Visual Numerics®

VERSION 5.5

IMSL

C Numerical Library™

User's Guide

VOLUME 2 of 4: C Math Library™ [CHAPTERS 8-12]

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

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

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

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

Visual Numerics SARL
Tour Europe
33 Place des Corolles Cedex
F-92049 Paris La Defense
France

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

Visual Numerics S. A. de C.V.
Florencia 57 Piso 10-01
Col. Juarez
Mexico D. F. C. P. 06000
Mexico
PHONE: +52-5514-9730 or 9628
FAX: +52-5514-5880

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
Germany

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

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

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

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

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

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

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

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

IMSL Fortran and C and Java Application Development Tools

CMath Library /V2- Table of Contents

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 $f : \mathbf{R}^n \to \mathbf{R}$ 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`, page 425 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`, page 429 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:

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

$$\text{subject to } g_i(x) = 0 \quad \text{for } i = 1, 2, \ldots, m_1$$

$$g_i(x) \ge 0 \quad \text{for } i = m_1 + 1, \ldots, m$$

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

The function `imsl_f_constrained_nlp`, page 447 uses a sequential equality constrained 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`,
    `IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,
    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

`IMSL_FCN_W_DATA`, *float* `fcn`(*float* `x`, *void* `*data`), *void* `*data`, (Input)
User supplied function to compute the value of the function to be minimized, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_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,

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

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,
IMSL_FCN_W_DATA, *float* fcn(), *void* *data,
IMSL_GRADIENT_W_DATA, *float* grad(), *void* *data,
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`.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data`, (Input)
User supplied function to compute the value of the function to be minimized, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_GRADIENT_W_DATA`, *float* `grad` (*float* `x`, *void* `*data`), *void* `*data`, (Input)
User supplied function to compute the first derivative of the function, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `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(),
        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,
        IMSL_FCN_W_DATA, float fcn(), void *data,
        IMSL_GRADIENT_W_DATA, void grad(), void *data,
        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

details.
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 details.
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 $(\varepsilon_1, \varepsilon_2)$ where,

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

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

`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

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

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.

`IMSL_FCN_W_DATA`, *float* `fcn` (*int* `n`, *float* `x`, *void* `*data`), *void* `*data`, (Input)
User supplied function to compute the value of the function to be minimized, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_GRADIENT_W_DATA`, *void* `grad` (*int* `n`, *float* `x[]`, *float* `g[]`, *void* `*data`), *void* `*data`, (Input)
User supplied function to compute the gradient at the point `x`, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `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:

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

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

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

Finally, the optimality condition $\|g(x)\| \le \varepsilon$ 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.

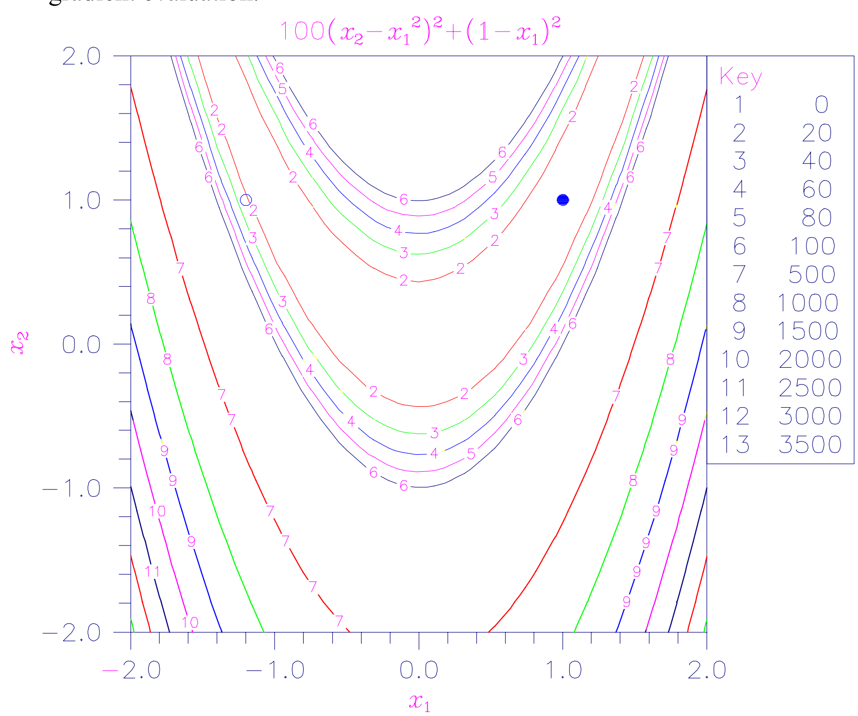

Figure 8-1 Plot of the Rosenbrock Function

Examples

Example 1

The function

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

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 413.

```
#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 ≤ 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,
        IMSL_FCN_W_DATA, void fcn(), void *data,
        IMSL_JACOBIAN_W_DATA, void jacobian(), void *data,
        0)
```

Optional Arguments

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `n` components containing 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[(i1)*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

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

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,

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

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

`IMSL_FCN_W_DATA`, *void* `fcn` (*int* `m`, *int* `n`, *float* `x[]`, *float* `f[]`, *void* `*data`), *void* `*data` (Input)
User supplied function to evaluate the function that defines the least-squares problem, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_JACOBIAN_W_DATA`, *void* `jacobian` (*int* `m`, *int* `n`, *float* `x[]`, *float* `fjac[]`, *int* `fjac_col_dim`, *void* `*data`), *void* `*data` (Input)
User supplied function to compute the Jacobian, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

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

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

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 $\delta_c \ge \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$ 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).

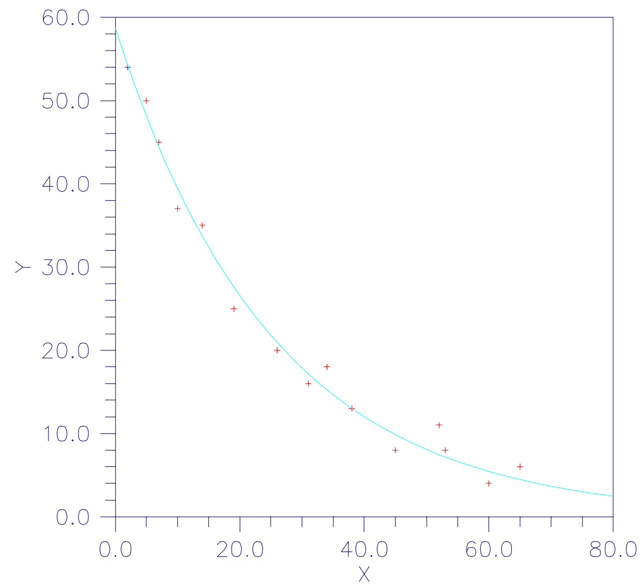

Figure 8-2 Plot of the Nonlinear Fit

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` (see Chapter 9, Special Functions) 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

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

$$x_l \le x \le x_u$$

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

$$\min f(x) = -x_1 - 3x_2$$

$$\begin{aligned}
\text{subject to}\quad & x_1 + x_2 + x_3 && = 1.5 \\
& x_1 + x_2 \quad - x_4 && = 0.5 \\
& x_1 \qquad\quad + x_5 && = 1.0 \\
& \quad x_2 \qquad\quad + x_6 && = 1.0 \\
& x_i \ge 0, \text{ for } i = 1, \dots, 6
\end{aligned}$$

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:

$$\min f(x) = -x_1 - 3x_2$$

$$\text{subject to}\quad 0.5 \le x_1 + x_2 \le 1.5$$

$$0 \le x_1 \le 1.0$$

$$0 \le x_2 \le 1.0$$

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 (moved to a vertex that is poorly conditioned). If float is currently being used, using double precision 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 m 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 m 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

$$\min_{x \in \mathbf{R}^n} g^T x + \frac{1}{2} x^T H x$$

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

$$A_2 x \ge b_2$$

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

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

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

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 $a_{k,0}x_0 + \ldots + a_{k,nvar-1}x_{nvar-1} \le b_k$, $k =$ `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*] ≤ 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 ≤ `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[],
IMSL_FCN_W_DATA, void fcn(), void *data,
IMSL_GRADIENT_W_DATA, void grad(), void *data,
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.

`IMSL_FCN_W_DATA`, *void* `fcn` (*int* `n`, *float* `x[]`, *float* `*f` , *void* `*data`), *void* `*data` (Input)
User supplied function to compute the value of the function to be minimized, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_GRADIENT_W_DATA`, *void* `gradient` (*int* `n`, *float* `x[]`, *float* `g[]`,*void* `*data`) , *void* `*data` (Input)
User-supplied function to compute the gradient at the point `x`, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_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

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

subject to

$$a_jd = 0, j \in I_k$$

$$a_jd \le 0, j \in J_k$$

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

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

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

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

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

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

$$0 \le x \le 10$$

is solved.

```
#include "imsl.h"

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

        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` ≤ `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.
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,
IMSL_FCN_W_DATA, *void* fcn(), *void* *data,
IMSL_JACOBIAN_W_DATA, *void* jacobian(), *void* *data,
0)

Optional Arguments

IMSL_XGUESS, *float* xguess[] (Input)
Array with n components containing 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 details.
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

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

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

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

Default: `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

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

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` = $\max(10^{-20}, \varepsilon^2)$, $\max(10^{-40}, \varepsilon^2)$ in double, where ε is the machine precision

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

Maximum allowable step size.

Default: `max_step` = 1000 $\max(\varepsilon_1, \varepsilon_2)$, where

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

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

IMSL_FCN_W_DATA, *void* fcn (*int* m, *int* n, *float* x[], *float* f[], *void* *data), *void* *data, (Input)
User-supplied function to evaluate the function that defines the least-squares problem, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSL_JACOBIAN_W_DATA, *void* jacobian (*int* m, *int* n, *float* x[], *float* fjac[], *int* fjac_col_dim, *void* *data), *void* *data, (Input)

User-supplied function to compute the Jacobian, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

The function `imsl_f_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 $m \ge n$, $F : \mathbf{R}^n \to \mathbf{R}^m$, 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

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

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
```

constrained_nlp

Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method.

Synopsis

#include `<imsl.h>`

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

The type *double* function is `imsl_d_constrained_nlp`.

Required Arguments

void `fcn`(*int* `n`, *float* `x[]`, *int* `iact`, *float* `*result`, *int* `*ierr`) (Input)
User supplied function to evaluate the objective function and constraints at a given point.

int `n` (Input)
Number of variables.

float `x[]` (Input)
The point at which the objective function or a constraint is evaluated.

int `iact` (Input)
Integer indicating whether evaluation of the function is requested or evaluation of a constraint is requested. If `iact` is zero, then an objective function evaluation is requested. If `iact` is nonzero then the value of `iact` indicates the index of the constraint to evaluate.

float `result[]` (Output)
If `iact` is zero, then `result` is the computed objective function at the point `x`. If `iact` is nonzero, then `result` is the requested constraint value at the point `x`.

int `*ierr` (Output)
Address of an integer. On input `ierr` is set to 0. If an error or other undesirable condition occurs during evaluation, then `ierr` should be set to 1. Setting `ierr` to 1 will result in the step size being reduced

and the step being tried again. (If `ierr` is set to 1 for `xguess`, then an error is issued.)

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 `imsl_f_machine(8)`.

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 `imsl_f_machine(7)`.

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 Arugments

#include `<imsl.h>`

float *`imsl_f_constrained_nlp` (*void* `fcn()`, *int* `m`, *int* `meq`, *int* `n`, `i` *int* `nt`
`ibtype`, *float* `xlb[]`, *float* `xub[]`,
`IMSL_GRADIENT`, *void* `grad()`,
`IMSL_PRINT`, *int* `iprint`,
`IMSL_XGUESS`, *float* `xguess[]`,
`IMSL_ITMAX`, *int* `itmax`,

IMSL_TAU0, *float* tau0,
IMSL_DEL0, *float* del0,
IMSL_SMALLW, *float* smallw,
IMSL_DELMIN, *float* delmin,
IMSL_SCFMAX, *float* scfmax,
IMSL_RETURN_USER, *float* x[],
IMSL_OBJ, *float* *obj,
IMSL_DIFFTYPE, *int* difftype,
IMSL_XSCALE, *float* xscale[],
IMSL_EPSDIF, *float* epsdif,
IMSL_EPSFCN, *float* epsfcn,
IMSL_TAUBND, *float* taubnd,
IMSL_FCN_W_DATA, *void* fcn(), *void* *data,
IMSL_GRADIENT_W_DATA, *void* grad(), *void* *data,
0)

Optional Arguments

IMSL_GRADIENT, *void* grad(*int* n, *float* x[], *int* iact, *float* result[]) (Input)
User-supplied function to evaluate the gradients at a given point where

int n (Input)
Number of variables.

float x[] (Input)
The point at which the gradient of the objective function or gradient of a constraint is evaluated

int iact (Input)
Integer indicating whether evaluation of the function gradient is requested or evaluation of a constraint gradient is requested. If iact is zero, then an objective function gradient evaluation is requested. If iact is nonzero then the value of iact indicates the index of the constraint gradient to evaluate.

float result[] (Output)
If iact is zero, then result is the computed gradient of the objective function at the point x. If iact is nonzero, then result is the computed gradient of the requested constraint value at the point x.

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

Iprint	**Action**
0	No output printed.
1	One line of intermediate results is printed in each iteration.
2	Lines of intermediate results summarizing the most important data for each step are printed.

Iprint	Action
3	Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking and etc are printed
4	Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated and etc are printed.

Default: `iprint` = 0.

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array of length `n` containing an initial guess of the solution. (Input)
Default: `xguess` = `X`, (with the smallest value of $\|x\|_2$) that satisfies the bounds.

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

`IMSL_TAU0`, *float* `tau0` (Input)
A universal bound describing how much the unscaled penalty-term may deviate from zero. (Input)
`imsl_f_constrained_nlp` assumes that within the region described by

$$\sum_{i=1}^{M_e} \left|g_i(x)\right| - \sum_{i=M_e+1}^{M} \min\left(0, g_i(x)\right) \le \texttt{tau0}$$

all functions may be evaluated safely. The initial guess, however, may violate these requirements. In that case an initial feasibility improvement phase is run by `imsl_f_constrained_nlp` until such a point is found. A small `tau0` diminishes the efficiency of `imsl_f_constrained_nlp`, because the iterates then will follow the boundary of the feasible set closely. Conversely, a large `tau0` may degrade the reliability of the code.
Default `tau0` = 1.0.

`IMSL_DEL0`, *float* `del0` (Input)
In the initial phase of minimization a constraint is considered binding if

$$\frac{g_i(x)}{\max\left(1, \left\|\nabla g_i(x)\right\|\right)} \le \texttt{del0} \qquad i = M_e + 1, \ldots, M$$

Good values are between .01 and 1.0. If `del0` is chosen too small then identification of the correct set of binding constraints may be delayed.

Contrary, if del0 is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well-scaled problems del0 = 1.0 is reasonable.
Default: del0 = .5* tau0

IMSL_SMALLW, *float* smallw (Input)
Scalar containing the error allowed in the multipliers. For example, a negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than smallw .
Default: smallw = exp(2*log(eps/3)) where eps is the machine precision.

IMSL_DELMIN, *float* delmin (Input)
Scalar which defines allowable constraint violations of the final accepted result. Constraints are satisfied if $|g_i(x)| \leq$ delmin for equality constraints, and $g_i(x) \geq$ (-delmin) for equality constraints.
Default: delmin = min(.1*del0, max(epsdif, max(1.e-6*del0, smallw))

IMSL_SCFMAX, *float* scfmax (Input)
Scalar containing the bound for the internal automatic scaling of the objective function. (Input)
Default: scfmax = 1.0e4

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

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

IMSL_FCN_W_DATA, *void* fcn(*int* n, *float* x[], *int* iact, *float* *result, *int* *ierr, *void* *data), *void* *data, (Input)
User supplied function to evaluate the objective function and constraints at a given point, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSL_GRADIENT_W_DATA, *void* grad(*int* n, *float* x[], *int* iact, *float* result[], *void* *data), *void* *data, (Input)
User-supplied function to evaluate the gradients at a given point, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

The following optional arguments are valid only if IMSL_GRADIENT is not supplied.

IMSL_DIFFTYPE, *int* difftype (Input)
Type of numerical differentiation to be used.
Default: difftype = 1

`difftype`	**Action**
1	Use a forward difference quotient with discretization stepsize 0.1 `(epsfcn)` $^{1/2}$ componentwise relative.
2	Use the symmetric difference quotient with discretization stepsize 0.1 `(epsfcn)` $^{1/3}$ componentwise relative.
3	Use the sixth order approximation computing a Richardson extrapolation of three symmetric difference quotient values. This uses a discretization stepsize 0.01(`epsfcn`)$^{1/7}$.

`IMSL_XSCALE`, *float* `xscale[]` (Input)
Vector of length `n` setting the internal scaling of the variables. The initial value given and the objective function and gradient evaluations however are always in the original unscaled variables. The first internal variable is obtained by dividing values `x[i]` by `xscale[i]`. (Input)
In the absence of other information, set all entries to 1.0.
Default: `xscale[]` = 1.0.

`IMSL_EPSDIF`, *float* `epsdif` (Input)
Relative precision in gradients.
Default: `epsdif` = ε where ε is the machine precision.

`IMSL_EPSFCN`, *float* `epsfcn` (Input)
Relative precision of the function evaluation routine. (Input)
Default: `epsfcn` = ε where ε is the machine precision

`IMSL_TAUBND`, *float* `taubnd` (Input)
Amount by which bounds may be violated during numerical differentiation. Bounds are violated by `taubnd` (at most) only if a variable is on a bound and finite differences are taken taken for gradient evaluations. (Input)
Default: `taubnd` = 1.0

Description

The function `constrained_nlp` provides an interface to a licensed version of subroutine `DONLP2`, a code developed by Peter Spellucci (1998). It uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the “working sets”). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armjijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: An SQP method for general nonlinear programs using only equality constrained subproblems. Math. Prog. 82, (1998), 413-448.

P. Spellucci: A new technique for inconsistent problems in the SQP method. Math. Meth. of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

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

subject to

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

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

$$x_l \leq x \leq x_u$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, `imsl_f_constrained_nlp` allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The DONLP2 Users Guide provides detailed descriptions of these parameters as well as strategies for maximizing the perfomance of the algorithm. The DONLP2 Users Guide is available in the "*help*" subdirectory of the main IMSL product installation directory. In addition, the following are a number of guidelines to consider when using `imsl_f_constrained_nlp`.

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See optional argument `IMSL_XGUESS`.
- Gradient approximation methods can have an effect on the success of `imsl_f_constrained_nlp`. Selecting a higher order approximation method may be necessary for some problems. See optional argument `IMSL_DIFFTYPE`.
- If a two sided constraint $l_i \leq g_i(x) \leq u_i$ is transformed into two constraints $g_{2i}(x) \geq 0$ and $g_{2i+1}(x) \geq 0$, then choose $\text{del0} < \frac{1}{2}(u_i - l_i) / max\{1, \left\|\nabla g_i(x)\right\|\}$, or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument `IMSL_DEL0`.
- The parameter `ierr` provided in the interface to the user supplied function `fcn` can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting `ierr` to `1` and returning without performing the evaluation will avoid the exception. `imsl_f_constrained_nlp` will then reduce the stepsize and try the step again. Note, if `ierr` is set to `1` for the initial guess, then an error is issued.

Example

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 grad(int n, float x[], int iact, float result[]);
void fcn(int n, float x[], int iact, float *result, int *ierr);

void main()
{
  int ibtype = 0;
  float *x, ans[2];
  static float xlb[N], xub[N];

  xlb[0] = xlb[1] = imsl_f_machine(8);
  xub[0] = xub[1] = imsl_f_machine(7);
  x = imsl_f_constrained_nlp(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 n, float x[], int iact, float *result, int *ierr)
{
  float  tmp1, tmp2;
  tmp1 = x[0] - 2.0e0;
  tmp2 = x[1] - 1.0e0;
  switch (iact) {
  case 0:
    *result = tmp1 * tmp1 + tmp2 * tmp2;
    break;
  case 1:
    *result = x[0] - 2.0e0 * x[1] + 1.0e0;
    break;
  case 2:
    *result = -(x[0]*x[0]) / 4.0e0 - x[1]*x[1] + 1.0e0;
    break;
  default: ;
    break;
  }
  *ierr = 0;
  return;
}
```

Output

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

Chapter 9: Special Functions

Routines

9.1 Error and Gamma Functions

9.2 Bessel Functions

Usage Notes

Users can perform financial computations by using pre-defined data types. Most of the financial functions require one or more of the following:

- Date
- Number of payments per year
- A variable to indicate when payments are due
- Day count basis

IMSL C/Math/Library provides the identifiers for the input, `frequency`, to indicate the number of payments for each year. The identifiers are `IMSL_ANNUAL`, `IMSL_SEMIANNUAL`, and `IMSL_QUARTERLY`.

Identifier (`frequency`)	Meaning
`IMSL_ANNUAL`	One payment per year (Annual payment)
`IMSL_SEMIANNUAL`	Two payments per year (Semi-annual payment)
`IMSL_QUARTERLY`	Four payments per year (Quarterly payment)

IMSL C/Math/Library provides the identifiers for the input, `when`, to indicate when payments are due. The identifiers are `IMSL_AT_END_OF_PERIOD`, `IMSL_AT_BEGINNING_OF_PERIOD`.

Identifier (when)	Meaning
IMSL_AT_END_OF_PERIOD	Payments are due at the end of the period
IMSL_AT_BEGINNING_OF_PERIOD	Payments are due at the beginning of the period

IMSL C/Math/Library provides the identifiers for the input, basis, to indicate the type of day count basis. Day count basis is the method for computing the number of days between two dates. The identifiers are IMSL_DAY_CNT_BASIS_NASD, IMSL_DAY_CNT_BASIS_ACTUALACTUAL, IMSL_DAY_CNT_BASIS_ACTUAL360, IMSL_DAY_CNT_BASIS_ACTUAL365, and IMSL_DAY_CNT_BASIS_30E360.

Identifier (basis)	Day count basis
IMSL_DAY_CNT_BASIS_NASD	US (NASD) 30/360
IMSL_DAY_CNT_BASIS_ACTUALACTUAL	Actual/Actual
IMSL_DAY_CNT_BASIS_ACTUAL360	Actual/360
IMSL_DAY_CNT_BASIS_ACTUAL365	Actual/365
IMSL_DAY_CNT_BASIS_30E360	European 30/360

IMSL C/Math/Library uses the C programming language structure, tm, provided in the standard header <time.h>, to represent a date. For a detailed description of tm, see Kernighan and Richtie 1988, *The C Programming Language*, Second Edition, p 255.

The structure tm is declared within <time.h> as follows:

```
struct  tm {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

For example, to declare a variable to represent Jan 1, 2001, use the following code segment:

```
struct tm date;

date.tm_year = 101;
date.tm_mon = 0;
date.tm_mday = 1;
```

NOTE: IMSL C/Math/Library only uses the `tm_year`, `tm_mon`, and `tm_mday` fields in structure `tm`.

Additional Information

In preparing the finance and bond functions we incorporated standards used by *SIA Standard Securities Calculation Methods*.

More detailed information on finance and bond functionality can be found in the following manuals:

- *SIA Standard Securities Calculation Methods* 1993, vols. 1 & 2, Third Edition.
- *Accountants' Handbook*, Volume 1, Sixth Edition.
- *Microsoft Excel 5, Worksheet Function Reference*.

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

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

All values of x are legal.

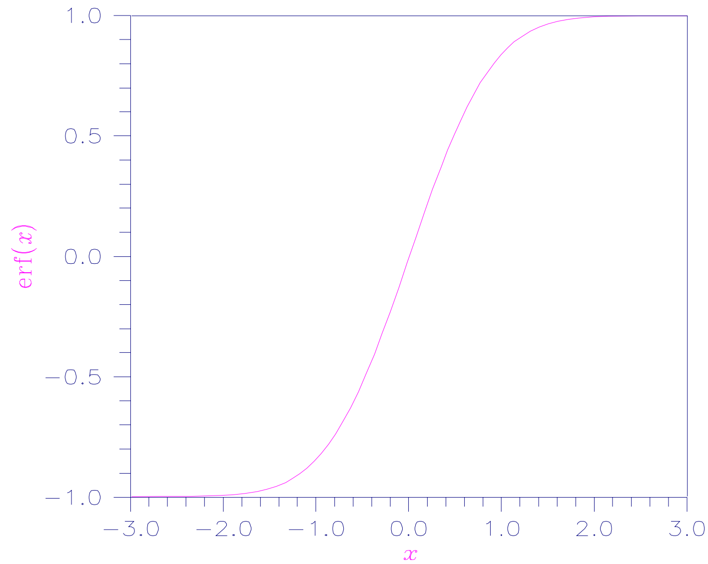

Figure 9-1 Plot of erf(x)

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

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

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.

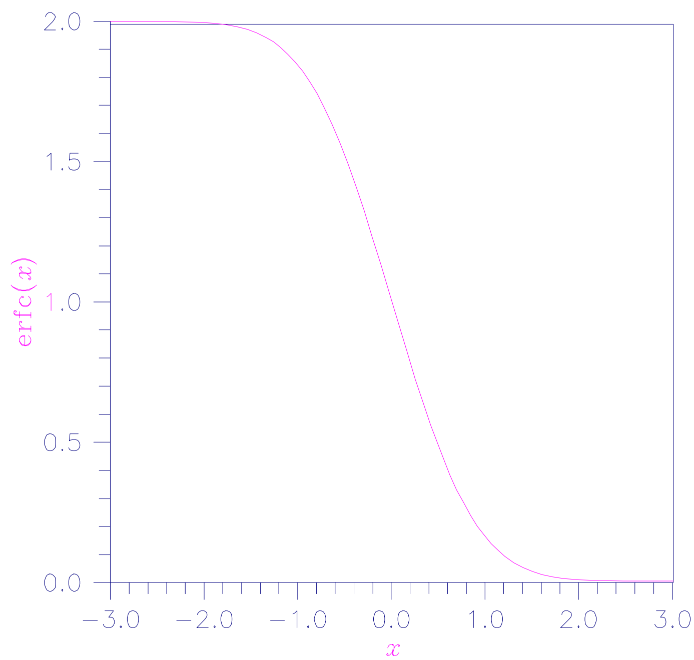

Figure 9-2 Plot of erfc(*x*)

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.

erfce

Evaluates the exponentially scaled complementary error function.

Synopsis

#include `<imsl.h>`

float `imsl_f_erfce` (*float* `x`)

The type *double* function is `imsl_d_erfce`.

Required Arguments

float `x` (Input)
Argument for which the function value is desired.

Return Value

Exponentially scaled complementary error function value.

Description

Function `imsl_f_erfce` computes

$$e^{x^2} \text{ erfc } (x)$$

where erfc(x) is the complementary error function. See `imsl_f_erfc` (page 461) for its definition.

To prevent the answer from underflowing, x must be greater than

$$x_{\min} \simeq -\sqrt{\ln(b/2)}$$

where b = `imsl_f_machine`(2) is the largest representable floating-point number.

Example

In this example, `imsl_f_erfce`(1.0) is computed and printed.

```
#include "imsl.h"
main()
{
   float value, x;

   x = 1.0;
   value = imsl_f_erfce(x);

   printf("erfce(%6.3f) = %6.3f \n", x, value);
}
```

Output

```
 erfce( 1.000) =  0.428
```

erfe

Evaluates a scaled function related to erfc(z).

Synopsis

#include `<imsl.h>`

f_complex `imsl_c_erfe` (*f_complex* `z`)

The type *double complex* function is `imsl_z_erfe`.

Required Arguments

f_complex `z` (Input)
Complex argument for which the function value is desired.

Return Value

Complex scaled function value related to erfc(z).

Description

Function `imsl_c_erfe` is defined to be

$$e^{-z^2}\operatorname{erfc}(-iz) = -ie^{-z^2}\frac{2}{\sqrt{\pi}}\int_z^\infty e^{t^2}\,dt$$

Let b = `imsl_f_machine`(2) be the largest floating-point number. The argument z must satisfy

$$|z| \le \sqrt{b}$$

or else the value returned is zero. If the argument z does not satisfy

$$(\Im z)^2 - (\Re z)^2 \le \log b,$$

then b is returned. All other arguments are legal (Gautschi 1969, 1970).

Example

In this example, `imsl_c_erfe`($2.5 + 2.5i$) is computed and printed.

```
#include "imsl.h"
main()
{
   f_complex value, z;

   z = imsl_cf_convert(2.5, 2.5);
   value = imsl_c_erfe(z);
printf("\n erfe(%2.3f + %2.3fi) = %2.3f + %2.3fi \n", z.re, z.im, value.re, value.im);
              z.re, z.im, value.re, value.im);
}
```

Output

```
 erfe(2.500 + 2.500i) = 0.117 + 0.108i
```

erf_inverse

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

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 $\mathrm{erf}^{-1}(x)$.

Description

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

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

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

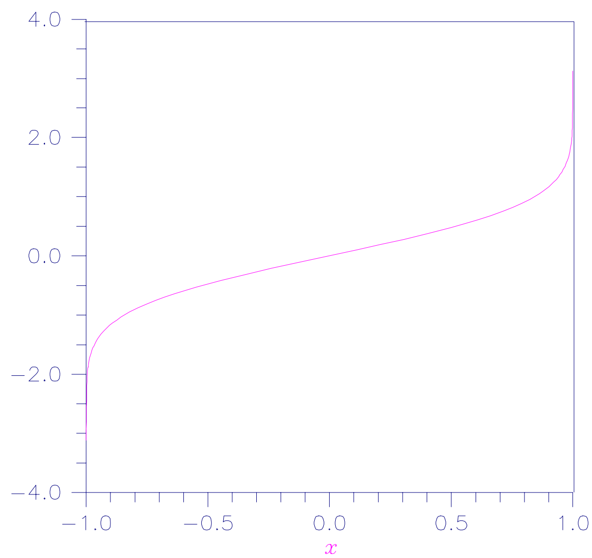

Figure 9-3 Plot of $\mathrm{erf}^{-1}(x)$

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 $\lvert x \rvert$ 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 $\mathrm{erfc}^{-1}(x)$.

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 $y = \mathrm{erfc}^{-1}(x)$ is such that $x = \mathrm{erfc}(y)$ where

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

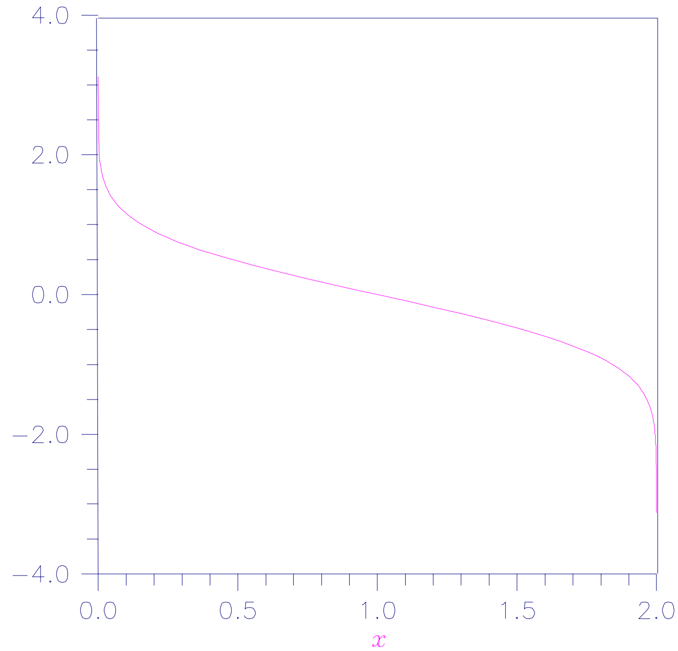

Figure 9-4 Plot of $\mathrm{erfc}^{-1}(x)$

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 β(*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 β (*x*, *y*). If no result can be computed, NaN is returned.

Description

The beta function, β (*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.

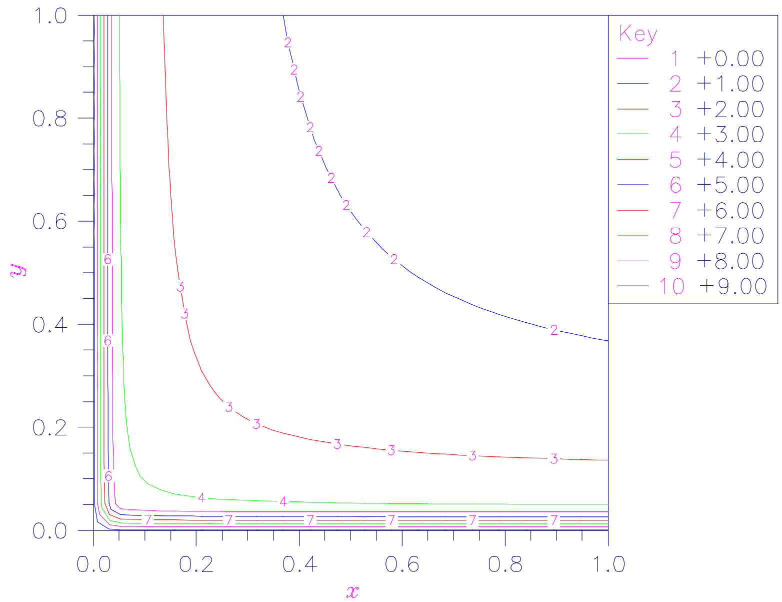

Figure 9-5 Plot of β(*x*,*y*)

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 β(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 β(x, y).

Description

The beta function, β (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 β(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 β(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 Γ(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 Γ(x).

Description

The gamma function, Γ(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.

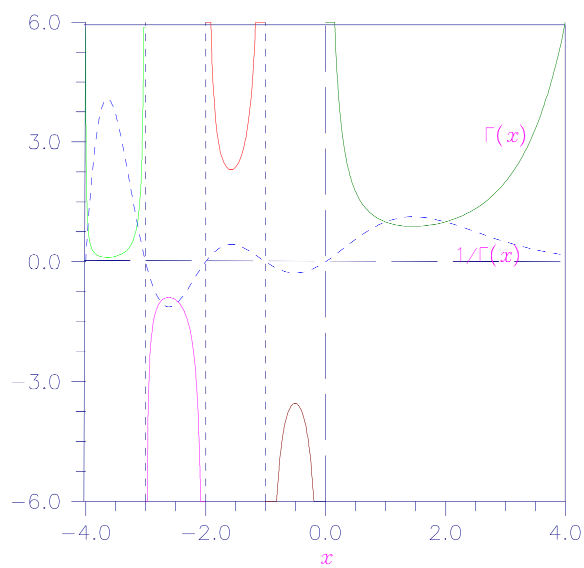

Figure 9-6 Plot of Γ(x) and 1/Γ(x)

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.

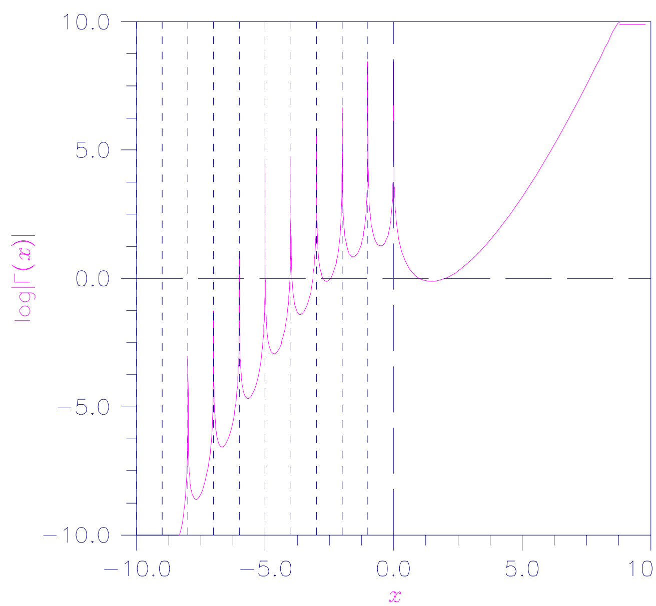

Figure 9-7 Plot of log $|\Gamma(x)|$

Example

In this example, log |Γ(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

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

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.

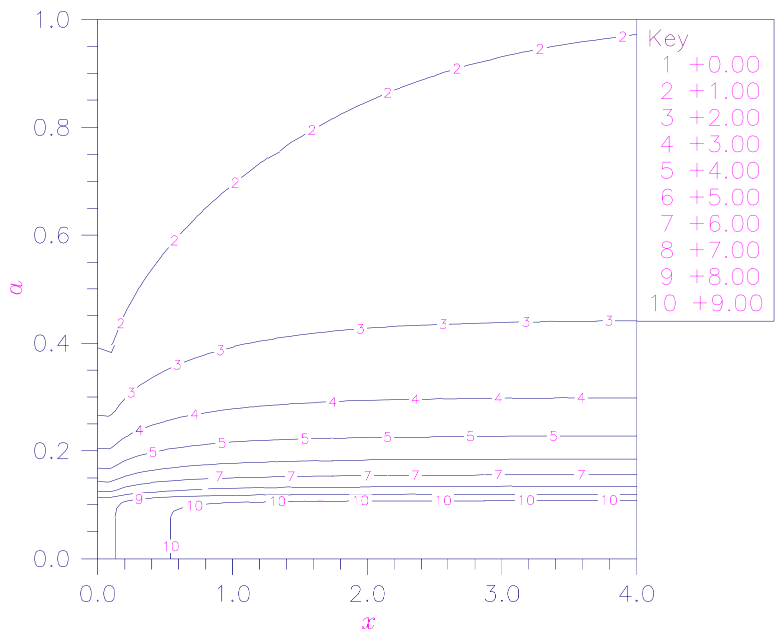

Figure 9-8 Plot of γ(a, x)

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.

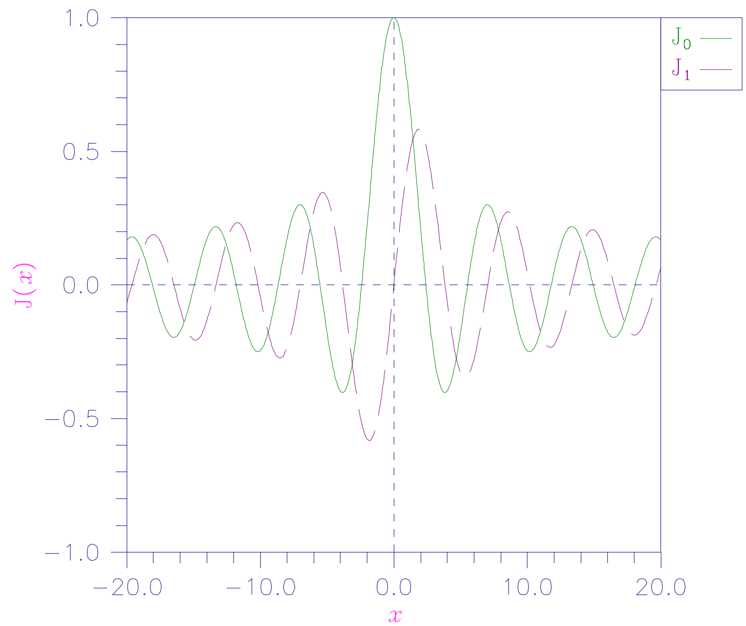

Figure 9-9   Plot of J0 (x) and J1 (x)

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/ε 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

$$J_1(x) = \frac{1}{\pi}\int_0^\pi \cos(x\sin\theta - \theta)\,d\theta$$

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`	$\lvert x \rvert$ 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`	$\lvert x \rvert$ should be less than 1/ε 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, …, *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

$$J_\nu(z) = \frac{1}{\pi}\int_0^\pi \cos(z\sin\theta - \nu\theta)\,d\theta - \frac{\sin(\nu\pi)}{\pi}\int_0^\infty e^{z\sinh t - \nu t}\,dt$$
$$\text{for } |\arg z| < \frac{\pi}{2}$$

This function is based on the code BESSCC of Barnett (1981) and Thompson and Barnett (1987). This code computes $J_\nu(z)$ 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

$$Y_0(x) = \frac{1}{\pi}\int_0^{\pi} \sin(x\sin\theta)\, d\theta - \frac{2}{\pi}\int_0^{\infty} e^{-z\sinh t}\, dt$$

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.

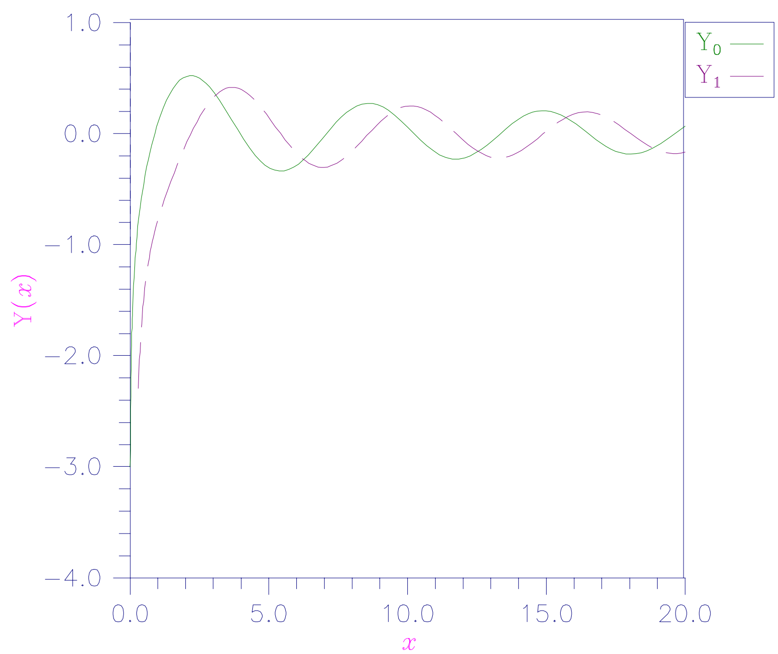

Figure 9-10 Plot of Y0(x) and Y1(x)

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/ε 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

$$Y_1(x) = -\frac{1}{\pi}\int_0^{\pi} \sin(\theta - x\sin\theta)\,d\theta - \frac{1}{\pi}\int_0^{\infty}\left\{e^t - e^{-t}\right\}e^{-z\sinh t}\,dt$$

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/ε 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, …, *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

$$Y_\nu(z) = \frac{1}{\pi}\int_0^\pi \sin(z\sin\theta - \nu\theta)\,d\theta - \frac{1}{\pi}\int_0^\infty \left[e^{\nu t} + e^{-\nu t}\cos(\nu\pi)\right]e^{-z\sinh t}\,dt$$

$$\text{for } |\arg z| < \frac{\pi}{2}$$

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:

$$Y_\nu\left(z e^{\pi i/2}\right) = e^{(\nu+1)\pi i/2} I_\nu(z) - \frac{2}{\pi} e^{-\nu\pi i/2} K_\nu(z) \qquad \text{for } -\pi < \arg z \le \frac{\pi}{2}$$

Example

In this example, $Y_{0.3+\nu-1}$ (1.2 + 0.5*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_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

$$I_0(x) = \frac{1}{\pi}\int_0^\pi \cosh(x\cos\theta)\,d\theta$$

If no solution can be computed, NaN is returned.

Description

For large $|x|$, `imsl_f_bessel_I0` will overflow.

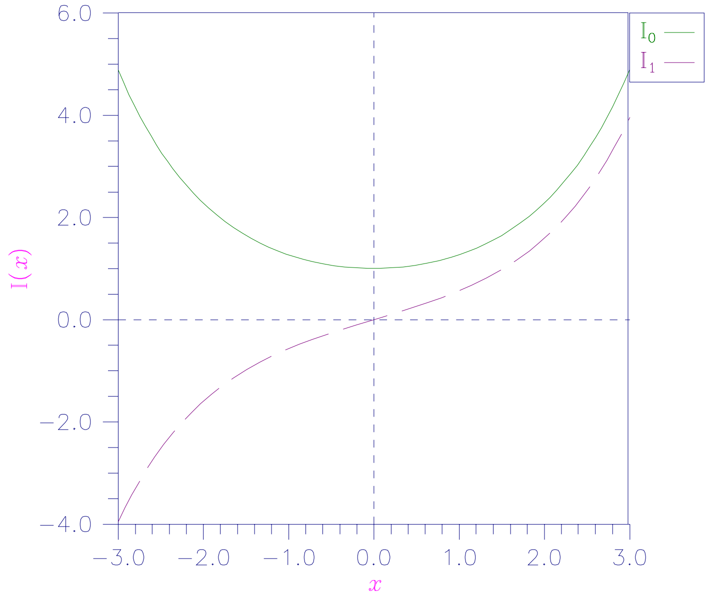

Figure 9-11 Plot of I0(x) and I1(x)

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 $e^{-|x|} I_0(x)$. If no solution can be computed, NaN is returned.

Description

The Bessel function is $I_0(x)$ is defined to be

$$I_0(x) = \frac{1}{\pi} \int_0^{\pi} \cosh(x \cos\theta)\, d\theta$$

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 $e^{-\lvert x \rvert} I_1(x)$. 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 $i = 0, \ldots, n - 1$.

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, $J_{0.3+\nu-1}$ $(1.2 + 0.5i)$, $\nu = 1, \ldots, 4$ 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_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 second 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

$$K_0(x) = \int_0^\infty \cos(x \sinh t)\, dt$$

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.

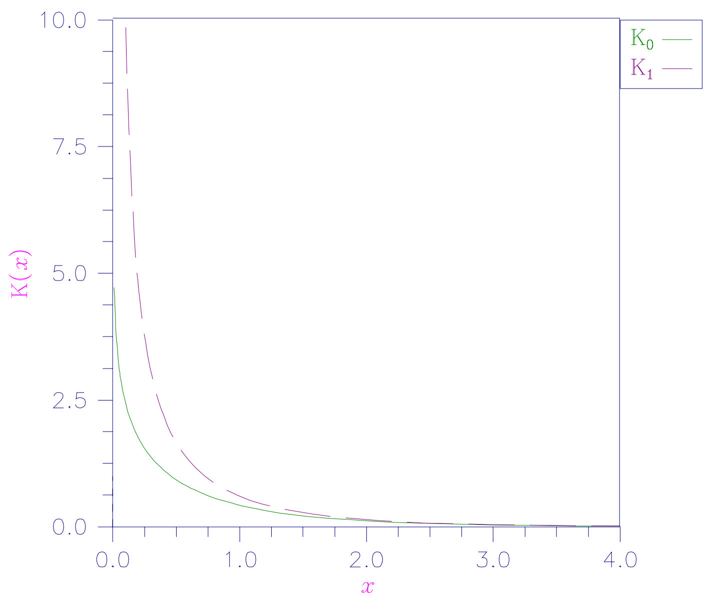

Figure 9-12 Plot of $K_0(x)$ and $K_1(x)$

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 second 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

$$K_0(x) = \int_0^\infty \cos(x \sinh t)\, dt$$

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 second 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

$$K_1(x) = \int_0^\infty \sin(x \sinh t) \sinh t \, dt$$

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 second 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

$$K_1(x) = \int_0^\infty \sin(x \sinh t) \sinh t \, dt$$

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 second 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 $i = 0, \ldots, n - 1$.

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

$$K_\nu\left(z\right) = \frac{\pi}{2} e^{\nu\pi i/2}\left[iJ_\nu\left(ze^{\pi i/2}\right) - Y_\nu\left(ze^{\pi i/2}\right)\right] \quad \text{for } -\pi < \arg z \le \frac{\pi}{2}$$

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 + 0.5*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_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

$$K(x) = \int_0^{\pi/2} \frac{d\theta}{\left[1 - x \sin^2 \theta\right]^{1/2}} \quad \text{for } 0 \le x < 1$$

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 502) 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 502) and `imsl_f_elliptic_integral_RD` (page 504). 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

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt$$

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 $K_v(x)$ is defined on page 495.

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

$$\mathrm{Bi}(x) = \frac{1}{\pi}\int_0^\infty \exp(xt - \frac{1}{3}t^3)dt + \frac{1}{\pi}\int_0^\infty \sin(xt + \frac{1}{3}t^3)dt$$

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` (page 492) and `bessel_Jx` (page 481)):

$$\mathrm{Bi}(x) = \sqrt{\frac{x}{3}}\left[I_{-1/3}(\frac{2}{3}x^{3/2}) + I_{1/3}(\frac{2}{3}x^{3/2}) \right] \text{ for } x > 0$$

and

$$\mathrm{Bi}(x) = \sqrt{\frac{-x}{3}}\left[J_{-1/3}(\frac{2}{3}|x|^{3/2}) - J_{1/3}(\frac{2}{3}|x|^{3/2}) \right] \text{ for } x < 0$$

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) (page 511). 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) (page 512). 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 $\mathrm{ber}_0(x)$ 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, 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 $\mathrm{bie}_0(x)$ 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, $\mathrm{bei}_0(0.4)$ 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, $\text{ker}_0(0.4)$ 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 $\mathrm{kei}_0(x)$ 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, $\mathrm{kei}_0(0.4)$ 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 `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, `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 `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 $\mathrm{ker}_0'(x)$ is defined to be

$$\frac{d}{dx}\mathrm{ker}_0(x)$$

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, $\mathrm{ker}_0'(0.6)$ 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 $\mathtt{kei}_0'(x)$ 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, $\mathtt{kei}_0'(0.6)$ 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:

$$\Phi(x) = \text{erfc}\left(-x/\sqrt{2.0}\right)/2$$

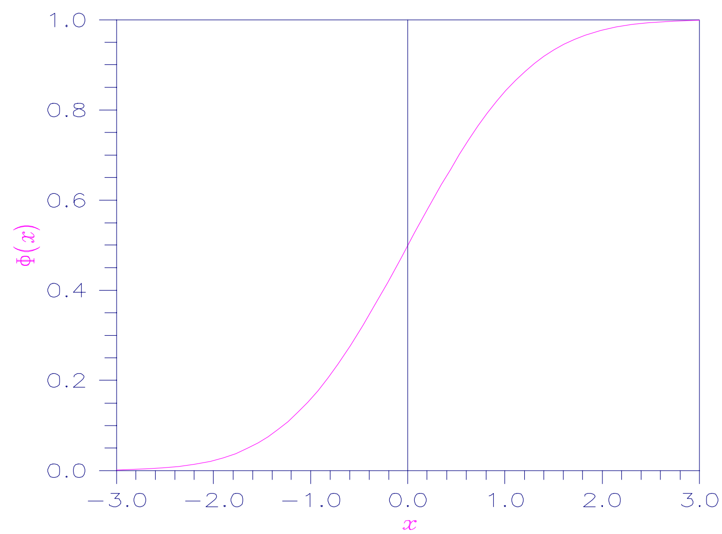

Figure 9-13 Plot of Φ(x)

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 Γ(·) 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 $\nu \le 65$, 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 $\nu \ge 100$, 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.

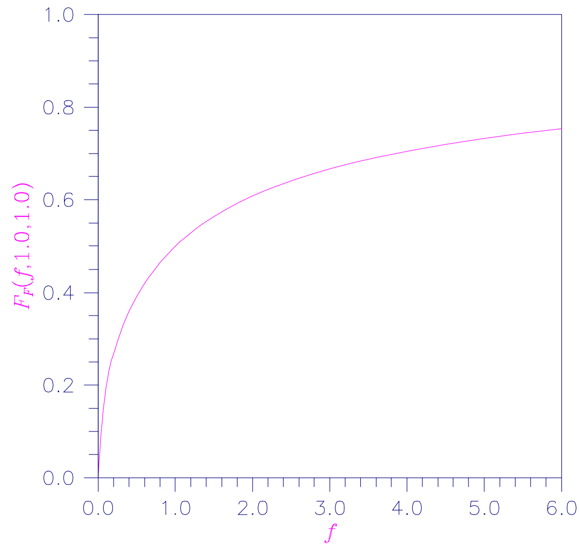

Figure 9-14 Plot of F_F (f, 1.0, 1.0)

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 $P \le 0.5$, `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
```

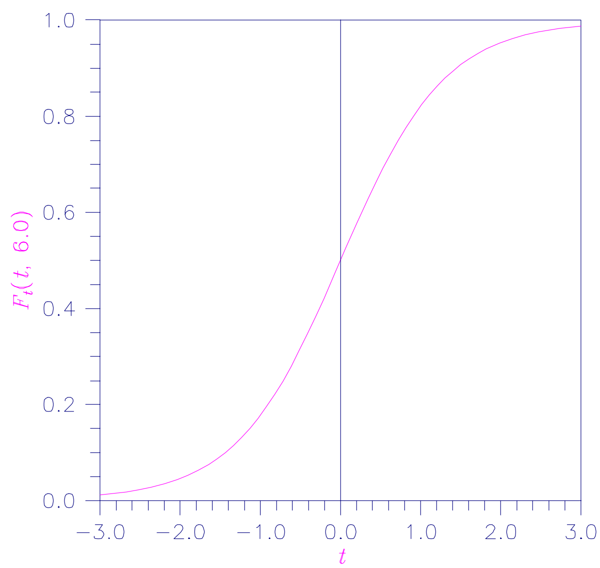

Figure 9-15   Plot of F_t(t, 6.0)

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

$$Pr\left(x=j\right)=\frac{\binom{m}{j}\binom{l-m}{n-j}}{\binom{l}{n}} \qquad \text{for } j=i, i+1, \ldots, \min\left(n,m\right)$$

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

$$f(x + 1) = f(x)q/(x + 1), \text{for } x = 0, 1, 2, \ldots, k - 1$$

with $f(0) = e^{-\theta}$.

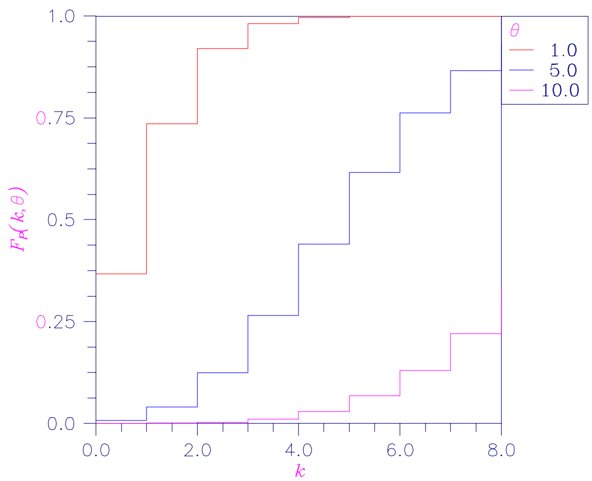

Figure 9-16 Plot of $F_p(k, \theta)$

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

$$P = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} \int_0^x t^{p-1}(1-t)^{q-1}\,dt$$

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 $X \le x$ 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.

Description

Function `imsl_f_bivariate_normal_cdf` evaluates the distribution function *F* of a bivariate normal distribution with means of zero, variances of one, and correlation of `rho`; that is, with ρ = `rho`, and |ρ| < 1,

$$F(x, y) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{x}\int_{-\infty}^{y} \exp\left(-\frac{u^2 - 2\rho uv + v^2}{2(1-\rho^2)}\right) du\ dv$$

To determine the probability that $U \le u_0$ and $V \le v_0$, where $(U, V)^T$ is a bivariate normal random variable with mean $\mu = (\mu_U, \mu_V)^T$ and variance-covariance matrix

$$\Sigma = \begin{pmatrix} \sigma_U^2 & \sigma_{UV} \\ \sigma_{UV} & \sigma_V^2 \end{pmatrix}$$

transform $(U, V)^T$ to a vector with zero means and unit variances. The input to `imsl_f_bivariate_normal_cdf` would be
`X` $= (u_0 - \mu_U)/\sigma_U$, `Y` $= (v_0 - \mu_V)/\sigma_V$, and $\rho = \sigma_{UV}/(\sigma_U\sigma_V)$.

Function `imsl_f_bivariate_normal_cdf` uses the method of Owen (1962, 1965). Computation of Owen's T-function is based on code by M. Patefield and D. Tandy (2000). For |ρ| = 1, the distribution function is computed based on the univariate statistic, *Z* = min(*x*, *y*), and on the normal distribution function `imsl_f_normal_cdf`, which can be found in Chapter 11, "Probablility Distribution Functions and Inverses."

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
```

cumulative_interest

Evaluates the cumulative interest paid between two periods.

Synopsis

#include `<imsl.h>`

float `imsl_f_cumulative_interest` (*float* `rate`, *int* `n_periods`,
float `present_value`, *int* `start`, *int* `end`, *int* `when`)

The type *double* function is `imsl_d_cumulative_interest`.

Required Arguments

float `rate` (Input)
Interest rate.

int `n_periods` (Input)
Total number of payment periods. `n_periods` cannot be less than or equal to 0.

float `present_value` (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

int `start` (Input)
Starting period in the calculation. `start` cannot be less than 1; or greater than `end`.

int `end` (Input)
Ending period in the calculation.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The cumulative interest paid between the first period and the last period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_cumulative_interest` evaluates the cumulative interest paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end} \text{interest}_i$$

where $interest_i$ is computed from `imsl_f_interest_payment` for the ith period.

Example

In this example, `imsl_f_cumulative_interest` computes the total interest paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

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

void main()
{
  float rate = 0.0725 / 12;
  int n_periods = 12 * 30;
  float present_value = 200000;
  int start = 1;
  int end = 12;
  float total;

  total = imsl_f_cumulative_interest (rate, n_periods, present_value,
                                      start, end, IMSL_AT_END_OF_PERIOD);

  printf ("First year interest = $%.2f.\n", total);
}
```

Output

```
First year interest = $-14436.52.
```

cumulative_principal

Evaluates the cumulative principal paid between two periods.

Synopsis

#include `<imsl.h>`

float `imsl_f_cumulative_principal` (*float* `rate`, *int* `n_periods`, *float* `present_value`, *int* `start`, *int* `end`, *int* `when`)

The type *double* function is `imsl_d_cumulative_principal`.

Required Arguments

float `rate` (Input)
Interest rate.

int `n_periods` (Input)
: Total number of payment periods. `n_periods` cannot be less than or equal to 0.

float `present_value` (Input)
: The current value of a stream of future payments, after discounting the payments using some interest rate.

int `start` (Input)
: Starting period in the calculation. `start` cannot be less than 1; or greater than `end`.

int `end` (Input)
: Ending period in the calculation.

int `when` (Input)
: Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The cumulative principal paid between the first period and the last period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_cumulative_principal` evaluates the cumulative principal paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end} \text{principal}_i$$

where $principal_i$ is computed from `imsl_f_principal_payment` for the ith period.

Example

In this example, `imsl_f_cumulative_principal` computes the total principal paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

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

void
main ()
{
  float rate = 0.0725 / 12;
  int n_periods = 12 * 30;
  float present_value = 200000;
  int start = 1;
```

```
  int end = 12;
  float total;

  total = imsl_f_cumulative_principal (rate, n_periods, present_value,
                                  start, end, IMSL_AT_END_OF_PERIOD);

  printf ("First year principal = $%.2f.\n", total);
}
```

Output

```
First year principal = $-1935.73.
```

depreciation_db

Evaluates the depreciation of an asset using the fixed-declining balance method.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_db` (*float* `cost`, *float* `salvage`, *int* `life`, *int* `period`, *int* `month`)

The type *double* function is `imsl_d_depreciation_db`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `life` (Input)
Number of periods over which the asset is being depreciated.

int `period` (Input)
Period for which the depreciation is to be computed. `period` cannot be less than or equal to 0, and cannot be greater than `life` +1.

int `month` (Input)
Number of months in the first year. `month` cannot be greater than 12 or less than 1.

Return Value

The depreciation of an asset for a specified period using the fixed-declining balance method. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_db` computes the depreciation of an asset for a specified period using the fixed-declining balance method. Routine `imsl_f_depreciation_db` varies depending on the specified value for the argument `period`, see table below.

period	Formula
period = 1	$\text{cost} \times \text{rate} \times \frac{\text{month}}{12}$
period = *life*	$\left(\text{cost} - \text{total depreciation from periods}\right) \times \text{rate} \times \frac{\text{12-month}}{12}$
period other than 1 or *life*	$\left(\text{cost} - \text{total depreciation from prior periods}\right) \times rate$

where

$$rate = 1 - \left(\frac{\text{salvage}}{\text{cost}}\right)^{\left(\frac{1}{life}\right)}$$

NOTE: *rate* is rounded to three decimal places.

Example

In this example, `imsl_f_depreciation_db` computes the depreciation of an asset, which costs $2,500 initially, a useful life of 3 periods and a salvage value of $500, for each period.

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

void main()
{
  float cost = 2500;
  float salvage = 500;
  int life = 3;
  int month = 6;
  float db;
  int period;

  for (period = 1; period <= life + 1; period++)
    {
      db = imsl_f_depreciation_db (cost, salvage, life, period, month);
      printf ("For period %i, db = $%.2f.\n", period, db);
    }
}
```

Output

```
For period 1, db = $518.75.
For period 2, db = $822.22.
For period 3, db = $481.00.
For period 4, db = $140.69.
```

depreciation_ddb

Evaluates the depreciation of an asset using the double-declining balance method.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_ddb` (*float* `cost`, *float* `salvage`, *int* `life`, *int* `period`, *float* `factor`)

The type *double* function is `imsl_d_depreciation_ddb`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `life` (Input)
Number of periods over which the asset is being depreciated.

int `period` (Input)
Period for which the depreciation is to be computed. `period` cannot be greater than `life`.

float `factor` (Input)
Rate at which the balance declines. `factor` must be positive.

Return Value

The depreciation of an asset using the double-declining balance method for a period specified by the user. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_ddb` computes the depreciation of an asset using the double-declining balance method for a specified period.

It is computed using the following:

$$\left[\text{cost} - \text{salvage}\left(\text{total depreciation from prior periods}\right)\right]\left(\frac{factor}{life}\right)$$

Example

In this example, `imsl_f_depreciation_ddb` computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500, for each period.

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

void main()
{
  float cost = 2500;
  float salvage = 500;
  float factor = 2;
  int life = 24;
  int period;
  float ddb;

  for (period = 1; period <= life; period++)
    {
      ddb = imsl_f_depreciation_ddb (cost, salvage, life, period, factor);
      printf ("For period %i, ddb = $%.2f.\n", period, ddb);
    }
}
```

Output

```
For period 1, ddb = $208.33.
For period 2, ddb = $190.97.
For period 3, ddb = $175.06.
For period 4, ddb = $160.47.
For period 5, ddb = $147.10.
For period 6, ddb = $134.84.
For period 7, ddb = $123.60.
For period 8, ddb = $113.30.
For period 9, ddb = $103.86.
For period 10, ddb = $95.21.
For period 11, ddb = $87.27.
For period 12, ddb = $80.00.
For period 13, ddb = $73.33.
For period 14, ddb = $67.22.
For period 15, ddb = $61.62.
For period 16, ddb = $56.48.
For period 17, ddb = $51.78.
For period 18, ddb = $47.46.
For period 19, ddb = $22.09.
For period 20, ddb = $0.00.
For period 21, ddb = $0.00.
For period 22, ddb = $0.00.
For period 23, ddb = $0.00.
For period 24, ddb = $0.00.
```

depreciation_sln

Evaluates the depreciation of an asset using the straight-line method.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_sln` (*float* `cost`, *float* `salvage`, *int* `life`)

The type *double* function is `imsl_d_depreciation_sln`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `life` (Input)
Number of periods over which the asset is being depreciated.

Return Value

The straight line depreciation of an asset for its life. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_sln` computes the straight line depreciation of an asset for its life.

It is computed using the following:

(cost-salvage)/life

Example

In this example, `imsl_f_depreciation_sln` computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500.

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

void main()
{
  float cost = 2500;
  float salvage = 500;
  int life = 24;
  float depreciation_sln;

  depreciation_sln = imsl_f_depreciation_sln (cost, salvage, life);
  printf ("The straight line depreciation of the asset for one ");
  printf ("period is $%.2f.\n", depreciation_sln);
}
```

Output

```
The straight line depreciation of the asset for one period is $83.33.
```

depreciation_syd

Evaluates the depreciation of an asset using the sum-of-years digits method.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_syd` (*float* `cost`, *float* `salvage`, *int* `life`, *int* `period`)

The type *double* function is `imsl_d_depreciation_syd`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `life` (Input)
Number of periods over which the asset is being depreciated.

int `period` (Input)
Period for which the depreciation is to be computed. `period` cannot be greater than `life`.

Return Value

The sum-of-years digits depreciation of an asset for a specified period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_syd` computes the sum-of-years digits depreciation of an asset for a specified period.

It is computed using the following:

$$(cost - salvage)(period)\ \frac{(life+1)(life)}{2}$$

Example

In this example, `imsl_f_depreciation_syd` computes the depreciation of an asset, which costs \$25,000 initially, lasts 15 years and a salvage value of \$5,000, for the 14th year.

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

void main()
```

```
{
  float cost = 25000;
  float salvage = 5000;
  int life = 15;
  int period = 14;
  float depreciation_syd;

  depreciation_syd = imsl_f_depreciation_syd (cost, salvage, life, period);
  printf ("The depreciation allowance for the 14th year ");
  printf ("is $%.2f.\n", depreciation_syd);
}
```

Output

```
The depreciation allowance for the 14th year is $333.33.
```

depreciation_vdb

Evaluates the depreciation of an asset for any given period using the variable-declining balance method.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_vdb` (*float* `cost`, *float* `salvage`, *int* `life`, *int* `start`, *int* `end`, *float* `factor`, *int* `sln`)

The type *double* function is `imsl_d_depreciation_vdb`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `life` (Input)
Number of periods over which the asset is being depreciated.

int `start` (Input)
Starting period in the calculation. `start` cannot be less than 1; or greater than `end`.

int `end` (Input)
Final period for the calculation. `end` cannot be greater than `life`.

float `factor` (Input)
Rate at which the balance declines. `factor` must be positive.

int `sln` (Input)
If equal to zero, do not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.

Return Value

The depreciation of an asset for any given period, including partial periods, using the variable-declining balance method. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_vdb` computes the depreciation of an asset for any given period using the variable-declining balance method using the following:

If `sln` = 0

$$\sum_{i=start+1}^{end} ddb_i$$

If `sln` $\neq 0$

$$A + \sum_{i=k}^{end} \frac{\text{cost} - A - salvage}{end - k + 1}$$

where ddb_i is computed from `imsl_f_depreciation_ddb` for the ith period. k = the first period where straight line depreciation is greater than the depreciation using the double-declining balance method. $A = \sum_{i=start+1}^{k-1} ddb_i$.

Example

In this example, `imsl_f_depreciation_vdb` computes the depreciation of an asset between the 10th and 15th year, which costs $25,000 initially, lasts 15 years and has a salvage value of $5,000.

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

void main()
{
  float cost = 25000;
  float salvage = 5000;
  int life = 15;
  int start = 10;
  int end = 15;
  float factor = 2.;
  int sln = 0;
  float vdb;

  vdb = imsl_f_depreciation_vdb (cost, salvage, life, start,
                                  end, factor, sln);
  printf ("The depreciation allowance between the 10th and 15th ");
  printf ("year is $%.2f.\n", vdb);
}
```

Output

```
The depreciation allowance between the 10th and 15th year is $976.69.
```

dollar_decimal

Converts a fractional price to a decimal price.

Synopsis

#include `<imsl.h>`

float `imsl_f_dollar_decimal` (*float* `fractional_dollar`*,* *int* `fraction`)

The type *double* function is `imsl_d_dollar_decimal`.

Required Arguments

float `fractional_dollar` (Input)
Whole number of dollars plus the numerator, as the fractional part.

int `fraction` (Input)
Denominator of the fractional dollar. `fraction` must be positive.

Return Value

The dollar price expressed as a decimal number. The dollar price is the whole number part of fractional-dollar plus its decimal part divided by fraction. If no result can be computed, NaN is returned.

Description

Function `imsl_f_dollar_decimal` converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number.

It is computed using the following:

$$idollar + \left[fractional\_dollar - idollar \right] * \frac{10^{(ifrac+1)}}{fraction}$$

where *idollar* is the integer part of *fractional_dollar*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, `imsl_f_dollar_decimal` converts $ 1 1/4 to $1.25.

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

void main()
{
  float fractional_dollar = 1.1;
```

```
  int fraction = 4;
  float dollardec;

  dollardec = imsl_f_dollar_decimal (fractional_dollar, fraction);
  printf ("The fractional dollar $1 1/4 = $%.2f.\n", dollardec);
}
```

Output

```
The fractional dollar $1 1/4 = $1.25.
```

dollar_fraction

Converts a decimal price to a fractional price.

Synopsis

#include `<imsl.h>`

float `imsl_f_dollar_fraction` (*float* `decimal_dollar`, *int* `fraction`)

The type *double* function is `imsl_d_dollar_fraction`.

Required Arguments

float `decimal_dollar` (Input)
Dollar price expressed as a decimal number.

int `fraction` (Input)
Denominator of the fractional dollar. `fraction` must be positive.

Return Value

The dollar price expressed as a fraction. The numerator is the decimal part of the return value. If no result can be computed, NaN is returned.

Description

Function `imsl_f_dollar_fraction` converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fractional price. If no result can be computed, NaN is returned.

It can be found by solving the following

$$\mathrm{i}dollar + \frac{\left[decimal\_dollar - idollar\right]}{10^{(ifrac+1)} / fraction}$$

where *idollar* is the integer part of the *decimal_dollar*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, `imsl_f_dollar_fraction` converts $ 1.25 to $1 1/4.

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

void main()
{
  float decimal_dollar = 1.25;
  int fraction = 4;
  int numerator;
  float dollarfrc;

  dollarfrc = imsl_f_dollar_fraction (decimal_dollar, fraction);
  numerator = dollarfrc*10.-((int)dollarfrc)*10;
  printf ("The decimal dollar $1.25 as a fractional dollar = $%i %i/%i.\n",
         (int)dollarfrc, numerator, fraction);
}
```

Output

```
The decimal dollar $1.25 as a fractional dollar = $1 1/4.
```

effective_rate

Evaluates the effective annual interest rate.

Synopsis

#include `<imsl.h>`

float `imsl_f_effective_rate` (*float* `nominal_rate`, *int* `n_periods`)

The type *double* function is `imsl_d_effective_rate`.

Required Arguments

float `nominal_rate` (Input)
The interest rate as stated on the face of a security.

int `n_periods` (Input)
Number of compounding periods per year.

Return Value

The effective annual interest rate. If no result can be computed, NaN is returned.

Description

Function `imsl_f_effective_rate` computes the continuously-compounded interest rate equivalent to a given periodically-compounded interest rate. The nominal interest rate is the periodically-compounded interest rate as stated on the face of a security.

It can found by solving the following:

$$\left(1+\frac{\text{nominal\_}rate}{n\_periods}\right)^{(n\_periods)}-1$$

Example

In this example, `imsl_f_effective_rate` computes the effective annual interest rate of the nominal interest rate, 6%, compounded quarterly.

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

void main()
{
  float nominal_rate = .06;
  int n_periods = 4;
  float effective_rate;

  effective_rate = imsl_f_effective_rate (nominal_rate, n_periods);
  printf ("The effective rate of the nominal rate, 6.0%%, ");
  printf ("compounded quarterly is %.2f%%.\n", effective_rate * 100.);
}
```

Output

```
The effective rate of the nominal rate, 6.0%, compounded quarterly is 6.14%.
```

future_value

Evaluates the future value of an investment.

Synopsis

#include `<imsl.h>`

float `imsl_f_future_value` (*float* `rate`, *int* `n_periods`, *float* `payment`, *float* `present_value`, *int* `when`)

The type *double* function is `imsl_d_future_value`.

Required Arguments

float `rate` (Input)
Interest rate.

int `n_periods` (Input)
Total number of payment periods.

float `payment` (Input)
Payment made in each period.

float `present_value` (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The future value of an investment. If no result can be computed, NaN is returned.

Description

Function `imsl_f_future_value` computes the future value of an investment. The future value is the value, at some time in the future, of a current amount and a stream of payments.

It can be found by solving the following:

If `rate` = 0

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If `rate` ≠ 0

$$present\_value(1+rate)^{\mathrm{n\_periods}} + payment\left[1+rate(when)\right]\frac{(1+rate)^{\mathrm{n\_periods}}-1}{rate}$$

$$+\mathrm{future\_value}=0$$

Example

In this example, `imsl_f_future_value` computes the value of $30,000 payment made annually at the beginning of each year for the next 20 years with an annual interest rate of 5%.

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

void
main ()
{
  float rate = .05;
  int n_periods = 20;
  float payment = -30000.00;
  float present_value = -30000.00;
  int when = IMSL_AT_BEGINNING_OF_PERIOD;
  float future_value;

  future_value = imsl_f_future_value (rate, n_periods, payment,
                                      present_value, when);
  printf ("After 20 years, the value of the investments ");
  printf ("will be $%.2f.\n", future_value);
}
```

Output

```
After 20 years, the value of the investments will be $1121176.63.
```

future_value_schedule

Evaluates the future value of an initial principal taking into consideration a schedule of compound interest rates.

Synopsis

#include `<imsl.h>`

float `imsl_f_future_value_schedule` (*float* `principal`, *int* `count`, *float* `schedule[]`)

The type *double* function is `imsl_d_future_value_schedule`.

Required Arguments

float `principal` (Input)
Principal or present value.

int `count` (Input)
Number of interest rates in `schedule`.

float `schedule[]` (Input)
Array of size `count` of interest rates to apply.

Return Value

The future value of an initial principal after applying a schedule of compound interest rates. If no result can be computed, NaN is returned.

Description

Function `imsl_f_future_value_schedule` computes the future value of an initial principal after applying a schedule of compound interest rates.

It is computed using the following:

$$\sum_{i=1}^{count} \left(principal * schedule_i \right)$$

where $schedule_i$ = interest rate at the ith period.

Example

In this example, `imsl_f_future_value_schedule` computes the value of a $10,000 investment after 5 years with interest rates of 5%, 5.1%, 5.2%, 5.3% and 5.4%, respectively.

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

```
#include "imsl.h"

void main()
{
  float principal = 10000.0;
  float schedule[5] = { .050, .051, .052, .053, .054 };
  float fvschedule;

  fvschedule = imsl_f_future_value_schedule (principal, 5, schedule);
  printf ("After 5 years the $10,000 investment will have grown ");
  printf ("to $%.2f.\n", fvschedule);
}
```

Output

```
After 5 years the $10,000 investment will have grown to $12884.77.
```

interest_payment

Evaluates the interest payment for an investment for a given period.

Synopsis

#include `<imsl.h>`

float `imsl_f_interest_payment` (*float* `rate`, *int* `period`, *int* `n_periods`, *float* `present_value`, *float* `future_value`, *int* `when`)

The type *double* function is `imsl_d_interest_payment`.

Required Arguments

float `rate` (Input)
Interest rate.

int `period` (Input)
Payment period.

int `n_periods` (Input)
Total number of periods.

float `present_value` (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

float `future_value` (Input)
The value, at some time in the future, of a current amount and a stream of payments.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on see the Usage Notes section of this chapter.

Return Value

The interest payment for an investment for a given period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_interest_payment` computes the interest payment for an investment for a given period.

It is computed using the following:

$$\left\{ present\_value\left(1+rate\right)^{n\_periods-1} + payment\left(1+rate*when\right)\left[\frac{\left(1+rate\right)^{n\_periods-1}}{rate}\right]\right\} rate$$

Example

In this example, `imsl_f_interest_payment` computes the interest payment for the second year of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

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

void main()
{
  float rate = .08;
  int period = 2;
  int n_periods = 25;
  float present_value = 100000.00;
  float future_value = 0.0;
  int when = IMSL_AT_END_OF_PERIOD;
  float interest_payment;

  interest_payment = imsl_f_interest_payment (rate, period, n_periods,
                                    present_value, future_value, when);
  printf ("The interest due the second year on the $100,000 ");
  printf ("loan is $%.2f.\n", interest_payment);
}
```

Output

```
The interest due the second year on the $100,000 loan is $-7890.57.
```

interest_rate_annuity

Evaluates the interest rate per period of an annuity.

Synopsis

#include `<imsl.h>`

float imsl_f_interest_rate_annuity (*int* n_periods, *float* payment, *float* present_value, *float* future_value, *int* when, …, 0)

The type *double* function is imsl_d_interest_rate_annuity.

Required Arguments

int n_periods (Input)
Total number of periods.

float payment (Input)
Payment made each period.

float present_value (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

float future_value (Input)
The value, at some time in the future, of a current amount and a stream of payments.

int when (Input)
Time in each period when the payment is made, either IMSL_AT_END_OF_PERIOD or IMSL_AT_BEGINNING_OF_PERIOD. For a more detailed discussion on when see the Usage Notes section of this chapter.

Return Value

The interest rate per period of an annuity. If no result can be computed, NaN is returned.

Synopsis with Optional Arguments

#include <imsl.h>

float imsl_f_interest_rate_annuity (*int* n_periods, *float* payment, *float* present_value, *float* future_value, *int* when, IMSL_XGUESS, *float* guess, IMSL_HIGHEST, *float* max, 0)

Optional Arguments

IMSL_XGUESS, *float* guess (Input)
Initial guess at the interest rate.

IMSL_HIGHEST, *float* max (Input)
Maximum value of the interest rate allowed.
Default: 1.0 (100%)

Description

Function imsl_f_interest_rate_annuity computes the interest rate per period of an annuity. An annuity is a security that pays a fixed amount at equally spaced intervals.

It can be found by solving the following:

If `rate` = 0

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If `rate` ≠ 0

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate}$$

$$+future\_value=0$$

Example

In this example, `imsl_f_interest_rate_annuity` computes the interest rate of a $20,000 loan that requires 70 payments of $350 each to pay off.

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

void main()
{
  float rate;
  int n_periods = 70;
  float payment = -350.;
  float present_value = 20000;
  float future_value = 0.;
  int when = IMSL_AT_BEGINNING_OF_PERIOD;

  rate = imsl_f_interest_rate_annuity (n_periods, payment, present_value,
                  future_value, when, 0) * 12;
  printf ("The computed interest rate on the loan is ");
  printf ("%.2f%%.\n", rate * 100.);
}
```

Output

```
The computed interest rate on the loan is 7.35%.
```

internal_rate_of_return

Evaluates the internal rate of return for a schedule of cash flows.

Synopsis

#include `<imsl.h>`

float `imsl_f_internal_rate_of_return` (*int* `count`, *float* `values[]`, …, `0`)

The type *double* function is `imsl_d_internal_rate_of_return`.

Required Arguments

int `count` (Input)
Number of cash flows in `values`. `count` must be greater than one.

float `values[]` (Input)
Array of size `count` of cash flows which occur at regular intervals, which includes the initial investment.

Return Value

The internal rate of return for a schedule of cash flows. If no result can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_internal_rate_of_rtn` (*int* `count`, *float* `values[]`,
`IMSL_XGUESS`, *float* `guess`, `0`)

Optional Arguments

`IMSL_XGUESS`, *float* `guess` (Input)
Initial guess at the internal rate of return.

`IMSL_HIGHEST`, *float* `max` (Input)
Maximum value of the internal rate of return allowed.
Default: 1.0 (100%).

Description

Function `imsl_f_internal_rate_of_return` computes the internal rate of return for a schedule of cash flows. The internal rate of return is the interest rate such that a stream of payments has a net present value of zero.

It is found by solving the following:

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the *i*th cash flow, *rate* is the internal rate of return.

Example

In this example, `imsl_f_internal_rate_of_return` computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

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

void main()
```

```
{
  float values[] = { -4500., -800., 800., 800., 600.,
                      600., 800., 800., 700., 3000. };
  float internal_rate;

  internal_rate = imsl_f_internal_rate_of_return (10, values, 0);
  printf ("After 9 years, the internal rate of return on the ");
  printf ("cows is %.2f%%.\n", internal_rate * 100.);
}
```

Output

```
After 9 years, the internal rate of return on the cows is 7.21%.
```

internal_rate_schedule

Evaluates the internal rate of return for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Synopsis

#include `<imsl.h>`

float `imsl_f_internal_rate_schedule` (*int* `count`, *float* `values[]`,
struct tm `dates[]`, …, `0`)

The type *double* function is `imsl_d_internal_rate_schedule`.

Required Arguments

int `count` (Input)
Number of cash flows in `values`. `count` must be greater than one.

float `values[]` (Input)
Array of size `count` of cash flows, which includes the initial investment.

struct tm `dates[]` (Input)
Array of size `count` of dates cash flows are made see the Usage Notes section of this chapter.

Return Value

The internal rate of return for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_internal_rate_schedule` (*int* `count`, *float* `values[]`,
struct tm `dates[]`, `IMSL_XGUESS`, *float* `guess`, `IMSL_HIGHEST`,
float `max`, `0`)

Optional Arguments

`IMSL_XGUESS`, *float* `guess` (Input)
Initial guess at the internal rate of return.

`IMSL_HIGHEST`, *float* `max` (Input)
Maximum value of the internal rate of return allowed.
Default: 1.0 (100%)

Description

Function `imsl_f_internal_rate_schedule` computes the internal rate of return for a schedule of cash flows that is not necessarily periodic. The internal rate such that the stream of payments has a net present value of zero.

It can be found by solving the following:

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{\frac{d_i - d_1}{365}}}$$

In the equation above, d_i represents the ith payment date. d_1 represents the 1st payment date. $value_i$ represents the ith cash flow. *rate* is the internal rate of return.

Example

In this example, `imsl_f_internal_rate_schedule` computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

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

void main()
{
  float values[10] = { -4500., -800., 800., 800., 600., 600.,
                         800., 800., 700., 3000. };
  struct tm dates[10];
  float xirr;

  dates[0].tm_year = 98; dates[0].tm_mon = 0; dates[0].tm_mday = 1;
  dates[1].tm_year = 98; dates[1].tm_mon = 9; dates[1].tm_mday = 1;
  dates[2].tm_year = 99; dates[2].tm_mon = 4; dates[2].tm_mday = 5;
  dates[3].tm_year = 100; dates[3].tm_mon = 4; dates[3].tm_mday = 5;
  dates[4].tm_year = 101; dates[4].tm_mon = 5; dates[4].tm_mday = 1;
  dates[5].tm_year = 102; dates[5].tm_mon = 6; dates[5].tm_mday = 1;
  dates[6].tm_year = 103; dates[6].tm_mon = 7; dates[6].tm_mday = 30;
  dates[7].tm_year = 104; dates[7].tm_mon = 8; dates[7].tm_mday = 15;
  dates[8].tm_year = 105; dates[8].tm_mon = 9; dates[8].tm_mday = 15;
  dates[9].tm_year = 106; dates[9].tm_mon = 10; dates[9].tm_mday = 1;

  xirr = imsl_f_internal_rate_schedule (10, values, dates, 0);
  printf ("After approximately 9 years, the internal\n");
  printf ("rate of return on the cows is %.2f%%.\n", xirr * 100.);
}
```

Output

```
After approximately 9 years, the internal
rate of return on the cows is 7.69%.
```

modified_internal_rate

Evaluates the modified internal rate of return for a schedule of periodic cash flows.

Synopsis

#include `<imsl.h>`

float `imsl_f_modified_internal_rate` (*int* `count`, *float* `values[]`, *float* `finance_rate`, *float* `reinvest_rate`)

The type *double* function is `imsl_d_modified_internal_rate`.

Required Arguments

int `count` (Input)
Number of cash flows in `values` and `count` must greater than one.

float `values[]` (Input)
Array of size `count` of cash flows.

float `finance_rate` (Input)
Interest paid on the money borrowed.

float `reinvest_rate` (Input)
Interest rate received on the cash flows.

Return Value

The modified internal rate of return for a schedule of periodic cash flows. If no result can be computed, NaN is returned.

Description

Function `imsl_f_modified_internal_rate` computes the modified internal rate of return for a schedule of periodic cash flows. The modified internal rate of return differs from the ordinary internal rate of return in assuming that the cash flows are reinvested at the cost of capital, not at the internal rate of return.

It also eliminates the multiple rates of return problem.

It is computed using the following:

$$\left\{\left[\frac{-(\text{pnpv})(1+\text{reinvest\_rate})^{n\_periods}}{(nnpv)(1+\text{finance\_rate})}\right]^{\frac{1}{n\_periods-1}}\right\}-1$$

where *pnpv* is calculated from `imsl_f_net_present_value` for positive values in `values` using `reinvest_rate`, and where *nnpv* is calculated from `imsl_f_net_present_value` for negative values in `values` using `finance_rate`.

Example

In this example, `imsl_f_modified_internal_rate` computes the modified internal rate of return for an investment of $4,500 with cash flows of $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000 for 9 years.

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

void main()
{
  float value[] = { -4500., -800., 800., 800., 600., 600., 800.,
                        800., 700., 3000. };
  float finance_rate = .08;
  float reinvest_rate = .055;
  float mirr;

  mirr = imsl_f_modified_internal_rate (10, value, finance_rate,
                                        reinvest_rate);
  printf ("After 9 years, the modified internal rate of return ");
  printf ("on the cows is %.2f%%.\n", mirr * 100.);
}
```

Output

```
After 9 years, the modified internal rate of return on the cows is 6.66%.
```

net_present_value

Evaluates the net present value of a stream of unequal periodic cash flows, which are subject to a given discount rate.

Synopsis

#include `<imsl.h>`

float `imsl_f_net_present_value` (*float* `rate`, *int* `count`, *float* `values[]`)

The type *double* function is `imsl_d_net_present_value`.

Required Arguments

float `rate` (Input)
Interest rate per period.

int `count` (Input)
Number of cash flows in `values`.

float `values[]` (Input)
Array of size `count` of equally-spaced cash flows.

Return Value

The net present value of an investment. If no result can be computed, NaN is returned.

Description

Function `imsl_f_net_present_value` computes the net present value of an investment. Net present value is the current value of a stream of payments, after discounting the payments using some interest rate.

It is found by solving the following:

$$\sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the *i*th cash flow.

Example

In this example, `imsl_f_net_present_value` computes the net present value of a $10 million prize paid in 20 years ($50,000 per year) with an annual interest rate of 6%.

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

void main()
{
  float rate = 0.06;
  int count = 20;
  float value[20];
  float net_present_value;
  int i;

  for (i = 0; i < count; i++)
      value[i] = 500000.;

  net_present_value = imsl_f_net_present_value (rate, count, value);

  printf ("The net present value of the $10 million prize is $%.2f.\n",
              net_present_value);
}
```

Output

```
The net present value of the $10 million prize is $5734963.00.
```

nominal_rate

Evaluates the nominal annual interest rate.

Synopsis

#include `<imsl.h>`

float `imsl_f_nominal_rate` (*float* `effective_rate`, *int* `n_periods`)

The type *double* function is `imsl_d_nominal_rate`.

Required Arguments

float `effective_rate` (Input)
: The amount of interest that would be charged if the interest was paid in a single lump sum at the end of the loan.

int `n_periods` (Input)
: Number of compounding periods per year.

Return Value

The nominal annual interest rate. If no result can be computed, NaN is returned.

Description

Function `imsl_f_nominal_rate` computes the nominal annual interest rate. The nominal interest rate is the interest rate as stated on the face of a security.

It is computed using the following:

$$\left[(1+\text{effective\_rate})^{\frac{1}{n\_periods}}-1\right]*\text{n\_periods}$$

Example

In this example, `imsl_f_nominal_rate` computes the nominal annual interest rate of the effective interest rate, 6.14%, compounded quarterly.

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

void main()
{
  double effective_rate = .0614;
  int n_periods = 4;
  double nominal_rate;

  nominal_rate = imsl_d_nominal_rate (effective_rate, n_periods);
  printf ("The nominal rate of the effective rate, 6.14%%, \n");
  printf ("compounded quarterly is %.2f%%.\n", nominal_rate * 100.);
}
```

Output

```
The nominal rate of the effective rate, 6.14%,
compounded quarterly is 6.00%.
```

number_of_periods

Evaluates the number of periods for an investment for which periodic and constant payments are made and the interest rate is constant.

Synopsis

#include `<imsl.h>`

float `imsl_f_number_of_periods` (*float* `rate`, *float* `payment`, *float* `present_value`, *float* `future_value`, *int* `when`)

The type *double* function is `imsl_d_number_of_periods`.

Required Arguments

float `rate` (Input)
Interest rate on the investment.

float `payment` (Input)
Payment made on the investment.

float `present_value` (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

float `future_value` (Input)
The value, at some time in the future, of a current amount and a stream of payments.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The number of periods for an investment.

Description

Function `imsl_f_number_of_periods` computes the number of periods for an investment based on periodic, constant payment and a constant interest rate.

It can be found by solving the following:

If `rate` = 0

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If `rate` ≠ 0

$$present\_value(1+rate)^{\text{n\_periods}} + payment\left[1+rate(when)\right]\frac{(1+rate)^{\text{n\_periods}}-1}{rate}$$

$$+\text{future\_value}=0$$

Example

In this example, `imsl_f_number_of_periods` computes the number of periods needed to pay off a $20,000 loan with a monthly payment of $350 and an annual interest rate of 7.25%. The payment is made at the beginning of each period.

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

void main()
{
  float rate = 0.0725 / 12;
  float payment = -350.;
  float present_value = 20000;
  float future_value = 0.;
  int when = IMSL_AT_BEGINNING_OF_PERIOD;
  float number_of_periods;

  number_of_periods = imsl_f_number_of_periods (rate, payment,
                                  present_value, future_value, when);

  printf ("Number of payment periods = %f.\n", number_of_periods);
}
```

Output

```
Number of payment periods = 70.
```

payment

Evaluates the periodic payment for an investment.

Synopsis

#include `<imsl.h>`

float `imsl_f_payment` (*float* `rate`, *int* `n_periods`, *float* `present_value`,
float `future_value`, *int* `when`)

The type *double* function is `imsl_d_payment`.

Required Arguments

float `rate` (Input)
Interest rate.

int `n_periods` (Input)
Total number of periods.

float `present_value` (Input)
: The current value of a stream of future payments, after discounting the payments using some interest rate.

float `future_value` (Input)
: The value, at some time in the future, of a current amount and a stream of payments.

int `when` (Input)
: Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The periodic payment for an investment. If no result can be computed, NaN is returned.

Description

Function `imsl_f_payment` computes the periodic payment for an investment.

It can be found by solving the following:

If `rate` = 0

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If `rate` ≠ 0

$$present\_value(1+rate)^{\text{n\_periods}} + payment\left[1+rate(when)\right]\frac{(1+rate)^{\text{n\_periods}}-1}{rate}$$

$$+\text{future\_value}=0$$

Example

In this example, `imsl_f_payment` computes the periodic payment of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

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

void main()
{
  float rate = .08;
  int n_periods = 25;
  float present_value = 100000.00;
  float future_value = 0.0;
  int when = IMSL_AT_END_OF_PERIOD;
  float payment;

  payment = imsl_f_payment (rate, n_periods, present_value,
```

```
                              future_value, when);
  printf ("The payment due each year on the $100,000 ");
  printf ("loan is $%.2f.\n", payment);
}
```

Output

```
The payment due each year on the $100,000 loan is $-9367.88.
```

present_value

Evaluates the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate..

Synopsis

#include `<imsl.h>`

float `imsl_f_present_value` (*float* `rate`, *int* `n_periods`, *float* `payment`, *float* `future_value`, *int* `when`)

The type *double* function is `imsl_d_present_value`.

Required Arguments

float `rate` (Input)
Interest rate.

int `n_periods` (Input)
Total number of periods.

float `payment` (Input)
Payment made in each period.

float `future_value` (Input)
The value, at some time in the future, of a current amount and a stream of payments.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The present value of an investment. If no result can be computed, NaN is returned.

Description

Function `imsl_f_present_value` computes the present value of an investment.

It can be found by solving the following:

If `rate` = 0

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If `rate` $\neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate}$$

$$+future\_value = 0$$

Example

In this example, `imsl_f_present_value` computes the present value of 20 payments of $500,000 per payment ($10 million) with an annual interest rate of 6%. The payment is made at the end of each period.

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

void main()
{
  float rate = 0.06;
  float payment = 500000.;
  float future_value = 0.;
  int n_periods = 20;
  int when = IMSL_AT_END_OF_PERIOD;
  float present_value;

  present_value = imsl_f_present_value (rate, n_periods, payment,
                                        future_value, when);

  printf ("The present value of the $10 million prize is ");
  printf ("$%.2f.\n", present_value);
}
```

Output

```
The present value of the $10 million prize is $-5734961.00.
```

present_value_schedule

Evaluates the present value for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Synopsis

#include `<imsl.h>`

float `imsl_f_present_value_schedule` (*float* `rate`, *int* `count`,
    float `values[]`, *struct tm* `dates[]`)

The type *double* function is `imsl_d_present_value_schedule`.

Required Arguments

float `rate` (Input)
Interest rate.

int `count` (Input)
Number of cash flows in `values` or number of dates in `dates`.

float `values[]` (Input)
Array of size `count` of cash flows.

struct tm `dates[]` (Input)
Array of size `count` of dates cash flows are made. For a more detailed discussion on dates see the Usage Notes section of this chapter.

Return Value

The present value for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Description

Function `imsl_f_present_value_schedule` computes the present value for a schedule of cash flows that is not necessarily periodic.

It can be found by solving the following:

$$\sum_{i=1}^{count} \frac{value_i}{\left(1+rate\right)^{\left(d_i-d_1\right)/365}}$$

In the equation above, d_i represents the ith payment date, d_1 represents the 1st payment date, and $value_I$ represents the ith cash flow.

Example

In this example, `imsl_f_present_value_schedule` computes the present value of 3 payments, $1,000, $2,000 and $1,000, with an interest rate of 5% made on January 3, 1997, January 3, 1999 and January 3, 2000.

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

void main()
{
  float rate = 0.05;
  float values[3] = { 1000.0, 2000.0, 1000.0 };
  struct tm dates[3];
  float xnpv;

  dates[0].tm_year = 97; dates[0].tm_mon = 0; dates[0].tm_mday = 3;
  dates[1].tm_year = 99; dates[1].tm_mon = 0; dates[1].tm_mday = 3;
  dates[2].tm_year = 100; dates[2].tm_mon = 0; dates[2].tm_mday = 3;
```

```
  xnpv = imsl_f_present_value_schedule (rate, 3, values, dates);
  printf ("The present value of the cash flows is $%.2f.\n", xnpv);
}
```

Output

```
The present value of the cash flows is $3677.90.
```

principal_payment

Evaluates the payment on the principal for a specified period.

Synopsis

#include `<imsl.h>`

float `imsl_f_principal_payment` (*float* `rate`, *int* `period`, *int* `n_periods`, *float* `present_value`, *float* `future_value`, *int* `when`)

The type *double* function is `imsl_d_principal_payment`.

Required Arguments

float `rate` (Input)
Interest rate.

int `period` (Input)
Payment period.

int `n_periods` (Input)
Total number of periods.

float `present_value` (Input)
The current value of a stream of future payments, after discounting the payments using some interest rate.

float `future_value` (Input)
The value, at some time in the future, of a current amount and a stream of payments.

int `when` (Input)
Time in each period when the payment is made, either `IMSL_AT_END_OF_PERIOD` or `IMSL_AT_BEGINNING_OF_PERIOD`. For a more detailed discussion on `when` see the Usage Notes section of this chapter.

Return Value

The payment on the principal for a given period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_principal_payment` computes the payment on the principal for a given period.

It is computed using the following:

$$payment_i - interest_i$$

where $payment_i$ is computed from `imsl_f_payment` for the *i*th period, $interest_i$ is calculated from `imsl_f_interest_payment` for the *i*th period.

Example

In this example, `imsl_f_principal_payment` computes the principal paid for the first year on a 30-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each year.

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

void main()
{
  float rate = .08;
  int period = 1;
  int n_periods = 30;
  float present_value = 100000.00;
  float future_value = 0.0;
  int when = IMSL_AT_END_OF_PERIOD;
  float principal;

  principal = imsl_f_principal_payment (rate, period, n_periods,
                                   present_value, future_value, when);
  printf ("The payment on the principal for the first year of \n");
  printf ("the $100,000 loan is $%.2f.\n", principal);
}
```

Output

```
The payment on the principal for the first year of
the $100,000 loan is $-882.74.
```

accr_interest_maturity

Evaluates the interest which has accrued on a security that pays interest at maturity.

Synopsis

#include `<imsl.h>`

float `imsl_f_accr_interest_maturity` (*struct tm* `issue`, *struct tm* `maturity`, *float* `coupon_rate`, *float* `par_value`, *int* `basis`)

The type *double* function is `imsl_d_accr_interest_maturity`.

Required Arguments

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `coupon_rate` (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float `par_value` (Input)
Nominal or face value of the security used to calculate interest payments.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The interest which has accrued on a security that pays interest at maturity. If no result can be computed, NaN is returned.

Description

Function `imsl_f_accr_interest_maturity` computes the accrued interest for a security that pays interest at maturity:

$$= (par\_value)(rate)\left(\frac{A}{D}\right)$$

In the above equation, A represents the number of days starting at issue date to maturity date and D represents the annual basis.

Example

In this example, `imsl_f_accr_interest_maturity` computes the accrued interest for a security that pays interest at maturity using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 2000, the maturity date of November 3, 2000, and a coupon rate of 6%.

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

void main()
{
```

```
  struct tm issue, maturity;
  float rate = .06;
  float par = 1000.;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float accrintm;

  issue.tm_year = 100;
  issue.tm_mon = 9;
  issue.tm_mday = 1;

  maturity.tm_year = 100;
  maturity.tm_mon = 10;
  maturity.tm_mday = 3;

  accrintm = imsl_f_accr_interest_maturity (issue, maturity,
                                       rate, par, basis);

  printf ("The accrued interest is $%.2f.\n", accrintm);
}
```

Output

```
The accrued interest is $5.33.
```

accr_interest_periodic

Evaluates the interest which has accrued on a security that pays interest periodically.

Synopsis

#include `<imsl.h>`

float `imsl_f_accr_interest_periodic` (*struct tm* `issue`, *struct tm* `first_coupon`, *struct tm* `settlement`, *float* `coupon_rate`, *float* `par_value`, *int* `frequency`, *int* `basis`)

The type *double* function is `imsl_d_accr_interest_periodic`.

Required Arguments

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `first_coupon` (Input)
First date on which an interest payment is due on the security (e.g. the coupon date). For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `coupon_rate` (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float `par_value` (Input)
Nominal or face value of the security used to calculate interest payments.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The accrued interest for a security that pays periodic interest. If no result can be computed, NaN is returned.

Description

Function `imsl_f_accr_interest_periodic` computes the accrued interest for a security that pays periodic interest.

In the equation below, A_i represents the number days which have accrued for the *i*th quasi-coupon period within the odd period. (The quasi-coupon periods are periods obtained by extending the series of equal payment periods to before or after the actual payment periods.) *NC* represents the number of quasi-coupon periods within the odd period, rounded to the next highest integer. (The odd period is a period between payments that differs from the usual equally spaced periods at which payments are made.) NL_i represents the length of the normal *i*th quasi-coupon period within the odd period. NL_I is expressed in days.

Function `imsl_f_accr_interest_periodic` can be found by solving the following:

$$(par\_value)\left(\frac{rate}{frequency}\left[\sum_{i=1}^{NC}\left(\frac{A_i}{NL_i}\right)\right]\right)$$

Example

In this example, `imsl_f_accr_interest_periodic` computes the accrued interest for a security that pays periodic interest using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 1999, the settlement date of November 3, 1999, the first coupon date of March 31, 2000, and a coupon rate of 6%.

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

```
void main()
{
  struct tm issue, first_coupon, settlement;
  float rate = .06;
  float par = 1000.;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float accrint;

  issue.tm_year = 99;
  issue.tm_mon = 9;
  issue.tm_mday = 1;

  first_coupon.tm_year = 100;
  first_coupon.tm_mon = 2;
  first_coupon.tm_mday = 31;

  settlement.tm_year = 99;
  settlement.tm_mon = 10;
  settlement.tm_mday = 3;

  accrint = imsl_f_accr_interest_periodic (issue, first_coupon,
                       settlement, rate, par, frequency, basis);

  printf ("The accrued interest is $%.2f.\n", accrint);
}
```

Output

```
The accrued interest is $5.33.
```

bond_equivalent_yield

Evaluates the bond-equivalent yield of a Treasury bill.

Synopsis

#include `<imsl.h>`

float `imsl_f_bond_equivalent_yield` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `discount_rate`)

The type *double* function is `imsl_d_bond_equivalent_yield`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `discount_rate` (Input)

The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Return Value

The bond-equivalent yield of a Treasury bill. If no result can be computed, NaN is returned.

Description

Function `imsl_f_bond_equivalent_yield` computes the bond-equivalent yield for a Treasury bill.

It is computed using the following:

$if\ DSM <= 182$

$$\frac{365 * discount\_rate}{360 - discount\_rate * DSM}$$

otherwise,

$$\frac{-\frac{DSM}{365} + \sqrt{\left(\frac{DSM}{365}\right)^2 - \left(2 * \frac{DSM}{365} - 1\right) * \frac{discount\_rate * DSM}{discount\_rate * DSM - 360}}}{\frac{DSM}{365} - 0.5}$$

In the above equation, *DSM* represents the number of days starting at settlement date to maturity date.

Example

In this example, `imsl_f_bond_equivalent_yield` computes the bond-equivalent yield for a Treasury bill with the settlement date of July 1, 1999, the maturity date of July 1, 2000, and discount rate of 5% at the issue date.

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

void main()
{
  struct tm settlement, maturity;
  float discount = .05;
  float yield;

  settlement.tm_year = 99;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 100;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;
```

```
  yield = imsl_f_bond_equivalent_yield (settlement, maturity, discount);
  printf ("The bond-equivalent yield for the T-bill is %.2f%%.\n",
             yield * 100.);
}
```

Output

```
The bond-equivalent yield for the T-bill is 5.29%.
```

convexity

Evaluates the convexity for a security.

Synopsis

#include `<imsl.h>`

float `imsl_f_convexity` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `coupon_rate`, *float* `yield`, *int* `frequency`, *int* `basis`)

The type *double* function is `imsl_d_convexity`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `coupon_rate` (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float `yield` (Input)
Annual yield of the security.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`.. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The convexity for a security. If no result can be computed, NaN is returned.

Description

Function `imsl_f_convexity` computes the convexity for a security. Convexity is the sensitivity of the duration of a security to changes in yield.

It is computed using the following:

$$\frac{\frac{1}{\left(q * frequency\right)^2}\left\{\sum_{t=1}^{n} t\left(t+1\right)\left(\frac{rate}{frequency}\right)q^{-t} + n\left(n+1\right)q^{-n}\right\}}{\left(\sum_{t=1}^{n}\left(\frac{rate}{frequency}\right)q^{-t} + q^{-n}\right)}$$

where n is calculated from `imsl_coupon_number`, and $q = 1 + \frac{yield}{frequency}$.

Example

In this example, `imsl_f_convexity` computes the convexity for a security with the settlement date of July 1, 1990, and maturity date of July 1, 2000, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float coupon = .075;
  float yield = .09;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float convexity;

  settlement.tm_year = 90;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 100;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  convexity = imsl_f_convexity (settlement, maturity,
                                coupon, yield, frequency, basis);

  printf ("The convexity of the bond with ");
  printf ("semiannual interest payments is %.4f.\n", convexity);
}
```

Output

```
The convexity of the bond with semiannual interest payments is 59.4050.
```

coupon_days

Evaluates the number of days in the coupon period containing the settlement date.

Synopsis

#include `<imsl.h>`

float `imsl_f_coupon_days` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

The type *double* function is `imsl_d_coupon_days`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The number of days in the coupon period which contains the settlement date. If no result can be computed, NaN is returned.

Description

Function `imsl_f_coupon_days` computes the number of days in the coupon period that contains the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, `imsl_f_coupon_days` computes the number of days in the coupon period of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float coupdays;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  coupdays = imsl_f_coupon_days (settlement, maturity, frequency, basis);
  printf ("The number of days in the coupon period that\n");
  printf ("contains the settlement date is %.2f.\n", coupdays);
}
```

Output

```
The number of days in the coupon period that
contains the settlement date is 182.50.
```

coupon_number

Evaluates the number of coupons payable between the settlement date and the maturity date.

Synopsis

#include `<imsl.h>`

int `imsl_coupon_number` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are

paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `frequency` (Input)
: Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
: The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on see the Usage Notes section of this chapter.

Return Value

The number of coupons payable between the settlement date and the maturity date.

Description

Function `imsl_coupon_number` computes the number of coupons payable between the settlement date and the maturity date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, `imsl_coupon_number` computes the number of coupons payable with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  int coupnum;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  coupnum = imsl_coupon_number (settlement, maturity, frequency, basis);
  printf ("The number of coupons payable between the\n");
  printf ("settlement date and the maturity date is %d.\n", coupnum);
}
```

Output

```
The number of coupons payable between the
settlement date and the maturity date is 25.
```

days_before_settlement

Evaluates the number of days starting with the beginning of the coupon period and ending with the settlement date.

Synopsis

#include `<imsl.h>`

int `imsl_days_before_settlement` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on see the Usage Notes section of this chapter.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Description

Function `imsl_days_before_settlement` computes the number of days from the beginning of the coupon period to the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, `imsl_days_before_settlement` computes the number of days from the beginning of the coupon period to November 11, 1996, of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  int days;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  days = imsl_days_before_settlement (settlement, maturity,
                                      frequency, basis);

  printf ("The number of days from the beginning of the\n");
  printf ("coupon period to the settlement date is %d.\n", days);
}
```

Output

```
The number of days from the beginning of the
coupon period to the settlement date is 71.
```

days_to_next_coupon

Evaluates the number of days starting with the settlement date and ending with the next coupon date.

Synopsis

#include `<imsl.h>`

int `imsl_days_to_next_coupon` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)

The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `frequency` (Input)

Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)

The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E36`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The number of days starting with the settlement date and ending with the next coupon date.

Description

Function `imsl_days_to_next_coupon` computes the number of days from the settlement date to the next coupon date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pp. 17-35.

Example

In this example, `imsl_days_to_next_coupon` computes the number of days from November 11, 1996, to the next coupon date of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  int days;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  days = imsl_days_to_next_coupon (settlement, maturity, frequency, basis);
```

```
  printf ("The number of days from the settlement date to ");
  printf ("the next coupon date is %d.\n", days);
}
```

Output

```
The number of days from the settlement date to the next coupon date is 110.
```

depreciation_amordegrc

Evaluates the depreciation for each accounting period. During the evaluation of the function a depreciation coefficient based on the asset life is applied.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_amordegrc` (*float* `cost`, *struct tm* `issue`, *struct tm* `first_period`, *float* `salvage`, *int* `period`, *float* `rate`, *int* `basis`)

The type *double* function is `imsl_d_depreciation_amordegrc`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `first_period` (Input)
Date of the end of the first period. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `period` (Input)
Depreciation for the accounting period to be computed.

float `rate` (Input)
Depreciation rate.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E36`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The depreciation for each accounting period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_amordegrc` computes the depreciation for each accounting period. This function is similar to `depreciation_amorlinc`. However, in this function a depreciation coefficient based on the asset life is applied during the evaluation of the function.

Example

In this example, `imsl_f_depreciation_amordegrc` computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

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

void main()
{
  struct tm issue, first_period;
  float cost = 2400.;
  float salvage = 300.;
  int period = 2;
  float rate = .15;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float amordegrc;

  issue.tm_year = 99;
  issue.tm_mon = 10;
  issue.tm_mday = 1;

  first_period.tm_year = 100;
  first_period.tm_mon = 10;
  first_period.tm_mday = 30;

  amordegrc = imsl_f_depreciation_amordegrc (cost, issue, first_period,
                                         salvage, period, rate, basis);

  printf ("The depreciation for the second accounting period ");
  printf ("is $%.2f.\n", amordegrc);
}
```

Output

```
The depreciation for the second accounting period is $335.00.
```

depreciation_amorlinc

Evaluates the depreciation for each accounting period. This function is similar to `depreciation_amordegrc`, except that `depreciation_amordegrc` has a depreciation coefficient that is applied during the evaluation that is based on the asset life.

Synopsis

#include `<imsl.h>`

float `imsl_f_depreciation_amorlinc` (*float* `cost`, *struct tm* `issue`, *struct tm* `first_period`, *float* `salvage`, *int* `period`, *float* `rate`, *int* `basis`)

The type *double* function is `imsl_d_depreciation_amordegrc`.

Required Arguments

float `cost` (Input)
Initial value of the asset.

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `first_period` (Input)
Date of the end of the first period. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `salvage` (Input)
The value of an asset at the end of its depreciation period.

int `period` (Input)
Depreciation for the accounting period to be computed.

float `rate` (Input)
Depreciation rate.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E36`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The depreciation for each accounting period. If no result can be computed, NaN is returned.

Description

Function `imsl_f_depreciation_amorlinc` computes the depreciation for each accounting period.

Example

In this example, `imsl_f_depreciation_amorlinc` computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

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

void main()
{
  struct tm issue, first_period;
  float cost = 2400.;
  float salvage = 300.;
  int period = 2;
  float rate = .15;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float amorlinc;

  issue.tm_year = 99;
  issue.tm_mon = 10;
  issue.tm_mday = 1;

  first_period.tm_year = 100;
  first_period.tm_mon = 10;
  first_period.tm_mday = 30;

  amorlinc = imsl_f_depreciation_amorlinc (cost, issue, first_period,
                                     salvage, period, rate, basis);
  printf ("The depreciation for the second accounting period ");
  printf ("is $%.2f.\n", amorlinc);
}
```

Output

```
The depreciation for the second accounting period is $360.00.
```

discount_price

Evaluates the price of a security sold for less than its face value.

Synopsis

#include `<imsl.h>`

float `imsl_f_discount_price` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `discount_rate`, *float* `redemption`, *int* `basis`)

The type *double* function is `imsl_d_discount_price`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on see the Usage Notes section of this chapter.

float `discount_rate` (Input)
The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

float `redemption` (Input)
Redemption value per $100 face value of the security.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The price per face value for a discounted security. If no result can be computed, NaN is returned.

Description

Function `imsl_f_discount_price` computes the price per $100 face value of a discounted security.

It is computed using the following:

$$redemption - (discount\_rate)\left[redemption\left(\frac{DSM}{B}\right)\right]$$

In the equation above, *DSM* represents the number of days starting at the settlement date and ending with the maturity date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, `imsl_f_discount_price` computes the price of the discounted bond with the settlement date of July 1, 2000, and maturity date of July 1, 2001, at the discount rate of 5% using the US (NASD) 30/360 day count method.

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

```
void main()
{
  struct tm settlement, maturity;
  float discount = .05;
  float redemption = 100.;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float price;

  settlement.tm_year = 100;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 101;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  price = imsl_f_discount_price (settlement, maturity, discount,
                                 redemption, basis);

  printf ("The price of the discounted bond is $%.2f.\n", price);
}
```

Output

```
The price of the discounted bond is $95.00.
```

discount_rate

Evaluates the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Synopsis

#include `<imsl.h>`

float `imsl_f_discount_rate` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `price`, *float* `redemption`, *int* `basis`)

The type *double* function is `imsl_d_discount_rate`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `price` (Input)
Price per $100 face value of the security.

float redemption (Input)
Redemption value per $100 face value of the security.

int basis (Input)
The method for computing the number of days between two dates. It should be one of IMSL_DAY_CNT_BASIS_ACTUALACTUAL, IMSL_DAY_CNT_BASIS_NASD, IMSL_DAY_CNT_BASIS_ACTUAL360, IMSL_DAY_CNT_BASIS_ACTUAL365, or IMSL_DAY_CNT_BASIS_30E360, For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The discount rate for a security. If no result can be computed, NaN is returned.

Description

Function imsl_f_discount_rate computes the discount rate for a security. The discount rate is the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, B represents the number of days in a year based on the annual basis and DSM represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, imsl_f_discount_rate computes the discount rate of a security which is selling at $97.975 with the settlement date of February 15, 2000, and maturity date of June 10, 2000, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float price = 97.975;
  float redemption = 100.;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float rate;

  settlement.tm_year = 100;
  settlement.tm_mon = 1;
  settlement.tm_mday = 15;

  maturity.tm_year = 100;
  maturity.tm_mon = 5;
  maturity.tm_mday = 10;
```

```
  rate = imsl_f_discount_rate (settlement, maturity, price,
                               redemption, basis);

  printf ("The discount rate for the security is %.2f%%.\n", rate * 100.);
}
```

Output

```
The discount rate for the security is 6.37%.
```

discount_yield

Evaluates the annual yield of a discounted security.

Synopsis

#include `<imsl.h>`

float `imsl_f_discount_yield` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `price`, *float* `redemption`, *int* `basis`)

The type *double* function is `imsl_d_discount_yield`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on see the Usage Notes section of this chapter.

float `price` (Input)
Price per $100 face value of the security.

float `redemption` (Input)
Redemption value per $100 face value of the security.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed see the Usage Notes section of this chapter.

Return Value

The annual yield for a discounted security. If no result can be computed, NaN is returned.

Description

Function `imsl_f_discount_yield` computes the annual yield for a discounted security.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DSM* represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, `imsl_f_discount_yield` computes the annual yield for a discounted security which is selling at $95.40663 with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the US (NASD) 30/360 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float price = 95.40663;
  float redemption = 105.;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float yielddisc;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  yielddisc = imsl_f_discount_yield (settlement, maturity,
                                    price, redemption, basis);
  printf ("The yield on the discounted bond is ");
  printf ("%.2f%%.\n", yielddisc * 100.);
}
```

Output

```
The yield on the discounted bond is 1.01%.
```

duration

Evaluates the annual duration of a security where the security has periodic interest payments.

Synopsis

#include `<imsl.h>`

float `imsl_f_duration` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `coupon_rate`, *float* `yield`, *int* `frequency`, *int* `basis`)

The type *double* function is `imsl_d_duration`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `coupon_rate` (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float `yield` (Input)
Annual yield of the security.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The annual duration of a security with periodic interest payments. If no result can be computed, NaN is returned.

Description

Function `imsl_f_duration` computes the Maccaluey's duration of a security with periodic interest payments. The Maccaluey's duration is the weighted-average time to the payments, where the weights are the present value of the payments.

It is computed using the following:

$$\left(\frac{ \dfrac{\frac{DSC}{E} * 100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}} + \sum_{k=1}^{N} \left(\left(\dfrac{100 * coupon\_rate}{freq * \left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}} \right) * \left(k-1+\frac{DSC}{E}\right) \right) }{ \dfrac{100}{\left(1+\frac{yield}{freq}\right)^{N-1+\frac{DSC}{E}}} + \sum_{k=1}^{N} \left(\dfrac{100 * coupon\_rate}{freq * \left(1+\frac{yield}{freq}\right)^{k-1+\frac{DSC}{E}}} \right) } \right) * \frac{1}{freq}$$

In the equation above, *DSC* represents the number of days starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable from the settlement date to the maturity date. *freq* represents the frequency of the coupon payments annually.

Example

In this example, `imsl_f_duration` computes the annual duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float coupon = .075;
  float yield = .09;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float duration;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  duration = imsl_f_duration (settlement, maturity, coupon,
                              yield, frequency, basis);
```

```
  printf ("The annual duration of the bond with ");
  printf ("semiannual interest payments is %.4f.\n", duration);
}
```

Output

```
The annual duration of the bond with semiannual interest payments is 7.0420.
```

interest_rate_security

Evaluates the interest rate of a fully invested security.

Synopsis

#include `<imsl.h>`

float `imsl_f_interest_rate_security` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `investment`, *float* `redemption`, *int* `basis`)

The type *double* function is `imsl_d_interest_rate_security`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `investment` (Input)
The total amount one has invested in the security..

float `redemption` (Input)
Amount to be received at maturity.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion see the Usage Notes section of this chapter.

Return Value

The interest rate for a fully invested security. If no result can be computed, NaN is returned.

Description

Function `imsl_f_interest_rate_security` computes the interest rate for a fully invested security.

It is computed using the following:

$$\left(\frac{redemption - investment}{investment}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, B represents the number of days in a year based on the annual basis, and DSM represents the number of days in the period starting with the settlement date and ending with the maturity date.

Example

In this example, `imsl_f_interest_rate_security` computes the interest rate of a $7,000 investment with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method. The total amount received at the end of the investment is $10,000.

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

void main()
{
  struct tm settlement, maturity;
  float investment = 7000.;
  float redemption = 10000.;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float intrate;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  intrate = imsl_f_interest_rate_security (settlement, maturity,
                                  investment, redemption, basis);

  printf ("The interest rate of the bond is %.2f%%.\n", intrate * 100.);
}
```

Output

```
The interest rate of the bond is 4.28%.
```

modified_duration

Evaluates the modified Macauley duration of a security.

Synopsis

#include <imsl.h>

float imsl_f_macauley_duration (*struct tm* settlement, *struct tm* maturity, *float* coupon_rate, *float* yield, *int* frequency, *int* basis)

The type *double* function is imsl_d_macauley_duration.

Required Arguments

struct tm settlement (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm maturity (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float coupon_rate (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float yield (Input)
Annual yield of the security.

int frequency (Input)
Frequency of the interest payments. It should be one of IMSL_ANNUAL, IMSL_SEMIANNUAL or IMSL_QUARTERLY. For a more detailed discussion on frequency see the Usage Notes section of this chapter.

int basis (Input)
The method for computing the number of days between two dates. It should be one of IMSL_DAY_CNT_BASIS_ACTUALACTUAL, IMSL_DAY_CNT_BASIS_NASD, IMSL_DAY_CNT_BASIS_ACTUAL360, IMSL_DAY_CNT_BASIS_ACTUAL365, or IMSL_DAY_CNT_BASIS_30E360. For a more detailed discussion on basis see the Usage Notes section of this chapter.

Return Value

The modified Macauley duration of a security is returned. The security has an assumed par value of $100. If no result can be computed, NaN is returned.

Description

Function imsl_f_macauley_duration computes the modified Macauley duration for a security with an assumed par value of $100.

It is computed using the following:

$$\frac{duration}{1+\left(\frac{yield}{frequency}\right)}$$

where *duration* is calculated from `imsl_f_duration`.

Example

In this example, `imsl_f_macauley_duration` computes the modified Macauley duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float coupon = .075;
  float yield = .09;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float mduration;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  mduration = imsl_f_macauley_duration (settlement, maturity,
                                 coupon, yield, frequency, basis);

  printf ("The modified Macauley duration of the bond with\n");
  printf ("semiannual interest payments is %.4f.\n", mduration);
}
```

Output

```
The modified Macauley duration of the bond with
semiannual interest payments is 6.7387.
```

next_coupon_date

Evaluates the first coupon date which follows the settlement date.

Synopsis

#include `<imsl.h>`

struct tm `imsl_next_coupon_date` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The first coupon date which follows the settlement date.

Description

Function `imsl_next_coupon_date` computes the next coupon date after the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, `imsl_next_coupon_date` computes the next coupon date of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity, date;
  char* month[] = { "January", "February", "March", "April", "May",
                    "June", "July", "August", "September",
                    "October", "November", "December" };
```

```
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  date = imsl_next_coupon_date (settlement, maturity, frequency, basis);
  printf ("The next coupon date after the settlement date ");
  printf ("is %s %d, %d.\n", month[date.tm_mon], date.tm_mday,
                             date.tm_year+1900);
}
```

Output

```
The next coupon date after the settlement date is March 1, 1997.
```

previous_coupon_date

Evaluates the coupon date which immediately precedes the settlement date.

Synopsis

#include `<imsl.h>`

struct tm `imsl_previous_coupon_date` (*struct tm* `settlement`, *struct tm* `maturity`, *int* `frequency`, *int* `basis`)

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `frequency` (Input)
Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`.

For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The coupon date which immediately precedes the settlement date.

Description

Function `imsl_previous_coupon_date` computes the coupon date which immediately precedes the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, `imsl_previous_coupon_date` computes the previous coupon date of a bond with the settlement date of November 11, 1986, and the maturity date of March 1, 1999, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity, date;
  char* month[] = { "January", "February", "March", "April", "May",
                   "June", "July", "August", "September",
                   "October", "November", "December" };
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;

  settlement.tm_year = 96;
  settlement.tm_mon = 10;
  settlement.tm_mday = 11;

  maturity.tm_year = 109;
  maturity.tm_mon = 2;
  maturity.tm_mday = 1;

  date = imsl_previous_coupon_date (settlement, maturity, frequency, basis);
  printf ("The previous coupon date before the settlement ");
  printf ("date is %s %d, %d.\n", month[date.tm_mon], date.tm_mday,
                                  date.tm_year+1900);
}
```

Output

```
The previous coupon date before the settlement date is September 1, 1996.
```

price

Evaluates the price, per $100 face value, of a security that pays periodic interest.

Synopsis

#include <imsl.h>

float imsl_f_price (*struct tm* settlement, *struct tm* maturity, *float* rate, *float* yield, *float* redemption, *int* frequency, *int* basis)

The type *double* function is imsl_d_price.

Required Arguments

struct tm settlement (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm maturity (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float rate (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float yield (Input)
Annual yield of the security.

float redemption (Input)
Redemption value per $100 face value of the security.

int frequency (Input)
Frequency of the interest payments. It should be one of IMSL_ANNUAL, IMSL_SEMIANNUAL or IMSL_QUARTERLY. For a more detailed discussion on frequency see the Usage Notes section of this chapter.

int basis (Input)
The method for computing the number of days between two dates. It should be one of IMSL_DAY_CNT_BASIS_ACTUALACTUAL, IMSL_DAY_CNT_BASIS_NASD, IMSL_DAY_CNT_BASIS_ACTUAL360, IMSL_DAY_CNT_BASIS_ACTUAL365, or IMSL_DAY_CNT_BASIS_30E360. For a more detailed discussion on basis see the Usage Notes section of this chapter.

Return Value

The price per $100 face value of a security that pays periodic interest. If no result can be computed, NaN is returned.

Description

Function `imsl_f_price` computes the price per $100 face value of a security that pays periodic interest.

It is computed using the following:

$$\left(\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right)+\left[\sum_{k=1}^{N}\frac{100*\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right]-\left(100*\frac{rate}{frequency}*\frac{A}{E}\right)$$

In the above equation, *DSC* represents the number of days in the period starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the timeframe from the settlement date to the redemption date. *A* represents the number of days in the timeframe starting with the beginning of coupon period and ending with the settlement date.

Example

In this example, `imsl_f_price` computes the price of a bond that pays coupon every six months with the settlement of July 1, 1995, the maturity date of July 1, 2005, a annual rate of 6%, annual yield of 7% and redemption value of $105 using the US (NASD) 30/360 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float rate = .06;
  float yield = .07;
  float redemption = 105.;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float price;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  price = imsl_f_price (settlement, maturity, rate, yield,
                        redemption, frequency, basis);
  printf ("The price of the bond is $%.2f.\n", price);
}
```

Output

```
The price of the bond is $95.41.
```

price_maturity

Evaluates the price, per $100 face value, of a security that pays interest at maturity.

Synopsis

#include `<imsl.h>`

float `imsl_f_price_maturity` (*struct tm* `settlement`, *struct tm* `maturity`, *struct tm* `issue`, *float* `rate`, *float* `yield`, *int* `basis`)

The type *double* function is `imsl_d_price_maturity`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on see the Usage Notes section of this chapter.

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `rate` (Input)
Annual interest rate set forth on the face of the security; the coupon rate.

float `yield` (Input)
Annual yield of the security.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The price per $100 face value of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Description

Function `imsl_f_price_maturity` computes the price per $100 face value of a security that pays interest at maturity.

It is computed using the following:

$$\left[\frac{100+\left(\frac{DIM}{B}*rate*100\right)}{1+\left(\frac{DSM}{B}*yield\right)}\right]-\left(\frac{A}{B}*rate*100\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date.

Example

In this example, `imsl_f_price_maturity` computes the price at maturity of a security with the settlement date of August 1, 2000, maturity date of July 1, 2001 and issue date of July 1, 2000, using the US (NASD) 30/360 day count method. The security has 5% annual yield and 5% interest rate at the date of issue.

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

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

void main()
{
  struct tm settlement, maturity, issue;
  float rate = .05;
  float yield = .05;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float pricemat;

  settlement.tm_year = 100;
  settlement.tm_mon = 7;
  settlement.tm_mday = 1;

  maturity.tm_year = 101;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  issue.tm_year = 100;
  issue.tm_mon = 6;
  issue.tm_mday = 1;

  pricemat = imsl_d_price_maturity (settlement, maturity, issue,
                                       rate, yield, basis);
```

```
  printf ("The price of the bond is $%.2f.\n", pricemat);
}
```

Output

```
The price of the bond is $99.98.
```

received_maturity

Evaluates the amount one receives when a fully invested security reaches the maturity date.

Synopsis

#include `<imsl.h>`

float `imsl_f_received_maturity` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `investment`, *float* `discount_rate`, *int* `basis`)

The type *double* function is `imsl_d_received_maturity`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.*struct tm* `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `investment` (Input)
The total amount one has invested in the security.

float `discount_rate` (Input)
The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The amount one receives when a fully invested security reaches its maturity date. If no result can be computed, NaN is returned.

Description

Function `imsl_f_received_maturity` computes the amount received at maturity for a fully invested security.

It is computed using the following:

$$\frac{investment}{1-\left(discount\_rate*\frac{DIM}{B}\right)}$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date.

Example

In this example, `imsl_f_received_maturity` computes the amount received of a $7,000 investment with the settlement date of July 1, 1995, maturity date of July 1, 2005 and discount rate of 6%, using the Actual/365 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float investment = 7000.;
  float discount = .06;
  int basis = IMSL_DAY_CNT_BASIS_ACTUAL365;
  float received;

  settlement.tm_year = 95;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 105;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  received = imsl_f_received_maturity (settlement, maturity,
                                       investment, discount, basis);
  printf ("The amount received at maturity for the ");
  printf ("bond is $%.2f.\n", received);
}
```

Output

```
The amount received at maturity for the bond is $17521.60.
```

treasury_bill_price

Evaluates the price per $100 face value of a Treasury bill.

Synopsis

#include `<imsl.h>`

float `imsl_f_treasury_bill_price` (*struct tm* `settlement`,
struct tm `maturity`, *float* `discount_rate`)

The type *double* function is `imsl_d_treasury_bill_price`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `discount_rate` (Input)
The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Return Value

The price per $100 face value of a Treasury bill. If no result can be computed, NaN is returned.

Description

Function `imsl_f_treasury_bill_price` computes the price per $100 face value for a Treasury bill.

It is computed using the following:

$$100\left(1-\frac{discount\_rate * DSM}{360}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, `imsl_f_treasury_bill_price` computes the price for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and a discount rate of 5% at the issue date.

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

void main()
{
  struct tm settlement, maturity;
  float discount = .05;
  float price;

  settlement.tm_year = 100;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 101;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  price = imsl_f_treasury_bill_price (settlement, maturity, discount);
  printf ("The price per $100 face value for the T-bill ");
  printf ("is $%.2f.\n", price);
}
```

Output

```
The price per $100 face value for the T-bill is $94.93.
```

treasury_bill_yield

Evaluates the yield of a Treasury bill.

Synopsis

#include `<imsl.h>`

float `imsl_f_treasury_bill_yield` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `price`)

The type *double* function is `imsl_d_treasury_bill_yield`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `price` (Input)
Price per $100 face value of the Treasury bill.

Return Value

The yield for a Treasury bill. If no result can be computed, NaN is returned.

Description

Function `imsl_f_treasury_bill_yield` computes the yield for a Treasury bill.

It is computed using the following:

$$\left(\frac{100 - price}{price}\right)\left(\frac{360}{DSM}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, `imsl_f_treasury_bill_yield` computes the yield for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and priced at $94.93.

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

void main()
{
  struct tm settlement, maturity;
  float price = 94.93;
  float yield;

  settlement.tm_year = 100;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 101;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  yield = imsl_f_treasury_bill_yield (settlement, maturity, price);
  printf ("The yield for the T-bill is %.2f%%.\n", yield * 100.);
}
```

Output

```
The yield for the T-bill is 5.27%.
```

year_fraction

Evaluates the fraction of a year represented by the number of whole days between two dates.

Synopsis

#include `<imsl.h>`

float `imsl_f_year_fraction` (*struct tm* `start`, *struct tm* `end`, *int* `basis`)

The type *double* function is `imsl_d_year_fraction`.

Required Arguments

struct tm `start` (Input)
Initial date. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `end` (Input)
Ending date. For a more detailed discussion on dates see the Usage Notes section of this chapter.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The fraction of a year represented by the number of whole days between two dates. If no result can be computed, NaN is returned.

Description

Function `imsl_f_year_fraction` computes the fraction of the year.

It is computed using the following:

$$A/D$$

where A = the number of days from `start` to `end`, D = annual basis.

Example

In this example, `imsl_f_year_fraction` computes the year fraction between August 1, 2000, and July 1, 2001, using the NASD day count method.

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

```
#include "imsl.h"

void main()
{
  struct tm start, end;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float yearfrac;

  start.tm_year = 100;
  start.tm_mon = 7;
  start.tm_mday = 1;

  end.tm_year = 101;
  end.tm_mon = 6;
  end.tm_mday = 1;

  yearfrac = imsl_f_year_fraction (start, end, basis);
  printf ("The year fraction of the 30/360 period is %f.\n", yearfrac);
}
```

Output

```
The year fraction of the 30/360 period is 0.916667.
```

yield_maturity

Evaluates the annual yield of a security that pays interest at maturity.

Synopsis

#include `<imsl.h>`

float `imsl_f_yield_maturity` (*struct tm* `settlement`, *struct tm* `maturity`, *struct tm* `issue`, *float* `rate`, *float* `price`, *int* `basis`)

The type *double* function is `imsl_d_yield_maturity`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `issue` (Input)
The date on which interest starts accruing. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `rate` (Input)
Interest rate at date of issue of the security.

float `price` (Input)
Price per $100 face value of the security.

int `basis` (Input)
The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The annual yield of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Description

Function `imsl_f_yield_maturity` computes the annual yield of a security that pays interest at maturity.

It is computed using the following:

$$\left\{\frac{\left[1+\left(\frac{DIM}{B}*rate\right)\right]-\left[\frac{price}{100}+\left(\frac{A}{B}*rate\right)\right]}{\frac{price}{100}+\left(\frac{A}{B}*rate\right)}\right\}*\left(\frac{B}{DSM}\right)$$

In the equation above, *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, `imsl_f_yield_maturity` computes the annual yield of a security that pays interest at maturity which is selling at $95.40663 with the settlement date of August 1, 2000, the issue date of July 1, 2000, the maturity date of July 1, 2010, and the interest rate of 6% at the issue using the US (NASD) 30/360 day count method.

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

void main()
{
  struct tm settlement, maturity, issue;
  float rate = .06;
  float price = 95.40663;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float yieldmat;
```

```
  settlement.tm_year = 100;
  settlement.tm_mon = 7;
  settlement.tm_mday = 1;

  maturity.tm_year = 110;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  issue.tm_year = 100;
  issue.tm_mon = 6;
  issue.tm_mday = 1;

  yieldmat = imsl_f_yield_maturity (settlement, maturity, issue,
                                     rate, price, basis);
  printf ("The yield on a bond which pays at maturity is ");
  printf ("%.2f%%.\n", yieldmat * 100.);
}
```

Output

```
The yield on a bond which pays at maturity is 6.74%.
```

yield_periodic

Evaluates the yield of a security that pays periodic interest.

Synopsis

#include `<imsl.h>`

float `imsl_f_yield_periodic` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `coupon_rate`, *float* `price`, *float* `redemption`, *int* `frequency`, *int* `basis`, …, `0`)

The type *double* function is `imsl_d_yield_periodic`.

Required Arguments

struct tm `settlement` (Input)
The date on which payment is made to settle a trade. For a more detailed discussion on dates see the Usage Notes section of this chapter.

struct tm `maturity` (Input)
The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see the Usage Notes section of this chapter.

float `coupon_rate` (Input)
Annual coupon rate.

float `price` (Input)
Price per $100 face value of the security.

float `redemption` (Input)
Redemption value per $100 face value of the security.

int `frequency` (Input)

Frequency of the interest payments. It should be one of `IMSL_ANNUAL`, `IMSL_SEMIANNUAL` or `IMSL_QUARTERLY`. For a more detailed discussion on `frequency` see the Usage Notes section of this chapter.

int `basis` (Input)

The method for computing the number of days between two dates. It should be one of `IMSL_DAY_CNT_BASIS_ACTUALACTUAL`, `IMSL_DAY_CNT_BASIS_NASD`, `IMSL_DAY_CNT_BASIS_ACTUAL360`, `IMSL_DAY_CNT_BASIS_ACTUAL365`, or `IMSL_DAY_CNT_BASIS_30E360`. For a more detailed discussion on `basis` see the Usage Notes section of this chapter.

Return Value

The yield of a security that pays interest periodically. If no result can be computed, NaN is returned.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `imsl_f_yield_periodic` (*struct tm* `settlement`, *struct tm* `maturity`, *float* `coupon_rate`, *float* `price`, *float* `redemption`, *int* `frequency`, *int* `basis`, `IMSL_XGUESS`, *float* `guess`, `IMSL_HIGHEST`, *float* `max`, `0`)

Optional Arguments

`IMSL_XGUESS`, *float* `guess` (Input)

Initial guess at the internal rate of return.

`IMSL_HIGHEST`, *float* `max` (Input)

Maximum value of the yield.

Default: 1.0 (100%)

Description

Function `imsl_f_yield_periodic` computes the yield of a security that pays periodic interest. If there is one coupon period use the following:

$$\left\{\frac{\left(\frac{redemption}{100}+\frac{coupon\_rate}{frequency}\right)-\left[\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)\right]}{\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)}\right\}\left(\frac{frequency*E}{DSR}\right)$$

In the equation above, *DSR* represents the number of days in the period starting with the settlement date and ending with the redemption date. *E* represents the number of days within the coupon period. *A* represents the number of days in the period starting with the beginning of coupon period and ending with the settlement date.

If there is more than one coupon period use the following:

$$price-\left(\left(\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right)+\left[\sum_{k=1}^{N}\frac{100*\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right]-\left(100*\frac{rate}{frequency}*\frac{A}{E}\right)\right)=0$$

In the equation above, *DSC* represents the number of days in the period from the settlement to the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the period starting with the settlement date and ending with the redemption date. *A* represents the number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Example

In this example, `imsl_f_yield_periodic` computes yield of a security which is selling at $95.40663 with the settlement date of July 1, 1985, the maturity date of July 1, 1995, and the coupon rate of 6% at the issue using the US (NASD) 30/360 day count method.

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

void main()
{
  struct tm settlement, maturity;
  float coupon_rate = .06;
  float price = 95.40663;
  float redemption = 105.;
  int frequency = IMSL_SEMIANNUAL;
  int basis = IMSL_DAY_CNT_BASIS_NASD;
  float yield;

  settlement.tm_year = 100;
  settlement.tm_mon = 6;
  settlement.tm_mday = 1;

  maturity.tm_year = 110;
  maturity.tm_mon = 6;
  maturity.tm_mday = 1;

  yield = imsl_f_yield_periodic (settlement, maturity, coupon_rate,
                                 price, redemption, frequency, basis, 0);
  printf ("The yield of the bond is %.2f%%.\n", yield * 100.);
}
```

Output

The yield of the bond is 7.00%.

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` (page 675). 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` (page 676). 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` (page 677) . 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` (page 676). 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` (page 675) and can be retrieved by `imsl_random_seed_get` (page 674). Prior to invoking any generator in this section, the user can call `imsl_random_seed_set` (page 675) 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` (page 675), 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` (page 674) 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` (page 675) and `imsl_random_seed_get` (page 674) 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

$$\text{median } \{x_i\} = \begin{cases} \text{middle } x_i \text{ after sorting if } n \text{ is odd} \\ \text{average of middle two } x_i\text{'s if } n \text{ is even} \end{cases}$$

Median Absolute Deviation

$$\text{MAD}=\text{median}\left\{\left|x_i - \text{median}\left\{x_j\right\}\right|\right\}$$

Simple Robust Estimate of Scale

$$\text{MAD}/\Phi^{-1}\left(3/4\right)$$

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 uppe r_ 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[],
        IMSL_FCN_W_DATA, float user_proc_cdf(), void *data,
        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.

IMSL_FCN_W_DATA, *float* user_proc_cdf (*float* y, *void* *data), *void* *data, (Input)
User supplied function that returns the hypothesized, cumulative distribution function at the point *y*, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

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

$$\lfloor mF(x_i)+1 \rfloor \text{ where } m = \text{n\_categories} \text{ and } \lfloor \cdot \rfloor$$

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 θ = 5.0 is generated via function `imsl_f_random_poisson` (page 680). In the call to `imsl_f_chi_squared_test`, function `imsl_f_poisson_cdf`(page 680) 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_{jk}, 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;
    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* × *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* × *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; $\beta_0, \beta_1, \ldots, \beta_k$ 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

$$\overline{y}$$

the so-called *corrected total sum of squares*. This statistic is computed as

$$\text{SST} = \sum_{i=1}^{n} \left(y_i - \overline{y} \right)^2$$

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:

$$\sum_{i=1}^{n} \left(x_i - \bar{x}_n\right)\left(x_i - \bar{x}_n\right)^T = \sum_{i=2}^{n} \frac{i}{i-1}\left(x_i - \bar{x}_i\right)\left(x_i - \bar{x}_i\right)^T$$

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

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i \qquad i = 1, 2, \dots, 9$$

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, $i = 1, \ldots$, 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*

Column	Description
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,

$$R^2 = \frac{\sum\left(\hat{y}_i - \bar{y}\right)^2}{\sum\left(y_1 - \bar{y}\right)^2} 100\%$$

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
```

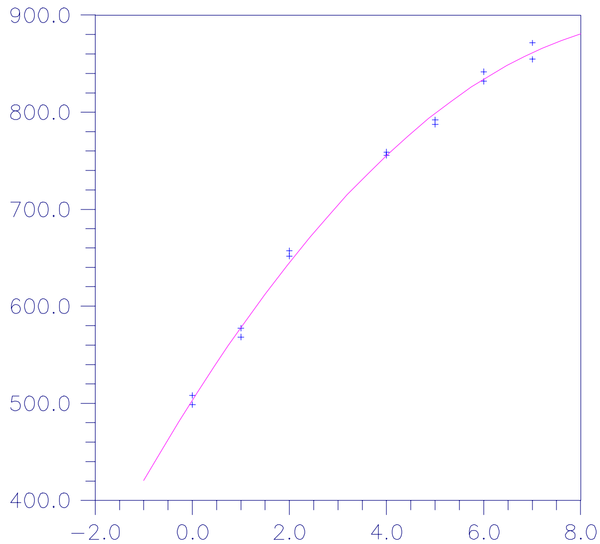

Figure 10-1 A Polynomial Fit

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 675).

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

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

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 674).

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 Γ(·) 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 – ε 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
```

faure_next_point

Computes a shuffled Faure sequence.

Synopsis

#include `<imsl.h>`

*Imsl_faure** `imsl_faure_sequence_init` (*int* `ndim`, …, `0`)

*float** `imsl_f_faure_next_point` (*Imsl_faure* `*state`, …, `0`)

void `imsl_faure_sequence_free` (*Imsl_faure* `*state`)

The type *double* function is `imsl_d_faure_next_point`. The functions `imsl_faure_sequence_init` and `imsl_faure_sequence_free` are precision independent.

Required Arguments for imsl_faure_sequence_init

int `ndim` (Input)
The dimension of the hyper-rectangle.

Return Value for imsl_faure_sequence_init

Returns a structure that contains information about the sequence. The structure should be freed using `imsl_faure_sequence_free` after it is no longer needed.

Required Arguments for imsl_faure_next_point

Imsl_faure *`state` (Input/Output)
Structure created by a call to `imsl_faure_sequence_init`.

Return Value for imsl_faure_next_point

Returns the next point in the shuffled Faure sequence. To release this space, use `free`.

Required Arguments for imsl_faure_sequence_free

Imsl_faure *`state` (Input/Output)
Structure created by a call to `imsl_faure_sequence_init`.

Synopsis with Optional Arguments

```
#include <imsl.h>
float *imsl_faure_sequence_init (int ndim,
        IMSL_BASE, int base,
        IMSL_SKIP, int skip,
        0)
float*  imsl_f_faure_next_point (Imsl_faure *state,
        IMSL_RETURN_USER, float *user,
        IMSL_RETURN_SKIP, int *skip,
        0)
```

Optional Arguments

`IMSL_BASE`, *int* `base` (Input)
The base of the Faure sequence.
Default: The smallest prime greater than or equal to `ndim`.

`IMSL_SKIP`, *int* `*skip` (Input)
The number of points to be skipped at the beginning of the Faure sequence.
Default: $\lfloor \texttt{base}^{m/2-1} \rfloor$, where $m = \lfloor \log \mathrm{B} / \log \texttt{base} \rfloor$ and B is the largest representable integer.

`IMSL_RETURN_USER`, *float* *user (Output)
User-supplied array of length `ndim` containing the current point in the sequence.

`IMSL_RETURN_SKIP`, *int* `*skip` (Output)
The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for `IMSL_SKIP`, and using the same dimension for `ndim`.

Description

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set $x_1, \ldots, x_n \in [0,1]^d$, $d \ge 1$, is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = [0, t_1) \times \cdots \times [0, t_d),\ 0 \le t_j \le 1,\ 1 \le j \le d,$$

λ is the Lebesque measure, and $A(E;n)$ is the number of the x_j contained in E.

The sequence $x_1, x_2, \ldots$ of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d, such that

$$D_n^{(d)} \le c(d)\frac{(\log n)^d}{n}$$

for all $n>1$.

Generalized Faure sequences can be defined for any prime base $b \ge d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \ge d$, so the optional argument `IMSL_BASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer n in its b-ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers, $0 \le a_i(n) < b$.

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series, $c_{k\,d}^{(j)}$, is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and $c_{k\,d}$ is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the b-ary Gray code. The function $G(n)$ maps the positive integer n into the integer given by its b-ary expansion.

The sequence computed by this function is $\boldsymbol{x}(G(n))$, where $\boldsymbol{x}$ is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that `imsl_faure_sequence_init` is used to create a structure that holds the state of the sequence. Each call to `imsl_f_faure_next_point` returns the next point in the sequence and updates the *Imsl_faure* structure. The final call to `imsl_faure_sequence_free` frees data items, stored in the structure, that were allocated by `imsl_faure_sequence_init`.

```
#include "stdio.h"
#include "imsl.h"


void main()
{
       Imsl_faure    *state;
       float         *x;
       int            ndim = 3;
       int            k;

       state = imsl_faure_sequence_init(ndim, 0);

       for (k = 0;  k < 5;  k++) {
              x = imsl_f_faure_next_point(state, 0);
              printf("%10.3f %10.3f  %10.3f\n", x[0], x[1], x[2]);
              free(x);
       }

       imsl_faure_sequence_free(state);
}
```

Output

```
     0.334      0.493       0.064
     0.667      0.826       0.397
     0.778      0.270       0.175
     0.111      0.604       0.509
     0.445      0.937       0.842
```

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.

Keyword	Action
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 697).

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 698).

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.

Option	Value	Meaning
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.

If multiple threads are used then default settings are valid for each thread. When using threads it is possible to set different output files for each thread by calling `imsl_output_file` from within each thread. See Example 2 for details.

Examples

Example 1

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
```

Example 2

The following example illustrates how to direct output from IMSL routines that run in separate threads to different files. First, two threads are created, each calling a different IMSL function, then the results are printed by calling `imsl_f_write_matrix` from within each thread. Note that `imsl_output_file` is called from within each thread to change the default output file.

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

```
void *ex1(void* arg);
void *ex2(void* arg);
void main()
{
  pthread_t       thread1;
  pthread_t       thread2;

  /* Disable IMSL signal trapping. */
  imsl_error_options(IMSL_SET_SIGNAL_TRAPPING, 0, 0);

  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}

void *ex1(void* arg)
{
  float *rand_nums = NULL;
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex1.out", "w");
  /* Set the output file for this thread. */
  imsl_output_file(IMSL_SET_OUTPUT_FILE, file_ptr, 0);
  /* Compute 5 random numbers. */
  imsl_random_seed_set(12345);
  rand_nums = imsl_f_random_uniform(5, 0);
  /* Output random numbers. */
  imsl_f_write_matrix("Random Numbers", 5, 1, rand_nums, 0);

  if (rand_nums) free(rand_nums);
  fclose(file_ptr);
```

```
}
void *ex2(void* arg)
{
  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};
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex2.out", "w");
  /* Set the output file for this thread. */
  imsl_output_file(IMSL_SET_OUTPUT_FILE, file_ptr, 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);

  if (x) free(x);
  fclose(file_ptr);
}
```

Output

ex1.out

```
Random Numbers
 1      0.0966
 2      0.8340
 3      0.9477
 4      0.0359
 5      0.0115
```

ex2.out

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

version

Returns 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 (page 711).

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 (page 711) 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`,
`IMSL_SET_SIGNAL_TRAPPING`, *int* `setting`,
0)

Optional Arguments

`IMSL_SET_PRINT`, *Imsl_error* `type`, *int* `setting` (Input)
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` = "`imslerr.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.

`IMSL_SET_SIGNAL_TRAPPING`, *int* `setting` (Input)
C/Math/Library will use its own signal handler if setting is 1; otherwise the C/Math/Library signal handler is not used. If C/Math/Library is called from a multi-threaded application, signal handling must be turned off. See Example 3 for details.

Default: setting = 1

Return Value

The return value for this function is void.

Description

This function allows the error handling system to be customized.

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options (excluding `IMSL_SET_SIGNAL_TRAPPING`) for each thread by calling `imsl_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsl_error_options(IMSL_SET_SIGNAL_TRAPPING, 0, 0);
```

See Example 3 and Example 4 for multithreaded examples.

Examples

Example 1

In this example, the `IMSL_TERMINAL` print setting is retrieved. Next, stopping on `IMSL_TERMINAL` errors is turned off, then 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 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.
```

Example 3

In this example, two threads are created and error options is called within each thread to set the error handling options differently for each thread. Since we expect to generate terminal errors in each thread, we must turn off stopping on terminal errors for each thread. Also notice that `imsl_error_options` is called from `main` to disable the IMSL signal-trapping mechanism. See Example 4 for a similar example using WIN32 threads. Note since multiple threads are executing, the order of the errors output may differ on some systems.

```
#include <pthread.h>
```

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

void *ex1(void* arg);
void *ex2(void* arg);

void main()
{
  pthread_t       thread1;
  pthread_t       thread2;

  /* Disable IMSL signal trapping. */
  imsl_error_options(IMSL_SET_SIGNAL_TRAPPING, 0, 0);

  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}

void *ex1(void* arg)
{
  float res;
  /*
   * Call imsl_error_options to set teh error handling
   * options for this thread.  Notice that the error printing
   * function wil lbe user defined for this thread only.
   */
  imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0, 0);

  res = imsl_f_beta(-1.0, .5);
}

void *ex2(void* arg)
{
  float res;
  /*
   * Call imsl_error_options to set the error handling
   * options for this thread.
   */
  imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0,
                  IMSL_SET_TRACEBACK, IMSL_TERMINAL, 1, 0);

  res = imsl_f_gamma(-1.0);
}
```

Output

```
*** TERMINAL Error from imsl_f_beta.  Both "x" = -1.000000e+00 and "y" =
***          5.000000e-01 must be greater than zero.


*** TERMINAL Error from imsl_f_gamma.  The argument for the function can not
***          be a negative integer. Argument "x" = -1.000000e+00.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code              Routine
  ----------        ----------              -------
 IMSL_TERMINAL    IMSL_NEGATIVE_INTEGER     imsl_f_gamma
                                            USER
```

Example 4

In this example the WIN32 API is used to demonstrate the same functionality as shown in Example 3 above. Note since multiple threads are executing, the order of the errors output may differ on some systems.

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

DWORD WINAPI ex1(void *arg);
DWORD WINAPI ex2(void *arg);

int main(int argc, char* argv[])
{
       HANDLE thread[2];

       imsl_error_options(IMSL_SET_SIGNAL_TRAPPING, 0, 0);

       thread[0] = CreateThread(NULL, 0, ex1, NULL, 0, NULL);
       thread[1] = CreateThread(NULL, 0, ex2, NULL, 0, NULL);

       WaitForMultipleObjects(2, thread, TRUE, INFINITE);

}
DWORD WINAPI ex1(void *arg)
{
  float res;
  /*
   * Call imsl_error_options to set the error handling
   * options for this thread.
   */
  imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0, 0);
  res = imsl_f_beta(-1.0, .5);
  return(0);
}
DWORD WINAPI ex2(void *arg)
{
  float res;
  /*
```

```
   * Call imsl_error_options to set the error handling
   * options for this thread.  Notice that tracebacks are
   * turned on for IMSL_TERMINAL errors.
   */
  imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0,
                     IMSL_SET_TRACEBACK, IMSL_TERMINAL, 1,
                     0);
  res = imsl_f_gamma(-1.0);
  return(0);
}
```

Output

```
*** TERMINAL Error from imsl_f_gamma.  The argument for the function can not
***          be a negative integer. Argument "x" = -1.000000e+00.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code              Routine
  ----------        ----------              -------
 IMSL_TERMINAL    IMSL_NEGATIVE_INTEGER     imsl_f_gamma
                                            USER

*** TERMINAL Error from imsl_f_beta.  Both "x" = -1.000000e+00 and "y" =
***          5.000000e-01 must be greater than zero.
```

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
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

Name	Description	Value	Reference
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	$5.67032 \times 10^{-8} W/K^4/m^2$	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
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	$A^{M_l} - 1$, 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

$$E_{\min_f} \le E \le E_{\max_f}$$

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 $\mathrm{perm}_i = i$, 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 $\mathtt{perm}_i = i$, 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

$$\sum_{i=0}^{n-1} \left|x_i\right|$$

is returned. If the option IMSL_INF_NORM is selected, the infinity norm

$$\max \left|x_i\right|$$

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

$$A \in \Re^{\mathrm{nrowa} \times \mathrm{ncola}}$$

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nlcb, *int* nucb, *float* *b (Input)
The sparse matrix

$$B \in \Re^{\mathrm{nrowb} \times \mathrm{xnolb}}$$

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^Tx, A^Hx and AA^H are computed.

```
#include <imsl.h>
#include <stdlib.h>
main()
{
        int             n = 4;
        int             nlca = 1;
        int             nuca = 1;
        f_complex      *b;
        f_complex      *z;
        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, 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, 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

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} \left|A_{ij}\right|$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\|A\|_\infty = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} \left|A_{ij}\right|$$

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

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} |A_{ij}|$$

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 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:

$a_{i,i} = 4$	$0 \le i < n$
$a_{i,i+1} = -1$	$0 \le i < \mathrm{n} - 1$
$a_{i+1,1} = -1$	$0 \le i < \mathrm{n} - 1$
$a_{i,i+c} = -1$	$0 \le i < \mathrm{n} - c$
$a_{i+c,i} = -1$	$0 \le i < \mathrm{n} - c$

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 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

$a_{i,i} = 4$	$0 \le i < n$
$a_{i,i+1} = -1$	$0 \le i < n - 1$
$a_{i+1,1} = -1$	$0 \le i < n - 1$
$a_{i,i+c} = -1$	$0 \le i < n - c$
$a_{i+c,i} = -1$	$0 \le i < n - c$

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:

$a_{i,i} = 1$	$0 \le i < n$
$a_{i,i+c} = i + 2$	$0 \le i < n - c$
$a_{i,i-n+c} = i + 2$	$n - c \le i < n$
$a_{i,i+c+1} = -(i + 1)$	$0 \le i < n - c - 1$
$a_{i,i-n+c+1} = -(i + 1)$	$n - c - 1 \le i < n$
$a_{i,i+c+2} = 16$	$0 \le i < n - c - 2$
$a_{i,i-n+c+2} = 16$	$n - c - 2 \le i < n$
$a_{i,n-11+i+j} = 100j$	$1 \le i < 11 - j, \quad 0 \le j < 10$

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:

$a_{i,i} = 6$	$0 \le i < n$
$a_{i,i+1} = -1 - i$	$0 \le i < n - 1$
$a_{i+1,1} = -1 - i$	$0 \le i < n - 1$
$a_{i,i+c} = -1 + i$	$0 \le i < n - c$
$a_{i+c,i} = -1 + i$	$0 \le i < n - c$

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:

$a_{i,i} = 1$	$0 \le i < n$
$a_{i,i+c} = i + 2$	$0 \le i < n - c$
$a_{i,i-n+c} = i + 2$	$n - c \le i < n$
$a_{i,i+c+1} = -(i + 1)$	$0 \le i < n - c - 1$
$a_{i,i+c+1} = -(i + 1)$	$n - c - 1 \le i < n$
$a_{i,i+c+2} = 16$	$0 \le i < n - c - 2$
$a_{i,i-n+c+2} = 16$	$n - c - 2 \le i < n$
$a_{i,n-11+i+j} = 100j$	$1 \le i < 11 - j, \quad 0 \le j < 10$

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 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`.

Threads and Error Handling

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options using `imsl_error_options` (excluding `IMSL_SET_SIGNAL_TRAPPING`) for each thread by calling `imsl_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsl_error_options(IMSL_SET_SIGNAL_TRAPPING, 0, 0);
```

See Examples 3 and 4 of `imsl_error_options` for multithreaded examples.

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)
$z = \overline{x}$	`z = imsl_c_conjg(x)`	`f_complex`	`f_complex`
$a = \|z\|$	`a = imsl_c_abs(z)`	`float`	`f_complex`
$a = \arg(z)$ $-\pi < a \le \pi$	`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)
$z = \bar{x}$	`z = imsl_z_conjg(x)`	`d_complex`	`d_complex`
$a = \|z\|$	`a = imsl_z_abs(z)`	`double`	`d_complex`
$a = \arg(z)$ $-\pi < a \le \pi$	`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
```

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 emailing:

- `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 license number
2. Include the product name and version number: IMSL C/Stat/Library Version 5.5
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

Appendix A: References

Abramowitz and Stegun

Abramowitz, Milton, and Irene A. Stegun (editors) (1964), *Handbook of Mathematical Functions with Formulas*, *Graphs*, *and Mathematical Tables*, National Bureau of Standards, Washington.

Ahrens and Dieter

Ahrens, J.H., and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing,* **12**, 223–246.

Akima

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, *Journal of the ACM*, **17**, 589–602.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, *ACM Transactions on Mathematical Software*, **4**, 148–159.

Ashcraft

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse symmetric positive definite systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

Ashcraft et al.

Ashcraft, C., R. Grimes, J. Lewis, B. Peyton, and H. Simon (1987), Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.*, **1**(**4**), 10–29.

Atkinson (1979)

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.

Atkinson (1978)

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

Barnett

Barnett, A.R. (1981), An algorithm for regular and irregular Coulomb and Bessel functions of real order to machine accuracy, *Computer Physics Communication*, **21**, 297–314.

Barrett and Healy

Barrett, J.C., and M. J.R. Healy (1978), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **27**, 379–380.

Bays and Durham

Bays, Carter, and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59–64.

Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

Boisvert

Boisvert, Ronald (1984), A fourth order accurate fast direct method of the Helmholtz equation, *Elliptic Problem solvers II*, (edited by G. Birkhoff and A. Schoenstadt), Academic Press, Orlando, Florida, 35–44.

Bosten and Battiste

Bosten, Nancy E., and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156–157.

Brent

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Brigham

Brigham, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

Burgoyne

Burgoyne, F.D. (1963), Approximations to Kelvin functions, *Mathematics of Computation*, **83**, 295-298.

Carlson

Carlson, B.C. (1979), Computing elliptic integrals by duplication, *Numerische Mathematik*, **33**, 1–16.

Carlson and Notis

Carlson, B.C., and E.M. Notis (1981), Algorithms for incomplete elliptic integrals, *ACM Transactions on Mathematical Software*, **7**, 398–403.

Carlson and Foley

Carlson, R.E., and T.A. Foley (1991),The parameter R^2 in multiquadric interpolation, *Computer Mathematical Applications*, **21**, 29–42.

Cheng

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.

Cohen and Taylor

Cohen, E. Richard, and Barry N. Taylor (1986), *The 1986 Adjustment of the Fundamental Physical Constants*, Codata Bulletin, Pergamon Press, New York.

Cooley and Tukey

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.

Cooper

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190–192.

Courant and Hilbert

Courant, R., and D. Hilbert (1962), *Methods of Mathematical Physics,* Volume II, John Wiley & Sons, New York, NY.

Craven and Wahba

Craven, Peter, and Grace Wahba (1979), Smoothing noisy data with spline functions, *Numerische Mathematik*, **31**, 377–403.

Crowe et al.

Crowe, Keith, Yuan-An Fan, Jing Li, Dale Neaderhouser, and Phil Smith (1990), *A direct sparse linear equation solver using linked list storage*, IMSL Technical Report 9006, IMSL, Houston.

Davis and Rabinowitz

Davis, Philip F., and Philip Rabinowitz (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

de Boor

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

Dennis and Schnabel

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Dongarra et al.

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK User's Guide*, SIAM, Philadelphia.

Draper and Smith

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, 2nd. ed., John Wiley & Sons, New York.

DuCroz et al.

Du Croz, Jeremy, P. Mayes, and G. Radicati (1990), Factorization of band matrices using Level-3 BLAS, *Proceedings of CONPAR 90-VAPP IV*, Lecture Notes in Computer Science, Springer, Berlin, 222.

Duff et al.

Duff, I. S., A. M. Erisman, and J. K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

Duff and Reid

Duff, I.S., and J.K. Reid (1983), The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, **9**, 302–325.

Duff, I.S., and J.K. Reid (1984), The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, **5**, 633–641.

Enright and Pryce

Enright, W.H., and J.D. Pryce (1987), Two FORTRAN packages for assessing initial value methods, *ACM Transactions on Mathematical Software*, **13**, 1–22.

Farebrother and Berry

Farebrother, R.W., and G. Berry (1974), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **23**, 477.

Fisher

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *Annals of Eugenics*, **7**, 179– 188.

Fishman and Moore

Fishman, George S. and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$, *Journal of the American Statistical Association*, **77**, 129–136.

Forsythe

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74–88.

Franke

Franke, R. (1982), Scattered data interpolation: Tests of some methods, *Mathematics of Computation*, **38**, 181–200.

Garbow et al.

Garbow, B.S., J.M. Boyle, K.J. Dongarra, and C.B. Moler (1977), *Matrix Eigensystem Routines - EISPACK Guide Extension*, Springer–Verlag, New York.

Garbow, B.S., G. Giunta, J.N. Lyness, and A. Murli (1988), Software for an implementation of Weeks' method for the inverse Laplace transform problem, *ACM Transactions on Mathematical Software*, **14**, 163–170.

Gautschi

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251–270.

Gautschi, Walter (1969), Complex error function, *Communications of the ACM*, **12**, 635. Gautschi, Walter (1970), Efficient computation of the complex error function, *SIAM Journal on Mathematical Analysis*, **7**, 187−198.

Gear

Gear, C.W. (1971), Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, New Jersey.

Gentleman

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448–454.

George and Liu

George, A., and J.W.H. Liu (1981), *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gill and Murray

Gill, Philip E., and Walter Murray (1976), *Minimization subject to bounds on the variables*, NPL Report NAC 92, National Physical Laboratory, England.

Gill et al.

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Goldfarb and Idnani

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1–33.

Golub

Golub, G.H. (1973), Some modified matrix eigenvalue problems, *SIAM Review*, **15**, 318–334.

Golub and Van Loan

Golub, G.H., and C.F. Van Loan (1989), *Matrix Computations*, Second Edition, The Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland.

Golub and Welsch

Golub, G.H., and J.H. Welsch (1969), Calculation of Gaussian quadrature rules, *Mathematics of Computation*, **23**, 221–230.

Gregory and Karney

Gregory, Robert, and David Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, John Wiley & Sons, New York.

Griffin and Redfish

Griffin, R., and K A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

Grosse

Grosse, Eric (1980), Tensor spline approximation, *Linear Algebra and its Applications*, **34**, 29–41.

Guerra and Tapia

Guerra, V., and R. A. Tapia (1974), *A local procedure for error detection and data smoothing*, MRC Technical Summary Report 1452, Mathematics Research Center, University of Wisconsin, Madison.

Hageman and Young

Hageman, Louis A., and David M. Young (1981), *Applied Iterative Methods*, Academic Press, New York.

Hanson

Hanson, Richard J. (1986), Least squares with bounds and linear constraints, *SIAM Journal Sci. Stat. Computing*, **7**, #3.

Hardy

Hardy, R.L. (1971), Multiquadric equations of topography and other irregular surfaces, *Journal of Geophysical Research*, **76**, 1905–1915.

Hart et al.

Hart, John F., E.W. Cheney, Charles L. Lawson, Hans J.Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, Jr., and Christoph Witzgall (1968), *Computer Approximations*, John Wiley & Sons, New York.

Healy

Healy, M.J.R. (1968), Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **17**, 195–197.

Herraman

Herraman, C. (1968), Sums of squares and products matrix, *Applied Statistics*, **17**, 289–292.

Higham

Higham, Nicholas J. (1988), FORTRAN Codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, *ACM Transactions on Mathematical Software*, **14**, 381-396.

Hill

Hill, G.W. (1970), Student's *t*-distribution, *Communications of the ACM*, **13**, 617–619.

Hindmarsh

Hindmarsh, A.C. (1974)*, GEAR: Ordinary Differential Equation System Solver*, Lawrence Livermore National Laboratory Report UCID-30001, Revision 3, Lawrence Livermore National Laboratory, Livermore, Calif.

Hinkley

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67–69.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hull et al.

Hull, T.E., W.H. Enright, and K.R. Jackson (1976), *User's guide for DVERK — A subroutine for solving non-stiff ODEs*, Department of Computer Science Technical Report 100, University of Toronto.

Irvine et al.

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129–151.

Jackson et al.

Jackson, K.R., W.H. Enright, and T.E. Hull (1978), A theoretical criterion for comparing Runge-Kutta formulas, *SIAM Journal of Numerical Analysis*, **15**, 618–641.

Jenkins

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178–189.

Jenkins and Traub

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, **7**, 545–566.

Jenkins, M.A., and J.F. Traub (1970), A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration, *Numerishe Mathematik*, **14**, 252–263.

Jenkins, M.A., and J.F. Traub (1972), Zeros of a complex polynomial, *Communications of the ACM*, **15**, 97– 99.

Jöhnk

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufalls-zahlen, *Metrika*, **8**, 5–15.

Kendall and Stuart

Kendall, Maurice G., and Alan Stuart (1973), *The Advanced Theory of Statistics*, Volume II, *Inference and Relationship*, Third Edition, Charles Griffin & Company, London, Chapter 30.

Kennedy and Gentle

Kennedy, William J., Jr., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kernighan and Richtie

Kernighan, Brian W., and Richtie, Dennis M. 1988, "The C Programming Language" Second Edition, **241**.

Kinnucan and Kuki

Kinnucan, P., and Kuki, H., (1968), *A single precision inverse error function subroutine*, Computation Center, University of Chicago.

Knuth

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume II: *Seminumerical Algorithms*, 2nd. ed., Addison-Wesley, Reading, Mass.

Learmonth and Lewis

Learmonth, G.P., and P.A.W. Lewis (1973), *Naval Postgraduate School Random Number Generator Package LLRANDOM, NPS55LW73061A*, Naval Postgraduate School, Monterey, California.

Lehmann

Lehmann, E.L. (1975), *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, San Francisco.

Levenberg

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

Leavenworth

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

Lentini and Pereyra

Pereyra, Victor (1978), PASVA3: An adaptive finite-difference FORTRAN program for first order nonlinear boundary value problems, in *Lecture Notes in Computer Science*, **76**, Springer-Verlag, Berlin, 67–88.

Lewis et al.

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/ 360, *IBM Systems Journal*, **8**, 136–146.

Liepman

Liepman, David S. (1964), Mathematical constants, in *Handbook of Mathematical Functions*, Dover Publications, New York.

Liu

Liu, J.W.H. (1987), *A collection of routines for an implementation of the multifrontal method*, Technical Report CS-87-10, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1989), The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, **15**, 310-325.

Liu, J.W.H. (1990), *The multifrontal method for sparse matrix solution: theory and practice*, Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1986), On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, **12**, 249-264.

Lyness and Giunta

Lyness, J.N. and G. Giunta (1986), A modification of the Weeks Method for numerical inversion of the Laplace transform, *Mathematics of Computation*, **47**, 313–322.

Madsen and Sincovec

Madsen, N.K., and R.F. Sincovec (1979), Algorithm 540: PDECOL, General collocation software for partial differential equations, *ACM Transactions on Mathematical Software*, **5**, #3, 326–351.

Maindonald

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

Marquardt

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431–441.

Martin and Wilkinson

Martin, R.S., and J.H. Wilkinson (1971), Reduction of the Symmetric Eigenproblem $\mathbf{A}x = \lambda\mathbf{B}x$ and Related Problems to Standard Form, *Volume II, Linear Algebra Handbook*, Springer, New York.

Martin, R.S., and J.H. Wilkinson (1971), The Modified LR Algorithm for Complex Hessenberg Matrices, *Handbook, Volume II, Linear Algebra*, Springer, New York.

Mayle

Mayle, Jan, (1993), Fixed Income Securities Formulas for Price, Yield, and Accrued Interest, *SIA Standard Securities Calculation Methods*, Volume I, Third Edition, pages 17-35.

Michelli

Micchelli, C.A. (1986), Interpolation of scattered data: Distance matrices and conditionally positive definite functions, *Constructive Approximation*, **2**, 11–22.

Michelli et al.

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth functions, *Numerische Mathematik*, **26**, 279–285.

Micchelli, C.A., Philip W. Smith, John Swetits, and Joseph D. Ward (1985), Constrained L_p approximation, *Constructive Approximation*, **1**, 93–102.

Moler and Stewart

Moler, C., and G.W. Stewart (1973), An algorithm for generalized matrix eigenvalue problems, *SIAM Journal on Numerical Analysis*, **10**, 241-256.

Moré et al.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74, Argonne, Illinois.

Müller

Müller, D.E. (1956), A method for solving algebraic equations using an automatic computer, *Mathematical Tables and Aids to Computation*, **10**, 208–215.

Murtagh

Murtagh, Bruce A. (1981), Advanced Linear Programming: Computation and Practice, McGraw-Hill, New York.

Murty

Murty, Katta G. (1983), *Linear Programming*, John Wiley and Sons, New York.

Neter and Wasserman

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Illinois.

Neter et al.

Neter, John, William Wasserman, and Michael H. Kutner (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Illinois.

Østerby and Zlatev

Østerby, Ole, and Zahari Zlatev (1982), Direct Methods for Sparse Matrices, *Lecture Notes in Computer Science*, **157**, Springer-Verlag, New York.

Owen

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley Publishing Company, Reading, Mass.

Owen, D.B. (1965), A special case of the bivariate non-central t distribution, *Biometrika*, **52**, 437–446.

Parlett

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. Überhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

Powell

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G. A. Watson), **630**, Springer-Verlag, Berlin, Germany, 144–157.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46–61.

Powell, M.J.D. (1988), *A tolerant algorithm for linearly constrained optimizations calculation*s, DAMTP Report NA17, University of Cambridge, England.

Powell, M.J.D. (1989), *TOLMIN: A fortran package for linearly constrained optimizations calculations*, DAMTP Report NA2, University of Cambridge, England.

Powell, M.J.D. (1983), *ZQPCVX a FORTRAN subroutine for convex quadratic programming*, DAMTP Report 1983/NA17, University of Cambridge, Cambridge, England.

Reinsch

Reinsch, Christian H. (1967), Smoothing by spline functions, *Numerische Mathematik*, **10**, 177–183.

Rice

Rice, J.R. (1983), *Numerical Methods, Software, and Analysis*, McGraw-Hill, New York.

Saad and Schultz

Saad, Y., and M. H. Schultz (1986), GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems, *SIAM Journal of Scientific and Statistical Computing*, **7**, 856-869.

Sallas and Lionti

Sallas, William M., and Abby M. Lionti (1988), Some useful computing formulas for the nonfull rank linear model with linear equality restrictions, IMSL Technical Report 8805, IMSL, Houston.

Savage

Savage, I. Richard (1956), Contributions to the theory of rank order statistics—the two-sample case, *Annals of Mathematical Statistics*, **27**, 590–615.

Schmeiser

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, in *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154–160.

Schmeiser and Babu

Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917–926.

Schmeiser and Kachitvichyanukul

Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81–4, School of Industrial Engineering, Purdue University, West Lafayette, Indiana.

Schmeiser and Lal

Schmeiser, Bruce W., and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679–682.

Seidler and Carmichael

Seidler, Lee J. and Carmichael, D.R., (editors) (1980), *Accountants' Handbook*, Volume I, Sixth Edition, The Ronald Press Company, New York.

Shampine

Shampine, L.F. (1975), Discrete least squares polynomial fits, *Communications of the ACM*, **18**, 179–180.

Shampine and Gear

Shampine, L.F. and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1–17.

Sincovec and Madsen

Sincovec, R.F., and N.K. Madsen (1975), Software for nonlinear partial differential equations, *ACM Transactions on Mathematical Software*, **1**, #3, 232–260.

Singleton

Singleton, T.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185–187.

Smith et al.

Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler (1976), *Matrix Eigensystem Routines — EISPACK Guide*, Springer-Verlag, New York.

Smith

Smith, P.W. (1990), On knots and nodes for spline interpolation, *Algorithms for Approximation II*, J.C. Mason and M.G. Cox, Eds., Chapman and Hall, New York.

Spellucci, Peter

Spellucci, P. (1998), An SQP method for general nonlinear programs using only equality constrained subproblems, *Math. Prog.,* **82**, 413-448, Physica Verlag, Heidelberg, Germany

Spellucci, P. (1998), A new technique for inconsistent problems in the SQP method. *Math. Meth. of Oper. Res.,***47**, 355-500, Physica Verlag, Heidelberg, Germany.

Stewart

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Strecok

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144–158.

Stroud and Secrest

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

Temme

Temme, N.M (1975), On the numerical evaluation of the modified Bessel Function of the third kind, *Journal of Computational Physics*, **19**, 324–337.

Tezuka

Tezuka, S. (1995), *Uniform Random Numbers: Theory and Practice*. Academic Publishers, Boston.

Thompson and Barnett

Thompson, I.J. and A.R. Barnett (1987), Modified Bessel functions $I_\nu(z)$ and $K_\nu(z)$ of real order and complex argument, *Computer Physics Communication*, **47**, 245–257.

Tukey

Tukey, John W. (1962), The future of data analysis, *Annals of Mathematical Statistics,* **33**, 1–67.

Velleman and Hoaglin

Velleman, Paul F., and David C. Hoaglin (1981), *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.

Walker

Walker, H.F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM Journal of Scientific and Statistical Computing*, **9**, 152-163.

Watkins

Watkins, David S., L. Elsner (1991), Convergence of algorithm of decomposition type for the eigenvalue problem, *Linear Algebra Applications*, **143**, pp. 29–47.

Weeks

Weeks, W.T. (1966), Numerical inversion of Laplace transforms using Laguerre functions, *J. ACM*, **13**, 419–429.

Appendix B: Alphabetical Summary of Routines

Function	Purpose Statement	Page
accr_interest_maturity	Evaluates the accrued interest for a security that pays at maturity.	580
accr_interest_periodic	Evaluates the accrued interest for a security that pays periodic interest.	582
airy_Ai	Evaluates the Airy function.	509
airy_Ai_derivative	Evaluates the derivative of the Airy function	511
airy_Bi	Evaluates the Airy function of the second kind.	510
airy_Bi_derivative	Evaluates the derivative of the Airy function of the second kind.	512
bessel_exp_I0	Evaluates the exponentially scale modified Bessel function of the first kind of order zero.	489
bessel_exp_I1	Evaluates the exponentially scaled modified Bessel function of the first kind of order one.	491
bessel_exp_K0	Evaluates the exponentially scaled modified Bessel function of the third kind of order zero.	495
bessel_exp_K1	Evaluates the exponentially scaled modified Bessel function of the third kind of order one.	497
bessel_I0	Evaluates the real modified Bessel function of the first kind of order zero $I0(x)$.	487
bessel_I1	Evaluates the real modified Bessel function of the first kind of order one $I1(x)$.	490
bessel_Ix	Evaluates a sequence of modified Bessel functions of the first kind with real order and complex arguments.	492
bessel_J0	Evaluates the real Bessel function of the first kind of order zero $J0(x)$.	478
bessel_J1	Evaluates the real Bessel function of the first kind of order one $J1(x)$.	480
bessel_Jx	Evaluates a sequence of Bessel functions of the first kind with real order and complex arguments.	481
bessel_K0	Evaluates the real modified Bessel function of the third kind of order zero $K0(x)$.	493

Function	Purpose Statement	Page
bessel_K1	Evaluates the real modified Bessel function of the third kind of order one $K1(x)$.	496
bessel_Kx	Evaluates a sequence of modified Bessel functions of the third kind with real order and complex arguments.	499
bessel_Y0	Evaluates the real Bessel function of the second kind of order zero $Y0(x)$.	482
bessel_Y1	Evaluates the real Bessel function of the second kind of order one $Y1(x)$.	484
bessel_Yx	Evaluates a sequence of Bessel functions of the second kind with real order and complex arguments.	485
beta	Evaluates the real beta function $\beta(x, y)$.	469
beta_cdf	Evaluates the beta probability distribution function	540
beta_incomplete	Evaluates the real incomplete beta function $I\mathrm{x} = \beta\mathrm{x}(a, b)/\beta(a, b)$.	472
beta_inverse_cdf	Evaluates the inverse of the beta distribution function.	542
binomial_cdf	Evaluates the binomial distribution function.	536
bivariate_normal_cdf	Evaluates the bivariate normal distribution function.	543
bond_equivalent_yield	Evaluates the bond-equivalent for a Treasury yield.	584
bounded_least_squares	Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.	439
bvp_finite_difference	Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite difference method with deferred corrections.	321
chi_squared_cdf	Evaluates the chi-squared distribution function	524
chi_squared_inverse_cdf	Evaluates the inverse of the chi-squared distribution function.	526
chi_squared_test	Performs a chi-squared goodness-of-fit test	638
constant	Returns the value of various mathematical and physical constants.	719
constrained_nlp	Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method.	447
convexity	Evaluates the convexity for a security.	586
convolution (complex)	Computes the convolution, and optionally, the correlation of two complex vectors.	370
convolution	Computes the convolution, and optionally, the correlation of two real vectors.	363

Function	Purpose Statement	Page
coupon_days	Evaluates the number of days in the coupon period that contains the settlement date.	588
coupon_number	Evaluates the number of coupons payable between the settlement date and maturity date.	589
covariances	Computes the sample variance-covariance or correlation matrix.	646
ctime	Returns the number of CPU seconds used.	709
cub_spline_integral	Computes the integral of a cubic spline.	160
cub_spline_interp_e_cnd	Computes a cubic spline interpolant, specifying various endpoint conditions.	145
cub_spline_interp_shape	Computes a shape-preserving cubic spline.	152
cub_spline_smooth	Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.	205
cub_spline_value	Computes the value of a cubic spline or the value of one of its derivatives.	157
cumalative_interest	Evaluates the cumulative interest paid between two periods.	545
cumalative_principal	Evaluates the cumulative principal paid between two periods.	546
date_to_days	Evaluates the number of days from January 1, 1900, to the given date.	709
days_before_settlement	Evaluates the number of days from the beginning of the coupon period to the settlement date.	591
days_to_date	Gives the date corresponding to the number of days since January 1, 1900.	711
days_to_next_coupon	Evaluates the number of days from settlement date to the next coupon date.	592
depreciation_amordegrc	Evaluates the depreciation for each accounting period. Similar to `depreciation_amorlinc`.	594
depreciation_amorlinc	Evaluates the depreciation for each accounting period. Similar to `depreciation_amordegrc`.	596
depreciation_db	Evaluates the depreciation of an asset for a specified period using the fixed-declining balance method.	548
depreciation_ddb	Evaluates the depreciation of an asset for a specified period using the double-declining method.	550
depreciation_sln	Evaluates the straight line depreciation of an asset for one period.	551
depreciation_syd	Evaluates the sum-of-years digits depreciation of an asset for a specified period.	553

Function	Purpose Statement	Page
`depreciation_vdb`	Evaluates the depreciation of an asset for any given period, including partial periods, using the double-declining balance method.	554
`discount_price`	Evaluates the price per $100 face value of a discounted security.	597
`discount_rate`	Evaluates the discount rate for a security.	599
`discount_yield`	Evaluates the annual yield for a discounted security.	601
`dollar_decimal`	Converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number.	556
`dollar_fraction`	Converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fraction.	557
`duration`	Evaluates the annual duration of a security with periodic interest payment.	603
`effective_rate`	Evaluates the effective annual interest rate.	558
`eig_gen (complex)`	Computes the eigenexpansion of a complex matrix *A*.	120
`eig_gen`	Computes the eigenexpansion of a real matrix *A*	118
`eig_herm (complex)`	Computes the eigenexpansion of a complex Hermitian matrix *A*.	126
`eig_sym`	Computes the eigenexpansion of a real symmetric matrix *A*.	123
`eig_symgen`	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. *A* and *B* are real and symmetric. *B* is positive definite.	129
`elliptic_integral_E`	Evaluates the complete elliptic integral of the second kind $E(x)$.	501
`elliptic_integral_K`	Evaluates the complete elliptic integral of the kind $K(x)$.	500
`elliptic_integral_RC`	Evaluates an elementary integral from which inverse circular functions, logarithms, and inverse hyperbolic functions can be computed.	506
`elliptic_integral_RD`	Evaluates Carlson's elliptic integral of the second kind $R\text{D}(x, y, z)$.	504
`elliptic_integral_RF`	Evaluates Carlson's elliptic integral of the first kind $R\text{F}(x, y, z)$.	502
`elliptic_integral_RJ`	Evaluates Carlson's elliptic integral of the third kind $R\text{J}(x, y, z, \rho)$.	505
`erf`	Evaluates the real error function erf(x).	460
`erf_inverse`	Evaluates the real inverse error function erf-1(x).	465
`erfc`	Evaluates the real complementary error function erfc(x).	461

Function	Purpose Statement	Page
erfc_inverse	Evaluates the real inverse complementary error function erfc-1(*x*).	467
erfce	Evaluates the exponentially scaled complementary error function.	463
erfe	Evaluates a scaled function related to erfc(z)	464
error_code	Gets the code corresponding to the error message from the last function called.	718
error_options	Sets various error handling options.	712
F_cdf	Evaluates the *F* distribution function.	528
F_inverse_cdf	Evaluates the inverse of the *F* distribution function.	530
fast_poisson_2d	Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.	332
faure_next_point	Evaluates a shuffled Faure sequence	687
fcn_derivative	Computes the first, second or third derivative of a user-supplied function.	286
fft_2d_complex	Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array.	359
fft_complex	Computes the complex discrete Fourier transform of a complex sequence.	346
fft_complex_init	Computes the parameters for imsl_c_fft_complex.	349
fft_cosine	Computes the discrete Fourier cosine transformation of an even sequence.	351
fft_cosine_init	Computes the parameters needed for imsl_f_fft_cosine.	353
fft_real	Computes the real discrete Fourier transform of a real sequence.	341
fft_real_init	Computes the parameters for imsl_f_fft_real	345
fft_sine	Computes the discrete Fourier sine transformation of an odd sequence.	355
fft_sine_init	Computes the parameters needed for imsl_f_fft_sine.	357
fresnel_integral_C	Evaluates the cosine Fresnel integral.	507
fresnel_integral_S	Evaluates the sine Fresnel integral.	508
future_value	Evaluates the future value of an investment.	559
future_value_schedule	Evaluates the future value of an initial principal after applying a series of compound interest rates.	561
gamma	Evaluates the real gamma function $\Gamma(x)$.	473
gamma_cdf	Evaluates the gamma distribution function	534

Function	Purpose Statement	Page
gamma_incomplete	Evaluates the incomplete gamma function $\gamma(a, x)$.	476
gauss_quad_rule	Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.	282
geneig (complex)	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B complex.	135
geneig	Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with A and B real.	132
generate_test_band (complex)	Generates test matrices of class Ec(n, c).	784
generate_test_band	Generates test matrices of class $E(n, c)$.	782
generate_test_coordinate (complex)	Generates test matrices of class $D(n, c)$ and $E(n, c)$.	791
generate_test_coordinate	Generates test matrices of class $D(n, c)$ and $E(n, c)$.	786
hypergeometric_cdf	Evaluates the hypergeometric distribution function.	537
int_fcn	Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.	241
int_fcn_2d	Computes a two-dimensional iterated integral	272
int_fcn_alg_log	Integrates a function with algebraic-logarithmic singularities.	249
int_fcn_cauchy	Computes integrals of the form $$\int_a^b \frac{f(x)}{x-c}dx$$ in the Cauchy principal value sense.	265
int_fcn_fourier	Computes a Fourier sine or cosine transform.	261
int_fcn_hyper_rect	Integrates a function on a hyper-rectangle.	276
int_fcn_inf	Integrates a function over an infinite or semi-infinite interval.	253
int_fcn_qmc	Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.	279
int_fcn_sing	Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules.	237
int_fcn_sing_pts	Integrates a function with singularity points given	245
int_fcn_smooth	Integrates a smooth function using a nonadaptive rule.	268
int_fcn_trig	Integrates a function containing a sine or a cosine factor.	257
interest_payment	Evaluates the interest payment for a given period for an investment.	562
interest_rate_annuity	Evaluates the interest rate per period for an annuity.	563

Function	Purpose Statement	Page
interest_rate_security	Evaluates the interest rate for a fully invested security.	605
internal_rate_of_return	Evaluates the internal rate of return for a schedule of cash flows.	565
internal_rate_schedule	Evaluates the internal rate of return for a schedule of cash flows that is not necessarily periodic.	567
inverse_laplace	Computes the inverse Laplace transform of a complex function.	376
kelvin_bei0	Evaluates the Kelvin function of the first kind, bei, of order zero.	514
kelvin_bei0_derivative	Evaluates the derivative of the Kelvin function of the first kind, bei, of order zero.	518
kelvin_ber0	Evaluates the Kelvin function of the first kind, ber, of order zero.	513
kelvin_ber0_derivative	Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero.	517
kelvin_kei0	Evaluates the Kelvin function of the second kind, kei, of order zero.	516
kelvin_kei0_derivative	Evaluates the derivative of the Kelvin function of the second kind, kei, of order zero.	520
kelvin_ker0	Evaluates the Kelvin function of the second kind, der, of order zero.	515
kelvin_ker0_derivative	Evaluates the derivative of the Kelvin function of the second kind, ker, of order zero.	519
lin_least_squares_gen	Solves a linear least-squares problem $Ax = b$.	84
lin_lsq_lin_constraints	Solves a linear least squares problem with linear constraints.	92
lin_prog	Solves a linear programming problem using the revised simplex algorithm.	425
lin_sol_def_cg	Solves a real symmetric definite linear system using a conjugate gradient method.	78
lin_sol_gen (complex)	Solves a complex general system of linear equations $Ax = b$.	11
lin_sol_gen	Solves a real general system of linear equations $Ax = b$.	4
lin_sol_gen_band (complex)	Solves a complex general system of linear equations $Ax = b$.	31
lin_sol_gen_band	Solves a real geeral band system of linear equations $Ax=b$.	26
lin_sol_gen_coordinate (complex)	Solves a system of linear equations $Ax = b$, with sparse complex coefficient matrix A.	54
lin_sol_gen_coordinate	Solves a sparse system of linear equations $Ax = b$.	44

Function	Purpose Statement	Page
`lin_sol_gen_min_residual`	Solves a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method.	73
`lin_sol_nonnegdef`	Solves a real symmetric nonnegative definite system of linear equations $Ax = b$.	107
`lin_sol_posdef (complex)`	Solves a complex Hermitian positive definite system of linear equations $Ax = b$.	22
`lin_sol_posdef`	Solves a real symmetric positive definite system of linear equations $Ax = b$.	17
`lin_sol_posdef_band (complex)`	Solves a complex Hermitian positive definite system of linear equations $Ax = b$ in band symmetric storage mode.	39
`lin_sol_posdef_band`	Solves a real symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode.	35
`lin_sol_posdef_coordinate (complex)`	Solves a sparse Hermitian positive definite system of linear equations $Ax = b$.	68
`lin_sol_posdef_coordinate`	Solves a sparse real symmetric positive definite system of linear equations $Ax = b$.	62
`lin_svd_gen (complex)`	Computes the SVD, $A = USV$H, of a complex rectangular matrix A.	102
`lin_svd_gen`	Computes the SVD, $A = USV$T, of a real rectangular matrix A.	96
`log_beta`	Evaluates the logarithm of the real beta function ln β (x, y).	471
`log_gamma`	Evaluates the logarithm of the absolute value of the gamma function log $\|\Gamma(x)\|$.	475
`machine (float)`	Returns information describing the computer's floating-point arithmetic.	725
`machine (integer)`	Returns integer information describing the computer's arithmetic.	723
`mat_add_band (complex)`	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.	764
`mat_add_band`	Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$.	760
`mat_add_coordinate (complex)`	Performs element-wise addition on two complex matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.	771
`mat_add_coordinate`	Performs element-wise addition of two real matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$.	768
`mat_mul_rect (complex)`	Computes the transpose of a matrix, the conjugate-transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product.	738

Function	Purpose Statement	Page
mat_mul_rect	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product.	735
mat_mul_rect_band (complex)	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices of complex type and stored in band form.	746
mat_mul_rect_band	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices stored in band form.	742
mat_mul_rect_coordinate (complex)	Computes the transpose of a matrix, a matrix-vector product or a matrix-matrix product, all matrices stored in sparse coordinate form.	755
mat_mul_rect_coordinate	Computes the transpose of a matrix, a matrix-vector product, or a matrix-matrix product, all matrices stored in sparse coordinate form.	751
matrix_norm	Computes various norms of a rectangular matrix.	775
matrix_norm_band	Computes various norms of a matrix stored in band storage mode.	777
matrix_norm_coordinate	Computes various norms of a matrix stored in coordinate format.	779
min_con_gen_lin	Minimizes a general objective function subject to linear equality/inequality constraints.	433
min_uncon	Finds the minimum point of a smooth function *f(x)* of a single variable using only function evaluations.	401
min_uncon_deriv	Finds the minimum point of a smooth function *f(x)* of a single variable using both function and first derivative evaluations.	405
min_uncon_multivar	Minimizes a function *f(x)* of *n* variables using a quasi-Newton method.	409
modified_duration	Evaluates the modified Macauley duration of a security.	607
modified_internal_rate	Evaluates the modified internal rate of return for a series of periodic cash flows.	569
net_present_value	Evaluates the net present value of an investment based on a series of periodic.	570
next_coupon_date	Evaluates the next coupon date after the settlement date.	608
nominal_rate	Evaluates the nominal annual interest rate.	571
nonlin_least_squares	Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm.	416
normal_cdf	Evaluates the standard normal (Gaussian) distribution function.	521
normal_inverse_cdf	Evaluates the inverse of the standard normal (Gaussian) distribution function.	523

Function	Purpose Statement	Page
number_of_periods	Evaluates the number of periods for an investment based on periodic and constant payment and a constant interest rate.	573
ode_adams_gear	Solves a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods.	297
ode_runge_kutta	Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.	291
output_file	Sets the output file or the error message output file.	704
page	Sets or retrieve the page width or length.	697
payment	Evaluates the periodic payment for an investment.	574
pde_method_of_lines	Solves a system of partial differential equations of the form ut + f(x, t, u, ux, $u$xx) using the method of lines.	304
poisson_cdf	Evaluates the Poisson distribution function.	539
poly_regression	Performs a polynomial least-squares regression.	660
present_value	Evaluates the present value of an investment.	576
present_value_schedule	Evaluates the present value for a schedule of cash flows that is not necessarily periodic.	577
previous_coupon_date	Evaluates the previous coupon date before the settlement date.	610
price	Evaluates the price per $100 face value of a security that pays periodic interest.	612
price_maturity	Evaluates the price per $100 face value of a security that pays interest at maturity.	614
principal_payment	Evaluates the payment on the principal for a given period.	579
quadratic_prog	Solves a quadratic programming problem subject to linear equality or inequality constraints.	429
radial_evaluate	Evaluates a radial basis fit.	231
radial_scattered_fit	Computes an approximation to scattered data in R^n for $n \geq 2$ using radial basis functions.	225
random_beta	Generates pseudorandom numbers from a beta distribution.	684
random_exponential	Generates pseudorandom numbers from a standard exponential distribution.	685
random_gamma	Generates pseudorandom numbers from a standard gamma distribution.	682
random_normal	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.	679
random_option	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.	676

Function	Purpose Statement	Page
random_poisson	Generates pseudorandom numbers from a Poisson distribution.	680
random_seed_get	Retrieves the current value of the seed used in the IMSL random number generators.	674
random_seed_set	Initializes a random seed for use in the IMSL random number generators.	675
random_uniform	Generates pseudorandom numbers from a uniform (0, 1) distribution.	677
ranks	Computes the ranks, normal scores, or exponential scores for a vector of observations.	667
received_maturity	Evaluates the amount received for a fully invested security.	616
regression	Fits a multiple linear regression model using least squares.	651
scattered_2d_interp	Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.	220
simple_statistics	Computes basic univariate statistics.	629
smooth_1d_data	Smooth one-dimensional data by error detection	216
sort (integer)	Sorts an integer vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned.	730
sort	Sorts a vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort permutation can be returned.	728
spline_2d_integral	Evaluates the integral of a tensor-product spline on a rectangular domain.	186
spline_2d_interp	Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data.	171
spline_2d_least_squares	Computes a two-dimensional, tensor-product spline approximant using least squares.	199
spline_2d_value	Computes the value of a tensor-product spline or the value of one of its partial derivatives.	182
spline_integral	Computes the integral of a spline.	180
spline_interp	Computes a spline interpolant.	161
spline_knots	Computes the knots for a spline interpolant.	167
spline_least_squares	Computes a least-squares spline approximation.	193
spline_lsq_constrained	Computes a least-squares constrained spline approximation.	209
spline_value	Computes the value of a spline or the value of one of its derivatives.	177
t_cdf	Evaluates the Student's *t* distribution function.	531

Function	Purpose Statement	Page
t_inverse_cdf	Evaluates the inverse of the Student's *t* distribution function.	533
table_oneway	Tallies observations into a one-way frequency table.	634
treasury_bill_price	Computes the price per $100 face value for a Treasury bill.	618
treasury_bill_yield	Computes the yield for a Treasury bill.	619
user_fcn_least_squares	Computes a least-squares fit using user-supplied functions.	189
vector_norm	Computes various norms of a vector or the difference of two vectors.	733
version	Returns integer information describing the version of the library, license number, operating system, and compiler.	708
write_matrix	Prints a rectangular matrix (or vector) stored in contiguous memory locations.	691
write_options	Sets or retrieve an option for printing a matrix.	698
year_fraction	Evaluates the year fraction that represents the number of whole days between two dates.	621
yield_maturity	Evaluates the annual yield of a security that pays interest at maturity.	622
yield_periodic	Evaluates the yield of a security that pays periodic interest.	624
zeros_fcn	Finds the real zeros of a real function using Müller's method.	388
zeros_poly (complex)	Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm.	386
zeros_poly	Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm.	384
zeros_sys_eqn	Solves a system of n nonlinear equations $f(x) = 0$ using a modified Powell hybrid algorithm.	393

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

Z