choice. For imsl_d_regression, tolerance = 100 × imsl_d_machine(4) is the default. See imsl_f_machine (page 635).

IMSL_RANK, *int* *rank (Output)
The rank of the fitted model is returned in *rank.

IMSL_COEF_COVARIANCES, *float* **p_coef_covariances (Output)
The address of a pointer to the $m \times m$ array containing the estimated variances and covariances of the estimated regression coefficients. Here, m is the number of regression coefficients in the model. If IMSL_NO_INTERCEPT is specified, m = n_independent; otherwise, m = n_independent + 1. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_coef_covariances is declared; &p_coef_covariances is used as an argument to this function; and free(p_coef_covariances) is used to free this array.

IMSL_COEF_COVARIANCES_USER, *float* coef_covariances[] (Output)
If specified, coef_covariances is an array of length $m \times m$ containing the estimated variances and covariances of the estimated coefficients where m is the number of regression coefficients in the model.

IMSL_COV_COL_DIM, *int* cov_col_dim (Input)
The column dimension of array coef_covariance.
Default: cov_col_dim = m where m is the number of regression coefficients in the model.

IMSL_X_MEAN, *float* **p_x_mean (Output)
The address of a pointer to the array containing the estimated means of the independent variables. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_x_mean is declared; &p_x_mean is used as an argument to this function; and free(p_x_mean) is used to free this array.

IMSL_X_MEAN_USER, *float* x_mean[] (Output)
If specified, x_mean is an array of length n_independent provided by the user. On return, x_mean contains the means of the independent variables.

IMSL_RESIDUAL, *float* **p_residual (Output)
The address of a pointer to the array containing the residuals. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_residual is declared; &p_residual is used as argument to this function; and free(p_residual) is used to free this array.

IMSL_RESIDUAL_USER, *float* residual[] (Output)
If specified, residual is an array of length n_observations provided by the user. On return, residual contains the residuals.

IMSL_ANOVA_TABLE, *float* **p_anova_table (Output)
The address of a pointer to the array containing the analysis of variance table. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_anova_table is declared; &p_anova_table is used as argument to this function; and free(p_anova_table) is used to free this array.

The analysis of variance statistics are given as follows:

Element	Analysis of Variance Statistics
0	degrees of freedom for the model
1	degrees of freedom for error
2	total (corrected) degrees of freedom
3	sum of squares for the model
4	sum of squares for error
5	total (corrected) sum of squares
6	model mean square
7	error mean square
8	overall *F*-statistic
9	*p*-value
10	R^2 (in percent)
11	adjusted R^2 (in percent)
12	estimate of the standard deviation
13	overall mean of *y*
14	coefficient of variation (in percent)

`IMSL_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
If specified, the 15 analysis of variance statistics listed above are computed and stored in the array `anova_table` provided by the user.

`IMSL_RETURN_USER`, *float* `coefficients[]` (Output)
If specified, the least-squares solution for the regression coefficients is stored in array `coefficients` provided by the user. If `IMSL_NO_INTERCEPT` is specified, the array requires *m* = `n_independent` units of memory; otherwise, the number of units of memory required to store the coefficients is *m* = `n_independent` + 1.

Description

The function `imsl_f_regression` fits a multiple linear regression model with or without an intercept. By default, the multiple linear regression model is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the y_i's (input in y) are the responses or values of the dependent variable; the x_{i1}'s, x_{i2}'s, …, x_{ik}'s (input in x) are the settings of the *k* (input in `n_independent`) independent variables; β_0, β_1, …, β_k are the regression coefficients whose estimated values are to be output by `imsl_f_regression`; and the ε_i's are independently distributed normal errors each with mean zero and variance σ^2. Here, *n* is the number of rows in the augmented matrix (x,y), i.e., *n* equals `n_observations`. Note that by default, β_0 is included in the model.

The function `imsl_f_regression` computes estimates of the regression coefficients by minimizing the sum of squares of the deviations of the observed response y_i from the fitted response

$$\hat{y}_i$$

for the n observations. This minimum sum of squares (the error sum of squares) is output as one of the analysis of variance statistics if `IMSL_ANOVA_TABLE` (or `IMSL_ANOVA_TABLE_USER`) is specified and is computed as

$$\text{SSE} = \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$$

Another analysis of variance statistic is the total sum of squares. By default, the total sum of squares is the sum of squares of the deviations of y_i from its mean

$$\bar{y}$$

the so-called *corrected total sum of squares*. This statistic is computed as

$$\text{SST} = \sum_{i=1}^{n} \left(y_i - \bar{y}\right)^2$$

When `IMSL_NO_INTERCEPT` is specified, the total sum of squares is the sum of squares of y_i, the so-called *uncorrected total sum of squares*. This is computed as

$$\text{SST} = \sum_{i=1}^{n} y_i^2$$

See Draper and Smith (1981) for a good general treatment of the multiple linear regression model, its analysis, and many examples.

In order to compute a least-squares solution, `imsl_f_regression` performs an orthogonal reduction of the matrix of regressors to upper-triangular form. The reduction is based on one pass through the rows of the augmented matrix (x, y) using fast Givens transformations. (See Golub and Van Loan 1983, pp. 156−162; Gentleman 1974.) This method has the advantage that the loss of accuracy resulting from forming the crossproduct matrix used in the normal equations is avoided.

By default, the current means of the dependent and independent variables are used to internally center the data for improved accuracy. Let x_i be a column vector containing the j-th row of data for the independent variables. Let $\bar{x}_i$ represent the mean vector for the independent variables given the data for rows 1, 2, …, i. The current mean vector is defined to be

$$\bar{x}_i = \frac{\sum_{j=1}^{i} x_j}{i}$$

The i-th row of data has $\bar{x}_i$ subtracted from it and is then weighted by $i/(i-1)$. Although a crossproduct matrix is not computed, the validity of this centering operation can be seen from the following formula for the sum of squares and crossproducts matrix:

$$\sum_{i=1}^{n}(x_i - \bar{x}_n)(x_i - \bar{x}_n)^T = \sum_{i=2}^{n}\frac{i}{i-1}(x_i - \bar{x}_i)(x_i - \bar{x}_i)^T$$

An orthogonal reduction on the centered matrix is computed. When the final computations are performed, the intercept estimate and the first row and column of the estimated covariance matrix of the estimated coefficients are updated (if `IMSL_COEF_COVARIANCES` or `IMSL_COEF_COVARIANCES_USER` is specified) to reflect the statistics for the original (uncentered) data. This means that the estimate of the intercept is for the uncentered data.

As part of the final computations, `imsl_regression` checks for linearly dependent regressors. In particular, linear dependence of the regressors is declared if any of the following three conditions are satisfied:

- A regressor equals zero.
- Two or more regressors are constant.
- $$\sqrt{1 - R^2_{i\cdot 1,2,\ldots,i-1}}$$

  is less than or equal to `tolerance`. Here, $R_{i\cdot 1,2,\ldots,i-1}$ is the multiple correlation coefficient of the i-th independent variable with the first $i-1$ independent variables. If no intercept is in the model, the "multiple correlation" coefficient is computed without adjusting for the mean.

On completion of the final computations, if the i-th regressor is declared to be linearly dependent upon the previous $i-1$ regressors, then the i-th coefficient estimate and all elements in the i-th row and i-th column of the estimated variance-covariance matrix of the estimated coefficients (if `IMSL_COEF_COVARIANCES` or `IMSL_COEF_COVARIANCES_USER` is specified) are set to zero. Finally, if a linear dependence is declared, an informational (error) message, code `IMSL_RANK_DEFICIENT`, is issued indicating the model is not full rank.

Examples

Example 1

A regression model

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i \qquad i = 1,2,\ldots,9$$

is fitted to data taken from Maindonald (1984, pp. 203–204).

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

```
#define INTERCEPT       1
#define N_INDEPENDENT   3
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_OBSERVATIONS  9

main()
{
    float       *coefficients;
    float       x[][N_INDEPENDENT] = {7.0, 5.0, 6.0,
                                       2.0,-1.0, 6.0,
                                       7.0, 3.0, 5.0,
                                      -3.0, 1.0, 4.0,
                                       2.0,-1.0, 0.0,
                                       2.0, 1.0, 7.0,
                                      -3.0,-1.0, 3.0,
                                       2.0, 1.0, 1.0,
                                       2.0, 1.0, 4.0};
    float       y[] = {7.0,-5.0, 6.0, 5.0, 5.0, -2.0, 0.0, 8.0, 3.0};

    coefficients = imsl_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
                                     (float *)x, y, 0);
    imsl_f_write_matrix("Least-Squares Coefficients", 1, N_COEFFICIENTS,
                        coefficients,
                        IMSL_COL_NUMBER_ZERO,
                        0);
}
```

Output

```
      Least-Squares Coefficients
    0           1           2           3
7.733      -0.200       2.333      -1.667
```

Example 2

A weighted least-squares fit is computed using the model

$$y_i = \beta_0 x_{i0} + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i \qquad i = 1, 2, \ldots, 4$$

and weights $1/i^2$ discussed by Maindonald (1984, pp. 67–68). In order to compute the weighted least-squares fit, using an ordinary least-squares function (`imsl_f_regression`), the regressors (including the column of ones for the intercept term) and the responses must be transformed prior to invocation of `imsl_f_regression`. Specifically, the *i*-th response and regressors are multiplied by a square root of the *i*-th weight. `IMSL_NO_INTERCEPT` must be specified since the column of ones corresponding to the intercept term in the untransformed model is transformed by the weights and is regarded as an additional independent variable.

In the example, `IMSL_ANOVA_TABLE` is specified. The minimum sum of squares for error in terms of the original untransformed regressors and responses for this weighted regression is

$$\text{SSE} = \sum_{i=1}^{4} w_i \left(y_i - \hat{y}_i \right)^2$$

where $w_i = 1/i^2$. Also, since IMSL_NO_INTERCEPT is specified, the uncorrected total sum-of-squares terms of the original untransformed responses is

$$\text{SST} = \sum_{i=1}^{4} w_i y_i^2$$

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

#define N_INDEPENDENT   3
#define N_COEFFICIENTS  N_INDEPENDENT
#define N_OBSERVATIONS  4

main()
{
    int         i, j;
    float       *coefficients, w, anova_table[15], power;
    float       x[][N_INDEPENDENT] = {1.0, -2.0, 0.0,
                                      1.0, -1.0, 2.0,
                                      1.0,  2.0, 5.0,
                                      1.0,  7.0, 3.0};
    float       y[] = {-3.0, 1.0, 2.0, 6.0};
    char        *anova_row_labels[] = {
                   "degrees of freedom for regression",
                   "degrees of freedom for error",
                   "total (uncorrected) degrees of freedom",
                   "sum of squares for regression",
                   "sum of squares for error",
                   "total (uncorrected) sum of squares",
                   "regression mean square",
                   "error mean square", "F-statistic",
                   "p-value", "R-squared (in percent)",
                   "adjusted R-squared (in percent)",
                   "est. standard deviation of model error",
                   "overall mean of y",
                   "coefficient of variation (in percent)"};
    power = 0.0;
    for (i = 0;  i < N_OBSERVATIONS;  i++)  {
        power += 1.0;
                                /* The square root of the weight */
        w = sqrt(1.0 / (power*power));
                                /* Transform response */
        y[i] *= w;
                                /* Transform regressors */
        for (j = 0;  j < N_INDEPENDENT;  j++)
            x[i][j] *= w;
    }

    coefficients = imsl_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
                                     (float *)x, y,
                                     IMSL_NO_INTERCEPT,
                                     IMSL_ANOVA_TABLE_USER,
                                     anova_table, 0);

    imsl_f_write_matrix("Least-Squares Coefficients", 1,
                        N_COEFFICIENTS, coefficients, 0);
    imsl_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
                        anova_table, IMSL_ROW_LABELS, anova_row_labels,
```

```
                        IMSL_WRITE_FORMAT, "%10.2f", 0);
}
```

Output

```
Least-Squares Coefficients
     1           2           3
-1.431       0.658       0.748

         * * * Analysis of Variance * * *

degrees of freedom for regression             3.00
degrees of freedom for error                  1.00
total (uncorrected) degrees of freedom        4.00
sum of squares for regression                10.93
sum of squares for error                      1.01
total (uncorrected) sum of squares           11.94
regression mean square                        3.64
error mean square                             1.01
F-statistic                                   3.60
p-value                                       0.37
R-squared (in percent)                       91.52
adjusted R-squared (in percent)              66.08
est. standard deviation of model error        1.01
overall mean of y                            -0.08
coefficient of variation (in percent)     -1207.73
```

Warning Errors

`IMSL_RANK_DEFICIENT`	The model is not full rank. There is not a unique least-squares solution.

poly_regression

Performs a polynomial least-squares regression.

Synopsis

#include `<imsl.h>`

float `*imsl_f_poly_regression` (*int* `n_observations`, *float* `x[]`, *float* `y[]`, *int* `degree`, …, 0)

The type *double* procedure is `imsl_d_poly_regression`.

Required Arguments

int `n_observations` (Input)
The number of observations.

float `x[]` (Input)
Array of length `n_observations` containing the independent variable.

float `y[]` (Input)
Array of length `n_observations` containing the dependent variable.

int `degree` (Input)
The degree of the polynomial.

Return Value

A pointer to the vector of size `degree` + 1 containing the coefficients of the fitted polynomial. If a fit cannot be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_poly_regression (`*int* `n_observations,` *float* `xdata[],` *float* `ydata[],` *int* `degree,`
`IMSL_WEIGHTS,` *float* `weights[],`
`IMSL_SSQ_POLY,` *float* `**p_ssq_poly,`
`IMSL_SSQ_POLY_USER,` *float* `ssq_poly[],`
`IMSL_SSQ_POLY_COL_DIM,` *int* `ssq_poly_col_dim,`
`IMSL_SSQ_LOF,` *float* `**p_ssq_lof,`
`IMSL_SSQ_LOF_USER,` *float* `ssq_lof[],`
`IMSL_SSQ_LOF_COL_DIM,` *int* `ssq_lof_col_dim,`
`IMSL_X_MEAN,` *float* `*x_mean,`
`IMSL_X_VARIANCE,` *float* `*x_variance,`
`IMSL_ANOVA_TABLE,` *float* `**p_anova_table,`
`IMSL_ANOVA_TABLE_USER,` *float* `anova_table[],`
`IMSL_DF_PURE_ERROR,` *int* `*df_pure_error,`
`IMSL_SSQ_PURE_ERROR,` *float* `*ssq_pure_error,`
`IMSL_RESIDUAL,` *float* `**p_residual,`
`IMSL_RESIDUAL_USER,` *float* `residual[],`
`IMSL_RETURN_USER,` *float* `coefficients[],`
0)

Optional Arguments

`IMSL_WEIGHTS,` *float* `weights[]` (Input)
Array with `n_observations` components containing the vector of weights for the observation. If this option is not specified, all observations have equal weights of one.

`IMSL_SSQ_POLY,` *float* `**p_ssq_poly` (Output)
The address of a pointer to the array containing the sequential sums of squares and other statistics. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*p_ssq_poly` is declared; `&p_ssq_poly` is used as an argument to this function; and `free(p_ssq_poly)` is used to free this array. Row *i* corresponds to x^i, $i = 1, \ldots,$ degree, and the columns are described as follows:

Column	Description
1	degrees of freedom
2	sums of squares
3	*F*-statistic
4	*p*-value

`IMSL_SSQ_POLY_USER`, *float* `ssq_poly[]` (Output)
Array of size `degree` × 4 containing the sequential sums of squares for a polynomial fit described under optional argument `IMSL_SSQ_POLY`.

`IMSL_SSQ_POLY_COL_DIM`, *int* `ssq_poly_col_dim` (Input)
The column dimension of `ssq_poly`.
Default: `ssq_poly_col_dim` = 4

`IMSL_SSQ_LOF`, *float* `**p_ssq_lof` (Output)
The address of a pointer to the array containing the lack-of-fit statistics. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*p_ssq_lof` is declared; `&p_ssq_lof` is used as an argument to this function; and `free(p_ssq_lof)` is used to free this array. Row *i* corresponds to x^i, *i* = 1, …, degree, and the columns are described in the following table.

Column	Description
1	degrees of freedom
2	lack-of-fit sums of squares
3	*F*-statistic for testing lack-of-fit for a polynomial model of degree *i*
4	*p*-value for the test

`IMSL_SSQ_LOF_USER`, *float* `ssq_lof[]` (Output)
Array of size `degree` × 4 containing the matrix of lack-of-fit statistics described under optional argument `IMSL_SSQ_LOF`.

`IMSL_SSQ_LOF_COL_DIM`, *int* `ssq_lof_col_dim` (Input)
The column dimension of `ssq_lof`.
Default: `ssq_lof_col_dim` = 4

`IMSL_X_MEAN`, *float* `*x_mean` (Output)
The mean of *x*.

`IMSL_X_VARIANCE`, *float* `*x_variance` (Output)
The variance of *x*.

`IMSL_ANOVA_TABLE`, *float* `**p_anova_table` (Output)
The address of a pointer to the array containing the analysis of variance table. On return, the pointer is initialized (through a memory allocation request to

malloc), and the array is stored there. Typically, *float* *p_anova_table is declared; &p_anova_table is used as an argument to this function; and free(p_anova_table) is used to free this array.

Element	Analysis of Variance Statistic
0	degrees of freedom for the model
1	degrees of freedom for error
2	total (corrected) degrees of freedom
3	sum of squares for the model
4	sum of squares for error
5	total (corrected) sum of squares
6	model mean square
7	error mean square
8	overall *F*-statistic
9	*p*-value
10	R^2 (in percent)
11	adjusted R^2 (in percent)
12	estimate of the standard deviation
13	overall mean of *y*
14	coefficient of variation (in percent)

IMSL_ANOVA_TABLE_USER, *float* anova_table[] (Output)
Array of size 15 containing the analysis variance statistics listed under optional argument IMSL_ANOVA_TABLE.

IMSL_DF_PURE_ERROR, *int* *df_pure_error (Output)
If specified, the degrees of freedom for pure error are returned in df_pure_error.

IMSL_SSQ_PURE_ERROR, *float* *ssq_pure_error (Output)
If specified, the sums of squares for pure error are returned in ssq_pure_error.

IMSL_RESIDUAL, *float* **p_residual (Output)
The address of a pointer to the array containing the residuals. On return, the pointer is initialized (through a memory allocation request to malloc), and the array is stored there. Typically, *float* *p_residual is declared; &p_residual is used as an argument to this function; and free(p_residual)is used to free this array.

IMSL_RESIDUAL_USER, *float* residual[] (Output)
If specified, residual is an array of length n_observations provided by the user. On return, residual contains the residuals.

`IMSL_RETURN_USER`, *float* `coefficients[]` (Output)
If specified, the least-squares solution for the regression coefficients is stored in array `coefficients` of size `degree` + 1 provided by the user.

Description

The function `imsl_f_poly_regression` computes estimates of the regression coefficients in a polynomial (curvilinear) regression model. In addition to the computation of the fit, `imsl_f_poly_regression` computes some summary statistics. Sequential sums of squares attributable to each power of the independent variable (stored in `ssq_poly`) are computed. These are useful in assessing the importance of the higher order powers in the fit. Draper and Smith (1981, pp. 101−102) and Neter and Wasserman (1974, pp. 278−287) discuss the interpretation of the sequential sums of squares. The statistic R^2 is the percentage of the sum of squares of y about its mean explained by the polynomial curve. Specifically,

$$R^2 = \frac{\sum (\hat{y}_i - \bar{y})^2}{\sum (y_1 - \bar{y})^2} 100\%$$

where $\hat{y}_i$ is the fitted y value at x_i and $\bar{y}$ is the mean of y. This statistic is useful in assessing the overall fit of the curve to the data. R^2 must be between 0% and 100%, inclusive. R^2 = 100% indicates a perfect fit to the data.

Estimates of the regression coefficients in a polynomial model are computed using orthogonal polynomials as the regressor variables. This reparameterization of the polynomial model in terms of orthogonal polynomials has the advantage that the loss of accuracy resulting from forming powers of the x-values is avoided. All results are returned to the user for the original model (power form).

The function `imsl_f_poly_regression` is based on the algorithm of Forsythe (1957). A modification to Forsythe's algorithm suggested by Shampine (1975) is used for computing the polynomial coefficients. A discussion of Forsythe's algorithm and Shampine's modification appears in Kennedy and Gentle (1980, pp. 342−347).

Examples

Example 1

A polynomial model is fitted to data discussed by Neter and Wasserman (1974, pp. 279−285). The data set contains the response variable y measuring coffee sales (in hundred gallons) and the number of self-service coffee dispensers. Responses for 14 similar cafeterias are in the data set. A graph of the results also is given.

```
#include <imsl.h>

#define DEGREE          2
#define NOBS           14

main()
{
    float       *coefficients;
```

```
    float       x[] = {0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 4.0,
                       4.0, 5.0, 5.0, 6.0, 6.0, 7.0, 7.0};
    float       y[] = {508.1, 498.4, 568.2, 577.3, 651.7, 657.0, 755.3,
                       758.9, 787.6, 792.1, 841.4, 831.8, 854.7, 871.4};

    coefficients = imsl_f_poly_regression (NOBS, x, y, DEGREE, 0);

    imsl_f_write_matrix("Least-Squares Polynomial Coefficients",
                        DEGREE + 1, 1, coefficients,
                        IMSL_ROW_NUMBER_ZERO,
                        0);
}
```

Output

```
Least-Squares Polynomial Coefficients
            0       503.3
            1        78.9
            2        -4.0
```

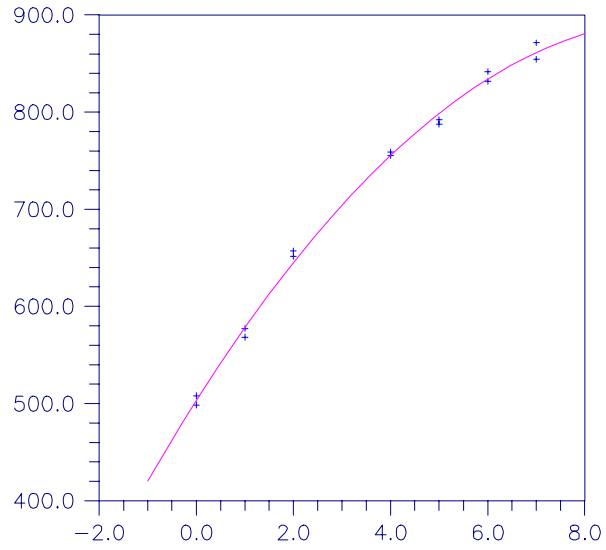


Figure 10-1 A Polynomial Fit

Example 2

This example is a continuation of the initial example. Here, many optional arguments are used.

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

#define DEGREE           2
#define NOBS            14

void main()
{
    int        iset = 1, dfpe;
```

```
    float      *coefficients, *anova, sspe, *sspoly, *sslof;
    float      x[] = {0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 4.0,
                      4.0, 5.0, 5.0, 6.0, 6.0, 7.0, 7.0};
    float      y[] = {508.1, 498.4, 568.2, 577.3, 651.7, 657.0, 755.3,
                      758.9, 787.6, 792.1, 841.4, 831.8, 854.7, 871.4};
    char       *coef_rlab[2];
    char       *coef_clab[] = {" ", "intercept", "linear", "quadratic"};
    char       *stat_clab[] = {" ", "Degrees of\nFreedom",
                               "Sum of\nSquares", "\nF-Statistic",
                               "\np-value"};
    char       *anova_rlab[] = {
                   "degrees of freedom for regression",
                   "degrees of freedom for error",
                   "total (corrected) degrees of freedom",
                   "sum of squares for regression",
                   "sum of squares for error",
                   "total (corrected) sum of squares",
                   "regression mean square",
                   "error mean square", "F-statistic",
                   "p-value", "R-squared (in percent)",
                   "adjusted R-squared (in percent)",
                   "est. standard deviation of model error",
                   "overall mean of y",
                   "coefficient of variation (in percent)"};

    coefficients = imsl_f_poly_regression (NOBS, x, y, DEGREE,
                                           IMSL_SSQ_POLY, &sspoly,
                                           IMSL_SSQ_LOF, &sslof,
                                           IMSL_ANOVA_TABLE, &anova,
                                           IMSL_DF_PURE_ERROR, &dfpe,
                                           IMSL_SSQ_PURE_ERROR, &sspe,
                                           0);
    imsl_write_options(-1, &iset);
    imsl_f_write_matrix("Least-Squares Polynomial Coefficients",
                        1, DEGREE + 1, coefficients,
                        IMSL_COL_LABELS, coef_clab, 0);
    coef_rlab[0] = coef_clab[2];
    coef_rlab[1] = coef_clab[3];
    imsl_f_write_matrix("Sequential Statistics", DEGREE, 4, sspoly,
                        IMSL_COL_LABELS, stat_clab,
                        IMSL_ROW_LABELS, coef_rlab,
                        IMSL_WRITE_FORMAT, "%3.1f%8.1f%6.1f%6.4f",
                        0);
    imsl_f_write_matrix("Lack-of-Fit Statistics", DEGREE, 4, sslof,
                        IMSL_COL_LABELS, stat_clab,
                        IMSL_ROW_LABELS, coef_rlab,
                        IMSL_WRITE_FORMAT,  "%3.1f%8.1f%6.1f%6.4f",
                        0);
    imsl_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
                                                              anova,
                        IMSL_ROW_LABELS, anova_rlab,
                        IMSL_WRITE_FORMAT, "%9.2f",
                        0);
}
```

Output

```
                Least-Squares Polynomial Coefficients
                    intercept      linear   quadratic
```

```
               503.3        78.9        -4.0

                Sequential Statistics
             Degrees of    Sum of
                Freedom   Squares  F-Statistic  p-value
linear              1.0  220644.2       3415.8   0.0000
quadratic           1.0    4387.7         67.9   0.0000

                Lack-of-Fit Statistics
             Degrees of    Sum of
                Freedom   Squares  F-Statistic  p-value
linear              5.0    4793.7         22.0   0.0004
quadratic           4.0     405.9          2.3   0.1548

           * * * Analysis of Variance * * *

  degrees of freedom for regression            2.00
  degrees of freedom for error                11.00
  total (corrected) degrees of freedom        13.00
  sum of squares for regression           225031.94
  sum of squares for error                   710.55
  total (corrected) sum of squares        225742.48
  regression mean square                  112515.97
  error mean square                           64.60
  F-statistic                               1741.86
  p-value                                      0.00
  R-squared (in percent)                      99.69
  adjusted R-squared (in percent)             99.63
  est. standard deviation of model error       8.04
  overall mean of y                          710.99
  coefficient of variation (in percent)        1.13
```

Warning Errors

IMSL_CONSTANT_YVALUES	The *y* values are constant. A zero-order polynomial is fit. High order coefficients are set to zero.
IMSL_FEW_DISTINCT_XVALUES	There are too few distinct *x* values to fit the desired degree polynomial. High order coefficients are set to zero.
IMSL_PERFECT_FIT	A perfect fit was obtained with a polynomial of degree less than `degree`. High order coefficients are set to zero.

Fatal Errors

IMSL_NONNEG_WEIGHT_REQUEST_2	All weights must be nonnegative.
IMSL_ALL_OBSERVATIONS_MISSING	Each (*x*, *y*) point contains NaN (not a number). There are no valid data.
IMSL_CONSTANT_XVALUES	The *x* values are constant.

ranks

Computes the ranks, normal scores, or exponential scores for a vector of observations.

Synopsis

#include `<imsl.h>`

float `*imsl_f_ranks` (*int* `n_observations`, *float* `x[]`, …, 0)

The type *double* function is `imsl_d_ranks`.

Required Arguments

int `n_observations` (Input)
The number of observations.

float `x[]` (Input)
Array of length `n_observations` containing the observations to be ranked.

Return Value

A pointer to a vector of length `n_observations` containing the rank (or optionally, a transformation of the rank) of each observation.

Synopsis with Optional Arguments

#include `<imsl.h>`

*float** `imsl_f_ranks` (*int* `n_observations`, *float* `x[]`,
`IMSL_AVERAGE_TIE`,
`IMSL_HIGHEST`,
`IMSL_LOWEST`,
`IMSL_RANDOM_SPLIT`,
`IMSL_FUZZ`, *float* `fuzz_value`,
`IMSL_RANKS`,
`IMSL_BLOM_SCORES`,
`IMSL_TUKEY_SCORES`,
`IMSL_VAN_DER_WAERDEN_SCORES`,
`IMSL_EXPECTED_NORMAL_SCORES`,
`IMSL_SAVAGE_SCORES`,
`IMSL_RETURN_USER`, *float* `ranks[]`,
0)

Optional Arguments

`IMSL_AVERAGE_TIE`, *or*
`IMSL_HIGHEST`, *or*
`IMSL_LOWEST`, *or*

IMSL_RANDOM_SPLIT

Exactly one of these optional arguments may be used to change the method used to assign a score to tied observations.

Keyword	Method
IMSL_AVERAGE_TIE	average of the scores of the tied observations (default)
IMSL_HIGHEST	highest score in the group of ties
IMSL_LOWEST	lowest score in the group of ties
IMSL_RANDOM_SPLIT	tied observations are randomly split using a random number generator.

IMSL_FUZZ, *float* fuzz_value (Input)

Value used to determine when two items are tied. If abs(x[i]-x[j]) is less than or equal to fuzz_value, then x[i] and x[j] are said to be tied. The default value for fuzz_value is 0.0.

IMSL_RANKS, *or*
IMSL_BLOM_SCORES, *or*
IMSL_TUKEY_SCORES, *or*
IMSL_VAN_DER_WAERDEN_SCORES, *or*
IMSL_EXPECTED_NORMAL_SCORES, *or*
IMSL_SAVAGE_SCORES

Exactly one of these optional arguments may be used to specify the type of values returned.

Keyword	Result
IMSL_RANKS	ranks (default)
IMSL_BLOM_SCORES	Blom version of normal scores
IMSL_TUKEY_SCORES	Tukey version of normal scores
IMSL_VAN_DER_WAERDEN_SCORES	Van der Waerden version of normal scores
IMSL_EXPECTED_NORMAL_SCORES	expected value of normal order statistics (For tied observations, the average of the expected normal scores.)
IMSL_SAVAGE_SCORES	Savage scores (the expected value of exponential order statistics)

IMSL_RETURN_USER, *float* ranks[] (Output)

If specified, the ranks are returned in the user-supplied array ranks.

Description

Ties

In data without ties, the output values are the ordinary ranks (or a transformation of the ranks) of the data in `x`. If `x[i]` has the smallest value among the values in `x` and there is no other element in `x` with this value, then `ranks[i]` = 1. If both `x[i]` and `x[j]` have the same smallest value, then the output value depends upon the option used to break ties.

Keyword	Result
`IMSL_AVERAGE_TIE`	`ranks[i]` = `ranks[j]` = 1.5
`IMSL_HIGHEST`	`ranks[i]` = `ranks[j]` = 2.0
`IMSL_LOWEST`	`ranks[i]` = `ranks [j]` = 1.0
`IMSL_RANDOM_SPLIT`	`ranks[i]` = 1.0 and `ranks[j]` = 2.0 or, randomly, `ranks[i]` = 2.0 and `ranks[j]` = 1.0

When the ties are resolved randomly, the function `imsl_f_random_uniform` is used to generate random numbers. Different results may occur from different executions of the program unless the "seed" of the random number generator is set explicitly by use of the function **`imsl_random_seed_set`** (page 564).

The Scores

Normal and other functions of the ranks can optionally be returned. Normal scores can be defined as the expected values, or approximations to the expected values, of order statistics from a normal distribution. The simplest approximations are obtained by evaluating the inverse cumulative normal distribution function, `imsl_f_normal_inverse_cdf`, at the ranks scaled into the open interval (0,1). In the Blom version (see Blom 1958), the scaling transformation for the rank r_i ($1 \le r_i \le n$ where n is the sample size, `n_observations`) is $(r_i - 3/8)/(n + 1/4)$. The Blom normal score corresponding to the observation with rank r_i is

$$\Phi^{-1}\left(\frac{r_i - 3/8}{n + 1/4}\right)$$

where $\Phi(\cdot)$ is the normal cumulative distribution function.

Adjustments for ties are made after the normal score transformation; that is, if `x[i]` equals `x[j]` (within `fuzz_value`) and their value is the k-th smallest in the data set, the Blom normal scores are determined for ranks of k and $k + 1$. Then, these normal scores are averaged or selected in the manner specified. (Whether the transformations are made first or ties are resolved first makes no difference except when `IMSL_AVERAGE` is specified.)

In the Tukey version (see Tukey 1962), the scaling transformation for the rank r_i is $(r_i - 1/3)/(n + 1/3)$. The Tukey normal score corresponding to the observation with rank r_i is

$$\Phi^{-1}\left(\frac{r_i - 1/3}{n + 1/3}\right)$$

Ties are handled in the same way as for the Blom normal scores.

In the Van der Waerden version (see Lehmann 1975, p. 97), the scaling transformation for the rank r_i is $r_i/(n + 1)$. The Van der Waerden normal score corresponding to the observation with rank r_i is

$$\Phi^{-1}\left(\frac{r_i}{n+1}\right)$$

Ties are handled in the same way as for the Blom normal scores.

When option `IMSL_EXPECTED_NORMAL_SCORES` is used, the output values are the expected values of the normal order statistics from a sample of size `n_observations`. If the value in `x[i]` is the k-th smallest, then the value output in `ranks[i]` is $E(z_k)$ where $E(\cdot)$ is the expectation operator, and z_k is the k-th order statistic in a sample of size `n_observations` from a standard normal distribution. Ties are handled in the same way as for the Blom normal scores.

Savage scores are the expected values of the exponential order statistics from a sample of size `n_observations`. These values are called Savage scores because of their use in a test discussed by Savage (1956) (see Lehmann 1975). If the value in `x[i]` is the k-th smallest, then the value output in `ranks[i]` is $E(y_k)$ where y_k is the k-th order statistic in a sample of size `n_observations` from a standard exponential distribution. The expected value of the k-th order statistic from an exponential sample of size n (`n_observations`) is

$$\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1}$$

Ties are handled in the same way as for the Blom normal scores.

Examples

Example 1

The data for this example, from Hinkley (1977), contains 30 observations. Note that the fourth and sixth observations are tied, and that the third and twentieth observations are tied.

```
#include <imsl.h>

#define N_OBSERVATIONS          30

main()
{
```

```
    float       *ranks;
    float       x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                       3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                       1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                       4.75, 2.48, 0.96, 1.89, 0.90, 2.05};

    ranks = imsl_f_ranks(N_OBSERVATIONS, x, 0);
    imsl_f_write_matrix("Ranks" , 1, N_OBSERVATIONS, ranks, 0);
}
```

Output

```
                          Ranks
  1           2           3           4           5           6
5.0        18.0         6.5        11.5        21.0        11.5

  7           8           9          10          11          12
2.0        15.0        29.0        24.0        27.0        28.0

  13          14          15          16          17          18
16.0        23.0         3.0        17.0        13.0         1.0

 19          20          21          22          23          24
4.0         6.5        26.0        19.0        10.0        14.0

  25          26          27          28          29          30
30.0        25.0         9.0        20.0         8.0        22.0
```

Example 2

This example uses all of the score options with the same data set, which contains some ties. Ties are handled in several different ways in this example.

```
#include <imsl.h>

#define N_OBSERVATIONS          30

void main()
{
    float       fuzz_value=0.0, score[4][N_OBSERVATIONS], *ranks;
    float       x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                       3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                       1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                       4.75, 2.48, 0.96, 1.89, 0.90, 2.05};
    char        *row_labels[] = {"Blom", "Tukey", "Van der Waerden",
                                 "Expected Value"};

                                /* Blom scores using largest ranks */
                                /* for ties */
    imsl_f_ranks(N_OBSERVATIONS, x,
                 IMSL_HIGHEST,
                 IMSL_BLOM_SCORES,
                 IMSL_RETURN_USER,   &score[0][0],
                 0);
                                /* Tukey normal scores using smallest */
                                /* ranks for ties */
    imsl_f_ranks(N_OBSERVATIONS, x,
                 IMSL_LOWEST,
                 IMSL_TUKEY_SCORES,
```

```
                IMSL_RETURN_USER,  &score[1][0],
                0);
                                /* Van der Waerden scores using */
                                /* randomly resolved ties */
    imsl_random_seed_set(123457);
    imsl_f_ranks(N_OBSERVATIONS, x,
                IMSL_RANDOM_SPLIT,
                IMSL_VAN_DER_WAERDEN_SCORES,
                IMSL_RETURN_USER, &score[2][0],
                0);
                                /* Expected value of normal order */
                                /* statistics using averaging to */
                                /* break ties */
    imsl_f_ranks(N_OBSERVATIONS, x,
                IMSL_EXPECTED_NORMAL_SCORES,
                IMSL_RETURN_USER, &score[3][0],
                0);
    imsl_f_write_matrix("Normal Order Statistics", 4, N_OBSERVATIONS,
                        (float *)score,
                        IMSL_ROW_LABELS, row_labels,
                        0);
                                /* Savage scores using averaging */
                                /* to break ties */
    ranks = imsl_f_ranks(N_OBSERVATIONS, x,
                         IMSL_SAVAGE_SCORES,
                         0);
    imsl_f_write_matrix("Expected values of exponential order "
                        "statistics", 1,
                        N_OBSERVATIONS, ranks,
                        0);
}
```

Output

```
                          Normal Order Statistics
                    1          2          3          4          5
Blom           -1.024      0.209     -0.776     -0.294      0.473
Tukey          -1.020      0.208     -0.890     -0.381      0.471
Van der Waerden -0.989     0.204     -0.753     -0.287      0.460
Expected Value -1.026      0.209     -0.836     -0.338      0.473

                    6          7          8          9         10
Blom           -0.294     -1.610     -0.041      1.610      0.776
Tukey          -0.381     -1.599     -0.041      1.599      0.773
Van der Waerden -0.372    -1.518     -0.040      1.518      0.753
Expected Value -0.338     -1.616     -0.041      1.616      0.777

                   11         12         13         14         15
Blom            1.176      1.361      0.041      0.668     -1.361
Tukey           1.171      1.354      0.041      0.666     -1.354
Van der Waerden  1.131     1.300      0.040      0.649     -1.300
Expected Value   1.179     1.365      0.041      0.669     -1.365

                   16         17         18         19         20
Blom            0.125     -0.209     -2.040     -1.176     -0.776
Tukey           0.124     -0.208     -2.015     -1.171     -0.890
Van der Waerden  0.122    -0.204     -1.849     -1.131     -0.865
Expected Value   0.125     -0.209     -2.043     -1.179     -0.836
```

```
                    21          22          23          24          25
Blom              1.024       0.294      -0.473      -0.125       2.040
Tukey             1.020       0.293      -0.471      -0.124       2.015
Van der Waerden   0.989       0.287      -0.460      -0.122       1.849
Expected Value    1.026       0.294      -0.473      -0.125       2.043

                    26          27          28          29          30
Blom              0.893      -0.568       0.382      -0.668       0.568
Tukey             0.890      -0.566       0.381      -0.666       0.566
Van der Waerden   0.865      -0.552       0.372      -0.649       0.552
Expected Value    0.894      -0.568       0.382      -0.669       0.568

           Expected values of exponential order statistics
        1           2           3           4           5           6
    0.179       0.892       0.240       0.474       1.166       0.474

        7           8           9          10          11          12
    0.068       0.677       2.995       1.545       2.162       2.495

       13          14          15          16          17          18
    0.743       1.402       0.104       0.815       0.555       0.033

       19          20          21          22          23          24
    0.141       0.240       1.912       0.975       0.397       0.614

       25          26          27          28          29          30
    3.995       1.712       0.350       1.066       0.304       1.277
```

random_seed_get

Retrieves the current value of the seed used in the IMSL random number generators.

Synopsis

#include `<imsl.h>`

int `imsl_random_seed_get ( )`

Return Value

The value of the seed.

Description

The function `imsl_random_seed_get` retrieves the current value of the "seed" used in the random number generators. A reason for doing this would be to restart a simulation, using `imsl_random_seed_set` to reset the seed.

Example

This example illustrates the statements required to restart a simulation using `imsl_random_seed_get` and `imsl_random_seed_set`. Also, the example shows that restarting the sequence of random numbers at the value of the seed last generated is the same as generating the random numbers all at once.

```
#include <imsl.h>

#define     N_RANDOM     5

main()
{
    int         seed = 123457;
    float       *r1, *r2, *r;

    imsl_random_seed_set(seed);
    r1 = imsl_f_random_uniform(N_RANDOM, 0);
    imsl_f_write_matrix ("First Group of Random Numbers", 1,
                         N_RANDOM, r1, 0);
    seed = imsl_random_seed_get();

    imsl_random_seed_set(seed);
    r2 = imsl_f_random_uniform(N_RANDOM, 0);
    imsl_f_write_matrix ("Second Group of Random Numbers", 1,
                         N_RANDOM, r2, 0);

    imsl_random_seed_set(123457);
    r = imsl_f_random_uniform(2*N_RANDOM, 0);
    imsl_f_write_matrix ("Both Groups of Random Numbers", 1,
                         2*N_RANDOM, r, 0);
}
```

Output

```
            First Group of Random Numbers
     1           2           3           4           5
0.9662      0.2607      0.7663      0.5693      0.8448

           Second Group of Random Numbers
     1           2           3           4           5
0.0443      0.9872      0.6014      0.8964      0.3809

                Both Groups of Random Numbers
     1           2           3           4           5           6
0.9662      0.2607      0.7663      0.5693      0.8448      0.0443

     7           8           9          10
0.9872      0.6014      0.8964      0.3809
```

random_seed_set

Initializes a random seed for use in the IMSL random number generators.

Synopsis

#include `<imsl.h>`

void `imsl_random_seed_set` (*int* `seed`)

Required Arguments

int `seed` (Input)

The seed of the random number generator. The argument `seed` must be in the range (0, 2147483646). If `seed` is zero, a value is computed using the system clock. Hence, the results of programs using the IMSL random number generators will be different at various times.

Description

The function `imsl_random_seed_set` is used to initialize the seed used in the IMSL random number generators. The form of the generators is

$$x_i \equiv cx_{i-1} \mod (2^{31} - 1)$$

The value of x_0 is the seed. If the seed is not initialized prior to invocation of any of the routines for random number generation by calling `imsl_random_seed_set`, the seed is initialized via the system clock. The seed can be reinitialized to a clock-dependent value by calling `imsl_random_seed_set` with `seed` set to 0.

The effect of `imsl_random_seed_set` is to set some global values used by the random number generators.

A common use of `imsl_random_seed_set` is in conjunction with `imsl_random_seed_get` to restart a simulation.

Example

See function **`imsl_random_seed_get`** (page 563).

random_option

Selects the uniform (0,1) multiplicative congruential pseudorandom number generator.

Synopsis

#include `<imsl.h>`

void `imsl_random_option` (*int* `generator_option`)

Required Arguments

int `generator_option` (Input)

Indicator of the generator. The random number generator is a multiplicative congruential generator with modulus $2^{31} - 1$. Argument `generator_option` is used to choose the multiplier and whether or not shuffling is done.

generator_option	Generator
1	multiplier 16807 used
2	multiplier 16807 used with shuffling
3	multiplier 397204094 used
4	multiplier 397204094 used with shuffling
5	multiplier 950706376 used
6	multiplier 950706376 used with shuffling

Description

The IMSL uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling. The value of the multiplier and whether or not to use shuffling are determined by `imsl_random_option`. The description of function `imsl_f_random_uniform` may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (Lewis et al. 1969).

Example

The C statement

```
imsl_random_option(1)
```

selects the simple multiplicative congruential generator with multiplier 16807. Since this is the same as the default, this statement has no effect unless `imsl_random_option` had previously been called in the same program to select a different generator.

random_uniform

Generates pseudorandom numbers from a uniform (0,1) distribution.

Synopsis

#include `<imsl.h>`

float `*imsl_f_random_uniform` (*int* `n_random`, …, 0)

The type *double* function is `imsl_d_random_uniform`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

A pointer to a vector of length `n_random` containing the random uniform (0, 1) deviates.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_random_uniform` (*int* `n_random`,
`IMSL_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `r[]` (Output)
If specified, the array of length `n_random` containing the random uniform (0, 1) deviates is returned in the user-provided array `r`.

Description

The function `imsl_f_random_uniform` generates pseudorandom numbers from a uniform (0, 1) distribution using a multiplicative congruential method. The form of the generator is

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0,1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by the function `imsl_random_option`. The choice of 16807 will result in the fastest execution time. If no selection is made explicitly, the functions use the multiplier 16807.

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

The user can select a shuffled version of these generators. In this scheme, a table is filled with the first 128 uniform (0, 1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number; and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table.

The values returned by `imsl_f_random_uniform` are positive and less than 1.0. Some values returned may be smaller than the smallest relative spacing, however. Hence, it may be the case that some value, for example `r[i]`, is such that 1.0 – `r[i]` = 1.0.

Deviates from the distribution with uniform density over the interval (a, b) can be obtained by scaling the output from `imsl_f_random_uniform`. The following

statements (in single precision) would yield random deviates from a uniform (*a*, *b*) distribution.

```
float *r;
r = imsl_f_random_uniform (n_random, 0);
for (i=0; i<n_random; i++) r[i]*(b-a) + a;
```

Example

In this example, `imsl_f_random_uniform` is used to generate five pseudorandom uniform numbers. Since `imsl_random_option` is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

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

#define N_RANDOM        5

void main()
{
    float        *r;

    imsl_random_seed_set(123457);

    r = imsl_f_random_uniform(N_RANDOM, 0);

    printf("Uniform random deviates: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Uniform random deviates:   0.9662  0.2607  0.7663  0.5693  0.8448
```

random_normal

Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.

Synopsis

#include `<imsl.h>`

float `*imsl_f_random_normal` (*int* `n_random`, …, 0)

The type *double* function is `imsl_d_random_normal`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

A pointer to a vector of length `n_random` containing the random standard normal deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_random_normal (*int* n_random,
IMSL_RETURN_USER, *float* r[],
0)

Optional Arguments

IMSL_RETURN_USER, *float* r[] (Output)
Pointer to a vector of length n_random that will contain the generated random standard normal deviates.

Description

Function imsl_f_random_normal generates pseudorandom numbers from a standard normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0, 1) random deviate is generated. Then, the inverse of the normal distribution function is evaluated at that point, using the function imsl_f_normal_inverse_cdf.

Deviates from the normal distribution with mean mean and standard deviation std_dev can be obtained by scaling the output from imsl_f_random_normal. The following statements (in single precision) would yield random deviates from a normal (mean, std_dev^2) distribution.

```
float *r;
r = imsl_f_random_normal (n_random, 0);
for (i=0; i<n_random; i++)
   r[i] = r[i]*std_dev + mean;
```

Example

In this example, imsl_f_random_normal is used to generate five pseudorandom deviates from a standard normal distribution.

```
#include <imsl.h>

#define N_RANDOM        5

void main()
{
    int         seed = 123457;
    int         n_random = N_RANDOM;
    float       *r;

    imsl_random_seed_set (seed);
    r = imsl_f_random_normal(n_random, 0);
    printf("%s: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
           "Standard normal random deviates",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Standard normal random deviates:   1.8279 -0.6412  0.7266  0.1747  1.0145
```

Remark

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

random_poisson

Generates pseudorandom numbers from a Poisson distribution.

Synopsis

#include `<imsl.h>`

int `*imsl_random_poisson` (*int* `n_random`, *float* `theta`, …, 0)

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `theta` (Input)
Mean of the Poisson distribution. The argument `theta` must be positive.

Return Value

If no optional arguments are used, `imsl_random_poisson` returns a pointer to a vector of length `n_random` containing the random Poisson deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

int `*imsl_random_poisson` (*int* `n_random`, *float* `theta`,
`IMSL_RETURN_USER`, *int* `r[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *int* `r[]` (Output)
If specified, the vector of length `n_random` of random Poisson deviates is returned in the user-provided array `r`.

Description

The function `imsl_random_poisson` generates pseudorandom numbers from a Poisson distribution with positive mean `theta`. The probability function (with θ = `theta`) is

$$f(x) = (e^{-\theta}\theta^{x})/x!, \qquad \text{for } x = 0, 1, 2, \ldots$$

If `theta` is less than 15, `imsl_random_poisson` uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) (see also Schmeiser 1983) is used. The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

Example

In this example, `imsl_random_poisson` is used to generate five pseudorandom deviates from a Poisson distribution with mean equal to 0.5.

```
#include <imsl.h>

#define N_RANDOM       5

void main()
{
    int         *r;
    int         seed = 123457;
    float       theta = 0.5;

    imsl_random_seed_set (seed);
    r = imsl_random_poisson (N_RANDOM, theta, 0);
    imsl_i_write_matrix ("Poisson(0.5) random deviates", 1, 5, r, 0);
}
```

Output

```
Poisson(0.5) random deviates
      1   2   3   4   5
      2   0   1   0   1
```

random_gamma

Generates pseudorandom numbers from a standard gamma distribution.

Synopsis

#include `<imsl.h>`

float `*imsl_f_random_gamma` (*int* `n_random`, *float* `a`, …, 0)

The type *double* procedure is `imsl_d_random_gamma`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float a (Input)
The shape parameter of the gamma distribution. This parameter must be positive.

Return Value

If no optional arguments are used, imsl_f_random_gamma returns a pointer to a vector of length n_random containing the random standard gamma deviates. To release this space, use free.

Synopsis with Optional Arguments

```
#include <imsl.h>
float *imsl_f_random_gamma (int n_random, float a,
        IMSL_RETURN_USER, float r[],
        0)
```

Optional Arguments

IMSL_USER_RETURN, *float* r[] (Output)
If specified, the vector of length n_random containing the random standard gamma deviates is returned in the user-provided array r.

Description

The function imsl_f_random_gamma generates pseudorandom numbers from a gamma distribution with shape parameter *a* and unit scale parameter. The probability density function is

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \quad \text{for } x \geq 0$$

Various computational algorithms are used depending on the value of the shape parameter *a*. For the special case of $a = 0.5$, squared and halved normal deviates are used; and for the special case of $a = 1.0$, exponential deviates are generated. Otherwise, if *a* is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used. If *a* is greater than 1.0, a ten-region rejection procedure developed by Schmeiser and Lal (1980) is used.

Deviates from the two-parameter gamma distribution with shape parameter *a* and scale parameter *b* can be generated by using imsl_f_random_gamma and then multiplying each entry in r by *b*. The following statements (in single precision) would yield random deviates from a gamma (*a*, *b*) distribution.

```
float *r;
r = imsl_f_random_gamma(n_random, a, 0);
for (i=0; i<n_random; i++) *(r+i) *= b;
```

The Erlang distribution is a standard gamma distribution with the shape parameter having a value equal to a positive integer; hence, imsl_f_random_gamma generates pseudorandom deviates from an Erlang distribution with no modifications required.

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

Example

In this example, `imsl_f_random_gamma` is used to generate five pseudorandom deviates from a gamma (Erlang) distribution with shape parameter equal to 3.0.

```
#include <imsl.h>

void main()
{
    int         seed = 123457;
    int         n_random = 5;
    float       a = 3.0;
    float       *r;

    imsl_random_seed_set(seed);
    r = imsl_f_random_gamma(n_random, a, 0);
    imsl_f_write_matrix("Gamma(3) random deviates", 1, n_random, r, 0);
}
```

Output

```
            Gamma(3) random deviates
    1           2           3           4           5
6.843       3.445       1.853       3.999       0.779
```

random_beta

Generates pseudorandom numbers from a beta distribution.

Synopsis

#include `<imsl.h>`

float `*imsl_f_random_beta` (*float* `n_random`, *float* `pin`, *float* `qin`, …, 0)

The type *double* function is `imsl_d_random_beta`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

If no optional arguments are used, `imsl_f_random_beta` returns a pointer to a vector of length `n_random` containing the random standard beta deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_random_beta` (*float* `n_random`, *float* `pin`, *float* `qin`,
`IMSL_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSL_RETURN_USER`, *float* `r[]` (Output)
If specified, the vector of length `n_random` containing the random standard beta deviates is returned in `r`.

Description

The function `imsl_f_random_beta` generates pseudorandom numbers from a beta distribution with parameters `pin` and `qin`, both of which must be positive. With p = `pin` and q = `qin`, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1}(1-x)^{q-1} \qquad \text{for } 0 \le x \le 1$$

where $\Gamma(\cdot)$ is the gamma function.

The algorithm used depends on the values of p and q. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all of the methods use acceptance/rejection. If p and q are both less than 1, the method of Jöhnk (1964) is used. If either p or q is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both p and q are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used if `n_random` is less than 4; and algorithm B4PE of Schmeiser and Babu (1980) is used if `n_random` is greater than or equal to 4. Note that for p and q both greater than 1, calling `imsl_f_random_beta` in a loop getting less than 4 variates on each call will not yield the same set of deviates as calling `imsl_f_random_beta` once and getting all the deviates at once.

The values returned in `r` are less than 1.0 and greater than ε where ε is the smallest positive number such that $1.0 - \varepsilon$ is less than 1.0.

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

Example

In this example, `imsl_f_random_beta` is used to generate five pseudorandom beta (3, 2) variates.

```
#include <imsl.h>

main()
{

    int         n_random = 5;
    int         seed = 123457;
    float       pin = 3.0;
    float       qin = 2.0;
    float       *r;

    imsl_random_seed_set (seed);
    r = imsl_f_random_beta (n_random, pin, qin, 0);
    imsl_f_write_matrix("Beta (3,2) random deviates", 1, n_random, r, 0);
}
```

Output

```
            Beta (3,2) random deviates
     1           2           3           4           5
0.2814      0.9483      0.3984      0.3103      0.8296
```

random_exponential

Generates pseudorandom numbers from a standard exponential distribution.

Synopsis

#include `<imsl.h>`

float `*imsl_f_random_exponential` (*int* `n_random`, …, 0)

The type *double* function is `imsl_d_random_exponential`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

A pointer to an array of length `n_random` containing the random standard exponential deviates.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_random_exponential` (*int* `n_random`,
`IMSL_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

IMSL_RETURN_USER, *float* r[] (Output)

If specified, the array of length n_random containing the random standard exponential deviates is returned in the user-provided array r.

Description

Function imsl_f_random_exponential generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. Function imsl_random_exponential uses an antithetic inverse CDF technique; that is, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at $1.0 - U$ to yield the exponential deviate.

Deviates from the exponential distribution with mean θ can be generated by using imsl_f_random_exponential and then multiplying each entry in r by θ.

Example

In this example, imsl_f_random_exponential is used to generate five pseudorandom deviates from a standard exponential distribution.

```
#include <imsl.h>

#define N_RANDOM    5

main()

{
    int             seed = 123457;
    int             n_random = N_RANDOM;
    float           *r;

    imsl_random_seed_set(seed);
    r = imsl_f_random_exponential(n_random, 0);
    printf("%s: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
           "Exponential random deviates",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Exponential random deviates:   0.0344  1.3443  0.2662  0.5633  0.1686
```

Chapter 11: Printing Functions

Routines

write_matrix

Prints a rectangular matrix (or vector) stored in contiguous memory locations.

Synopsis

#include `<imsl.h>`

void `imsl_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`, ..., 0)

For *int* a[], use `imsl_i_write_matrix`.

For *double* `a[]`, use `imsl_d_write_matrix`.

For *f_complex* `a[]`, use `imsl_c_write_matrix`.

For *d_complex* `a[]`, use `imsl_z_write_matrix`.

Required Arguments

char `*title` (Input)
The matrix title. Use `\n` within a title to create a new line. Long titles are automatically wrapped.

int `nra` (Input)
The number of rows in the matrix.

int `nca` (Input)
The number of columns in the matrix.

float `a[]` (Input)
Array of size `nra` × `nca` containing the matrix to be printed.

Synopsis with Optional Arguments

#include <imsl.h>

void imsl_f_write_matrix (*char* *title, *int* nra, *int* nca, *float* a[],
IMSL_TRANSPOSE,
IMSL_A_COL_DIM, *int* a_col_dim,
IMSL_PRINT_ALL,
IMSL_PRINT_LOWER,
IMSL_PRINT_UPPER,
IMSL_PRINT_LOWER_NO_DIAG,
IMSL_PRINT_UPPER_NO_DIAG,
IMSL_WRITE_FORMAT, *char* *fmt,
IMSL_ROW_LABELS, *char* *rlabel[],
IMSL_NO_ROW_LABELS,
IMSL_ROW_NUMBER,
IMSL_ROW_NUMBER_ZERO,
IMSL_COL_LABELS, *char* *clabel[],
IMSL_NO_COL_LABELS,
IMSL_COL_NUMBER,
IMSL_COL_NUMBER_ZERO,
IMSL_RETURN_STRING, *char* **string,
IMSL_WRITE_TO_CONSOLE,
0)

Optional Arguments

IMSL_TRANSPOSE
Print a^T.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of *a*.
Default: a_col_dim = nca

IMSL_PRINT_ALL, *or*
IMSL_PRINT_LOWER, *or*
IMSL_PRINT_UPPER, *or*
IMSL_PRINT_LOWER_NO_DIAG, *or*
IMSL_PRINT_UPPER_NO_DIAG
Exactly one of these optional arguments can be specified in order to indicate that either a triangular part of the matrix or the entire matrix is to be printed. If omitted, the entire matrix is printed.

Keyword	Action
IMSL_PRINT_ALL	The entire matrix is printed (the default).
IMSL_PRINT_LOWER	The lower triangle of the matrix is printed, including the diagonal.
IMSL_PRINT_UPPER	The upper triangle of the matrix is printed, including the diagonal.
IMSL_PRINT_LOWER_NO_DIAG	The lower triangle of the matrix is printed, without the diagonal.
IMSL_PRINT_UPPER_NO_DIAG	The upper triangle of the matrix is printed, without the diagonal.

IMSL_WRITE_FORMAT, *char* *fmt (Input)

Character string containing a list of C conversion specifications (formats) to be used when printing the matrix. Any list of C conversion specifications suitable for the data type may be given. For example, fmt = "%10.3f" specifies the conversion character f for the entire matrix. (For the conversion character f, the matrix must be of type *float*, *double*, *f_complex*, or *d_complex*). Alternatively, fmt = "%10.3e%10.3e%10.3f%10.3f%10.3f" specifies the conversion character e for columns 1 and 2 and the conversion character f for columns 3, 4, and 5. (For *complex* matrices, two conversion specifications are required for each column of the matrix so the conversion character e is used in column 1. The conversion character f is used in column 2 and the real part of column 3.) If the end of fmt is encountered and if some columns of the matrix remain, format control continues with the first conversion specification in fmt.

Aside from restarting the format from the beginning, other exceptions to the usual C formatting rules are as follows:

1. Characters not associated with a conversion specification are not allowed. For example, in the format fmt = "1%d2%d", the characters 1 and 2 are not allowed and result in an error.
2. A conversion character d can be used for floating-point values (matrices of type *float*, *double*, *f_complex*, or *d_complex*). The integer part of the floating-point value is printed.
3. For printing numbers whose magnitudes are unknown, the conversion character g is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The w (or W) conversion character is a special conversion character used by this function to select a conversion specification so that the decimal points will be aligned. The conversion specification ending with w is specified as "%n.dw". Here, n is the field width and d is the number of significant digits generally printed. Valid values for n are 3, 4, …, 40. Valid values for d are 1, 2, …, n-2. If fmt specifies one conversion specification ending with w, all elements of a are examined to determine one conversion specification for printing.

If `fmt` specifies more than one conversion specification, separate conversion specifications are generated for each conversion specification ending with `w`. Set `fmt` = "`10.4w`" if you want a single conversion specification selected automatically with field width 10 and with four significant digits.

`IMSL_NO_ROW_LABELS`, *or*
`IMSL_ROW_NUMBER`, *or*
`IMSL_ROW_NUMBER_ZERO`, *or*
`IMSL_ROW_LABELS`, *char* `*rlabel[]` (Input)

If `IMSL_ROW_LABELS` is specified, `rlabel` is a vector of length `nra` containing pointers to the character strings comprising the row labels. Here, `nra` is the number of rows in the printed matrix. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no row labels are desired, use the `IMSL_NO_ROW_LABELS` optional argument. If the numbers 1, 2, …, `nra` are desired, use the `IMSL_ROW_NUMBER` optional argument. If the numbers 1, 2, …, `nra` – 1 are desired, use the `IMSL_ROW_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nra` are used for the row labels by default whenever `nra` > 1. If `nra` = 1, the default is no row labels.

`IMSL_NO_COL_LABELS`, *or*
`IMSL_COL_NUMBER`, *or*
`IMSL_COL_NUMBER_ZERO`, *or*
`IMSL_COL_LABELS`, *char* `*clabel[]` (Input)

If `IMSL_COL_LABELS` is specified, `clabel` is a vector of length `nca` + 1 containing pointers to the character strings comprising the column headings. The heading for the row labels is `clabel[0]`, and `clabel[i]`, `i` = 1, …, `nca`, is the heading for the `i`-th column. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no column labels are desired, use the `IMSL_NO_COL_LABELS` optional argument. If the numbers 1, 2, …, `nca`, are desired, use the `IMSL_COL_NUMBER` optional argument. If the numbers 0, 1, …, `nca` – 1 are desired, use the `IMSL_COL_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nca` are used for the column labels by default whenever `nca` > 1. If `nca` = 1, the default is no column labels.

`IMSL_RETURN_STRING`, *char* `**string` (Output)

The address of a pointer to a NULL-terminated string containing the matrix to be printed. Lines are new-line separated and the last line does not have a trailing new-line character. Typically *char* `*string` is declared, and `&string` is used as the argument.

`IMSL_WRITE_TO_CONSOLE`

This matrix is printed to a console window. If a console has not been allocated, a default console (80 × 24, white on black, no scrollbars) is created.

Description

The function `imsl_write_matrix` prints a real rectangular matrix (stored in *a*) with optional row and column labels (specified by `rlabel` and `clabel`, respectively, regardless of whether *a* or a^T is printed). An optional format, `fmt`, may be used to specify a conversion specification for each column of the matrix.

In addition, the write matrix functions can restrict printing to the elements of the upper or lower triangles of a matrix via the `IMSL_TRIANGLE` option. Generally, the `IMSL_TRIANGLE` option is used with symmetric matrices, but this is not required. Vectors can be printed by specifying a row or column dimension of 1.

Output is written to the file specified by the function **`imsl_output_file`**, Chapter 12, "Utilities." The default output file is standard output (corresponding to the file pointer `stdout`).

A page width of 78 characters is used. Page width and page length can be reset by invoking function **`imsl_page`** (page 583).

Horizontal centering, the method for printing large matrices, paging, the method for printing NaN (Not a Number), and whether or not a title is printed on each page can be selected by invoking function **`imsl_write_options`** (page 584).

Examples

Example 1

This example is representative of the most common situation in which no optional arguments are given.

```
#include <imsl.h>

#define NRA     3
#define NCA     4

main()
{
    int         i, j;
    f_complex   a[NRA][NCA];

    for (i = 0;  i < NRA;  i++) {
        for (j = 0;  j < NCA;  j++) {
            a[i][j].re = (i+1+(j+1)*0.1);
            a[i][j].im = -a[i][j].re+100;
        }
    }
                                /* Write matrix */
    imsl_c_write_matrix ("matrix\na", NRA, NCA, (f_complex *)a, 0);
}
```

Output

```
                                matrix
                                   a
                          1                         2                         3
1 (       1.1,      98.9)  (       1.2,      98.8)  (       1.3,      98.7)
2 (       2.1,      97.9)  (       2.2,      97.8)  (       2.3,      97.7)
3 (       3.1,      96.9)  (       3.2,      96.8)  (       3.3,      96.7)

                          4
1 (       1.4,      98.6)
2 (       2.4,      97.6)
3 (       3.4,      96.6)
```

Example 2

In this example, some of the optional arguments available in the `write_matrix` functions are demonstrated.

```
#include <imsl.h>

#define NRA     3
#define NCA     4

main()
{
    int         i, j;
    float       a[NRA][NCA];
    char        *fmt = "%10.6W";
    char        *rlabel[] = {"row 1", "row 2", "row 3"};
    char        *clabel[] = { "", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0;  i < NRA;  i++) {
        for (j = 0;  j < NCA;  j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }
    }
                                /* Write matrix */
    imsl_f_write_matrix ("matrix\na", NRA, NCA, (float *)a,
                         IMSL_WRITE_FORMAT, fmt,
                         IMSL_ROW_LABELS, rlabel,
                         IMSL_COL_LABELS, clabel,
                         IMSL_PRINT_UPPER_NO_DIAG,
                         0);
}
```

Output

```
                    matrix
                       a
             col 2       col 3       col 4
row 1          1.2         1.3         1.4
row 2                      2.3         2.4
row 3                                  3.4
```

Example 3

In this example, a row vector of length four is printed.

```
#include <imsl.h>

#define NRA     1
#define NCA     4

main()
{
    int         i;
    float       a[NCA];
    char        *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0;  i < NCA;  i++) {
         a[i] = i + 1;
    }
                                /* Write matrix */
    imsl_f_write_matrix ("matrix\na", NRA, NCA, a,
                         IMSL_COL_LABELS, clabel,
                         0);
}
```

Output

```
                matrix
                   a
     col 1       col 2       col 3       col 4
         1           2           3           4
```

page

Sets or retrieves the page width or length.

Synopsis

#include `<imsl.h>`

void `imsl_page` (*Imsl_page_options* `option`, *int* `*page_attribute`)

Required Arguments

Imsl_page_options `option` (Input)

Option giving which page attribute is to be set or retrieved. The possible values are:

`option`	Description
`IMSL_SET_PAGE_WIDTH`	Set the page width.
`IMSL_GET_PAGE_WIDTH`	Retrieve the page width.
`IMSL_SET_PAGE_LENGTH`	Set the page length.
`IMSL_GET_PAGE_LENGTH`	Retrieve the page length.

int `*page_attribute` (Input, if the attribute is set; Output, otherwise)

The value of the page attribute to be set or retrieved. The page width is the number of characters per line of output (default 78), and the page length is the number of lines of output per page (default 60). Ten or more characters per line and 10 or more lines per page are required.

Example

The following example illustrates the use of `imsl_page` to set the page width to 40 characters. The IMSL function `imsl_f_write_matrix` is then used to print a 3×4 matrix A, where $a_{ij} = i + j/10$.

```
#include <imsl.h>

#define NRA     3
#define NCA     4

main()
{
    int         i, j, page_attribute;
    float       a[NRA][NCA];

    for (i = 0;  i < NRA;  i++) {
         for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
    page_attribute = 40;
    imsl_page(IMSL_SET_PAGE_WIDTH, &page_attribute);
    imsl_f_write_matrix("a", NRA, NCA, (float *)a, 0);
}
```

Output

```
                a
              1           2           3
1           1.1         1.2         1.3
2           2.1         2.2         2.3
3           3.1         3.2         3.3

              4
1           1.4
2           2.4
3           3.4
```

write_options

Sets or retrieves an option for printing a matrix.

Synopsis

#include `<imsl.h>`

void `imsl_write_options` (*Imsl_write_options* `option`, *int** `option_value`)

Required Arguments

Imsl_write_options `option` (Input)
Option giving the type of the printing attribute to set or retrieve.

`option` for Setting	`option` for Retrieving	Attribute Description
`IMSL_SET_DEFAULTS`		Use the default settings for all parameters
`IMSL_SET_CENTERING`	`IMSL_GET_CENTERING`	Horizontal centering
`IMSL_SET_ROW_WRAP`	`IMSL_GET_ROW_WRAP`	Row wrapping
`IMSL_SET_PAGING`	`IMSL_GET_PAGING`	Paging
`IMSL_SET_NAN_CHAR`	`IMSL_GET_NAN_CHAR`	Method for printing NaN (not a number)
`IMSL_SET_TITLE_PAGE`	`IMSL_GET_TITLE_PAGE`	Whether or not titles appear on each page
`IMSL_SET_FORMAT`	`IMSL_GET_FORMAT`	Default format for real and complex numbers

int `*option_value` (Input, if `option` is to be set; Output, otherwise)
The value of the option attribute selected by `option`. The values to be used when setting attributes are described in a table in the description section.

Description

The function `imsl_write_options` allows the user to set or retrieve an option for printing a matrix. Options controlled by `imsl_write_options` are horizontal centering, method for printing large matrices, paging, method for printing NaN (not a number), method for printing titles, and the default format for real and complex numbers. (NaN can be retrieved by functions **`imsl_f_machine`** and **`imsl_d_machine`**, Chapter 12, "Utilities.")

The values that may be used for the attributes are as follows:

Option	Value	Meaning
CENTERING	0	Matrix is left justified.
	1	Matrix is centered.
ROW_WRAP	0	A complete row is printed before the next row is printed. Wrapping is used if necessary.
	m	Here *m* is a positive integer. Let n_1 be the maximum number of columns that fit across the page, as determined by the widths in the conversion specifications starting with column 1. First, columns 1 through n_1 are printed for rows 1 through *m*. Let n_2 be the maximum number of columns that fit across the page, starting with column $n_1 + 1$. Second, columns n_1+1 through $n_1 + n_2$ are printed for rows 1 through *m*. This continues until the last columns are printed for rows 1 through *m*. Printing continues in this fashion for the next *m* rows, etc.
PAGING	–2	No paging occurs.
	–1	Paging is on. Every invocation of a `imsl_f_write_matrix` function begins on a new page, and paging occurs within each invocation as is needed.
	0	Paging is on. The first invocation of a `imsl_f_write_matrix` function begins on a new page, and subsequent paging occurs as is needed. Paging occurs in the second and all subsequent calls to a `imsl_f_write_matrix` function only as needed.
	k	Turn paging on and set the number of lines printed on the current page to *k* lines. If *k* is greater than or equal to the page length, then the first invocation of a `imsl_f_write_matrix` function begins on a new page. In any case, subsequent paging occurs as is needed.
NAN_CHAR	0	. . . . . is printed for NaN.
	1	A blank field is printed for NaN.
TITLE_PAGE	0	Title appears only on first page.
	1	Title appears on the first page and all continuation pages.
FORMAT	0	Format is "`%10.4x`".
	1	Format is "`%12.6w`".
	2	Format is "`%22.5e`".

The w conversion character used by the FORMAT option is a special conversion character that can be used to automatically select a pretty C conversion specification ending in either e, f, or d. The conversion specification ending with w is specified as "%n.dw". Here, n is the field width, and d is the number of significant digits generally printed.

The function imsl_write_options can be invoked repeatedly before using a write_matrix function to print a matrix. The matrix printing functions retrieve the values set by imsl_write_options to determine the printing options. It is not necessary to call imsl_write_options if a default value of a printing option is desired. The defaults are as follows:

Option	Default Value	
CENTERING	0	Left justified
ROW_WRAP	1000	Lines before wrapping
PAGING	–2	No paging
NAN_CHAR	0	. . . . .
TITLE_PAGE	0	Title appears only on the first page
FORMAT	0	%10.4w

Example

The following example illustrates the effect of imsl_write_options when printing a 3×4 real matrix A with IMSL function imsl_f_write_matrix, where $a_{ij} = i + j/10$. The first call to imsl_write_options sets horizontal centering so that the matrix is printed centered horizontally on the page. In the next invocation of imsl_f_write_matrix, the left-justification option has been set via function imsl_write_options, so the matrix is left justified when printed.

```
#include <imsl.h>

#define NRA     4
#define NCA     3

main()
{
    int         i, j, option_value;
    float       a[NRA][NCA];

    for (i = 0;  i < NRA;  i++) {
        for (j = 0;  j < NCA;  j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
                                /* Activate centering option */
    option_value = 1;
    imsl_write_options (IMSL_SET_CENTERING, &option_value);
                                /* Write a matrix */
```

```
    imsl_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
                              /* Activate left justification */
    option_value = 0;
    imsl_write_options (IMSL_SET_CENTERING, &option_value);
    imsl_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
}
```

Output

```
                                    a
                                  1           2           3
                      1         1.1         1.2         1.3
                      2         2.1         2.2         2.3
                      3         3.1         3.2         3.3
                      4         4.1         4.2         4.3

                    a
              1           2           3
1         1.1         1.2         1.3
2         2.1         2.2         2.3
3         3.1         3.2         3.3
4         4.1         4.2         4.3
```

Chapter 12: Utilities

Routines

output_file

Sets the output file or the error message output file.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `imsl_output_file` (
`IMSL_SET_OUTPUT_FILE`, *FILE* `*ofile`,
`IMSL_GET_OUTPUT_FILE`, *FILE* `**pofile`,
`IMSL_SET_ERROR_FILE`, *FILE* `*efile`,
`IMSL_GET_ERROR_FILE`, *FILE* `**pefile`,
0)

Optional Arguments

`IMSL_SET_OUTPUT_FILE`, *FILE* `*ofile` (Input)
Set the output file to `ofile`.
Default: `ofile` = `stdout`

`IMSL_GET_OUTPUT_FILE`, *FILE* `**pfile` (Output)
Set the *FILE* pointed to by `pfile` to the current output file.

`IMSL_SET_ERROR_FILE`, *FILE* `*efile` (Input)
Set the error message output file to `efile`.
Default: `efile` = `stderr`

`IMSL_GET_ERROR_FILE`, *FILE* `**pefile` (Output)
Set the *FILE* pointed to by `pefile` to the error message output file.

Description

This function allows the file used for printing by IMSL routines to be changed.

Example

This example opens the file `myfile` and changes the output file to this new file. The function `imsl_f_write_matrix` then writes to this file.

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

main()
{
    FILE        *ofile;
    float       x[] = {3.0, 2.0, 1.0};

    imsl_f_write_matrix ("x (default file)", 1, 3, x, 0);

    ofile = fopen("myfile", "w");
    imsl_output_file(IMSL_SET_OUTPUT_FILE, ofile,
                     0);
    imsl_f_write_matrix ("x (myfile)", 1, 3, x, 0);
}
```

Output

```
x (default file)
1           2           3
3           2           1
```

File myfile

```
x (myfile)
1           2           3
3           2           1
```

version

Returns integer information describing the version of the library, serial number, operating system, and compiler.

Synopsis

#include `<imsl.h>`

char`*` `imsl_version` (*Imsl_keyword* `code`)

Required Arguments

Imsl_keyword `code` (Input)
Index indicating which value is to be returned. It must be `IMSL_LIBRARY_VERSION`, `IMSL_OS_VERSION`, `IMSL_COMPILER_VERSION`, or `IMSL_LICENSE_NUMBER`.

Return Value

The requested value is returned. If `code` is out of range, then `NULL` is returned. Use `free` to release the returned string.

Description

The function `imsl_version` returns information describing the version of this library, the version of the operating system under which it was compiled, the compiler used, and the IMSL number.

Example

This example prints all the values returned by `imsl_version` on a particular machine. The output is omitted because the results are system dependent.

```
#include <imsl.h>

main()
{
    char        *library_version, *os_version;
    char        *compiler_version, *license_number;

    library_version  = imsl_version(IMSL_LIBRARY_VERSION);
    os_version       = imsl_version(IMSL_OS_VERSION);
    compiler_version = imsl_version(IMSL_COMPILER_VERSION);
    license_number   = imsl_version(IMSL_LICENSE_NUMBER);

    printf("Library version = %s\n", library_version);
    printf("OS version = %s\n", os_version);
    printf("Compiler version = %s\n", compiler_version);
    printf("Serial number = %s\n", license_number);
}
```

ctime

Returns the number of CPU seconds used.

Synopsis

#include `<imsl.h>`

double `imsl_ctime ()`

Return Value

The number of CPU seconds used so far by the program.

Example

The CPU time needed to compute

$$\sum_{k=0}^{1,000,000} k$$

is obtained and printed. The time needed is, of course, machine dependent. The CPU time needed will also vary slightly from run to run on the same machine.

```
#include <imsl.h>

main()
{
    int     k;
    double  sum, time;
                                /* Sum 1 million values */
    for (sum=0, k=1;  k<=1000000; k++)
         sum += k;
                                /* Get amount of CPU time used */
    time = imsl_ctime();
    printf("sum = %f\n", sum);
    printf("time = %f\n", time);
}
```

Output

```
sum = 500000500000.000000
time = 2.260000
```

date_to_days

Computes the number of days from January 1, 1900, to the given date.

Synopsis

#include `<imsl.h>`

int `imsl_date_to_days` (*int* `day`, *int* `month`, *int* `year`)

Required Arguments

int `day` (Input)
Day of the input date.

int `month` (Input)
Month of the input date.

int `year` (Input)
Year of the input date. The year 1950 would correspond to the year 1950 A.D., and the year 50 would correspond to year 50 A.D.

Return Value

Number of days from January 1, 1900, to the given date. If negative, it indicates the number of days prior to January 1, 1900.

Description

The function `imsl_date_to_days` returns the number of days from January 1, 1900, to the given date. The function `imsl_date_to_days` returns negative values for days prior to January 1, 1900. A negative `year` can be used to specify B.C. Input dates in

year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, `imsl_date_to_days` issues a terminal error.

The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use.

Example

The following example uses `imsl_date_to_days` to compute the number of days from January 15, 1986, to February 28, 1986.

```
#include <imsl.h>

main()
{
    int         day0, day1;

    day0 = imsl_date_to_days(15, 1, 1986);
    day1 = imsl_date_to_days(28, 2, 1986);
    printf("Number of days = %d\n", day1 - day0);
}
```

Output

```
Number of days = 44
```

days_to_date

Gives the date corresponding to the number of days since January 1, 1900.

Synopsis

#include `<imsl.h>`

void `imsl_days_to_date` (*int* `days`, *int* `*day`, *int* `*month`, *int* `*year`)

Required Arguments

int `days` (Input)
Number of days since January 1, 1900.

int `*day` (Output)
Day of the output date.

int `*month` (Output)
Month of the output date.

int `*year` (Output)
Year of the output date. The year 1950 would correspond to the year 1950 A.D., and the year 50 would correspond to year 50 A.D.

Description

The function `imsl_days_to_date` computes the date corresponding to the number of days since January 1, 1900. For a negative input value of `days`, the date computed is prior to January 1, 1900. This function is the inverse of function **`imsl_date_to_days`** (see page 593).

The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use.

Example

The following example uses `imsl_days_to_date` to compute the date for the 100th day of 1986. This is accomplished by first using IMSL function **`imsl_date_to_days`** (see page 593) to get the "day number" for December 31, 1985.

```
#include <imsl.h>

main()
{
    int         day0, day, month, year;

    day0 = imsl_date_to_days(31, 12, 1985);
    imsl_days_to_date(day0+100, &day, &month, &year);
    printf("Day 100 of 1986 is (day-month-year) %d-%d-%d\n",
            day, month, year);
}
```

Output

```
Day 100 of 1986 is (day-month-year) 10-4-1986
```

error_options

Sets various error handling options.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `imsl_error_options(`
`IMSL_SET_PRINT,` *Imsl_error* `type,` *int* `setting,`
`IMSL_SET_STOP,` *Imsl_error* `type,` *int* `setting,`
`IMSL_SET_TRACEBACK,` *Imsl_error* `type,` *int* `setting,`
`IMSL_FULL_TRACEBACK,` *int* `setting,`
`IMSL_GET_PRINT,` *Imsl_error* `type,` *int* `*psetting,`
`IMSL_GET_STOP,` *Imsl_error* `type,` *int* `*psetting,`
`IMSL_GET_TRACEBACK,` *Imsl_error* `type,` *int* `*psetting,`
`IMSL_SET_ERROR_FILE,` *FILE* `*file,`
`IMSL_GET_ERROR_FILE,` *FILE* `**pfile,`
`IMSL_ERROR_MSG_PATH,` *char* `*path,`
`IMSL_ERROR_MSG_NAME,` *char* `*name,`
`IMSL_ERROR_PRINT_PROC,` *Imsl_error_print_proc* `print_proc,`
0)

Optional Arguments

`IMSL_SET_PRINT`, *Imsl_error* `type`, *int* `setting` (Output)
Printing of type `type` error messages is turned off if `setting` is 0; otherwise, printing is turned on.
Default: Printing turned on for `IMSL_WARNING`, `IMSL_FATAL`, `IMSL_TERMINAL`, `IMSL_FATAL_IMMEDIATE`, and `IMSL_WARNING_IMMEDIATE` messages

`IMSL_SET_STOP`, *Imsl_error* `type`, *int* `setting` (Input)
Stopping on type `type` error messages is turned off if `setting` is 0; otherwise, stopping is turned on.
Default: Stopping turned on for `IMSL_FATAL`, `IMSL_TERMINAL`, and `IMSL_FATAL_IMMEDIATE` messages

`IMSL_SET_TRACEBACK`, *Imsl_error* `type`, *int* `setting` (Input)
Printing of a traceback on type `type` error messages is turned off if `setting` is 0; otherwise, printing of the traceback turned on.
Default: Traceback turned off for all message types

`IMSL_FULL_TRACEBACK`, *int* `setting` (Input)
Only documented functions are listed in the traceback if `setting` is 0; otherwise, internal function names also are listed.
Default: Full traceback turned off

`IMSL_GET_PRINT`, *Imsl_error* `type`, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for printing of type `type` error messages.

`IMSL_GET_STOP`, *Imsl_error* `type`, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for stopping on type `type` error messages.

`IMSL_GET_TRACEBACK`, *Imsl_error* type, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for printing of a traceback for type `type` error messages.

`IMSL_SET_ERROR_FILE`, *FILE* `*file` (Input)
Sets the error output file.
Default: `file` = `stderr`

`IMSL_GET_ERROR_FILE`, *FILE* `**pfile` (Output)
Sets the *FILE* `*` pointed to by `pfile` to the error output file.

`IMSL_ERROR_MSG_PATH`, *char* `*path` (Input)
Sets the error message file path. On UNIX systems, this is a colon-separated list of directories to be searched for the file containing the error messages.
Default: system dependent

`IMSL_ERROR_MSG_NAME`, *char* `*name` (Input)
Sets the name of the file containing the error messages.
Default: `file` = "`imslerror.bin`"

`IMSL_ERROR_PRINT_PROC`, *Imsl_error_print_proc* `print_proc` (Input)
Sets the error printing function. The procedure `print_proc` has the form *void* `print_proc` (*Imsl_error* `type`, *long* `code`, *char* `*function_name`, *char* `*message`).

In this case, `type` is the error message type number (`IMSL_FATAL`, etc.), code is the error message code number (`IMSL_MAJOR_VIOLATION`, etc.), `function_name` is the name of the function setting the error, and message is the error message to be printed. If `print_proc` is `NULL`, then the default error printing function is used.

Return Value

The return value for this function is void.

Description

This function allows the error handling system to be customized.

Examples

Example 1

In this example, the `IMSL_TERMINAL` print setting is retrieved. Next, stopping on `IMSL_TERMINAL` errors is turned off, output to standard output is redirected, and an error is deliberately caused by calling `imsl_error_options` with an illegal value.

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

main()
{
    int         setting;
                              /* Turn off stopping on IMSL_TERMINAL */
                              /* error messages and write error */
                              /* messages to standard output */
    imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0,
                       IMSL_SET_ERROR_FILE, stdout,
                       0);
                              /* Call imsl_error_options() with */
                              /* an illegal value */
    imsl_error_options(-1);
                              /* Get setting for IMSL_TERMINAL */
    imsl_error_options(IMSL_GET_PRINT, IMSL_TERMINAL, &setting,
                       0);
    printf("IMSL_TERMINAL error print setting = %d\n", setting);
}
```

Output

```
*** TERMINAL Error from imsl_error_options.  There is an error with
*** argument number 1.  This may be caused by an incorrect number of
*** values following a previous optional argument name.

IMSL_TERMINAL error print setting = 1
```

Example 2

In this example, IMSL's own error printing function has been substituted for the standard function. Only the first four lines are printed below.

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

void         print_proc(Imsl_error, long, char*, char*);

main()
{
                            /* Turn off tracebacks on IMSL_TERMINAL */
                            /* error messages and use a custom */
                            /* print function */
    imsl_error_options(IMSL_ERROR_PRINT_PROC, print_proc,
                       0);
                            /* Call imsl_error_options() with an */
                            /* illegal value */
    imsl_error_options(-1);
}

void print_proc(Imsl_error type, long code, char *function_name,
                char *message)
{
    printf("Error message type %d\n", type);
    printf("Error code %d\n", code);
    printf("From function %s\n", function_name);
    printf("%s\n", message);
}
```

Output

```
Error message type 5
Error code 103
From function imsl_error_options
There is an error with argument number 1.  This may be caused by an
incorrect number of values following a previous optional argument name.
```

error_code

Gets the code corresponding to the error message from the last function called.

Synopsis

#include `<imsl.h>`

long `imsl_error_code ()`

Return Value

This function returns the error message code from the last IMSL function called. The include file `imsl.h` defines a name for each error code.

Example

This example turns off stopping on IMSL_TERMINAL error messages and generates an error by calling imsl_error_options with an illegal value for IMSL_SET_PRINT. The error message code number is retrieved and printed. In imsl.h, IMSL_INTEGER_OUT_OF_RANGE is defined to be 132.

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

main()
{
    long       code;
                                /* Turn off stopping IMSL_TERMINAL */
                                /* messages and print error messages */
                                /* on standard output. */
    imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0,
                       IMSL_SET_ERROR_FILE, stdout,
                       0);
                                /* Call imsl_error_options() with */
                                /* an illegal value */
    imsl_error_options(IMSL_SET_PRINT, 100, 0,
                       0);
                                /* Get the error message code */
    code = imsl_error_code();
    printf("error code = %d\n", code);
}
```

Output

```
*** TERMINAL Error from imsl_error_options."type" must be between 1 and 5,
***          but "type" = 100.

error code = 132
```

constant

Returns the value of various mathematical and physical constants.

Synopsis

#include <imsl.h>

float imsl_f_constant (*char* name, *char* unit)

The type *double* function is imsl_d_constant.

Required Arguments

char *name (Input)

Character string containing the name of the desired constant. The case of the character string name does not matter. The names “PI”, “Pi”, “pI”, and “pi” are equivalent. Spaces and underscores are allowed and ignored.

char `*unit` (Input)

Character string containing the units of the desired constant. If `NULL`, then Système International d'Unités (SI) units are assumed. The case of the character string `unit` does not matter. The names "`METER`", "`Meter`" and "`meter`" are equivalent. `unit` has the form `U1*U2*... *Um/V1/.../Vn`, where `Ui` and `Vi` are the names of basic units or are the names of basic units raised to a power. Basic units must be separated by `*` or `/`. Powers are indicated by `^`, as in "`m^2`" for m^2. Examples are, "`METER*KILOGRAM/SECOND`", "`M*KG/S`", "`METER`", or "`M/KG^2`".

Return Value

By default, `imsl_f_constant` returns the desired constant. If no value can be computed, NaN is returned.

Description

The names allowed are listed in the following table. Values marked with a ‡ are exact (to machine precision). The references in the right-hand column are indicated by the code numbers: [1] for Cohen and Taylor (1986), [2] for Liepman (1964), and [3] for precomputed mathematical constants.

Name	Description	Value	Reference
amu	Atomic mass unit	$1.6605655 \times 10^{-27}$ kg	1
ATM	Standard atm pressure	1.01325×10^{5} N/m^2 ‡	2
AU	Astronomical unit	1.496×10^{11} m	
Avogadro	Avogadro's number, N	6.022045×10^{23} 1/mole	1
Boltzman	Boltzman's constant, k	1.380662×10^{-23} J/K	1
C	Speed of light, c	$2.997924580 \times 10^{8}$ m/s	1
Catalan	Catalan's constant	0.915965… ‡	3
E	Base of natural logs, e	2.718… ‡	3
ElectronCharge	Electron charge, e	$1.6021892 \times 10^{-19}$ C	1
ElectronMass	Electron mass, m_e	9.109534×10^{-31} kg	1
ElectronVolt	ElectronVolt, ev	1.6021892 x10 $^{-19}$J	1
Euler	Euler's constant, γ	0.577… ‡	3
Faraday	Faraday constant, F	9.648456×10^{4} C/mole	1
FineStructure	Fine structure, α	7.2973506×10^{-3}	1
Gamma	Euler's constant, γ	0.577… ‡	3
Gas	Gas constant, R_0	8.31441 J/mole/K	1

Name	Description	Value	Reference
Gravity	Gravitational constant, *G*	6.6720×10^{-11} N m^2/kg^2	1
Hbar	Planck's constant/2π	$1.0545887 \times 10^{-34}$ J s	1
PerfectGasVolume	Std vol ideal gas	2.241383×10^{-2} m^3/mole	1
Pi	Pi, π	3.141… ‡	3
Planck	Planck's constant, *h*	6.626176×10^{-34} J s	1
ProtonMass	Proton mass, M_p	$1.6726485 \times 10^{-27}$ kg	1
Rydberg	Rydberg's constant, R_∞	$1.097373177 \times 10^{7}$/m	1
Speedlight	Speed of light, *c*	$2.997924580 \times 10^{8}$ m/s	1
StandardGravity	Standard *g*	9.80665 m/s^2 ‡	2
StandardPressure	Standard atm pressure	1.01325×10^{5} N/m^2 ‡	2
StefanBoltzman	Stefan-Boltzman, σ	5.67032×10^{-8} $W/K^4/m^2$	1
WaterTriple	Triple point of water	2.7316×10^{2} K	2

The units allowed are as follows:

Unit	Description
Time	day, hour = hr, min, minute, s = sec = second, year
Frequency	Hertz = Hz
Mass	AMU, g = gram, lb = pound, ounce = oz, slug
Distance	Angstrom, AU, feet = foot, in = inch, m = meter = metre, micron, mile, mill, parsec, yard
Area	acre
Volume	1 = liter=litre
Force	dyne, N = Newton
Energy	BTU, Erg, J = Joule
Work	W = watt
Pressure	ATM = atmosphere, bar
Temperature	degC = Celsius, degF = Fahrenheit, degK = Kelvin
Viscosity	poise, stoke
Charge	Abcoulomb, C = Coulomb, statcoulomb
Current	A = ampere, abampere, statampere
Voltage	Abvolt, V = volt

Unit	Description
Magnetic induction	T = Telsa, Wb = Weber
Other units	I, farad, mole, Gauss, Henry, Maxwell, Ohm

The following metric prefixes may be used with the above units. The one or two letter prefixes may only be used with one letter unit abbreviations.

a	atto	10^{-18}	d	deci	10^{-1}
f	femto	10^{-15}	dk	deca	10^{2}
p	pico	10^{-12}	k	kilo	10^{3}
n	nano	10^{-9}		myria	10^{4}
u	micro	10^{-6}		mega	10^{6}
m	milli	10^{-3}	g	giga	10^{9}
c	centi	10^{-2}	t	tera	10^{12}

There is no one letter unit abbreviation for *myria* or *mega* since *m* means *milli.*

Examples

Example 1

In this example, Euler's constant γ is obtained and printed. Euler's constant is defined to be

$$\gamma = \lim_{n\to\infty}\left[\sum_{k=1}^{n-1}\frac{1}{k} - \ln n\right]$$

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

main()
{
    float       gamma;
                                /* Get gamma */
    gamma = imsl_f_constant("gamma", 0);
                                /* Print gamma */
    printf("gamma = %f\n", gamma);
}
```

Output

```
gamma = 0.577216
```

Example 2

In this example, the speed of light is obtained using several different units.

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

main()
{
    float       speed_light;
                           /* Get speed of light in meters/second */
    speed_light = imsl_f_constant("Speed Light", "meter/second");
    printf("speed of light = %g meter/second\n", speed_light);
                           /* Get speed of light in miles/second */
    speed_light = imsl_f_constant("Speed Light", "mile/second");
    printf("speed of light = %g mile/second\n", speed_light);
                           /* Get speed of light in */
                           /* centimeters/nanosecond */
    speed_light = imsl_f_constant("Speed Light", "cm/ns");
    printf("speed of light = %g cm/ns\n", speed_light);
}
```

Output

```
speed of light = 2.99792e+08 meter/second
speed of light = 186282 mile/second
speed of light = 29.9793 cm/ns
```

Warning Errors

IMSL_MASS_TO_FORCE	A conversion of units of mass to units of force was required for consistency.

machine (integer)

Returns integer information describing the computer's arithmetic.

Synopsis

#include `<imsl.h>`

int `imsl_i_machine` (*int* `n`)

Required Arguments

int `n` (Input)
Index indicating which value is to be returned. It must be between 0 and 12.

Return Value

The requested value is returned. If `n` is out of range, then NaN is returned.

Description

The function `imsl_i_machine` returns information describing the computer's arithmetic. This can be used to make programs machine independent.

`imsl_1_machine`(0) = Number of bits per byte

Assume that integers are represented in M-digit, base-A form as

$$\sigma \sum_{k=0}^{M} x_k A^k$$

where σ is the sign and $0 \le x_k < A$ for $k = 0, \ldots, M$. Then,

n	Definition
0	C, bits per character
1	A, the base
2	M_s, the number of base-A digits in a *short int*
3	$A^{M_s} - 1$, the largest *short int*
4	M_l, the number of base-A digits in a *long int*
5	$A^{M_1} - 1$, the largest *long int*

Assume that floating-point numbers are represented in N-digit, base B form as

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k}$$

where σ is the sign and $0 \le x_k < B$ for $k = 1, \ldots, N$ for and $E_{\min} \le E \le E_{\max}$. Then,

n	Definition
6	B, the base
7	N_f, the number of base-B digits in *float*
8	E_{min_f} , the smallest *float* exponent
9	E_{max_f} , the largest *float* exponent
10	N_d, the number of base-B digits in *double*
11	E_{min_d} , the smallest double exponent
12	E_{max_d} , the largest double exponent

Example

This example prints all the values returned by `imsl_i_machine` on a machine with IEEE (Institute for Electrical and Electronics Engineer) arithmetic.

```
#include <imsl.h>

main()
{
    int         n, ans;

    for (n = 0;  n <= 12;  n++) {
        ans = imsl_i_machine(n);
```

```
        printf("imsl_i_machine(%d) = %d\n", n, ans);
    }
}
```

Output

```
imsl_i_machine(0) = 8
imsl_i_machine(1) = 2
imsl_i_machine(2) = 15
imsl_i_machine(3) = 32767
imsl_i_machine(4) = 31
imsl_i_machine(5) = 2147483647
imsl_i_machine(6) = 2
imsl_i_machine(7) = 24
imsl_i_machine(8) = -125
imsl_i_machine(9) = 128
imsl_i_machine(10) = 53
imsl_i_machine(11) = -1021
imsl_i_machine(12) = 1024
```

machine (float)

Returns information describing the computer's floating-point arithmetic.

Synopsis

#include `<imsl.h>`

float `imsl_f_machine` (*int* `n`)

The type *double* function is `imsl_d_machine`.

Required Arguments

int `n` (Input)
Index indicating which value is to be returned.The index must be between 1 and 8.

Return Value

The requested value is returned. If `n` is out of range, then NaN is returned.

Description

The function `imsl_f_machine` returns information describing the computer's floating-point arithmetic. This can be used to make programs machine independent. In addition, some of the functions are also important in setting missing values (see below).

Assume that *float* numbers are represented in N_f-digit, base B form as

$$\sigma B^E \sum_{k=1}^{N_f} x_k B^{-k}$$

where σ is the sign, $0 \le x_k < B$ for $k = 1, 2, \ldots, N_f$, and

$$E_{min_f} \leq E \leq E_{max_f}$$

Note that B = `imsl_i_machine`(6), N_f = `imsl_i_machine`(7),

$$E_{min_f} = \text{imsl\_i\_machine(8)}$$

and

$$E_{max_f} = \text{imsl\_i\_machine(9)}$$

The ANSI/IEEE Std 754-1985 standard for binary arithmetic uses NaN (not a number) as the result of various otherwise illegal operations, such as computing 0/0. On computers that do not support NaN, a value larger than `imsl_d_machine`(2) is returned for `imsl_f_machine`(6). On computers that do not have a special representation for infinity, `imsl_f_machine`(2) returns the same value as `imsl_f_machine`(7).

The function `imsl_f_machine` is defined by the following table:

n	Definition
1	$B^{E_{min_f}-1}$, the smallest positive number
2	$B^{E_{max_f}}(1-B^{-N_f})$, the largest number
3	B^{-N_f}, the smallest relative spacing
4	B^{1-N_f}, the largest relative spacing
5	$\log_{10}(B)$
6	NaN (not a number)
7	positive machine infinity
8	negative machine infinity

The function `imsl_d_machine` retrieves machine constants which define the computer's double arithmetic. Note that for *double* B = `imsl_i_machine`(6), N_d = `imsl_i_machine`(10),

$$E_{min_f} = \text{imsl\_i\_machine(11)}$$

and

$$E_{max_f} = \text{imsl\_i\_machine(12)}$$

Missing values in IMSL functions are always indicated by NaN (Not a Number). This is `imsl_f_machine`(6) in single precision and `imsl_d_machine`(6) in double. There is no missing-value indicator for integers. Users will almost always have to convert from their missing value indicators to NaN.

Example

This example prints all eight values returned by `imsl_f_machine` and by `imsl_d_machine` on a machine with IEEE arithmetic.

```
#include <imsl.h>

main()
{
    int         n;
    float       fans;
    double      dans;

    for (n = 1;  n <= 8;  n++) {
        fans = imsl_f_machine(n);
        printf("imsl_f_machine(%d) = %g\n", n, fans);
    }

    for (n = 1;  n <= 8;  n++) {
        dans = imsl_d_machine(n);
        printf("imsl_d_machine(%d) = %g\n", n, dans);
    }
}
```

Output

```
imsl_f_machine(1) = 1.17549e-38
imsl_f_machine(2) = 3.40282e+38
imsl_f_machine(3) = 5.96046e-08
imsl_f_machine(4) = 1.19209e-07
imsl_f_machine(5) = 0.30103
imsl_f_machine(6) = NaN
imsl_f_machine(7) = Inf
imsl_f_machine(8) = -Inf
imsl_d_machine(1) = 2.22507e-308
imsl_d_machine(2) = 1.79769e+308
imsl_d_machine(3) = 1.11022e-16
imsl_d_machine(4) = 2.22045e-16
imsl_d_machine(5) = 0.30103
imsl_d_machine(6) = NaN
imsl_d_machine(7) = Inf
imsl_d_machine(8) = -Inf
```

sort

Sorts a vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned.

Synopsis

#include `<imsl.h>`

float `*imsl_f_sort` (*int* `n`, *float* `*x`, …, 0)

The type *double* function is `imsl_d_sort`.

Required Arguments

int `n` (Input)
The length of the input vector.

float `*x` (Input)
Input vector to be sorted.

Return Value

A vector of length `n` containing the values of the input vector `x` sorted into ascending order. If an error occurs, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*imsl_f_sort` (*int* `n`, *float* `*x`,
`IMSL_ABSOLUTE`,
`IMSL_PERMUTATION`, *int* `**perm`,
`IMSL_PERMUTATION_USER`, *int* `perm_user[]`,
`IMSL_RETURN_USER`, *float* `y[]`,
0)

Optional Arguments

`IMSL_ABSOLUTE`
Sort `x` by absolute value.

`IMSL_PERMUTATION`, *int* `**perm` (Output)
Return a pointer to the sort permutation.

`IMSL_PERMUTATION_USER`, *int* `perm_user[]` (Output)
Return the sort permutation in user-supplied space.

`IMSL_RETURN_USER`, *float* `y[]` (Output)
Return the sorted data in user-supplied space.

Description

By default, `imsl_f_sort` sorts the elements of `x` into ascending order by algebraic value. The vector is divided into two parts by choosing a central element `T` of the vector. The first and last elements of `x` are compared with `T` and exchanged until the three values appear in the vector in ascending order. The elements of the vector are rearranged until all elements greater than or equal to the central elements appear in the second part of the vector and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the vector. On completion, $x_j \le x_i$ for $j < i$. If the option `IMSL_ABSOLUTE` is selected, the elements of `x` are sorted into ascending order by absolute value. If we denote the return vector by y, on completion, $|y_j| \le |y_i|$ for $j < i$.

If the option IMSL_PERMUTATION is chosen, a record of the permutations to the array x is returned. That is, after the initialization of $perm_i = i$, the elements of perm are moved in the same manner as are the elements of x.

Examples

Example 1

In this example, an input vector is sorted algebraically.

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

main()
{
    float x[] = {1.0, 3.0, -2.0, 4.0};
    float *sorted_result;
    int         n;

    n = 4;
    sorted_result = imsl_f_sort (n, x, 0);

    imsl_f_write_matrix("Sorted vector", 1, 4, sorted_result, 0);
}
```

Output

```
           Sorted vector
 1           2           3           4
-2           1           3           4
```

Example 2

This example sorts an input vector by absolute value and prints the result stored in user-allocated space.

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

main()
{
    float x[] = {1.0, 3.0, -2.0, 4.0};
    float  sorted_result[4];
    int         n;

    n = 4;
    imsl_f_sort (n, x,
                 IMSL_ABSOLUTE,
                 IMSL_RETURN_USER, sorted_result,
                 0);

    imsl_f_write_matrix("Sorted vector", 1, 4, sorted_result, 0);
}
```

Output

```
        Sorted vector
1           2           3           4
1          -2           3           4
```

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.

Synopsis

#include `<imsl.h>`

int `*imsl_i_sort` (*int* `n`, *int* `*x`, …, 0)

Required Arguments

int `n` (Input)
The length of the input vector.

int `*x` (Input)
Input vector to be sorted.

Return Value

A vector of length `n` containing the values of the input vector `x` sorted into ascending order. If an error occurs, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

int `*imsl_i_sort` (*int*, `n` *int* `*x`,
`IMSL_ABSOLUTE`,
`IMSL_PERMUTATION`, *int* `**perm`,
`IMSL_PERMUTATION_USER`, *int* `perm_user[]`,
`IMSL_RETURN_USER`, *int* `y[]`,
0)

Optional Arguments

`IMSL_ABSOLUTE`
Sort `x` by absolute value.

`IMSL_PERMUTAION`, *int* `**perm` (Output)
Return a pointer to the sort permutation.

`IMSL_PERMUTATION_USER`, *int* `perm_user[]` (Output)
Return the sort permutation in user-supplied space.

`IMSL_RETURN_USER`, *int* `y[]` (Output)
Return the sorted data in user-supplied space.

Description

By default, `imsl_i_sort` sorts the elements of `x` into ascending order by algebraic value. The vector is divided into two parts by choosing a central element `T` of the vector. The first and last elements of `x` are compared with `T` and exchanged until the three values appear in the vector in ascending order. The elements of the vector are rearranged until all elements greater than or equal to the central elements appear in the second part of the vector and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the vector. On completion, $x_j \le x_i$ for $j < i$. If the option `IMSL_ABSOLUTE` is selected, the elements of `x` are sorted into ascending order by absolute value. If we denote the return vector by y, on completion, $|y_j| \le |y_i|$ for $j < i$.

If the option `IMSL_PERMUTATION` is chosen, a record of the permutations to the array `x` is returned. That is, after the initialization of $\texttt{perm}_i = i$, the elements of `perm` are moved in the same manner as are the elements of `x`.

Examples

Example 1

In this example, an input vector is sorted algebraically.

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

main()
{
    int x[] = {1, 3, -2, 4};
    int      *sorted_result;
    int       n;

    n = 4;
    sorted_result = imsl_i_sort (n, x, 0);

    imsl_i_write_matrix("Sorted vector", 1, 4, sorted_result, 0);
}
```

Output

```
 Sorted vector
 1   2   3   4
-2   1   3   4
```

Example 2

This example sorts an input vector by absolute value and prints the result stored in user-allocated space.

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

main()
{
    int x[] = {1, 3, -2, 4};
    int      sorted_result[4];
    int      n;

    n = 4;
    imsl_i_sort (n, x,
                 IMSL_ABSOLUTE,
                 IMSL_RETURN_USER, sorted_result,
                 0);

    imsl_i_write_matrix("Sorted vector", 1, 4, sorted_result, 0);
}
```

Output

```
 Sorted vector
 1   2   3   4
 1  -2   3   4
```

vector_norm

Computes various norms of a vector or the difference of two vectors.

Synopsis

#include `<imsl.h>`

float `imsl_f_vector_norm` (*int* `n`, *float* `*x`, …., 0)

The type *double* function is `imsl_d_vector_norm`.

Required Arguments

int `n` (Input)
The length of the input vector(s).

float `*x` (Input)
Input vector for which the norm is to be computed

Return Value

The requested norm of the input vector. If the norm cannot be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float imsl_f_vector_norm (int n, float *x,
        IMSL_ONE_NORM,
        IMSL_INF_NORM,
        IMSL_SECOND_VECTOR, float *y,
        0)
```

Description

By default, `imsl_f_vector_norm` computes the Euclidean norm

$$\left(\sum_{i=0}^{n-1} x_i^2\right)^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\sum_{i=0}^{n-1} |x_i|$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\max |x_i|$$

is returned. In the case of the infinity norm, the program also returns the index of the element with maximum modulus. If `IMSL_SECOND_VECTOR` is selected, then the norm of $x - y$ is computed.

Examples

Example 1

In this example, the Euclidean norm of an input vector is computed.

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

main()
{
    float x[] = {1.0, 3.0, -2.0, 4.0};
    float norm;
    int        n;

    n = sizeof(x)/sizeof(*x);
    norm = imsl_f_vector_norm (n, x, 0);

    printf("Euclidean norm of x = %f\n", norm);
}
```

Output

```
Euclidean norm of x = 5.477226
```

Example 2

This example computes max $|x_i - y_i|$ and prints the norm and index.

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

main()
{
    float x[] = {1.0, 3.0, -2.0, 4.0};
    float y[] = {4.0, 2.0, -1.0, -5.0};
    float norm;
    int   index;
    int   n;

    n = sizeof(x)/sizeof(*x);
    norm = imsl_f_vector_norm (n, x,
                IMSL_SECOND_VECTOR, y,
                IMSL_INF_NORM, &index, 0);

    printf("Infinity norm of x-y = %f ", norm);
    printf("at location %d\n", index);
}
```

Output

```
Infinity norm of x-y = 9.000000 at location 3
```

mat_mul_rect

Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product.

Synopsis

#include `<imsl.h>`

float `*imsl_f_mat_mul_rect` (*char* `*string`, …, 0)

The type *double* procedure is `imsl_d_mat_mul_rect`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication. This is always a pointer to a *float*, even if the result is a single number. To release this space, use `free`. If no answer was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float *imsl_f_mat_mul_rect (*char* *string,
IMSL_A_MATRIX, *int* nrowa, *int* ncola, *float* a[],
IMSL_A_COL_DIM, *int* a_col_dim,
IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *float* b[],
IMSL_B_COL_DIM, *int* b_col_dim,
IMSL_X_VECTOR, *int* nx, *float* *x,
IMSL_Y_VECTOR, *int* ny, *float* *y,
IMSL_RETURN_USER, *float* ans[],
IMSL_RETURN_COL_DIM, *int* return_col_dim,
0)

Optional Arguments

IMSL_A_MATRIX, *int* nrowa, *int* ncola, *float* a[] (Input)
The nrowa × ncola matrix *A*.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of *A*.
Default: a_col_dim = ncola

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *float* b[] (Input)
The nrowb × ncolb matrix *A*.

IMSL_B_COL_DIM, *int* b_col_dim (Input)
The column dimension of *B*.
Default: b_col_dim = ncolb

IMSL_X_VECTOR, *int* nx, *float* *x (Input)
The vector *x* of size nx.

IMSL_Y_VECTOR, *int* ny, *float* *y (Input)
The vector *y* of size ny.

IMSL_RETURN_USER, *float* ans[] (Output)
A user-allocated array containing the result.

IMSL_RETURN_COL_DIM, *int* return_col_dim (Input)
The column dimension of the answer.
Default: return_col_dim = the number of columns in the answer

Description

This function computes a matrix-vector product, a matrix-matrix product, a bilinear form of a matrix, or a triple product according to the specification given by string. For example, if "A*x" is given, Ax is computed. In string, the matrices A and B and the vectors x and y can be used. Any of these four names can be used with trans, indicating transpose. The vectors x and y are treated as $n \times 1$ matrices.

If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose, is returned. If `string` contains one multiplication, such as "`A*x`" or "`B*A`", then the indicated product is returned. Some other legal values for `string` are "`trans(y)*A`", "`A*trans(B)`", "`x*trans(y)`", or "`trans(x)*y`".

The matrices and/or vectors referred to in string must be given as optional arguments. If `string` is "`B*x`", then `IMSL_B_MATRIX` and `IMSL_X_VECTOR` must be given.

Example

Let

$$A = \begin{bmatrix} 1 & 2 & 9 \\ 5 & 4 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 7 & 4 \\ 9 & 1 \end{bmatrix} \quad x = \begin{bmatrix} 7 \\ 2 \\ 1 \end{bmatrix} \quad y = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

The arrays A^T, Ax, x^TA^T, AB, B^TA^T, x^Ty, xy^T, and x^TAy are computed and printed.

```
#include <imsl.h>

main()
{
    float       A[] = {1, 2, 9,
                       5, 4, 7};
    float       B[] = {3, 2,
                       7, 4,
                       9, 1};
    float       x[] = {7, 2, 1};
    float       y[] = {3, 4, 2};
    float       *ans;

    ans = imsl_f_mat_mul_rect("trans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
                              0);
    imsl_f_write_matrix("trans(A)", 3, 2, ans, 0);

    ans = imsl_f_mat_mul_rect("A*x",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_X_VECTOR, 3, x,
                              0);
    imsl_f_write_matrix("A*x", 1, 2, ans, 0);

    ans = imsl_f_mat_mul_rect("trans(x)*trans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_X_VECTOR, 3, x,
                              0);
    imsl_f_write_matrix("trans(x)*trans(A)", 1, 2, ans, 0);

    ans = imsl_f_mat_mul_rect("A*B",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_B_MATRIX, 3, 2, B,
                              0);
    imsl_f_write_matrix("A*B", 2, 2, ans, 0);

    ans = imsl_f_mat_mul_rect("trans(B)*trans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
```

```
                            IMSL_B_MATRIX, 3, 2, B,
                            0);
    imsl_f_write_matrix("trans(B)*trans(A)", 2, 2, ans, 0);

    ans = imsl_f_mat_mul_rect("trans(x)*y",
                            IMSL_X_VECTOR, 3, x,
                            IMSL_Y_VECTOR, 3, y,
                            0);
    imsl_f_write_matrix("trans(x)*y", 1, 1, ans, 0);

    ans = imsl_f_mat_mul_rect("x*trans(y)",
                            IMSL_X_VECTOR, 3, x,
                            IMSL_Y_VECTOR, 3, y,
                            0);
    imsl_f_write_matrix("x*trans(y)", 3, 3, ans, 0);

    ans = imsl_f_mat_mul_rect("trans(x)*A*y",
                            IMSL_A_MATRIX, 2, 3, A,
                              /* use only the first 2 components of x */
                            IMSL_X_VECTOR, 2, x,
                            IMSL_Y_VECTOR, 3, y,
                            0);
    imsl_f_write_matrix("trans(x)*A*y", 1, 1, ans, 0);
}
```

Output

```
        trans(A)
            1           2
1           1           5
2           2           4
3           9           7

          A*x
         1           2
        20          50

   trans(x)*trans(A)
         1           2
        20          50


            A*B
            1           2
1          98          19
2         106          33

    trans(B)*trans(A)
            1           2
1          98         106
2          19          33

trans(x)*y
        31

             x*trans(y)
            1           2           3
1          21          28          14
```

```
2           6           8           4
3           3           4           2

trans(x)*A*y
        293
```

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.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_mat_mul_rect` (*char* `*string`, …, 0)

The type *d_complex* function is `imsl_z_mat_mul_rect`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication. This is always a pointer to a *f_complex*, even if the result is a single number. To release this space, use `free`. If no answer was computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_mat_mul_rect` (*char* `*string`,
`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *f_complex* `*a`,
`IMSL_A_COL_DIM`, *int* `a_col_dim`,
`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *f_complex* `*b`,
`IMSL_B_COL_DIM`, *int* `b_col_dim`,
`IMSL_X_VECTOR`, *int* `nx`, *f_complex* `*x`,
`IMSL_Y_VECTOR`, *int* `ny`, *f_complex* `*y`,
`IMSL_RETURN_USER`, *f_complex* `ans[]`,
`IMSL_RETURN_COL_DIM`, *int* `return_col_dim`,
0)

Optional Arguments

`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *f_complex* `*a` (Input)
The `nrowa` × `ncola` matrix *A*.

`IMSL_A_COL_DIM`, *int* `a_col_dim` (Input)
The column dimension of *A*.
Default: `a_col_dim` = `ncola`

`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *f_complex* `*b` (Input)
The `nrowb` × `ncolb` matrix `B`.

`IMSL_B_COL_DIM`, *int* `b_col_dim` (Input)
The column dimension of *B*.
Default: `b_col_dim` = `ncolb`

`IMSL_X_VECTOR`, *int* `nx`, *f_complex* `*x` (Input)
The vector *x* of size `nx`.

`IMSL_Y_VECTOR`, *int* `ny`, *f_complex* `*y` (Input)
The vector *y* of size `ny`.

`IMSL_RETURN_USER`, *f_complex* `ans[]` (Output)
A user-allocated array containing the result.

`IMSL_RETURN_COL_DIM`, *int* `return_col_dim` (Input)
The column dimension of the answer.
Default: `return_col_dim` = the number of columns in the answer

Description

This function computes a matrix-vector product, a matrix-matrix product, a bilinear form of a matrix, or a triple product according to the specification given by `string`. For example, if "`A*x`" is given, *Ax* is computed. In `string`, the matrices *A* and *B* and the vectors *x* and *y* can be used. Any of these four names can be used with `trans`, indicating transpose, or with `ctrans`, indicating conjugate (or Hermitian) transpose. The vectors *x* and *y* are treated as $n \times 1$ matrices.

If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose, is returned. If `string` contains one multiplication, such as "`A*x`" or "`B*A`", then the indicated product is returned. Some other legal values for `string` are "`trans(y)*A`", "`A*ctrans(B)`", "`x*trans(y)`", or "`ctrans(x)*y`".

The matrices and/or vectors referred to in string must be given as optional arguments. If `string` is "`B*x`", then `IMSL_B_MATRIX` and `IMSL_X_VECTOR` must be given.

Example

Let

$$A = \begin{bmatrix} 1+4i & 2+3i & 9+6i \\ 5+2i & 4-3i & 7+i \end{bmatrix} \quad B = \begin{bmatrix} 3-6i & 2+4i \\ 7+3i & 4-5i \\ 9+2i & 1+3i \end{bmatrix}$$

$$x = \begin{bmatrix} 7+4i \\ 2+2i \\ 1-5i \end{bmatrix} \qquad y = \begin{bmatrix} 3+4i \\ 4-2i \\ 2+3i \end{bmatrix}$$

The arrays A^H, Ax, x^TA^T, AB, B^HA^T, x^Ty, and xy^H are computed and printed.

```
#include <imsl.h>

main()
{
    f_complex   A[] = {{1,4}, {2, 3}, {9,6},
                       {5,2}, {4,-3}, {7,1}};

    f_complex   B[] = {{3,-6}, {2, 4},
                       {7, 3}, {4,-5},
                       {9, 2}, {1, 3}};

    f_complex   x[] = {{7,4}, {2, 2}, {1,-5}};
    f_complex   y[] = {{3,4}, {4,-2}, {2, 3}};
    f_complex   *ans;

    ans = imsl_c_mat_mul_rect("ctrans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
                              0);
    imsl_c_write_matrix("ctrans(A)", 3, 2, ans, 0);

    ans = imsl_c_mat_mul_rect("A*x",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_X_VECTOR, 3, x,
                              0);
    imsl_c_write_matrix("A*x", 1, 2, ans, 0);

    ans = imsl_c_mat_mul_rect("trans(x)*trans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_X_VECTOR, 3, x,
                              0);
    imsl_c_write_matrix("trans(x)*trans(A)", 1, 2, ans, 0);

    ans = imsl_c_mat_mul_rect("A*B",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_B_MATRIX, 3, 2, B,
                              0);
    imsl_c_write_matrix("A*B", 2, 2, ans, 0);

    ans = imsl_c_mat_mul_rect("ctrans(B)*trans(A)",
                              IMSL_A_MATRIX, 2, 3, A,
                              IMSL_B_MATRIX, 3, 2, B,
                              0);
    imsl_c_write_matrix("ctrans(B)*trans(A)", 2, 2, ans, 0);

    ans = imsl_c_mat_mul_rect("trans(x)*y",
                              IMSL_X_VECTOR, 3, x,
                              IMSL_Y_VECTOR, 3, y,
                              0);
    imsl_c_write_matrix("trans(x)*y", 1, 1, ans, 0);

    ans = imsl_c_mat_mul_rect("x*ctrans(y)",
                              IMSL_X_VECTOR, 3, x,
                              IMSL_Y_VECTOR, 3, y,
                              0);
    imsl_c_write_matrix("x*ctrans(y)", 3, 3, ans, 0);
}
```

Output

```
                    ctrans(A)
                        1                       2
1  (         1,        -4)  (         5,        -2)
2  (         2,        -3)  (         4,         3)
3  (         9,        -6)  (         7,        -1)

                      A*x
                     1                      2
(        28,         3)  (        53,         2)

               trans(x)*trans(A)
                     1                      2
(        28,         3)  (        53,         2)

                        A*B
                         1                       2
1  (       101,       105)  (         0,        47)
2  (       125,       -10)  (         7,        14)

               ctrans(B)*trans(A)
                         1                       2
1  (        95,        69)  (        87,        -2)
2  (        38,         5)  (        59,       -28)

      trans(x)*y
(        34,        37)

                                  x*ctrans(y)
                         1                       2                       3
1 (        37,       -16)  (        20,        30)  (        26,       -13)
2 (        14,        -2)  (         4,        12)  (        10,        -2)
3 (       -17,       -19)  (        14,       -18)  (       -13,       -13)
```

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.

Synopsis

#include `<imsl.h>`

float `*imsl_f_mat_mul_rect_band` (*char* `*string`, ..., 0)

The equivalent *double* function is `imsl_d_mat_mul_rect_band`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include <imsl.h>

void *imsl_f_mat_mul_rect_band (*char* *string,
        IMSL_A_MATRIX, *int* nrowa, *int* ncola, *int* nlca, *int* nuca,
                float *a,
        IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nlcb, *int* nucb,
                float *b,
        IMSL_X_VECTOR, *int* nx, *float* *x,
        IMSL_RETURN_MATRIX_CODIAGONALS, *int* *nlc_result,
                int *nuc_result,
        IMSL_RETURN_USER_VECTOR, *float* vector_user[],
        0)

Optional Arguments

IMSL_A_MATRIX, *int* nrowa, *int* ncola, *int* nlca, *int* nuca, *float* *a (Input)
The sparse matrix

$$A \in \Re^{\text{nrowa} \times \text{ncola}}$$

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nlcb, *int* nucb, *float* *b (Input)
The sparse matrix

$$B \in \Re^{\text{nrowb} \times \text{xnolb}}$$

IMSL_X_VECTOR, *int* nx, *float* *x, (Input)
The vector x of length nx.

IMSL_RETURN_MATRIX_CODIAGONALS, *int* *nlc_result, *int* *nuc_result, (Output)
If the function imsl_f_mat_mul_rect_band returns data for a band matrix, use this option to retrieve the number of lower and upper codiagonals of the return matrix.

IMSL_RETURN_USER_VECTOR, *float* vector_user[], (Output)
If the result of the computation in a vector, return the answer in the user supplied sparse vector_user.

Description

The function imsl_f_mat_mul_rect_band computes a matrix-matrix product or a matrix-vector product, where the matrices are specified in band format. The operation performed is specified by string. For example, if "A*x" is given, Ax is computed. In string, the matrices A and B and the vector x can be used. Any of these names can be used with trans, indicating transpose. The vector x is treated as a dense $n \times 1$ matrix. If string contains only one item, such as "x" or "trans(A)", then a copy of the array, or its transpose is returned.

The matrices and/or vector referred to in string must be given as optional arguments. Therefore, if string is "A*x", then IMSL_A_MATRIX and IMSL_X_VECTOR must be given.

Examples

Example 1

Consider the matrix

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -3 & 1 & -2 & 0 \\ 0 & 0 & -1 & 2 \\ 0 & 0 & 2 & 1 \end{bmatrix}$$

After storing A in band format, multiply A by $x = (1, 2, 3, 4)^T$ and print the result.

```
#include <imsl.h>
main()
{
        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 x[] = {1.0, 2.0, 3.0, 4.0};
        int n = 4;
        int nuca = 1;
        int nlca = 1;
        float *b;

                        /* Set b = A*x */

        b = imsl_f_mat_mul_rect_band ("A*x",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                0);

        imsl_f_write_matrix ("Product, Ax", 1, n, b, 0);
}
```

Output

```
          Product, Ax
 1           2           3           4
 0          -7           5          10
```

Example 2

This example uses the power method to determine the dominant eigenvector of $E(100, 10)$. The same computation is performed by using `imsl_f_eig_sym`. The iteration stops when the component-wise absolute difference between the dominant eigenvector found by `imsl_f_eig_sym` and the eigenvector at the current iteration is less than the square root of machine unit roundoff.

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

void main()
{
        int                 i;
        int                 j;
        int                 k;
        int                 n;
        int                 c;
        int                 nz;
        int                 index;
        int                 start;
        int                 stop;
        float              *a;
        float              *z;
        float              *q;
        float              *dense_a;
        float              *dense_evec;
        float              *dense_eval;
        float               norm;
        float              *evec;
        float               error;
        float               tolerance;

        n = 100;
        c = 10;
        tolerance = sqrt(imsl_f_machine(4));
        error = 1.0;

        evec = (float*) malloc (n*sizeof(*evec));
        z = (float*) malloc (n*sizeof(*z));
        q = (float*) malloc (n*sizeof(*q));
        dense_a = (float*) calloc (n*n, sizeof(*dense_a));
        a = imsl_f_generate_test_band (n, c, 0);

                        /* Convert to dense format,
                           starting with upper triangle */

        start = c;
        for (i=0; i<c; i++, start--)
                for (k=0, j=start; j<n; j++, k++)
                        dense_a[k*n + j] = a[i*n + j];

                        /* Convert diagonal */

        for (j=0; j<n; j++)
                       dense_a[j*n + j] = a[c*n + j];

                        /* Convert lower triangle */
        stop = n-1;
        for (i=c+1; i<2*c+1; i++, stop--)
                for (k=i-c, j=0; j<stop; j++, k++)
                        dense_a[k*n + j] = a[i*n + j];

                        /* Determine dominant eigenvector by a dense method
*/

        dense_eval = imsl_f_eig_sym (n, dense_a,
```

```
                IMSL_VECTORS, &dense_evec,
                0);
        for (i=0; i<n; i++) evec[i] = dense_evec[n*i];

                        /* Normalize */

        norm = imsl_f_vector_norm (n, evec, 0);
        for (i=0; i<n; i++) evec[i] /= norm;

        for (i=0; i<n; i++) q[i] = 1.0/sqrt((float) n);

                        /* Do power method */

        while (error > tolerance) {
                imsl_f_mat_mul_rect_band ("A*x",
                        IMSL_A_MATRIX, n, n, c, c, a,
                        IMSL_X_VECTOR, n, q,
                        IMSL_RETURN_USER_VECTOR, z,
                        0);

                        /* Normalize */

                norm = imsl_f_vector_norm (n, z, 0);
                for (i=0; i<n; i++) q[i] = z[i]/norm;

                        /* Compute maximum absolute error between any
                           two elements */

                error = imsl_f_vector_norm (n, q,
                        IMSL_SECOND_VECTOR, evec,
                        IMSL_INF_NORM, &index,
                        0);
        }
        printf ("Maximum absolute error = %e\n", error);
}
```

Output

```
Maximum absolute error = 3.367960e-04
```

mat_mul_rect_band (complex)

Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product for all matrices of complex type and stored in band form.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_mat_mul_rect_band` (*char* `*string`, ..., 0)

The equivalent *d_complex* function is `imsl_z_mat_mul_rect_band`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication is returned. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `*imsl_c_mat_mul_rect_band` (*char* `*string`,
`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *int* `nlca`, *int* `nuca`,
f_complex `*a`,
`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *int* `nlcb`, *int* `nucb`,
f_complex `*b`,
`IMSL_X_VECTOR`, *int* `nx`, *f_complex* `*x`,
`IMSL_RETURN_MATRIX_CODIAGONALS`, *int* `*nlc_result`,
int `*nuc_result`,
`IMSL_RETURN_USER_VECTOR`, *f_complex* `vector_user[]`,
0)

Optional Arguments

`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *int* `nlca`, *int* `nuca`, *f_complex* `*a` (Input)
The sparse matrix

$$A \in \Re^{\text{nrowa} \times \text{ncola}}$$

`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *int* `nlcb`, *int* `nucb`, *f_complex* `*b` (Input)
The sparse matrix

$$B \in \Re^{\text{nrowb} \times \text{xnolb}}$$

`IMSL_X_VECTOR`, *int* `nx`, *f_complex* `*x`, (Input)
The vector x of length `nx`.

`IMSL_RETURN_MATRIX_CODIAGONALS`, *int* `*nlc_result`, *int* `*nuc_result`, (Output)
If the function `imsl_c_mat_mul_rect_band` returns data for a band matrix, use this option to retrieve the number of lower and upper codiagonals of the return matrix.

`IMSL_RETURN_USER_VECTOR`, *f_complex* `vector_user[]`, (Output)
If the result of the computation in a vector, return the answer in the user supplied sparse `vector_user`.

Description

The function `imsl_c_mat_mul_rect_band` computes a matrix-matrix product or a matrix-vector product, where the matrices are specified in band format. The operation performed is specified by `string`. For example, if "`A*x`" is given, Ax is computed. In `string`, the matrices A and B and the vector x can be used. Any of these names can be used with `trans`, indicating transpose. The vector x is treated as a dense $n \times 1$ matrix. If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose is returned.

The matrices and/or vector referred to in string must be given as optional arguments. Therefore, if `string` is "`A*x`", then `IMSL_A_MATRIX` and `IMSL_X_VECTOR` must be given.

Examples

Example 1

Let

$$A = \begin{bmatrix} -2 & 4 & 0 & 0 \\ 6+i & -0.5+3i & -2+2i & 0 \\ 0 & 1+i & 3-3i & -4-i \\ 0 & 0 & 2i & 1-i \end{bmatrix}$$

and

$$x = \begin{bmatrix} 3 \\ -1+i \\ 3 \\ -1+i \end{bmatrix}$$

This example computes the product Ax.

```
#include <imsl.h>

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

                        /* 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    x[] =
                {{3.0, 0.0}, {-1.0, 1.0}, {3.0, 0.0}, {-1.0, 1.0}};
```

```
                        /* Set b = A*x */

        b = imsl_c_mat_mul_rect_band ("A*x",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                0);

        imsl_c_write_matrix ("Product, Ax", 1, n, b, 0);
}
```

Output

```
                                Product, Ax
                      1                          2                          3
(     -10.0,      -5.0)  (       9.5,       5.5)  (      12.0,     -12.0)

                      4
(       0.0,       8.0)
```

Example 2

Using the same matrix A and vector x given in the last example, the products Ax, $A^T x$, $A^H x$ and AA^H are computed.

```
#include <imsl.h>

main()
{
        int          n = 4;
        int          nlca = 1;
        int          nuca = 1;
        f_complex   *b;
        f_complex   *z;
        int          i;
        int          nlca_z;
        int          nuca_z;

                        /* 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    x[] =
                {{3.0, 0.0}, {-1.0, 1.0}, {3.0, 0.0}, {-1.0, 1.0}};

                        /* Set b = A*x */

        b = imsl_c_mat_mul_rect_band ("A*x",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                0);

        imsl_c_write_matrix ("Ax", 1, n, b, 0);
        free(b);
```

```
                        /* Set b = trans(A)*x */

        b = imsl_c_mat_mul_rect_band ("trans(A)*x",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                0);

        imsl_c_write_matrix ("\n\ntrans(A)x", 1, n, b, 0);
        free(b);

                        /* Set b = ctrans(A)*x */

        b = imsl_c_mat_mul_rect_band ("ctrans(A)*x",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                0);

        imsl_c_write_matrix ("\n\nctrans(A)x", 1, n, b, 0);
        free(b);

                        /* Set z = A*ctrans(A) */

        z = imsl_c_mat_mul_rect_band ("A*ctrans(A)",
                IMSL_A_MATRIX, n, n, nlca, nuca, a,
                IMSL_X_VECTOR, n, x,
                IMSL_RETURN_MATRIX_CODIAGONALS, &nlca_z, &nuca_z,
                0);

        imsl_c_write_matrix("A*ctrans(A)", nlca_z+nuca_z+1, n, z, 0);
}
```

Output

```
                                Ax
                  1                       2                       3
(     -10.0,      -5.0)  (       9.5,       5.5)  (      12.0,     -12.0)

                  4
(       0.0,       8.0)


                             trans(A)x
                  1                       2                       3
(     -13.0,      -4.0)  (      12.5,      -0.5)  (       7.0,     -15.0)

                  4
(     -12.0,      -1.0)


                            ctrans(A)x
                  1                       2                       3
(     -11.0,      16.0)  (      18.5,      -0.5)  (      15.0,      11.0)

                  4
(     -14.0,       3.0)

                             A*ctrans(A)
                     1                       2                       3
```

```
1 (      0.00,      0.00)  (      0.00,      0.00)  (      4.00,     -4.00)
2 (      0.00,      0.00)  (    -17.00,    -28.00)  (     -9.50,      3.50)
3 (     29.00,      0.00)  (     54.25,      0.00)  (     37.00,      0.00)
4 (    -17.00,     28.00)  (     -9.50,     -3.50)  (     -9.00,     11.00)
5 (      4.00,      4.00)  (      4.00,     -4.00)  (      0.00,      0.00)

                         4
1  (      4.00,      4.00)
2  (     -9.00,    -11.00)
3  (      6.00,      0.00)
4  (      0.00,      0.00)
5  (      0.00,      0.00)
```

mat_mul_rect_coordinate

Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product for all matrices stored in sparse coordinate form.

Synopsis

#include `<imsl.h>`

void `*imsl_f_mat_mul_rect_coordinate` (*char* `*string`, ..., 0)

The equivalent *double* function is `imsl_d_mat_mul_rect_coordinate`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication. If the result is a vector, the return type is pointer to *float*. If the result of the multiplication is a sparse matrix, the return type is pointer to *Imsl_f_sparse_elem*. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `*imsl_f_mat_mul_rect_coordinate` (*char* `*string`,
`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *int* `nza`, *Imsl_f_sparse_elem* `*a`,
`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *int* `nzb`, *Imsl_f_sparse_elem* `*b`,
`IMSL_X_VECTOR`, *int* `nx`, *float* `*x`,
`IMSL_RETURN_MATRIX_SIZE`, *int* `*size`,
`IMSL_RETURN_USER_VECTOR`, *float* `vector_user[]`,
0)

Optional Arguments

`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *int* `nza`, *Imsl_f_sparse_elem* `*a` (Input)
The sparse matrix

$$A \in \Re^{\text{nrowa} \times \text{ncola}}$$

with `nza` nonzero elements.

`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *int* `nzb`, *Imsl_f_sparse_elem* `*b` (Input)
The sparse matrix

$$B \in \Re^{\text{nrowb} \times \text{xnolb}}$$

with `nzb` nonzero elements.

`IMSL_X_VECTOR`, *int* `nx`, *float* `*x`, (Input)
The vector x of length `nx`.

`IMSL_RETURN_MATRIX_SIZE`, *int* `*size`, (Output)
If the function `imsl_f_mat_mul_rect_coordinate` returns a vector of type *Imsl_f_sparse_elem*, use this option to retrieve the length of the return vector, i.e. the number of nonzero elements in the sparse matrix generated by the requested computations.

`IMSL_RETURN_USER_VECTOR`, *float* `vector_user[]`, (Output)
If the result of the computation in a vector, return the answer in the user supplied sparse `vector_user`. It's size depends on the computation.

Description

The function `imsl_f_mat_mul_rect_coordinate` computes a matrix-matrix product or a matrix-vector product, where the matrices are specified in coordinate representation. The operation performed is specified by `string`. For example, if "`A*x`" is given, Ax is computed. In `string`, the matrices A and B and the vector x can be used. Any of these names can be used with `trans`, indicating transpose. The vector x is treated as a dense $n \times 1$ matrix.

If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose is returned. Some multiplications, such as "`A*trans(A)`" or "`trans(x)*B`", will produce a sparse matrix in coordinate format as a result. Other products such as "`B*x`" will produce a pointer to a floating type, containing the resulting vector.

The matrices and/or vector referred to in string must be given as optional arguments. Therefore, if `string` is "`A*x`", then `IMSL_A_MATRIX` and `IMSL_X_VECTOR` must be given.

Examples

Example 1

In this example, a sparse matrix in coordinate form is multipled by a vector.

```
#include <imsl.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;

                        /* Set x = A*b */

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

        imsl_f_write_matrix ("Product Ab", 1, n, x, 0);
}
```

Output

```
                          Product Ab
  1           2           3           4           5           6
100         -98         675         344        -302         162
```

Example 2

This example uses the power method to determine the dominant eigenvector of E(100, 10). The same computation is performed by using `imsl_f_eig_sym`. The iteration stops when the component-wise absolute difference between the dominant eigenvector found by `imsl_f_eig_sym` and the eigenvector at the current iteration is less than the square root of machine unit roundoff.

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

void main()
{
        int                 i;
        int                 n;
        int                 c;
        int                 nz;
        int                 index;
        Imsl_f_sparse_elem *a;
        float              *z;
        float              *q;
        float              *dense_a;
        float              *dense_evec;
        float              *dense_eval;
        float               norm;
        float              *evec;
        float               error;
        float               tolerance;

        n = 100;
        c = 10;
        tolerance = sqrt(imsl_f_machine(4));
        error = 1.0;

        evec = (float*) malloc (n*sizeof(*evec));
        z = (float*) malloc (n*sizeof(*z));
        q = (float*) malloc (n*sizeof(*q));
        dense_a = (float*) calloc (n*n, sizeof(*dense_a));
        a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

                /* Convert to dense format */

        for (i=0; i<nz; i++)
                dense_a[a[i].col + n*a[i].row] = a[i].val;

                /* Determine dominant eigenvector by a dense method */

        dense_eval = imsl_f_eig_sym (n, dense_a,
                IMSL_VECTORS, &dense_evec,
                0);
        for (i=0; i<n; i++) evec[i] = dense_evec[n*i];

                /* Normalize */

        norm = imsl_f_vector_norm (n, evec, 0);
        for (i=0; i<n; i++) evec[i] /= norm;

        for (i=0; i<n; i++) q[i] = 1.0/sqrt((float) n);

                /* Do power method */

        while (error > tolerance) {
                imsl_f_mat_mul_rect_coordinate ("A*x",
                        IMSL_A_MATRIX, n, n, nz, a,
                        IMSL_X_VECTOR, n, q,
                        IMSL_RETURN_USER_VECTOR, z,
                        0);
```

```
                /* Normalize */

                norm = imsl_f_vector_norm (n, z, 0);
                for (i=0; i<n; i++) q[i] = z[i]/norm;

                /* Compute maximum absolute error between any
                           two elements */
                error = imsl_f_vector_norm (n, q,
                        IMSL_SECOND_VECTOR, evec,
                        IMSL_INF_NORM, &index,
                        0);
        }
        printf ("Maximum absolute error = %e\n", error);
}
```

Output

```
Maximum absolute error = 3.368035e-04
```

mat_mul_rect_coordinate (complex)

Computes the transpose of a matrix, a matrix-vector produce, or a matrix-matrix product for all matrices stored in sparse coordinate form.

Synopsis

#include `<imsl.h>`

void `*imsl_c_mat_mul_rect_coordinate` (*char* `*string`, ..., 0)

The equivalent *double* function is `imsl_d_mat_mul_rect_coordinate`.

Required Arguments

char `*string` (Input)
String indicating matrix multiplication to be performed.

Return Value

The result of the multiplication. If the result is a vector, the return type is pointer to *f_complex*. If the result of the multiplication is a sparse matrix, the return type is pointer to *Imsl_c_sparse_elem*.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `*imsl_c_mat_mul_rect_coordinate` (*char* `*string`,
`IMSL_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *int* `nza`, *Imsl_c_sparse_elem* `*a`,
`IMSL_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *int* `nzb`, *Imsl_c_sparse_elem* `*b`,
`IMSL_X_VECTOR`, *int* `nx`, *f_complex* `*x`,

IMSL_RETURN_MATRIX_SIZE, *int* *size,
IMSL_RETURN_USER_VECTOR, *f_complex* vector_user[],
0)

Optional Arguments

IMSL_A_MATRIX, *int* nrowa, *int* ncola, *int* nza, *Imsl_c_sparse_elem* *a
(Input)
The sparse matrix

$$A \in C^{\text{nrowa} \times \text{ncola}}$$

with nza nonzero elements.

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nzb, *Imsl_c_sparse_elem* *b
(Input)
The sparse matrix

$$B \in C^{\text{nrowb} \times \text{xnolb}}$$

with nzb nonzero elements.

IMSL_X_VECTOR, *int* nx, *f_complex* *x, (Input)
The vector *x* of length nx.

IMSL_RETURN_MATRIX_SIZE, *int* *size, (Output)
If the function imsl_c_mat_mul_rect_coordinate returns a vector of type *Imsl_c_sparse_elem*, use this option to retrieve the length of the return vector, i.e. the number of nonzero elements in the sparse matrix generated by the requested computations.

IMSL_RETURN_USER_VECTOR, *f_complex* vector_user[], (Output)
If the result of the computation is a vector, return the answer in the user supplied space vector_user. It's size depends on the computation.

Description

The function imsl_c_mat_mul_rect_coordinate computes a matrix-matrix product or a matrix-vector product, where the matrices are specified in coordinate representation. The operation performed is specified by string. For example, if "A*x" is given, *Ax* is computed. In string, the matrices *A* and *B* and the vector *x* can be used. Any of these names can be used with trans or ctrans, indicating transpose and conjugate transpose, respectively. The vector *x* is treated as a dense $n \times 1$ matrix.

If string contains only one item, such as "x" or "trans(A)", then a copy of the array, or its transpose is returned. Some multiplications, such as "A*ctrans(A)" or "trans(x)*B", will produce a sparse matrix in coordinate format as a result. Other products such as "B*x" will produce a pointer to a complex type, containing the resulting vector.

The matrix and/or vector referred to in string must be given as optional arguments. Therefore, if string is "A*x", IMSL_A_MATRIX and IMSL_X_VECTOR must be given.

To release this space, use free.

Examples

Example 1

Let

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

and

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

This example computes the product Ax.

```
#include <imsl.h>

main()
{
        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}};
        f_complex     b[] = {{1.0, 1.0}, {2.0, 2.0}, {3.0, 3.0},
                             {4.0, 4.0}, {5.0, 5.0}, {6.0, 6.0}};

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

                        /* Set x = A*b */

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

        imsl_c_write_matrix ("Product Ab", 1, n, x, 0);
}
```

Output

```
                          Product Ab
                    1                         2                         3
(         3,        17)  (       -19,         5)  (         6,        18)

                    4                         5                         6
(       -38,        32)  (       -63,        49)  (       -57,        83)
```

Example 2

Using the same matrix A and vector x given in the last example, the products Ax, $A^T x$, $A^H x$ and AA^H are computed.

```
#include <imsl.h>

main()
{
        Imsl_c_sparse_elem *z;
        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}};
        f_complex     x[] = {{1.0, 1.0}, {2.0, 2.0}, {3.0, 3.0},
                             {4.0, 4.0}, {5.0, 5.0}, {6.0, 6.0}};

        int           n = 6;
        int           nz = 15;
        int           nz_z;
        int           i;
        f_complex    *b;

                        /* Set b = A*x */

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

        imsl_c_write_matrix ("Ax", 1, n, b, 0);
        free(b);

                        /* Set b = trans(A)*x */

        b = imsl_c_mat_mul_rect_coordinate ("trans(A)*x",
                IMSL_A_MATRIX, n, n, nz, a,
```

```
                IMSL_X_VECTOR, n, x,
                0);

        imsl_c_write_matrix ("\n\ntrans(A)x", 1, n, b, 0);
        free(b);

                        /* Set b = ctrans(A)*x */

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

        imsl_c_write_matrix ("\n\nctrans(A)x", 1, n, b, 0);
        free(b);

                        /* Set z = A*ctrans(A) */

        z = imsl_c_mat_mul_rect_coordinate ("A*ctrans(A)",
                IMSL_A_MATRIX, n, n, nz, a,
                IMSL_X_VECTOR, n, x,
                IMSL_RETURN_MATRIX_SIZE, &nz_z,
                0);

        printf("\n\n\t\t\t    z = A*ctrans(A)\n\n");

        for (i=0; i<nz_z; i++)
                printf ("\t\t\tz(%1d,%1d) = (%6.1f, %6.1f)\n",
                        z[i].row, z[i].col, z[i].val.re, z[i].val.im);
}
```

Output

```
                                  Ax
                    1                       2                       3
(         3,        17)  (       -19,        5)  (         6,        18)

                    4                       5                       6
(       -38,        32)  (       -63,       49)  (       -57,        83)



                               trans(A)x
                    1                       2                       3
(      -112,        54)  (       -58,       46)  (         0,        12)

                    4                       5                       6
(       -51,         5)  (        34,       78)  (       -94,        60)



                              ctrans(A)x
                    1                       2                       3
(        54,      -112)  (        46,      -58)  (        12,         0)

                    4                       5                       6
(         5,       -51)  (        78,       34)  (        60,       -94)
```

```
    z = A*ctrans(A)

z(0,0) = ( 149.0,    0.0)
z(0,3) = ( -48.0,   26.0)
z(0,4) = ( -22.0,  -75.0)
z(0,5) = (  74.0, -127.0)
z(1,1) = (  27.0,    0.0)
z(1,2) = ( -12.0,    6.0)
z(1,3) = (  11.0,    8.0)
z(1,4) = (   5.0,  -10.0)
z(1,5) = (  10.0,  -28.0)
z(2,1) = ( -12.0,   -6.0)
z(2,2) = (  20.0,    0.0)
z(3,0) = ( -48.0,  -26.0)
z(3,1) = (  11.0,   -8.0)
z(3,3) = (  67.0,    0.0)
z(3,4) = ( -17.0,   36.0)
z(3,5) = ( -46.0,   28.0)
z(4,0) = ( -22.0,   75.0)
z(4,1) = (   5.0,   10.0)
z(4,3) = ( -17.0,  -36.0)
z(4,4) = ( 312.0,    0.0)
z(4,5) = (  81.0,  126.0)
z(5,0) = (  74.0,  127.0)
z(5,1) = (  10.0,   28.0)
z(5,3) = ( -46.0,  -28.0)
z(5,4) = (  81.0, -126.0)
z(5,5) = ( 271.0,    0.0)
```

mat_add_band

Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.

Synopsis

#include `<imsl.h>`

float `*imsl_f_mat_add_band` (*int* `n`, *int* `nlca`, *int* `nuca`, *float* `alpha`, *float* `a[]`, *int* `nlcb`, *int* `nucb`, *float* `beta`, *float* `b[]`, *int* `*nlcc`, *int* `*nucc`, ..., 0)

The type *double* function is `imsl_d_mat_add_band`.

Required Arguments

int `n` (Input)
The order of the matrices *A* and *B*.

int `nlca` (Input)
Number of lower codiagonals of *A*.

int `nuca` (Input)
Number of upper codiagonals of *A*.

float `alpha` (Input)
Scalar multiplier for *A*.

float `a[]` (Input)
An *n* by *n* band matrix with *nlca* lower codiagonals and *nuca* upper codiagonals stored in band mode with dimension (*nlca* + *nuca* + 1) by *n*.

int `nlcb` (Input)
Number of lower codiagonals of *B*.

int `nucb` (Input)
Number of upper codiagonals of *B*.

float `beta` (Input)
Scalar multiplier for *B*.

float `b[]` (Input)
An *n* by *n* band matrix with *nlcb* lower codiagonals and *nucb* upper codiagonals stored in band mode with dimension (*nlcb* + *nucb* + 1) by *n*.

int `*nlcc` (Output)
Number of lower codiagonals of *C*.

int `*nucc` (Output)
Number of upper codiagonals of *C*.

Return Value

A pointer to an array of type *float* containing the computed sum. `NULL` is returned in the event of an error or if the return matrix has no nonzero elements.

Synopsis with Optional Arguments

```
#include <imsl.h>

float *imsl_f_mat_add_band (int n, int nlca, int nuca, float alpha, float
        a[], int nlcb, int nucb, float beta, float b[], int *nlcc, int *nucc,
        IMSL_A_TRANSPOSE,
        IMSL_B_TRANSPOSE,
        IMSL_SYMMETRIC,
        0)
```

Optional Arguments

`IMSL_A_TRANSPOSE`,
Replace *A* with A^T in the expression $\alpha A + \beta B$.

`IMSL_B_TRANSPOSE`,
Replace *B* with B^T in the expression $\alpha A + \beta B$.

`IMSL_SYMMETRIC`,
A, *B* and *C* are stored in band symmetric storage mode.

Description

The function `imsl_f_mat_add_band` forms the sum $\alpha A + \beta B$, given the scalars α and β, and, the matrices *A* and *B* in band format. The transpose of *A* and/or *B* may be used during the computation if optional arguments are specified. Symmetric storage mode may be used if the optional argument is specified.

If `IMSL_SYMMETRIC` is specified, the return value for the number of lower codiagonals, *nlcc*, will be equal to 0.

If the return matrix equals `NULL`, the return value for the number of lower codiagonals, *nlcc*, will be equal to −1 and the number of upper codiagonals, *nucc*, will be equal to 0.

Examples

Example 1

Add two real matrices of order 4 stored in band mode. Matrix *A* has one upper codiagonal and one lower codiagonal. Matrix *B* has no upper codiagonals and two lower codiagonals.

```
#include <imsl.h>

void main()
{
        float a[] = {0.0, 2.0, 3.0, -1.0,
                     1.0, 1.0, 1.0, 1.0,
                     0.0, 3.0, 4.0, 0.0};
        float b[] = {3.0, 3.0, 3.0, 3.0,
                     1.0, -2.0, 1.0, 0.0,
                    -1.0, 2.0, 0.0, 0.0};
        int       nucb = 0, nlcb = 2;
        int       nuca = 1, nlca = 1;
        int       nucc, nlcc;
        int       n = 4, m;
        float     alpha = 1.0, beta = 1.0;
        float    *c;

        c = imsl_f_mat_add_band(n, nlca, nuca, alpha, a,
                                nlcb, nucb, beta, b,
                                &nlcc, &nucc, 0);

        m = nlcc + nucc + 1;
        imsl_f_write_matrix("C = A + B", m, n, c, 0);
        free(c);
}
```

```
                  C = A + B
            1           2           3           4
1           0           2           3          -1
2           4           4           4           4
3           1           1           5           0
4          -1           2           0           0
```

Example 2

Compute 4*A + 2*B, where

$$A = \begin{bmatrix} 3 & 4 & 0 & 0 \\ 4 & 2 & 3 & 0 \\ 0 & 3 & 1 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 2 & 0 & 0 \\ 2 & 1 & 3 & 0 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

```
#include <imsl.h>

void main()
{
        float a[] = {0.0, 4.0, 3.0, 1.0,
                     3.0, 2.0, 1.0, 2.0};
        float b[] = {0.0, 2.0, 3.0, 1.0,
                     5.0, 1.0, 2.0, 2.0};
        int     nuca = 1, nlca = 1;
        int     nucb = 1, nlcb = 1;
        int     n = 4, m, nlcc, nucc;
        float   alpha = 4.0, beta = 2.0;
        float  *c;

        c = imsl_f_mat_add_band(n, nlca, nuca, alpha, a,
                                nlcb, nucb, beta, b,
                                &nlcc, &nucc,
                                IMSL_SYMMETRIC, 0);

        m = nucc + nlcc + 1;
        imsl_f_write_matrix("C = 4*A + 2*B\n", m, n, c, 0);
        free(c);
}
```

Output

```
          C = 4*A + 2*B

           1           2           3           4
1          0          20          18           6
2         22          10           8          12
```

mat_add_band (complex)

Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_mat_add_band` (*int* `n`, *int* `nlca`, *int* `nuca`, *f_complex* `alpha`, *f_complex* `a[]`, *int* `nlcb`, *int* `nucb`, *f_complex* `beta`, *f_complex* `b[]`, *int* `*nlcc`, *int* `*nucc`, ..., 0)

The type *double* function is `imsl_z_mat_add_band`.

Required Arguments

int `n` (Input)
The order of the matrices *A* and *B*.

int `nlca` (Input)
Number of lower codiagonals of *A*.

int `nuca` (Input)
Number of upper codiagonals of *A*.

f_complex `alpha` (Input)
Scalar multiplier for *A*.

f_complex `a[]` (Input)
An *n* by *n* band matrix with *nlca* lower codiagonals and *nuca* upper codiagonals stored in band mode with dimension (*nlca* + *nuca* + 1) by *n*.

int `nlcb` (Input)
Number of lower codiagonals of *B*.

int `nucb` (Input)
Number of upper codiagonals of *B*.

f_complex `beta` (Input)
Scalar multiplier for *B*.

f_complex `b[]` (Input)
An *n* by *n* band matrix with *nlcb* lower codiagonals and *nucb* upper codiagonals stored in band mode with dimension (*nlcb* + *nucb* + 1) by *n*.

int `*nlcc` (Output)
Number of lower codiagonals of *C*.

int `*nucc` (Output)
Number of upper codiagonals of *C*.

Return Value

A pointer to an array of type *f_complex* containing the computed sum. In the event of an error or if the return matrix has no nonzero elements, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

f_complex `*imsl_c_mat_add_band` (*int* `n`, *int* `nlca`, *int* `nuca`, *f_complex* `alpha`, *f_complex* `a[]`, *int* `nlcb`, *int* `nucb`, *f_complex* `beta`, *f_complex* `b[]`, *int* `*nlcc`, *int* `*nucc`,
`IMSL_A_TRANSPOSE`,
`IMSL_B_TRANSPOSE`,
`IMSL_A_CONJUGATE_TRANSPOSE`,
`IMSL_B_CONJUGATE_TRANSPOSE`,
`IMSL_SYMMETRIC`,
0)

Optional Arguments

`IMSL_A_TRANSPOSE`,
Replace A with A^T in the expression $\alpha A + \beta B$.

`IMSL_B_TRANSPOSE`,
Replace B with B^T in the expression $\alpha A + \beta B$.

`IMSL_A_CONJUGATE_TRANSPOSE`,
Replace A with A^H in the expression $\alpha A + \beta B$.

`IMSL_B_CONJUGATE_TRANSPOSE`,
Replace B with B^H in the expression $\alpha A + \beta B$.

`IMSL_SYMMETRIC`,
Matrix A, B, and C are stored in band symmetric storage mode.

Description

The function `imsl_c_mat_add_band` forms the sum $\alpha A + \beta B$, given the scalars α and β, and the matrices A and B in band format. The transpose or conjugate transpose of A and/or B may be used during the computation if optional arguments are specified. Symmetric storage mode may be used if the optional argument is specified.

If `IMSL_SYMMETRIC` is specified, the return value for the number of lower codiagonals, *nlcc*, will be equal to 0.

If the return matrix equals `NULL`, the return value for the number of lower codiagonals, *nlcc*, will be equal to −1 and the number of upper codiagonals, *nucc*, will be equal to 0.

Examples

Example 1

Add two complex matrices of order 4 stored in band mode. Matrix A has one upper codiagonal and one lower codiagonal. Matrix B has no upper codiagonals and two lower codiagonals.

```
#include <imsl.h>

void main()
{
        f_complex a[] =
                   {{0.0, 0.0}, {2.0, 1.0}, {3.0, 3.0}, {-1.0, 0.0},
                    {1.0, 1.0}, {1.0, 3.0}, {1.0, -2.0}, {1.0, 5.0},
                    {0.0, 0.0}, {3.0, -2.0}, {4.0, 0.0}, {0.0, 0.0}};
        f_complex b[] =
                   {{3.0, 1.0}, {3.0, 5.0}, {3.0, -1.0}, {3.0, 1.0},
                    {1.0, -3.0}, {-2.0, 0.0}, {1.0, 2.0}, {0.0, 0.0},
                    {-1.0, 4.0}, {2.0, 1.0}, {0.0, 0.0}, {0.0, 0.0}};
        int        nucb = 0, nlcb = 2;
        int        nuca = 1, nlca = 1;
        int        nucc, nlcc;
        int        n = 4, m;
        f_complex *c;
        f_complex  alpha = {1.0, 0.0};
```

```
        f_complex  beta = {1.0, 0.0};

        c = imsl_c_mat_add_band(n, nlca, nuca, alpha, a,
                                nlcb, nucb, beta, b,
                                &nlcc, &nucc, 0);

        m = nlcc + nucc + 1;
        imsl_c_write_matrix("C = A + B", m, n, c, 0);
        free(c);
}
```

Output

```
                                C = A + B
                         1                        2                        3
1 (         0,          0)  (         2,          1)  (         3,          3)
2 (         4,          2)  (         4,          8)  (         4,         -3)
3 (         1,         -3)  (         1,         -2)  (         5,          2)
4 (        -1,          4)  (         2,          1)  (         0,          0)

                         4
1 (        -1,          0)
2 (         4,          6)
3 (         0,          0)
4 (         0,          0)
```

Example 2

Compute

$$(3 + 2i)A^H + (4 + i)\,B^H$$

where

$$A = \begin{bmatrix} 2+3i & 1+3i & 0 & 0 \\ 0 & 6+2i & 3+i & 0 \\ 0 & 0 & 4+i & 2+5i \\ 0 & 0 & 0 & 1+2i \end{bmatrix} \text{ and } B = \begin{bmatrix} 1+2i & 5+i & 0 & 0 \\ 4+i & 1+3i & 2+3i & 0 \\ 0 & 2+3i & 3+2i & 4+2i \\ 0 & 0 & 2+6i & 1+4i \end{bmatrix}$$

```
#include <imsl.h>

void main()
{
        f_complex a[] =
                        {{0.0, 0.0}, {1.0, 3.0}, {3.0, 1.0}, {2.0, 5.0},
                         {2.0, 3.0}, {6.0, 2.0}, {4.0, 1.0}, {1.0, 2.0}};
        f_complex b[] =
                        {{0.0, 0.0}, {5.0, 1.0}, {2.0, 3.0}, {4.0, 2.0},
                         {1.0, 2.0}, {1.0, 3.0}, {3.0, 2.0}, {1.0, 4.0},
                         {4.0, 1.0}, {2.0, 3.0}, {2.0, 6.0}, {0.0, 0.0}};
        int        nuca = 1, nlca = 0;
        int        nucb = 1, nlcb = 1;
        int        n = 4, m, nlcc, nucc;
        f_complex *c;
        f_complex  alpha = {3.0, 2.0};
        f_complex  beta = {4.0, 1.0};
        c = imsl_c_mat_add_band(n, nlca, nuca, alpha, a,
                                nlcb, nucb, beta, b,
```

```
                                &nlcc, &nucc,
                                IMSL_A_CONJUGATE_TRANSPOSE,
                                IMSL_B_CONJUGATE_TRANSPOSE, 0);

        m = nlcc + nucc + 1;
        imsl_c_write_matrix("C = (3+2i)*ctrans(A) + (4+i)*ctrans(B)\n",
                m, n, c, 0);
        free(c);
}
```

Output

```
                C = (3+2i)*ctrans(A) + (4+i)*ctrans(B)

                     1                          2                          3
1 (         0,         0)  (        17,         0)  (        11,       -10)
2 (        18,       -12)  (        29,        -5)  (        28,         0)
3 (        30,        -6)  (        22,        -7)  (        34,       -15)

                     4
1 (        14,       -22)
2 (        15,       -19)
3 (         0,         0)
```

mat_add_coordinate

Performs element-wise addition on two real matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.

Synopsis

#include `<imsl.h>`

Imsl_f_sparse_elem `*imsl_f_mat_add_coordinate` (*int* `n`, *int* `nz_a`, *float* `alpha`, *Imsl_f_sparse_elem* `a[]`, *int* `nz_b`, *float* `beta`, *Imsl_f_sparse_elem* `b[]`, *int* `*nz_c`, ..., 0)

The type *double* function is `imsl_d_mat_add_coordinate`.

Required Arguments

int `n` (Input)
The order of the matrices *A* and *B*.

int `nz_a` (Input)
Number of nonzeros in the matrix *A*.

float alpha (Input)
Scalar multiplier for *A*.

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

int `nz_b` (Input)
Number of nonzeros in the matrix *B*.

float `beta` (Input)
Scalar multiplier for *B*.

Imsl_f_sparse_elem `b[]` (Input)
Vector of length `nz_b` containing the location and value of each nonzero entry in the matrix *B*.

int `*nz_c` (Output)
The number of nonzeros in the sum $\alpha A + \beta B$.

Return Value

A pointer to an array of type *Imsl_f_sparse_elem* containing the computed sum. `In` the event of an error or if the return matrix has no nonzero elements, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_f_sparse_elem `*imsl_f_mat_add_coordinate` (*int* `n`, *int* `nz_a`, *float* `alpha`, *Imsl_f_sparse_elem* `a[]`, *int* `nz_b`, *float* `beta`, *Imsl_f_sparse_elem* `b[]`, *int* `*nz_c`,
`IMSL_A_TRANSPOSE`,
`IMSL_B_TRANSPOSE`,
0)

Optional Arguments

`IMSL_A_TRANSPOSE`,
Replace *A* with A^T in the expression $\alpha A + \beta B$.

`IMSL_B_TRANSPOSE`,
Replace *B* with B^T in the expression $\alpha A + \beta B$.

Description

The function `imsl_f_mat_add_coordinate` forms the sum $\alpha A + \beta B$, given the scalars α and β, and the matrices *A* and *B* in coordinate format. The transpose of *A* and/or *B* may be used during the computation if optional arguments are specified. The method starts by storing *A* in a linked list data structure, and performs the multiply by α. Next the data in matrix *B* is traversed and if the coordinates of a nonzero element correspond to those of a nonzero element in *A*, that entry in the linked list is updated. Otherwise, a new node in the linked list is created. The multiply by β occurs at this time. Lastly, the linked list representation of *C* is converted to coordinate representation, omitting any elements that may have become zero through cancellation.

Examples

Example 1

Add two real matrices of order 4 stored in coordinate format. Matrix *A* has five nonzero elements. Matrix *B* has seven nonzero elements.

```
#include <imsl.h>

void main ()
{
        Imsl_f_sparse_elem a[] = {0, 0, 3,
                                  0, 3, -1,
                                  1, 2, 5,
                                  2, 0, 1,
                                  3, 1, 3};
        Imsl_f_sparse_elem b[] = {0, 1, -2,
                                  0, 3, 1,
                                  1, 0, 3,
                                  2, 2, 5,
                                  2, 3, 1,
                                  3, 0, 4,
                                  3, 1, 3};
        int                       nz_a = 5, nz_b = 7, nz_c;
        int                       n = 4, i;
        float                     alpha = 1.0, beta = 1.0;
        Imsl_f_sparse_elem        *c;

        c = imsl_f_mat_add_coordinate(n, nz_a, alpha, a,
                                  nz_b, beta, b, &nz_c, 0);

        printf(" row  column  value\n");
        for (i = 0; i < nz_c; i++)
             printf("%3d %5d %8.2f\n", c[i].row, c[i].col, c[i].val);

        free(c);
}
```

Output

```
 row  column  value
  0     0     3.00
  0     1    -2.00
  1     0     3.00
  1     2     5.00
  2     0     1.00
  2     2     5.00
  2     3     1.00
  3     0     4.00
  3     1     6.00
```

Example 2

Compute $2*A^T + 2*B^T$, where

$$A = \begin{bmatrix} 3 & 0 & 0 & -1 \\ 0 & 0 & 5 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & -2 & 0 & 1 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 5 & 1 \\ 4 & 3 & 0 & 0 \end{bmatrix}$$

```
#include <imsl.h>

void main ()
{
        Imsl_f_sparse_elem a[] = {0, 0, 3,
```

```
                                0, 3, -1,
                                1, 2, 5,
                                2, 0, 1,
                                3, 1, 3};
        Imsl_f_sparse_elem b[] = {0, 1, -2,
                                0, 3, 1,
                                1, 0, 3,
                                2, 2, 5,
                                2, 3, 1,
                                3, 0, 4,
                                3, 1, 3};
        int                     nz_a = 5, nz_b = 7, nz_c;
        int                     n = 4, i;
        float                   alpha = 2.0, beta = 2.0;
        Imsl_f_sparse_elem      *c;


        c = imsl_f_mat_add_coordinate(n, nz_a, alpha, a,
                              nz_b, beta, b, &nz_c,
                              IMSL_A_TRANSPOSE,
                              IMSL_B_TRANSPOSE, 0);

        printf(" row  column  value\n");
        for (i = 0; i < nz_c; i++)
             printf("%3d %5d %8.2f\n", c[i].row, c[i].col, c[i].val);

        free(c);
}
```

Output

```
 row  column  value
  0     0     6.00
  0     1     6.00
  0     2     2.00
  0     3     8.00
  1     0    -4.00
  1     3    12.00
  2     1    10.00
  2     2    10.00
  3     2     2.00
```

mat_add_coordinate (complex)

Performs element-wise addition on two complex matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.

Synopsis

#include `<imsl.h>`

Imsl_c_sparse_elem `*imsl_c_mat_add_coordinate` (*int* `n`, *int* `nz_a`, *f_complex* `alpha`, *Imsl_c_sparse_elem* `a[]`, *int* `nz_b`, *f_complex* `beta`, *Imsl_c_sparse_elem* `b[]`, *int* `*nz_c`, ..., 0)

The type *double* function is `imsl_z_mat_add_coordinate`.

Required Arguments

int `n` (Input)
The order of the matrices *A* and *B*.

int `nz_a` (Input)
Number of nonzeros in the matrix *A*.

f_complex `alpha` (Input)
Scalar multiplier for *A*.

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

int `nz_b` (Input)
Number of nonzeros in the matrix *B*.

f_complex `beta` (Input)
Scalar multiplier for *B*.

Imsl_c_sparse_elem `b[]` (Input)
Vector of length `nz_b` containing the location and value of each nonzero entry in the matrix *B*.

int `*nz_c` (Output)
The number of nonzeros in the sum $\alpha A + \beta B$.

Return Value

A pointer to an array of type *Imsl_c_sparse_elem* containing the computed sum. In the event of an error or if the return matrix has no nonzero elements, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

Imsl_c_sparse_elem `*imsl_c_mat_add_coordinate` (*int* `n`, *int* `nz_a`,
f_complex `alpha`, *Imsl_c_sparse_elem* `a[]`, *int* `nz_b`, *f_complex* `beta`,
Imsl_c_sparse_elem `b[]`, *int* `*nz_c`,
`IMSL_A_TRANSPOSE`,
`IMSL_B_TRANSPOSE`,
`IMSL_A_CONJUGATE_TRANSPOSE`,
`IMSL_B_CONJUGATE_TRANSPOSE`,
0)

Optional Arguments

`IMSL_A_TRANSPOSE`,
Replace *A* with A^T in the expression $\alpha A + \beta B$.

`IMSL_B_TRANSPOSE`,
Replace *B* with B^T in the expression $\alpha A + \beta B$.

`IMSL_A_CONJUGATE_TRANSPOSE`,
Replace A with A^H in the expression $\alpha A + \beta B$.

`IMSL_B_CONJUGATE_TRANSPOSE`,
Replace B with B^H in the expression $\alpha A + \beta B$.

Description

The function `imsl_c_mat_add_coordinate` forms the sum $\alpha A + \beta B$, given the scalars α and β, and the matrices A and B in coordinate format. The transpose or conjugate transpose of A and/or B may be used during the computation if optional arguments are specified. The method starts by storing A in a linked list data structure, and performs the multiply by α. Next the data in matrix B is traversed and if the coordinates of a nonzero element correspond to those of a nonzero element in A, that entry in the linked list is updated. Otherwise, a new node in the linked list is created. The multiply by β occurs at this time. Lastly, the linked list representation of C is converted to coordinate representation, omitting any elements that may have become zero through cancellation.

Examples

Example 1

Add two complex matrices of order 4 stored in coordinate format. Matrix A has five nonzero elements. Matrix B has seven nonzero elements.

```
#include <imsl.h>

void main ()
{
        Imsl_c_sparse_elem a[] = {0, 0, 3, 4,
                                  0, 3, -1, 2,
                                  1, 2, 5, -1,
                                  2, 0, 1, 2,
                                  3, 1, 3, 0};
        Imsl_c_sparse_elem b[] = {0, 1, -2, 1,
                                  0, 3, 1, -2,
                                  1, 0, 3, 0,
                                  2, 2, 5, 2,
                                  2, 3, 1, 4,
                                  3, 0, 4, 0,
                                  3, 1, 3, -2};
        int                       nz_a = 5, nz_b = 7, nz_c;
        int                       n = 4, i;
        f_complex                 alpha = {1.0, 0.0}, beta = {1.0, 0.0};
        Imsl_c_sparse_elem        *c;

        c = imsl_c_mat_add_coordinate(n, nz_a, alpha, a,
                                 nz_b, beta, b, &nz_c, 0);

        printf(" row  column     value\n");
        for (i = 0; i < nz_c; i++)
             printf("%3d %5d %8.2f %8.2f\n",
                    c[i].row, c[i].col, c[i].val.re, c[i].val.im);
```

```
        free(c);
}
```

Output

```
 row  column     value
  0     0     3.00     4.00
  0     1    -2.00     1.00
  1     0     3.00     0.00
  1     2     5.00    -1.00
  2     0     1.00     2.00
  2     2     5.00     2.00
  2     3     1.00     4.00
  3     0     4.00     0.00
  3     1     6.00    -2.00
```

Example 2

Compute $2+3i*A^T + 2-i*B^T$, where

$$A = \begin{bmatrix} 3+4i & 0 & 0 & -1+2i \\ 0 & 0 & 5-i & 0 \\ 1+2i & 0 & 0 & 0 \\ 0 & 3+0i & 0 & 0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & -2+i & 0 & 1-2i \\ 3+0i & 0 & 0 & 0 \\ 0 & 0 & 5+2i & 1+4i \\ 4+0i & 3-2i & 0 & 0 \end{bmatrix}$$

```
#include <imsl.h>

void main ()
{
        Imsl_c_sparse_elem a[] = {0, 0, 3, 4,
                                  0, 3, -1, 2,
                                  1, 2, 5, -1,
                                  2, 0, 1, 2,
                                  3, 1, 3, 0};
        Imsl_c_sparse_elem b[] = {0, 1, -2, 1,
                                  0, 3, 1, -2,
                                  1, 0, 3, 0,
                                  2, 2, 5, 2,
                                  2, 3, 1, 4,
                                  3, 0, 4, 0,
                                  3, 1, 3, -2};
        int                       nz_a = 5, nz_b = 7, nz_c;
        int                       n = 4, i;
        f_complex                 alpha = {2.0, 3.0}, beta = {2.0, -1.0};
        Imsl_c_sparse_elem        *c;

        c = imsl_c_mat_add_coordinate(n, nz_a, alpha, a,
                                 nz_b, beta, b, &nz_c,
                                 IMSL_A_TRANSPOSE,
                                 IMSL_B_TRANSPOSE, 0);

        printf(" row  column     value\n");
        for (i = 0; i < nz_c; i++)
             printf("%3d %5d %8.2f %8.2f\n",
                    c[i].row, c[i].col, c[i].val.re, c[i].val.im);
```

```
        free(c);
}
```

Output

```
 row  column     value
  0     0    -6.00    17.00
  0     1     6.00    -3.00
  0     2    -4.00     7.00
  0     3     8.00    -4.00
  1     0    -3.00     4.00
  1     3    10.00     2.00
  2     1    13.00    13.00
  2     2    12.00    -1.00
  3     0    -8.00    -4.00
  3     2     6.00     7.00
```

matrix_norm

Computes various norms of a rectangular matrix.

Synopsis

#include `<imsl.h>`

float `imsl_f_matrix_norm` (*int* `m`, *int* `n`, *float* `a[]`, ..., 0)

The type *double* function is `imsl_d_matrix_norm`.

Required Arguments

int `m` (Input)
The number of rows in matrix *A*.

int `n` (Input)
The number of columns in matrix *A*.

float `a[]` (Input)
Matrix for which the norm will be computed.

Return Value

The requested norm of the input matrix. If the norm cannot be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_matrix_norm` (*int* `m`, *int* `n`, *float* `a[]`,
`IMSL_ONE_NORM`,
`IMSL_INF_NORM`,
0)

Description

By default, `imsl_f_matrix_norm` computes the Frobenius norm

$$\|A\|_2 = \left[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{ij}^2\right]^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} |A_{ij}|$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\|A\|_\infty = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} |A_{ij}|$$

is returned.

Example

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*.

```
#include <imsl.h>

void main()
{
        float a[] = {1.0, 2.0, -2.0, 3.0,
                     -2.0, 1.0, 3.0, 0.0,
                     0.0, 3.0, 1.0, -7.0,
                     5.0, -2.0, 7.0, 6.0,
                     4.0, 3.0, 4.0, 0.0};
        int          m = 5, n = 4;
        float        frobenius_norm, inf_norm, one_norm;

        frobenius_norm = imsl_f_matrix_norm(m, n, a, 0);

        inf_norm = imsl_f_matrix_norm(m, n, a, IMSL_INF_NORM, 0);

        one_norm = imsl_f_matrix_norm(m, n, a, IMSL_ONE_NORM, 0);

        printf("Frobenius norm = %f\n", frobenius_norm);
        printf("Infinity norm  = %f\n", inf_norm);
        printf("One norm       = %f\n", one_norm);
}
```

Output

```
Frobenius norm = 15.684387
Infinity norm  = 20.000000
One norm       = 17.000000
```

matrix_norm_band

Computes various norms of a matrix stored in band storage mode.

Synopsis

#include `<imsl.h>`

float `imsl_f_matrix_norm_band` (*int* `n`, *float* `a[]`, *int* `nlc`, *int* `nuc`, ..., 0)

The type *double* function is `imsl_d_matrix_norm_band`.

Required Arguments

int `n` (Input)
The order of matrix *A*.

float `a[]` (Input)
Matrix for which the norm will be computed.

int `nlc` (Input)
Number of lower codiagonals of *A*.

int `nuc` (Input)
Number of upper codiagonals of *A*.

Return Value

The requested norm of the input matrix, by default, the Frobenius norm. If the norm cannot be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_matrix_norm_band` (*int* `n`, *float* `a[]`, *int* `nlc`, *int* `nuc`,
`IMSL_ONE_NORM`,
`IMSL_INF_NORM`,
`IMSL_SYMMETRIC`,
0)

Optional Arguments

`IMSL_ONE_NORM`,
Compute the 1-norm of matrix *A*,

`IMSL_INF_NORM`,
Compute the infinity norm of matrix *A*,

`IMSL_SYMMETRIC`,
Matrix *A* is stored in band symmetric storage mode.

Description

By default, `imsl_f_matrix_norm_band` computes the Frobenius norm

$$\|A\|_2 = \left[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{ij}^2\right]^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} |A_{ij}|$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\|A\|_\infty = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} |A_{ij}|$$

is returned.

Examples

Example 1

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in band storage mode.

```
#include <imsl.h>

void main()
{
        float a[] = {0.0, 2.0, 3.0, -1.0,
                     1.0, 1.0, 1.0, 1.0,
                     0.0, 3.0, 4.0, 0.0};
        int          nlc = 1, nuc = 1;
        int          n = 4;
        float        frobenius_norm, inf_norm, one_norm;

        frobenius_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc, 0);

        inf_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc,
                                    IMSL_INF_NORM, 0);

        one_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc,
                                    IMSL_ONE_NORM, 0);

        printf("Frobenius norm = %f\n", frobenius_norm);
        printf("Infinity norm  = %f\n", inf_norm);
        printf("One norm       = %f\n", one_norm);
}
```

Output

```
Frobenius norm = 6.557438
```

```
Infinity norm  = 5.000000
One norm       = 8.000000
```

Example 2

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in symmetric band storage mode.

```
#include <imsl.h>

void main()
{
        float a[] = {0.0, 0.0, 7.0, 3.0, 1.0, 4.0,
                     0.0, 5.0, 1.0, 2.0, 1.0, 2.0,
                     1.0, 2.0, 4.0, 6.0, 3.0, 1.0};
        int          nlc = 2, nuc = 2;
        int          n = 6;
        float        frobenius_norm, inf_norm, one_norm;

        frobenius_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc,
                                    IMSL_SYMMETRIC, 0);

        inf_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc,
                                    IMSL_INF_NORM,
                                    IMSL_SYMMETRIC, 0);

        one_norm = imsl_f_matrix_norm_band(n, a, nlc, nuc,
                                    IMSL_ONE_NORM,
                                    IMSL_SYMMETRIC, 0);

        printf("Frobenius norm = %f\n", frobenius_norm);
        printf("Infinity norm  = %f\n", inf_norm);
        printf("One norm       = %f\n", one_norm);
}
```

Output

```
Frobenius norm = 16.941074
Infinity norm  = 16.000000
One norm       = 16.000000
```

matrix_norm_coordinate

Computes various norms of a matrix stored in coordinate format.

Synopsis

#include `<imsl.h>`

float `imsl_f_matrix_norm_coordinate` (*int* `m`, *int* `n`, *int* `nz`,
Imsl_f_sparse_elem `a[]`, ..., 0)

The type *double* function is `imsl_d_matrix_norm_coordinate`.

Required Arguments

int `m` (Input)
The number of rows in matrix *A*.

int `n` (Input)
The number of columns in matrix *A*.

int `nz` (Input)
The number of nonzeros in the matrix *A*.

Imsl_f_sparse_elem `a[]` (Input)
Matrix for which the norm will be computed.

Return Value

The requested norm of the input matrix, by default, the Frobenius norm. If the norm cannot be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_matrix_norm_coordinate` (*int* `m`, *int* `n`, *int* `nz`,
Imsl_f_sparse_elem `a[]`,
`IMSL_ONE_NORM`,
`IMSL_INF_NORM`,
`IMSL_SYMMETRIC`,
0)

Optional Arguments

`IMSL_ONE_NORM`,
Compute the 1-norm of matrix *A*.

`IMSL_INF_NORM`,
Compute the infinity norm of matrix *A*.

`IMSL_SYMMETRIC`,
Matrix *A* is stored in symmetric coordinate format.

Description

By default, `imsl_f_matrix_norm_coordinate` computes the Frobenius norm

$$\|A\|_2 = \left[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{ij}^2\right]^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} \left|A_{ij}\right|$$

is returned. If the option IMSL_INF_NORM is selected, the infinity norm

$$\|A\|_{\infty} = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} |A_{ij}|$$

is returned.

Examples

Example 1

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in coordinate format.

```
#include <imsl.h>

void 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};
        int                       m = 6, n = 6;
        int                       nz = 15;
        float                     frobenius_norm, inf_norm, one_norm;

        frobenius_norm = imsl_f_matrix_norm_coordinate (m, n, nz, a, 0);

        inf_norm = imsl_f_matrix_norm_coordinate(m, n, nz, a,
                                    IMSL_INF_NORM, 0);

        one_norm = imsl_f_matrix_norm_coordinate(m, n, nz, a,
                                    IMSL_ONE_NORM, 0);

        printf("Frobenius norm = %f\n", frobenius_norm);
        printf("Infinity norm  = %f\n", inf_norm);
        printf("One norm       = %f\n", one_norm);
}
```

Output

```
Frobenius norm = 24.839485
Infinity norm  = 15.000000
One norm       = 18.000000
```

Example 2

Compute the Frobenius norm, infinity norm and one norm of matrix *A*. Matrix *A* is stored in symmetric coordinate format.

```
#include <imsl.h>

void main()
{
        Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                                  0, 2, -1.0,
                                  0, 5, 5.0,
                                  1, 3, 2.0,
                                  1, 4, 3.0,
                                  2, 2, 3.0,
                                  2, 5, 4.0,
                                  4, 4, -1.0,
                                  4, 5, 4.0};
        int                       m = 6, n = 6;
        int                       nz = 9;
        float                     frobenius_norm, inf_norm, one_norm;

        frobenius_norm = imsl_f_matrix_norm_coordinate (m, n, nz, a,
                                    IMSL_SYMMETRIC, 0);

        inf_norm = imsl_f_matrix_norm_coordinate(m, n, nz, a,
                                    IMSL_INF_NORM,
                                    IMSL_SYMMETRIC, 0);

        one_norm = imsl_f_matrix_norm_coordinate(m, n, nz, a,
                                     IMSL_ONE_NORM,
                                     IMSL_SYMMETRIC, 0);

        printf("Frobenius norm = %f\n", frobenius_norm);
        printf("Infinity norm  = %f\n", inf_norm);
        printf("One norm       = %f\n", one_norm);
}
```

Output

```
Frobenius norm = 15.874508
Infinity norm  = 16.000000
One norm       = 16.000000
```

generate_test_band

Generates test matrices of class and $E(n, c)$. Returns in band or band symmetric format.

Synopsis

#include `<imsl.h>`

float `*imsl_f_generate_test_band` (*int* `n`, *int* `c`, ..., 0)

The function `imsl_d_generate_test_band` is the *double* precision analogue.

Required Arguments

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

int c (Input)
Parameter used to alter structure, also the number of upper/lower codiagonals.

Return Value

A pointer to a vector of type *float*. To release this space, use `free`. If no test was generated, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsl.h>

void *imsl_f_generate_sparse_test (int n, int c,
        IMSL_SYMMETRIC_STORAGE,
        0)
```

Optional Arguments

`IMSL_SYMMETRIC_STORAGE`,
Return matrix stored in band symmetric format.

Description

The same nomenclature as Østerby and Zlatev (1982) is used. Test matrices of class $E(n, c)$, to which we will generally refer to as E-matrices, are symmetric, positive definite matrices of order n with 4 in the diagonal and −1 in the superdiagonal and subdiagonal. In addition there are two bands with −1 at a distance c from the diagonal. More precisely:

$$a_{i,i} = 4 \qquad 0 \le i < n$$

$$a_{i,i+1} = -1 \qquad 0 \le i < n - 1$$

$$a_{i+1,1} = -1 \qquad 0 \le i < n - 1$$

$$a_{i,i+c} = -1 \qquad 0 \le i < n - c$$

$$a_{i+c,i} = -1 \qquad 0 \le i < n - c$$

for any $n \ge 3$ and $2 \le c \le n - 1$.

E-matrices are similar to those obtained from the five-point formula in the discretization of elliptic partial differential equations.

By default, `imsl_f_generate_test_band` returns an E-matrix in band storage mode. Option `IMSL_SYMMETRIC_STORAGE` returns a matrix in band symmetric storage mode.

Example

This example generates the matrix

$$E(5,3) = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 \\ -1 & 4 & -1 & 0 & -1 \\ 0 & -1 & 4 & -1 & 0 \\ -1 & 0 & -1 & 4 & -1 \\ 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

and prints the result.

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        float *a;

        a = imsl_f_generate_test_band (n, c, 0);

        imsl_f_write_matrix ("E(5,3) in band storage", 2*c + 1, n,
                a, 0);
}
```

Output

```
                E(5,3) in band storage
             1           2           3           4           5
1            0           0           0          -1          -1
2            0           0           0           0           0
3            0          -1          -1          -1          -1
4            4           4           4           4           4
5           -1          -1          -1          -1           0
6            0           0           0           0           0
7           -1          -1           0           0           0
```

generate_test_band (complex)

Generates test matrices of class $E_c(n, c)$. Returns in band or band symmetric format.

Synopsis

#include `<imsl.h>`

f_complex `*imsl_c_generate_test_band` (*int* `n`, *int* `c`, ..., 0)

The function `imsl_z_generate_test_band` is the double precision analogue.

Required Arguments

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

int `c` (Input)
Parameter used to alter structure, also the number of upper/lower codiagonals

Return Value

A pointer to a vector of type *f_*complex. To release this space, use `free`. If no test was generated, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `*imsl_c_generate_sparse_test (`*int* `n, `*int* `c,`
`IMSL_SYMMETRIC_STORAGE,`
`0)`

Optional Arguments

`IMSL_SYMMETRIC_STORAGE`,
Return matrix stored in band symmetric format.

Description

We use the same nomenclature as Østerby and Zlatev (1982). Test matrices of class $E(n, c)$, to which we will generally refer to as E-matrices, are symmetric, positive definite matrices of order `n` with (6.0, 0.0) in the diagonal, (−1.0, 1.0) in the superdiagonal and (−1.0, −1.0) subdiagonal. In addition there are two bands at a distance `c` from the diagonal with (−1.0, 1.0) in the upper codiagonal and (−1.0, −1.0) in the lower codiagonal. More precisely:

$$a_{i,i} = 6 \qquad 0 \le i < n$$

$$a_{i,i+1} = -1 - i \qquad 0 \le i < n - 1$$

$$a_{i+1,1} = -1 - i \qquad 0 \le i < n - 1$$

$$a_{i,i+c} = -1 + i \qquad 0 \le i < n - c$$

$$a_{i+c,i} = -1 + i \qquad 0 \le i < n - c$$

for any $n \ge 3$ and $2 \le c \le n - 1$.

E-matrices are similar to those obtained from the five-point formula in the discretization of elliptic partial differential equations.

By default, `imsl_c_generate_test_band` returns an E-matrix in band storage mode. Option `IMSL_SYMMETRIC_STORAGE` returns a matrix in band symmetric storage mode.

Example

This example generates the following matrix and prints the result:

$$E_c(5,3) = \begin{bmatrix} 6 & -1-i & 0 & -1+i & 0 \\ -1-i & 6 & -1+i & 0 & -1+i \\ 0 & -1-i & 6 & -1+i & 0 \\ -1-i & 0 & -1-i & 6 & -1+i \\ 0 & -1-i & 0 & -1-i & 6 \end{bmatrix}$$

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        f_complex *a;

        a = imsl_c_generate_test_band (n, c, 0);

        imsl_c_write_matrix ("E(5,3) in band storage", 2*c + 1, n,
                a, 0);
}
```

Output

```
                          E(5,3) in band storage
                           1                          2                          3
1 (         0,         0)  (         0,         0)  (         0,         0)
2 (         0,         0)  (         0,         0)  (         0,         0)
3 (         0,         0)  (        -1,         1)  (        -1,         1)
4 (         6,         0)  (         6,         0)  (         6,         0)
5 (        -1,        -1)  (        -1,        -1)  (        -1,        -1)
6 (         0,         0)  (         0,         0)  (         0,         0)
7 (        -1,        -1)  (        -1,        -1)  (         0,         0)

                           4                          5
1 (        -1,         1)  (        -1,         1)
2 (         0,         0)  (         0,         0)
3 (        -1,         1)  (        -1,         1)
4 (         6,         0)  (         6,         0)
5 (        -1,        -1)  (         0,         0)
6 (         0,         0)  (         0,         0)
7 (         0,         0)  (         0,         0)
```

generate_test_coordinate

Generates test matrices of class $D(n, c)$ and $E(n, c)$. Returns in either coordinate format.

Synopsis

#include `<imsl.h>`

Imsl_f_sparse_elem `*imsl_f_generate_test_coordinate` (*int* `n`, *int* `c`, *int* `*nz`, ..., 0)

The function `imsl_d_generate_test_coordinate` is the *double* precision analogue.

Required Arguments

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

int `c` (Input)
Parameter used to alter structure.

int `*nz` (Output)
Length of the return vector.

Return Value

A pointer to a vector of length `nz` of type *Imsl_f_sparse_elem*. To release this space, use `free`. If no test was generated, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

void `*imsl_f_generate_test_coordinate` (*int* `n`, *int* `c`, *int* `*nz`,
`IMSL_D_MATRIX`,
`IMSL_SYMMETRIC_STORAGE`,
0)

Optional Arguments

`IMSL_D_MATRIX`
Return a matrix of class $D(n, c)$.
Default: Return a matrix of class $E(n, c)$.

`IMSL_SYMMETRIC_STORAGE`,
For coordinate representation, return only values for the diagonal and lower triangle. This option is not allowed if `IMSL_D_MATRIX` is specified.

Description

We use the same nomenclature as Østerby and Zlatev (1982).Test matrices of class $E(n, c)$, to which we will generally refer to as E-matrices, are symmetric, positive definite matrices of order `n` with 4 in the diagonal and −1 in the superdiagonal and subdiagonal. In addition there are two bands with −1 at a distance `c` from the diagonal. More precisely

$$
\begin{aligned}
a_{i,i} &= 4 && 0 \le i < n \\
a_{i,i+1} &= -1 && 0 \le i < n-1 \\
a_{i+1,1} &= -1 && 0 \le i < n-1 \\
a_{i,i+c} &= -1 && 0 \le i < n-c \\
a_{i+c,i} &= -1 && 0 \le i < n-c
\end{aligned}
$$

for any $n \ge 3$ and $2 \le c \le n-1$.

E-matrices are similar to those obtained from the five-point formula in the discretization of elliptic partial differential equations.

Test matrices of class $D(n, c)$ are square matrices of order `n` with a full diagonal, three bands at a distance `c` above the diagonal and reappearing cyclically under the diagonal, and a 10×10 triangle of elements in the upper right corner. More precisely:

$$
\begin{aligned}
a_{i,i} &= 1 && 0 \le i < n \\
a_{i,i+c} &= i + 2 && 0 \le i < n - c \\
a_{i,i-n+c} &= i + 2 && n - c \le i < n \\
a_{i,i+c+1} &= -(i + 1) && 0 \le i < n - c - 1 \\
a_{i,i-n+c+1} &= -(i + 1) && n - c - 1 \le i < n \\
a_{i,i+c+2} &= 16 && 0 \le i < n - c - 2 \\
a_{i,i-n+c+2} &= 16 && n - c - 2 \le i < n \\
a_{i,n-11+i+j} &= 100j && 1 \le i < 11 - j, \quad 0 \le j < 10
\end{aligned}
$$

for any $n \ge 14$ and $1 \le c \le n - 13$.

We now show the sparsity pattern of $D(20, 5)$

x					x	x	x			x	x	x	x	x	x	x	x	x	x
	x					x	x	x			x	x	x	x	x	x	x	x	x
		x					x	x	x			x	x	x	x	x	x	x	x
			x					x	x	x			x	x	x	x	x	x	x
				x					x	x	x			x	x	x	x	x	x
					x					x	x	x			x	x	x	x	x
						x					x	x	x			x	x	x	x
							x					x	x	x			x	x	x
								x					x	x	x			x	x
									x					x	x	x			x
										x					x	x	x		
											x					x	x	x	
												x					x	x	x
x													x					x	x
x	x													x					x
x	x	x													x				
	x	x	x													x			
		x	x	x													x		
			x	x	x													x	
				x	x	x													x

By default `imsl_f_generate_test_coordinate` returns an *E*-matrix in coordinate representation. By specifying the `IMSL_SYMMETRIC_STORAGE` option, only the diagonal and lower triangle are returned. The scalar `nz` will contain the number of nonzeros in this representation.

The option `IMSL_D_MATRIX` will return a matrix of class $D(n, c)$. Since D-matrices are not symmetric, the `IMSL_SYMMETRIC_STORAGE` option is not allowed.

Examples

Example 1

This example generates the matrix

$$E(5,3) = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 \\ -1 & 4 & -1 & 0 & -1 \\ 0 & -1 & 4 & -1 & 0 \\ -1 & 0 & -1 & 4 & -1 \\ 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

and prints the result.

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        int nz;
        Imsl_f_sparse_elem *a;

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

        printf ("row    col    val\n");
        for (i=0; i<nz; i++)
                printf (" %d      %d   %5.1f\n",
                        a[i].row, a[i].col, a[i].val);
}
```

Output

```
row    col    val
 0      0     4.0
 1      1     4.0
 2      2     4.0
 3      3     4.0
 4      4     4.0
 1      0    -1.0
 2      1    -1.0
 3      2    -1.0
 4      3    -1.0
 0      1    -1.0
 1      2    -1.0
 2      3    -1.0
 3      4    -1.0
 3      0    -1.0
 4      1    -1.0
 0      3    -1.0
 1      4    -1.0
```

Example 2

In this example, the matrix $E(5, 3)$ is returned in symmetric storage and printed.

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        int nz;
        Imsl_f_sparse_elem *a;

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

        printf ("row    col    val\n");
        for (i=0; i<nz; i++)
                printf (" %d      %d   %5.1f\n",
                        a[i].row, a[i].col, a[i].val);
}
```

Output

```
row    col    val
 0      0     4.0
 1      1     4.0
 2      2     4.0
 3      3     4.0
 4      4     4.0
 1      0    -1.0
 2      1    -1.0
 3      2    -1.0
 4      3    -1.0
 3      0    -1.0
 4      1    -1.0
```

generate_test_coordinate (complex)

Generates test matrices of class $D(n, c)$ and $E(n, c)$. Returns in either coordinate or band storage format, where possible.

Synopsis

#include `<imsl.h>`

void `*imsl_c_generate_test_coordinate` (*int* `n`, *int* `c`, *int* `*nz`, ..., 0)

The function is `imsl_z_generate_test_coordinate` is the *double* precision analogue.

Required Arguments

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

int c (Input)
Parameter used to alter structure.

int *nz (Output)
Length of the return vector.

Return Value

A pointer to a vector of length nz of type *imsl_c_sparse_elem*. To release this space, use free. If no test was generated, then NULL is returned.

Synopsis with Optional Arguments

#include <imsl.h>

void *imsl_c_generate_test_coordinate (*int* n, *int* c, *int* *nz,
IMSL_D_MATRIX,
IMSL_SYMMETRIC_STORAGE,
0)

Optional Arguments

IMSL_D_MATRIX
Return a matrix of class $D(n, c)$.
Default: Return a matrix of class $E(n, c)$.

IMSL_SYMMETRIC_STORAGE,
For coordinate representation, return only values for the diagonal and lower triangle. This option is not allowed if IMSL_D_MATRIX is specified.

Description

The same nomenclature as Østerby and Zlatev (1982) is used. Test matrices of class $E(n, c)$, to which we will generally refer to as *E*-matrices, are symmetric, positive definite matrices of order n with (6.0, 0.0) in the diagonal, (−1.0, 1.0) in the superdiagonal and (−1.0, −1.0) subdiagonal. In addition there are two bands at a distance c from the diagonal with (−1.0, 1.0) in the upper codiagonal and (−1.0, −1.0) in the lower codiagonal. More precisely:

$$a_{i,i} = 6 \qquad 0 \le i < n$$
$$a_{i,i+1} = -1 - i \qquad 0 \le i < n - 1$$
$$a_{i+1,1} = -1 - i \qquad 0 \le i < n - 1$$
$$a_{i,i+c} = -1 + i \qquad 0 \le i < n - c$$
$$a_{i+c,i} = -1 + i \qquad 0 \le i < n - c$$

for any $n \ge 3$ and $2 \le c \le n - 1$.

Test matrices of class $D(n, c)$ are square matrices of order n with a full diagonal, three bands at a distance c above the diagonal and reappearing cyclically under the diagonal, and a 10×10 triangle of elements in the upper-right corner. More precisely:

$$
\begin{aligned}
a_{i,i} &= 1 && 0 \le i < n \\
a_{i,i+c} &= i + 2 && 0 \le i < n - c \\
a_{i,i-n+c} &= i + 2 && n - c \le i < n \\
a_{i,i+c+1} &= -(i + 1) && 0 \le i < n - c - 1 \\
a_{i,i+c+1} &= -(i + 1) && n - c - 1 \le i < n \\
a_{i,i+c+2} &= 16 && 0 \le i < n - c - 2 \\
a_{i,i-n+c+2} &= 16 && n - c - 2 \le i < n \\
a_{i,n-11+i+j} &= 100j && 1 \le i < 11 - j, \quad 0 \le j < 10
\end{aligned}
$$

for any $n \ge 14$ and $1 \le c \le n - 13$.

The sparsity pattern of D(20, 5) is as follows:

x					x	x	x			x	x	x	x	x	x	x	x	x	x
	x					x	x	x			x	x	x	x	x	x	x	x	x
		x					x	x	x			x	x	x	x	x	x	x	x
			x					x	x	x			x	x	x	x	x	x	x
				x					x	x	x			x	x	x	x	x	x
					x					x	x	x			x	x	x	x	x
						x					x	x	x			x	x	x	x
							x					x	x	x			x	x	x
								x					x	x	x			x	x
									x					x	x	x			x
										x					x	x	x		
											x					x	x	x	
												x					x	x	x
x													x					x	x
x	x													x					x
x	x	x													x				
	x	x	x													x			
		x	x	x													x		
			x	x	x													x	
				x	x	x													x

By default `imsl_c_generate_test_coordinate` returns an *E*-matrix in coordinate representation. By specifying the `IMSL_SYMMETRIC_STORAGE` option, only the diagonal and lower triangle are returned. The scalar `nz` will contain the number of non-zeros in this representation.

The option `IMSL_D_MATRIX` will return a matrix of class $D(n, c)$. Since *D*-matrices are not symmetric, the `IMSL_SYMMETRIC_STORAGE` option is not allowed.

Examples

Example 1

This example generates the matrix

$$E_c(5,3) = \begin{bmatrix} 6 & -1-i & 0 & -1+i & 0 \\ -1-i & 6 & -1-i & 0 & -1+i \\ 0 & -1-i & 6 & -1-i & 0 \\ -1-i & 0 & -1-i & 6 & -1+i \\ 0 & -1-i & 0 & -1-i & 6 \end{bmatrix}$$

and prints the result.

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        int nz;
        Imsl_c_sparse_elem *a;

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

        printf ("row    col    val\n");
        for (i=0; i<nz; i++)
                printf (" %d      %d   (%5.1f, %5.1f)\n",
                        a[i].row, a[i].col, a[i].val.re, a[i].val.im);
}
```

Output

```
row    col    val
 0      0   (  6.0,   0.0)
 1      1   (  6.0,   0.0)
 2      2   (  6.0,   0.0)
 3      3   (  6.0,   0.0)
 4      4   (  6.0,   0.0)
 1      0   ( -1.0,  -1.0)
 2      1   ( -1.0,  -1.0)
 3      2   ( -1.0,  -1.0)
 4      3   ( -1.0,  -1.0)
 0      1   ( -1.0,   1.0)
 1      2   ( -1.0,   1.0)
 2      3   ( -1.0,   1.0)
 3      4   ( -1.0,   1.0)
 3      0   ( -1.0,  -1.0)
 4      1   ( -1.0,  -1.0)
 0      3   ( -1.0,   1.0)
 1      4   ( -1.0,   1.0)
```

Example 2

In this example, the matrix E(5, 3) is returned in symmetric storage and printed.

```
#include <imsl.h>

main()
{
        int i;
        int n = 5;
        int c = 3;
        int nz;
        Imsl_c_sparse_elem *a;

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

        printf ("row    col    val\n");
        for (i=0; i<nz; i++)
                printf (" %d      %d   (%5.1f, %5.1f)\n",
                        a[i].row, a[i].col, a[i].val.re, a[i].val.im);
}
```

Output

```
row    col    val
 0      0   (  6.0,   0.0)
 1      1   (  6.0,   0.0)
 2      2   (  6.0,   0.0)
 3      3   (  6.0,   0.0)
 4      4   (  6.0,   0.0)
 1      0   ( -1.0,  -1.0)
 2      1   ( -1.0,  -1.0)
 3      2   ( -1.0,  -1.0)
 4      3   ( -1.0,  -1.0)
 3      0   ( -1.0,  -1.0)
 4      1   ( -1.0,  -1.0)
```

Reference Material

User Errors

IMSL functions attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, we recognize various levels of severity of errors, and we also consider the extent of the error in the context of the purpose of the function; a trivial error in one situation may be serious in another. Functions attempt to report as many errors as they can reasonably detect. Multiple errors present a difficult problem in error detection because input is interpreted in an uncertain context after the first error is detected.

What Determines Error Severity

In some cases, the user's input may be mathematically correct, but because of limitations of the computer arithmetic and of the algorithm used, it is not possible to compute an answer accurately. In this case, the assessed degree of accuracy determines the severity of the error. In cases where the function computes several output quantities, if some are not computable but most are, an error condition exists; and its severity depends on an assessment of the overall impact of the error.

Kinds of Errors and Default Actions

Five levels of severity of errors are defined in the IMSL C/Math/Library. Each level has an associated PRINT attribute and a STOP attribute. These attributes have default settings (YES or NO), but they may also be set by the user. The purpose of having multiple error types is to provide independent control of actions to be taken for errors of different levels of severity. Upon return from a Visual Numerics function, exactly one error state exists. (A code 0 "error" is no error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except `IMSL_TERMINAL` may be informational errors. The include file, `imsl.h`, defines `IMSL_NOTE`, `IMSL_ALERT`, `IMSL_WARNING`, `IMSL_FATAL`, `IMSL_TERMINAL`, `IMSL_WARNING_IMMEDIATE`, and `IMSL_FATAL_IMMEDIATE` as an enumerated data type `Imsl_error`.

`IMSL_NOTE`. A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations.

Default attributes: PRINT=NO, STOP=NO.

`IMSL_ALERT`. An *alert* indicates that a function value has been set to 0 due to underflow.

Default attributes: PRINT=NO, STOP=NO.

`IMSL_WARNING`. A *warning* indicates the existence of a condition that may require corrective action by the user or calling routine. A warning error may be issued because the results are accurate to only a few decimal places, because some of the output may be erroneous, but most of the output is correct, or because some assumptions underlying the analysis technique are violated. Usually no corrective action is necessary, and the condition can be ignored.

Default attributes: PRINT=YES, STOP=NO.

`IMSL_FATAL`. A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling routine must take corrective action to recover.

Default attributes: PRINT=YES, STOP=YES.

`IMSL_TERMINAL`. A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors may also be caused by various programming errors impossible to diagnose correctly in C. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the function with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error, because corrective action within the program is generally not reasonable. In normal usage, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur.

Default attributes: PRINT=YES, STOP=YES.

`IMSL_WARNING_IMMEDIATE`. An *immediate warning* error is identical to a warning error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=NO.

`IMSL_FATAL_IMMEDIATE`. An *immediate fatal* error is identical to a fatal error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=YES.

The user can set PRINT and STOP attributes by calling **`imsl_error_options`** as described in Chapter 12, "Utilities."

Errors in Lower-Level Functions

It is possible that a user's program may call an IMSL C/Math/Library function that in turn calls a nested sequence of lower-level functions. If an error occurs at a lower level in such a nest of functions, and if the lower-level function cannot pass the information up to the original user-called function, then a traceback of the functions is produced. The only common situation in which this can occur is when an IMSL C/Math/Library function calls a user-supplied routine that in turn calls another IMSL C/Math/Library function.

Functions for Error Handling

There are two ways in which the user may interact with the error handling system: (1) to change the default actions and (2) to determine the code of an informational error so as to take corrective action. The functions to use are `imsl_error_options` and `imsl_error_code`. Function `imsl_error_options` sets the actions to be taken when errors occur. Function `imsl_error_code` retrieves the integer code for an informational error. See functions **`imsl_error_options`** and **`imsl_error_code`**.

Use of Informational Error to Determine Program Action

In the program segment below, the Cholesky factorization of a matrix is to be performed. If it is determined that the matrix is not nonnegative definite (and often this is not immediately obvious), the program is to take a different branch.

```
x = imsl_f_lin_sol_nonnegdef (n, a, b, 0);
if (imsl_error_code() == IMSL_NOT_NONNEG_DEFINITE) {
       /*  Handle matrix that is not nonnegative
           definite  */
}
```

Additional Examples

See functions **`imsl_error_options`** and **`imsl_error_code`** in Chapter 12, "Utilities" for additional examples.

Complex Data Types and Functions

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

- f_complex z for single-precision complex values
- d_complex w for double-precision complex values

Each complex value is a C language *structure* that consists of a pair of real values, the *real* and *imaginary* part of the complex number. To access the real part of a single-precision complex number *z*, use the subexpression `z.re`. For the imaginary part, use the subexpression `z.im`. Use subexpressions `w.re` and `w.im` for the real and imaginary

parts of a double-precision complex number *w*. The structure is declared within `imsl.h` as follows:

```
typedef struct{
            float re;
            float im;
} f_complex;
```

Several standard operations and functions are available for users to perform calculations with complex numbers within their programs. The operations are provided for both single and double precision data types. Notice that even the ordinary arithmetic operations of "+", "-", "*", and "/" must be performed using the appropriate functions.

A uniform prefix name is used as part of the names for the operations and functions. The prefix `imsl_c_` is used for `f_complex` data. The prefix `imsl_z_` is used with `d_complex` data.

Single-Precision Complex Operations and Functions

Operation	Function Name	Function Result	Function Argument(s)
$z = -x$	`z = imsl_c_neg(x)`	`f_complex`	`f_complex`
$z = x + y$	`z = imsl_c_add(x,y)`	`f_complex`	`f_complex` (both)
$z = x - y$	`z = imsl_c_sub(x,y)`	`f_complex`	`f_complex` (both)
$z = x * y$	`z = imsl_c_mul(x,y)`	`f_complex`	`f_complex` (both)
$z = x / y$	`z = imsl_c_div(x,y)`	`f_complex`	`f_complex` (both)
$x = = y$[a]	`z = imsl_c_eq(x,y)`	`int`	`f_complex` (both)
$z = x$ *Drop Precision*	`z = imsl_cz_convert(x)`	`f_complex`	`d_complex`

[a] Result has the value 1 if *x* and *y* are valid numbers with real and imaginary parts identical; otherwise, result has the value 0.

Operation	Function Name	Function Result	Function Argument(s)
$z = a + ib$ *Ascend Data*	`z = imsl_cf_convert(a,b)`	`f_complex`	`float` (both)
$z = \bar{x}$	`z = imsl_c_conjg(x)`	`f_complex`	`f_complex`
$a = \|z\|$	`a = imsl_c_abs(z)`	`float`	`f_complex`
$a = \arg(z)$ $-\pi < a \leq \pi$	`a = imsl_c_arg(z)`	`float`	`f_complex`
$z = \sqrt{x}$	`z = imsl_c_sqrt(z)`	`f_complex`	`f_complex`
$z = \cos(x)$	`z = imsl_c_cos(z)`	`f_complex`	`f_complex`
$z = \sin(x)$	`z = imsl_c_sin(z)`	`f_complex`	`f_complex`
$z = \exp(x)$	`z = imsl_c_exp(z)`	`f_complex`	`f_complex`
$z = \log(x)$	`z = imsl_c_log(z)`	`f_complex`	`f_complex`
$z = x^a$	`z = imsl_cf_power(x,a)`	`f_complex`	`f_complex`, `float`
$z = x^y$	`z = imsl_cc_power(x,y)`	`f_complex`	`f_complex` (both)
$c = a^k$	`c = imsl_fi_power(a,k)`	`float`	`float`, `int`
$c = a^b$	`c = imsl_ff_power(a,b)`	`float`	`float` (both)
$m = j^k$	`m = imsl_ii_power(j,k)`	`int`	`int` (both)

Double-Precision Complex Operations and Functions

Operation	Function Name	Function Result	Function Argument(s)
$z = -x$	`z = imsl_z_neg(x)`	`d_complex`	`d_complex`
$z = x + y$	`z = imsl_z_add(x,y)`	`d_complex`	`d_complex` (both)
$z = x - y$	`z = imsl_z_sub(x,y)`	`d_complex`	`d_complex` (both)
$z = x * y$	`z = imsl_z_mul(x,y)`	`d_complex`	`d_complex` (both)
$z = x / y$	`z = imsl_z_div(x,y)`	`d_complex`	`d_complex` (both)
$x==y^b$	`z = imsl_z_eq(x,y)`	`int`	`d_complex` (both)
$z = x$ *Drop Precision*	`z = imsl_zc_convert(x)`	`d_complex`	`f_complex`
$z = a + ib$ *Ascend Data*	`z = imsl_zd_convert(a,b)`	`d_complex`	`double` (both)

[b] Result has the value 1 if x and y are valid numbers with real and imaginary parts identical; otherwise, result has the value 0.

Operation	Function Name	Function Result	Function Argument(s)
$z = x$	`z = imsl_z_conjg(x)`	`d_complex`	`d_complex`
$a = \|z\|$	`a = imsl_z_abs(z)`	`double`	`d_complex`
$a = \arg(z)$ $-\pi < a \leq \pi$	`a = imsl_z_arg(z)`	`double`	`d_complex`
$z = \sqrt{x}$	`z = imsl_z_sqrt(z)`	`d_complex`	`d_complex`
$z = \cos(x)$	`z = imsl_z_cos(z)`	`d_complex`	`d_complex`
$z = \sin(x)$	`z = imsl_z_sin(z)`	`d_complex`	`d_complex`
$z = \exp(x)$	`z = imsl_z_exp(z)`	`d_complex`	`d_complex`
$z = \log(x)$	`z = imsl_z_log(z)`	`d_complex`	`d_complex`
$z = x^a$	`z = imsl_zd_power(x,a)`	`d_complex`	`d_complex, double`
$z = x^y$	`z = imsl_zz_power(x,y)`	`d_complex`	`d_complex` (both)
$c = a^k$	`c = imsl_di_power(a,k)`	`double`	`double, int`
$c = a^b$	`c = imsl_dd_power(a,b)`	`double`	`double` (both)
$m = j^k$	`m = imsl_ii_power(j,k)`	`int`	`int` (both)

The following sample code computes and prints several quantities associated with complex numbers. Note that the quantity

$$w = \sqrt{3+4i}$$

has a rounding error associated with it. Also the quotient $z = (1 + 2i) / (3 + 4i)$ has a rounding error. The result is acceptable in both cases because the relative errors $|w - (2 + 2i)|/|w|$ and $|z * (3 + 4i) - (1 + 2i)|/|(1 + 2i)|$ are approximately the size of machine precision.

```
#include <imsl.h>

main()
{
    f_complex           x = {1,2};
    f_complex           y = {3,4};
    f_complex           z;
    f_complex           w;
    int                 isame;
    float               eps = imsl_f_machine(4);
                                /* Echo inputs x and y */
    printf("Data:  x = (%g, %g)\n       y = (%g, %g)\n\n",
           x.re, x.im, y.re, y.im);
                                /* Add inputs */
    z = imsl_c_add(x,y);
    printf("Sum:   z = x + y = (%g, %g)\n\n", z.re, z.im);
                                /* Compute square root of y */
```

```
    w = imsl_c_sqrt(y);
    printf("Square Root: w = sqrt(y) = (%g, %g)\n", w.re, w.im);
                                /* Check results */
    z = imsl_c_mul(w,w);
    printf("Check:       w*w = (%g, %g)\n", z.re, z.im);
    isame = imsl_c_eq(y,z);
    printf("             y == w*w  = %d\n", isame);
    z = imsl_c_sub(z,y);
    printf("Difference:  w*w - y = (%g, %g) = (%g, %g) * eps\n\n",
           z.re, z.im, z.re/eps, z.im/eps);
                                /* Divide inputs */
    z = imsl_c_div(x,y);
    printf("Quotient:    z = x/y = (%g, %g)\n", z.re, z.im);
                                /* Check results */
    w = imsl_c_sub(x, imsl_c_mul(z, y));
    printf("Check:       w = x - z*y = (%g, %g) = (%g, %g) * eps\n",
           w.re, w.im, w.re/eps, w.im/eps);
}
```

Output

```
Data:  x = (1, 2)
       y = (3, 4)

Sum:   z = x + y = (4, 6)

Square Root: w = sqrt(y) = (2, 1)
Check:       w*w = (3, 4)
             y == w*w  = 0
Difference:  w*w - y = (-2.38419e-07, 4.76837e-07) = (-2, 4) * eps

Quotient:    z = x/y = (0.44, 0.08)
Check:       w = x - z*y = (5.96046e-08, 0) = (0.5, 0) * eps
```

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 fo 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.

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 2: *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.

Kinnucan and Kuki

Kinnucan, P., and H. Kuki (1968), *A single precision inverse error function subroutine*, Computation Center, University of Chicago.

Knuth

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume 2: *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.

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.

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.

Schittkowski

Schittkowski, K. (1987), *More test examples for nonlinear programming codes*, Springer-Verlag, Berlin, **74**.

Schittkowski, K. (1986), NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems, (edited by Clyde L. Monma), *Annals of Operations Research*, **5**, 485–500.

Schittkowski, K. (1980), Nonlinear programming codes, *Lecture Notes in Economics and Mathematical Systems*, **183**, Springer-Verlag, Berlin, Germany.

Schittkowski, K. (1983), On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function, *Mathematik Operationsforschung und Statistik, Serie Optimization*, **14**, 197–216.

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.

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.

Stewart

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Stoer

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

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.

Thompson and Barnett

Thompson, I.J. and A.R. Barnett (1987), Modified Bessel functions $I_V(z)$ and $K_V(z)$ 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

bessel_Kx	Evaluates a sequence of modified Bessel functions of the third kind with real order and complex arguments	470
bessel_Y0	Evaluates the real Bessel function of the second kind of order zero $Y0(x)$	453
bessel_Y1	Evaluates the real Bessel function of the second kind of order one $Y1(x)$	455
bessel_Yx	Evaluates a sequence of Bessel functions of the second kind with real order and complex arguments	456
beta	Evaluates the real beta function $\beta(x, y)$	439
beta_cdf	Evaluates the beta probability distribution function	512
beta_incomplete	Evaluates the real incomplete beta function $Ix = \beta x(a, b)/\beta(a, b)$	442
beta_inverse_cdf	Evaluates the inverse of the beta distribution function	513
binomial_cdf	Evaluates the binomial distribution function	507
bivariate_normal_cdf	Evaluates the bivariate normal distribution function	514
bounded_least_squares	Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm	416
chi_squared_cdf	Evaluates the chi-squared distribution function	495
chi_squared_inverse_cdf	Evaluates the inverse of the chi-squared distribution function	497
chi_squared_test	Performs a chi-squared goodness-of-fit test	528
constant	Returns the value of various mathematical and physical constants	599
convolution	Computes the convolution, and optionally, the correlation of two real vectors	342
convolution (complex)	Computes the convolution, and optionally, the correlation of two complex vectors	348
covariances	Computes the sample variance-covariance or correlation matrix	536
ctime	Returns the number of CPU seconds used	592
cub_spline_integral	Computes the integral of a cubic spline	158
cub_spline_interp_e_cnd	Computes a cubic spline interpolant, specifying various endpoint conditions	143
cub_spline_interp_shape	Computes a shape-preserving cubic spline	150
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	203
cub_spline_value	Computes the value of a cubic spline or the value of one of its derivatives	155
date_to_days	Computes the number of days from January 1, 1900, to the given date	593

days_to_date	Gives the date corresponding to the number of days since January 1, 1900	594
eig_gen	Computes the eigenexpansion of a real matrix *A*	116
eig_gen (complex)	Computes the eigenexpansion of a complex matrix *A*	118
eig_herm (complex)	Computes the eigenexpansion of a complex Hermitian matrix *A*	124
eig_sym	Computes the eigenexpansion of a real symmetric matrix *A*	121
eig_symgen	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. *A* and *B* are real and symmetric. *B* is positive definite	127
elliptic_integral_E	Evaluates the complete elliptic integral of the second kind $E(x)$	472
elliptic_integral_K	Evaluates the complete elliptic integral of the kind $K(x)$	471
elliptic_integral_RC	Evaluates an elementary integral from which inverse circular functions, logarithms, and inverse hyperbolic functions can be computed	477
elliptic_integral_RD	Evaluates Carlson's elliptic integral of the second kind $RD(x, y, z)$	474
elliptic_integral_RF	Evaluates Carlson's elliptic integral of the first kind $RF(x, y, z)$	473
elliptic_integral_RJ	Evaluates Carlson's elliptic integral of the third kind $RJ(x, y, z, \rho)$	476
erf	Evaluates the real error function $\text{erf}(x)$	433
erf_inverse	Evaluates the real inverse error function $\text{erf-1}(x)$	436
erfc	Evaluates the real complementary error function $\text{erfc}(x)$	434
erfc_inverse	Evaluates the real inverse complementary error function $\text{erfc-1}(x)$	437
error_code	Gets the code corresponding to the error message from the last function called	598
error_options	Sets various error handling options	595
F_cdf	Evaluates the *F* distribution function	498
F_inverse_cdf	Evaluates the inverse of the *F* distribution function	501
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	311
fcn_derivative	Computes the first, second or third derivative of a user-supplied function	277

fft_2d_complex	Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array	338
fft_complex	Computes the complex discrete Fourier transform of a complex sequence	325
fft_complex_init	Computes the parameters for imsl_c_fft_complex	328
fft_cosine	Computes the discrete Fourier cosine transformation of an even sequence	329
fft_cosine_init	Computes the parameters needed for imsl_f_fft_cosine	332
fft_real	Computes the real discrete Fourier transform of a real sequence	319
fft_real_init	Computes the parameters for imsl_f_fft_real	323
fft_sine	Computes the discrete Fourier sine transformation of an odd sequence	334
fft_sine_init	Computes the parameters needed for imsl_f_fft_sine	336
fresnel_integral_C	Evaluates the cosine Fresnel integral	478
fresnel_integral_S	Evaluates the sine Fresnel integral	479
gamma	Evaluates the real gamma function $\Gamma(x)$	443
gamma_cdf	Evaluates the gamma distribution function	505
gamma_incomplete	Evaluates the incomplete gamma function $\gamma(a, x)$	446
gauss_quad_rule	Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions	273
geneig	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B real	130
geneig (complex)	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B complex	133
generate_test_band	Generates test matrices of class $E(n, c)$	660
generate_test_band (complex)	Generates test matrices of class $Ec(n, c)$	662
generate_test_coordinate	Generates test matrices of class $D(n, c)$ and $E(n, c)$	664
generate_test_coordinate (complex)	Generates test matrices of class $D(n, c)$ and $E(n, c)$	668
hypergeometric_cdf	Evaluates the hypergeometric distribution function	509
int_fcn	Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules	237
int_fcn_2d	Computes a two-dimensional iterated integral	266
int_fcn_alg_log	Integrates a function with algebraic-logarithmic singularities	245

int_fcn_cauchy	Computes integrals of the form $\int_a^b \frac{f(x)}{x-c}dx$ in the Cauchy principal value sense	260
int_fcn_fourier	Computes a Fourier sine or cosine transform	256
int_fcn_hyper_rect	Integrates a function on a hyper-rectangle	270
int_fcn_inf	Integrates a function over an infinite or semi-infinite interval	248
int_fcn_sing	Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules	233
int_fcn_sing_pts	Integrates a function with singularity points given	241
int_fcn_smooth	Integrates a smooth function using a nonadaptive rule	263
int_fcn_trig	Integrates a function containing a sine or a cosine factor	252
inverse_laplace	Computes the inverse Laplace transform of a complex function	354
kelvin_bei0	Evaluates the Kelvin function of the first kind, bei, of order zero	485
kelvin_bei0_derivative	Evaluates the derivative of the Kelvin function of the first kind, bei, of order zero	499
kelvin_ber0	Evaluates the Kelvin function of the first kind, ber, of order zero	484
kelvin_ber0_derivative	Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero	488
kelvin_kei0	Evaluates the Kelvin function of the second kind, kei, of order zero	486
kelvin_kei0_derivative	Evaluates the derivative of the Kelvin function of the second kind, kei, of order zero	491
kelvin_ker0	Evaluates the Kelvin function of the second kind, der, of order zero	486
kelvin_ker0_derivative	Evaluates the derivative of the Kelvin function of the second kind, ker, of order zero	490
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	91
lin_prog	Solves a linear programming problem using the revised simplex algorithm	402
lin_sol_def_cg	Solves a real symmetric definite linear system using a conjugate gradient method	78
lin_sol_gen	Solves a real general system of linear equations $Ax = b$	4

lin_sol_gen (complex)	Solves a complex general system of linear equations $Ax = b$	11
lin_sol_gen_band	Solves a real geeral band system of linear equations $Ax=b$	26
lin_sol_gen_band (complex)	Solves a complex general system of linear equations $Ax = b$	31
lin_sol_gen_coordinate	Solves a sparse system of linear equations $Ax = b$	44
lin_sol_gen_coordinate (complex)	Solves a system of linear equations $Ax = b$, with sparse complex coefficient matrix A	54
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 $Ax = b$	106
lin_sol_posdef	Solves a real symmetric positive definite system of linear equations $Ax = b$	17
lin_sol_posdef (complex)	Solves a complex Hermitian positive definite system of linear equations $Ax = b$	22
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_band (complex)	Solves a complex Hermitian positive definite system of linear equations $Ax = b$ in band symmetric storage mode	39
lin_sol_posdef_coordinate	Solves a sparse real symmetric positive definite system of linear equations $Ax = b$	62
lin_sol_posdef_coordinate (complex)	Solves a sparse Hermitian positive definite system of linear equations $Ax = b$	68
lin_svd_gen	Computes the SVD, $A = USV$T, of a real rectangular matrix A	96
lin_svd_gen (complex)	Computes the SVD, $A = USV$H, of a complex rectangular matrix A	101
log_beta	Evaluates the logarithm of the real beta function ln $\beta(x, y)$	441
log_gamma	Evaluates the logarithm of the absolute value of the gamma function log $\lvert\Gamma(x)\rvert$	445
machine (float)	Returns information describing the computer's floating-point arithmetic	605
machine (integer)	Returns integer information describing the computer's arithmetic	603
mat_add_band	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$	639
mat_add_band (complex)	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$	642
mat_add_coordinate	Performs element-wise addition of two real matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$	646
mat_add_coordinate (complex)	Performs element-wise addition on two complex matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$	649

mat_mul_rect	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product	614
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	618
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	621
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	625
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	630
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	634
matrix_norm	Computes various norms of a rectangular matrix	653
matrix_norm_band	Computes various norms of a matrix stored in band storage mode	654
matrix_norm_coordinate	Computes various norms of a matrix stored in coordinate format	657
min_con_gen_lin	Minimizes a general objective function subject to linear equality/inequality constraints	410
min_con_nonlin	Solves a general nonlinear programming problem using the successive quadratic programming algorithm	423
min_uncon	Finds the minimum point of a smooth function *f(x)* of a single variable using only function evaluations	379
min_uncon_deriv	Finds the minimum point of a smooth function *f(x)* of a single variable using both function and first derivative evaluations	383
min_uncon_multivar	Minimizes a function *f(x)* of *n* variables using a quasi-Newton method	387
nonlin_least_squares	Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm	394
normal_cdf	Evaluates the standard normal (Gaussian) distribution function	492
normal_inverse_cdf	Evaluates the inverse of the standard normal (Gaussian) distribution function	494
ode_adams_gear	Solves a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods	288

ode_runge_kutta	Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method	283
output_file	Sets the output file or the error message output file	590
page	Sets or retrieve the page width or length	583
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	295
poisson_cdf	Evaluates the Poisson distribution function	510
poly_regression	Performs a polynomial least-squares regression	549
quadratic_prog	Solves a quadratic programming problem subject to linear equality or inequality constraints	406
radial_evaluate	Evaluates a radial basis fit	229
radial_scattered_fit	Computes an approximation to scattered data in Rn for $n \geq 2$ using radial basis functions	223
random_beta	Generates pseudorandom numbers from a beta distribution	573
random_exponential	Generates pseudorandom numbers from a standard exponential distribution	575
random_gamma	Generates pseudorandom numbers from a standard gamma distribution	571
random_normal	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method	568
random_option	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator	565
random_poisson	Generates pseudorandom numbers from a Poisson distribution	570
random_seed_get	Retrieves the current value of the seed used in the IMSL random number generators	563
random_seed_set	Initializes a random seed for use in the ISML random number generators	564
random_uniform	Generates pseudorandom numbers from a uniform (0, 1) distribution	566
ranks	Computes the ranks, normal scores, or exponential scores for a vector of observations	557
regression	Fits a multiple linear regression model using least squares	541
scattered_2d_interp	Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables	218
simple_statistics	Computes basic univariate statistics	519
sort	Sorts a vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned	607
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	610

spline_2d_integral	Evaluates the integral of a tensor-product spline on a rectangular domain	184
spline_2d_interp	Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data	169
spline_2d_least_squares	Computes a two-dimensional, tensor-product spline approximant using least squares	197
spline_2d_value	Computes the value of a tensor-product spline or the value of one of its partial derivatives	180
spline_integral	Computes the integral of a spline	178
spline_interp	Computes a spline interpolant	159
spline_knots	Computes the knots for a spline interpolant	165
spline_least_squares	Computes a least-squares spline approximation	191
spline_lsq_constrained	Computes a least-squares constrained spline approximation	207
spline_value	Computes the value of a spline or the value of one of its derivatives	175
t_cdf	Evaluates the Student's *t* distribution function	506
t_inverse_cdf	Evaluates the inverse of the Student's *t* distribution function	504
table_oneway	Tallies observations into a one-way frequency table	524
user_fcn_least_squares	Computes a least-squares fit using user-supplied functions	187
vector_norm	Computes various norms of a vector or the difference of two vectors	612
version	Returns integer information describing the version of the library, license number, operating system, and compiler	591
write_matrix	Prints a rectangular matrix (or vector) stored in contiguous memory locations	577
write_options	Sets or retrieve an option for printing a matrix	584
zeros_fcn	Finds the real zeros of a real function using Müller's method	368
zeros_poly	Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm	364
zeros_poly (complex)	Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm	366
zeros_sys_eqn	Solves a system of n nonlinear equations $f(x) = 0$ using a modified Powell hybrid algorithm	373

Index

Q

R

S

T

U

V

Z

Product Support

Contacting Visual Numerics Support

Users within support warranty may contact Visual Numerics regarding the use of the IMSL C Numerical Libraries. Visual Numerics can consult on the following topics:

- Clarity of documentation
- Possible Visual Numerics-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem
- Evolution of the IMSL Libraries

Not included in these consultation topics are mathematical/statistical consulting and debugging of your program.

Consultation

Contact Visual Numerics Product Support by faxing 713/781-9260 or by emailing:

- for PC support, `pcsupport@houston.vni.com`.
- for non-PC support, `support@houston.vni.com`

Electronic addresses are not handled uniformly across the major networks, and some local conventions for specifying electronic addresses might cause further variations to occur; contact your E-mail postmaster for further details.

The following describes the procedure for consultation with Visual Numerics:

1. Include your serial (or license) number

2. Include the product name and version number: IMSL C/Math/Library Version 3.0

3. Include compiler and operating system version numbers

4. Include the name of the routine for which assistance is needed and a description of the problem