# IMSL© C MATH LIBRARY

**Version 8.6.0**

# Contents

## Chapter 6 Transforms 699



## Chapter 7 Nonlinear Equations 759



## Chapter 8 Optimization 783

# Introduction

## Table of Contents

# IMSL C Math Library

The IMSL® C Math Library, a component of the IMSL® C Numerical Library, is a library of C functions useful in scientific programming. Each function is designed and documented for use in research activities as well as by technical specialists. A number of the example programs also show graphs of resulting output.

# Organization of the Documentation

This manual contains a concise description of each function with at least one example demonstrating the use of each function, including sample input and results. All information pertaining to a particular function is in one place within a chapter.

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

- **Section Name:** Usually, the common root for the type *float* and type *double* versions of the function is given.
- **Purpose:** A statement of the purpose of the function.
- **Synopsis:** The form for referencing the subprogram with required arguments listed.
- **Required Arguments:** A description of the required arguments in the order of their occurrence, as follows:
  - **Input:** Argument must **be** initialized; it is not changed by the function.
  - **Input/Output: Argument** must be initialized; the function returns output through this argument. The argument cannot be a constant or an expression.
  - **Output:** No **initialization** is necessary. The argument cannot be a constant or an expression; the function returns output through this argument.
- **Return Value:** The value returned by the function.
- **Synopsis with Optional Arguments:** The form for referencing the function with both required and optional arguments listed.
- **Optional Arguments:** A description of the optional arguments in the order of their occurrence.
- **Description:** A description of the algorithm and references to detailed information. In many cases, other IMSL functions with similar or complementary functions are noted.
- **Examples:** At least one application of this function showing input and optional arguments.
- **Errors:** Listing of any errors that may occur with a particular function. A discussion on error types is given in the User Errors section of the Reference Material. The errors are listed by their type as follows:
  - **Informational Errors:** List of **informational** errors that may occur with the function.
  - **Alert Errors:** List of alert errors that may **occur** with the function.
  - **Warning Errors:** List of warning **errors** that may occur with the function.
  - **Fatal Errors:** List of fatal errors that may occur with the function.

# Finding the Right Function

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

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

# Naming Conventions

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

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

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

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

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

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

# Getting Started and the imsl.h file

## Getting Started

To use any of the IMSL C Math Library functions, you first must write a program in C to call the function. Each function conforms to established conventions in programming and documentation. We give first priority in development to efficient algorithms, clear documentation, and accurate results. The uniform design of the functions makes it easy to use more than one function in a given application. Also, you will find that the design consistency enables you to apply your experience with one IMSL C Math Library function to all other IMSL functions that you use.

## The imsl.h File

The include file `<imsl.h>` is used in all of the examples in this manual. This file contains prototypes for all IMSL-defined functions; the spline structures, ***Imsl_f_ppoly***, ***Imsl_d_ppoly***, ***Imsl_f_spline***, and ***Imsl_d_spline***; enumerated data types, ***Imsl_quad***, ***Imsl_write_options***, ***Imsl_page_options***, ***Imsl_ode***, and ***Imsl_error***; and the IMSL-defined data types ***f_complex*** (which is the type ***float*** complex) and ***d_complex*** (which is the type ***double*** complex).

# Error Handling, Underflow, Overflow, and Document Examples

The functions in the IMSL C Math Library attempt to detect and report errors and invalid input. This error-handling capability provides automatic protection for the user without requiring the user to make any specific provisions for the treatment of error conditions. Errors are classified according to severity and are assigned a code number. By default, errors of moderate or higher severity result in messages being automatically printed by the function. Moreover, errors of highest severity cause program execution to stop. The severity level, as well as the general nature of the error, is designated by an "error type" with symbolic names `IMSL_FATAL`, `IMSL_WARNING`, etc. See the User Errors section in the "Reference Material" for further details.

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

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

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

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

# Memory Allocation for Output Arrays

Many functions return a pointer to an array containing the computed answers. By default, an array returned as the value of a C Numerical Library function is stored in memory allocated by that function. To release this space, use `imsl_free`. To return the array in memory allocated by the calling program, use the optional argument

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

In this way, the allocation of space for the computed answers can be made either by the user or internally by the function.

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

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

`IMSL_INVERSE`, *float* `**p_inva` (Output)

specify two mutually exclusive optional arguments. If the first option is chosen, the inverse of the matrix is stored in the user-provided array `inva`.

In the second option, *float* `**p_inva` refers to the address of a pointer to the inverse. The called function allocates memory for the array and sets `*p_inva` to point to this memory. Typically, *float* `*p_inva` is declared, `&p_inva` is used as an argument to this function. Use `imsl_free(p_inva)` to release the space.

# Printing Results

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

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

# Complex Arithmetic

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

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

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

# Missing Values

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

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

# Passing Data to User-Supplied Functions

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

## Example

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

float fcn_w_data(float x, void *data);

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

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

/*
* User function that accepts additional data in a (void*) pointer.
* This (void*) pointer can be cast to any type and dereferenced to
* get at any sort of data-type or structure that is needed.
* For example, to get at the data in this example
*  *((float*)data)   and usr_data[0] contains the value  5.0
*  *((float*)data+1) and usr_data[1] contains the value 10.0
*/
float fcn_w_data(float x, void *data)
{
    float *usr_data = (float*)data;
```

```
    return exp(x) - usr_data[0]*x + usr_data[1];
}
```

# Return Values from User-Supplied Functions

All values returned by user-supplied functions must be valid real numbers. It is the user’s responsibility to check that the values returned by a user-supplied function do not contain NaN, infinity, or negative infinity values.

In addition to the techniques described below, it is also possible to instruct the IMSL C Numerical Library to return control to the calling program in case an unrecoverable error occurs within a user-supplied function. See function `imsl_set_user_fcn_return_flag` for a description of this feature.

## Example

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

void fcn(int, int, float[], float[]);
int main()
{
    int m=3, n=1;
    float *result, fx[3];
    float xguess[]={1.0};
    result = imsl_f_nonlin_least_squares(fcn, m, n, IMSL_XGUESS,
        xguess, 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);
}

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++)
    {
        /*      check for x=0
        do not want to return infinity to nonlin_least_squares    */
        if (x[0] == 0.0) {
            f[i] = 10000.;
        } else {
            f[i] = t[i]/x[0] - y[i];
        }
    }
}
```

# Thread Safe Usage

The IMSL C Math Library is thread safe based on OpenMP. That means it can be safely called from a multi-threaded application if the calling program adheres to a few important guidelines. In particular, IMSL C Math Library's implementation of error handling and I/O must be understood.

## Error Handling

C Math Library's error handling in a multithreaded application behaves similarly to how it behaves in a single-threaded application. The major difference is that an error stack exists for each thread calling C Math Library functions. The result of separate error stacks for each thread is greater control of the error handler options for each thread. Each thread can set its own options for the C Math Library error handler using `imsl_error_options`. For an example of setting error handler options for separate threads, see Chapter 12, ***Utilities***, Example 3 of `imsl_error_options`.

## Routines that Produce Output

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

# OpenMP Usage

Thread safety of the IMSL C Numerical Library is based on OpenMP. Users of the IMSL C Numerical Library are also able to leverage shared-memory parallelism by means of native support for the OpenMP API specification within parts of the Library. Those parts are flagged by the OpenMP icon shown below.

OpenMP

Parallelism in OpenMP is implemented by means of threads. In the OpenMP programming model, it is assumed that memory is shared among threads, such as in multi-core machines. These threads are spawned by OpenMP in response to directives embedded in source code.

The Library's use of OpenMP is largely transparent to the user. Codes that have been enhanced with OpenMP directives will still work properly in serial execution environments. Error handling routines have been extended so that the most severe error during a parallel run will be returned to the user.

OpenMP is used by the Library in these main ways:

1. To implement thread safety within the C Numerical Library.
2. To speed up computationally intensive functions by exploiting data parallelism in their processing.
3. To parallelize the evaluation of user-supplied functions in routines that use them, e.g. in numerical integration routines.

In the last case, the user must explicitly signal to the Library that the user-supplied functions themselves are thread-safe, or by default the user's function(s) will not evaluate in parallel. The utility `imsl_omp_options` allows the user to assert that all routines passed to the library are thread-safe.

Thread safety implies that function(s) may be executed simultaneously by multiple threads and still function correctly. Requiring that user-supplied functions be thread-safe is crucial, because the different threads spawned by OpenMP may call user-supplied functions simultaneously, and/or in an arbitrary order, and/or with differing inputs. Care must therefore be taken to ensure that the parallelized algorithm acts in the same way as its serial "ancestor". Functions whose results depend on the order in which they are executed are not thread-safe and are thus not good candidates for parallelization; neither are functions which access and modify global data.

Specifications of the OpenMP standards are provided at (`http://openmp.org/wp/`).

# Vendor Supplied Libraries Usage

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


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

# C++ Usage

IMSL C Numerical Library functions can be used in both C and C++ applications. It is also possible to wrap library functions into C++ classes.

The function `imsl_f_int_fcn_sing` computes the integral of a user defined function. For C++ usage the user defined function is defined as a member function of the abstract class `IntFcnSingFunction` defined as follows.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>
class IntFcnSingFunction
{
public:
    virtual float f(float x) = 0;
};
```

The function `imsl_f_int_fcn_sing` is wrapped as the C++ class `IntFcnSing`. This implementation uses the optional argument, `IMSL_FCN_W_DATA`, to call `local_function` which in turn calls the method `f` to evaluate the user defined function. For simplicity, this implementation only wraps a single optional argument, `IMSL_MAX_SUBINTER`, the maximum number of subintervals. More could be included in a similar manner.

```
#include <imsl.h>

class IntFcnSing
{
public:
   int max_subinter;
   IntFcnSing();
   float integrate(IntFcnSingFunction *F, float a, float b);
};

static float local_function(float x, void *data)
{
   IntFcnSingFunction *F = (IntFcnSingFunction*)data;
   return F->f(x);
}

IntFcnSing::IntFcnSing()
{
   max_subinter = 500;
}

float IntFcnSing::integrate(IntFcnSingFunction *F, float a, float b)
{
   float  result;

   result = imsl_f_int_fcn_sing(NULL, a, b,
```

```
        IMSL_FCN_W_DATA, local_function, F,
        IMSL_MAX_SUBINTER, max_subinter,
        0);
    if (imsl_error_type() >= 3)
    {
        throw imsl_error_message();
    }
    return result;
}
```

To use this `IntFcnSing` the user defined function must be defined as the method `f` in a class that extends `IntFcnSingFunction`. The following class, `MyClass`, defines the function $f(x) = e^x - ax$, where $a$ is a parameter.

```
class MyClass : public IntFcnSingFunction
{
public:
    MyClass();
    float f(double x);
private:
    float my_parameter;
};

MyClass::MyClass()
{
    my_parameter = 5.0;
}

float MyClass::f(float x)
{
    return exp(x) - my_parameter*x;
}
```

The following is an example of the use of these classes. Since the C++ throws an exception on fatal or terminal IMSL errors, printing and stopping on these errors is turned off by a call to `imsl_error_options`. Also, since the user defined function is thread-safe, a call is made to `imsl_omp_options` to declare this. With this setting, the quadrature code will use OpenMP to evaluate the function in parallel. Both of these calls need be made once per run.

The second part of this example sets the maximum number of subintevals to 5, an unrealistically small number, to show the error handling.

```
int main()
{
    imsl_error_options(
        IMSL_SET_PRINT, IMSL_FATAL, 0,
        IMSL_SET_PRINT, IMSL_TERMINAL, 0,
        IMSL_SET_STOP, IMSL_FATAL, 0,
        IMSL_SET_STOP, IMSL_TERMINAL, 0,
        0);
```

```
    imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    IntFcnSing *intFcnSing = new IntFcnSing();
    MyClass *myClass = new MyClass();
    float x = intFcnSing->integrate(myClass, -1.0, 1.0);
    printf("Solution in [-1,+1]: %g\n", x);
    try {
        intFcnSing->max_subinter = 5;
        x = intFcnSing-> integrate (myClass, -100.0, 1000.0);
        printf("Solution in [-100,1000]: %g\n", x);
    } catch(char * exception) {
        printf("Exception raised: %s\n", exception);
    }
}
```

## Output

```
Integral over [-1,+1] = 2.3504
Exception raised: The maximum number of subintervals allowed "maxsub" = 5
has been reached. Increase "maxsub".
```

# Matrix Storage Modes

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

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

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

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

## General Mode

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

## Rectangular Mode

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

## Symmetric Mode

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

## Hermitian Mode

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

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

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

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

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

## Sparse Coordinate Storage Format

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

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

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

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

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

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

As an example consider the 6 × 6 matrix:

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

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

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| row | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 |
| col | 0 | 1 | 2 | 3 | 2 | 0 | 3 | 4 | 0 | 3 | 4 | 5 | 0 | 1 | 5 |
| val | 2 | 9 | −3 | −1 | 5 | −2 | −7 | −1 | −1 | −5 | 1 | −3 | −1 | −2 | 6 |

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

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| row | 5 | 4 | 3 | 0 | 5 | 1 | 2 | 1 | 4 | 3 | 1 | 4 | 3 | 5 | 4 |
| col | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 |
| val | −1 | −1 | −2 | 2 | −2 | 9 | 5 | −3 | −5 | −7 | −1 | 1 | −1 | 6 | −3 |

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

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

        b[0].row = b[0].col = 0;          b[0].val = 2.0;
        b[1].row = b[1].col = 1;          b[1].val = 9.0;
        b[2].row = 1; b[2].col = 2;       b[2].val = -3.0;
        b[3].row = 1; b[3].col = 3;       b[3].val = -1.0;
```

```
        b[4].row = b[4].col = 2;           b[4].val = 5.0;
        b[5].row = 3; b[5].col = 0;        b[5].val = -2.0;
        b[6].row = b[6].col = 3;           b[6].val = -7.0;
        b[7].row = 3; b[7].col = 4;        b[7].val = -1;
        b[8].row = 4; b[8].col = 0;        b[8].val = -1.0;
        b[9].row = 4; b[9].col = 3;        b[9].val = -5.0;
        b[10].row = b[10].col = 4;         b[10].val = 1.0;
        b[11].row = 4; b[11].col = 5;      b[11].val = -3.0;
        b[12].row = 5; b[12].col = 0;      b[12].val = -1.0;
        b[13].row = 5; b[13] = 1;          b[13].val = -2.0;
        b[14].row = b[14].col = 5;         b[14].val = 6.0;
}
```

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

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

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

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

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

As before, an equivalent form would be

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

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

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

```
                {2, 1, {1.0, 1.0}},
                {3, 2, {1.0, 1.0}}
            }
            Imsl_c_sparse_elem b[7];

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

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

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

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

## Band Storage Format

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

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

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

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

In band storage format, the data would be arranged as

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

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

As an example, consider the following tridiagonal matrix:

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

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

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

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

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

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

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

The following Hermitian example illustrates the procedure:

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

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

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

or equivalently

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

## Choosing Between Banded and Coordinate Forms

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

# Compressed Sparse Column (CSC) Format

Functions that accept data in coordinate format can also accept data stored in the format described in the Users' Guide for the Harwell-Boeing Sparse Matrix Collection (via optional argument `IMSL_CSC_FORMAT`). The scheme is column oriented, with each column held as a sparse vector, represented by a list of the row indices of the entries in an integer array ("`rowind`" below) and a list of the corresponding values in a separate ***float*** (***double***, ***f_complex***, ***d_complex***) array ("`values`" below). Data for each column are stored consecutively and the columns are stored in order. A third array ("`colptr`" below) indicates the location in array "`values`" in which to place the first nonzero value of each succeeding column of the original sparse matrix. So `colptr[i]` contains the index of the first free location in array "`values`" in which to place the values from the $i^{th}$ column of the original sparse matrix. In other words, `values[colptr[i]]` holds the first nonzero value of the `i`-th column of the original sparse matrix. Only entries in the lower triangle and diagonal are stored for symmetric and Hermitian matrices. All arrays are based at zero, which is in contrast to the Harwell-Boeing test suite's one-based arrays.

As in the Harwell-Boeing user guide (link above), the storage scheme is illustrated with the following example: The 5 × 5 matrix

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

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

| **Subscripts** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `Colptr` | 0 | 3 | 5 | 7 | 9 | 11 | | | | | |
| `Rowind` | 0 | 2 | 4 | 0 | 3 | 1 | 4 | 0 | 3 | 1 | 4 |
| `Values` | 1 | 2 | 5 | -3 | 4 | -2 | -5 | -1 | -4 | 3 | 6 |

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

```
void main()
{
    int i, j, k, n=5, nz, start, stop;
    int colptr[] = {0, 3, 5, 7, 9, 11};
    int rowind[] = {0, 2, 4, 0, 3, 1, 4, 0, 3, 1, 4};
    int values[] = {1.0, 2.0, 5.0, -3.0, 4.0, -2.0,
          -5.0, -1.0, -4.0, 3.0, 6.0};
    Imsl_d_sparse_elem a[11];
    k = 0;
    for (i=0; i<n; i++) {
        start = colptr[i];
        stop = colptr[i+1];
        for (j=start; j<stop; j++) {
            a[k].row = rowind[j];
```

```
                a[k].col = i;
                a[k++].val = values[j];
            }
        }
        nz =k;
    }
```

# Chapter 1 Linear Systems

## Functions

## Linear Least-squares with Full Matrices

# Usage Notes

## Solving Systems of Linear Equations

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

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

## Matrix Factorizations

In some applications, it is desirable to just factor the $n \times n$ matrix $A$ into a product of two triangular matrices. This can be done by calling the appropriate function for solving the system of linear equations $Ax = b$. Suppose that in addition to the solution $x$ of a linear system of equations $Ax = b$, the $LU$ factorization of $A$ is desired. Use the keyword `IMSL_FACTOR` in the function `imsl_f_lin_sol_gen` to obtain access to the factorization. If only the factorization is desired, use the keywords `IMSL_FACTOR_ONLY` and `IMSL_FACTOR`. For function `imsl_f_superlu`, use keyword `IMSL_RETURN_SPARSE_LU_FACTOR` in order to get the $LU$ factorization. If only the factorization is desired, then keywords `IMSL_RETURN_SPARSE_LU_FACTOR` and `IMSL_FACTOR_SOLVE` with value 1 are required.

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

## Matrix Inversions

The inverse of an $n \times n$ nonsingular matrix can be obtained by using the keyword `IMSL_INVERSE` in functions for solving systems of linear equations. The inverse of a matrix need not be computed if the purpose is to ***solve*** one or more systems of linear equations. Even with multiple right-hand sides, solving a system of linear equations by computing the inverse and performing matrix multiplication is usually more expensive than the method discussed in the next section.

## Multiple Right-Hand Sides

Consider the case where a system of linear equations has more than one right-hand side vector. It is most economical to find the solution vectors by first factoring the coefficient matrix *A* into products of triangular matrices. Then, the resulting triangular systems of linear equations are solved for each right-hand side. When *A* is a real general matrix, access to the *LU* factorization of *A* is computed by using the keywords `IMSL_FACTOR` and `IMSL_FACTOR_ONLY` in function `imsl_f_lin_sol_gen`. The solution $x_k$ for the *k*-th right-hand side vector $b_k$ is then found by two triangular solves, $Ly_k = b_k$ and $Ux_{kk} = y_k$. The keyword `IMSL_SOLVE_ONLY` in the function `imsl_f_lin_sol_gen` is used to solve each right-hand side. These arguments are found in other functions for solving systems of linear equations. For function `imsl_f_superlu`, use the keywords `IMSL_RETURN_SPARSE_LU_FACTOR` and `IMSL_FACTOR_SOLVE` with value 1 to get the *LU* factorization, and then keyword `IMSL_FACTOR_SOLVE` with value 2 to get the solution for different right-hand sides.

## Least-Squares Solutions and *QR* Factorizations

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

Least-squares problems with linear constraints and one right-hand side can be solved. These equations are

$$A_{m\times n}\, x = b,$$

subject to constraints and simple bounds

$$b_l \le Cx \le b_u$$

$$x_l \le x \le x_u$$

Here *A* is the coefficient matrix of the least-squares equations, *b* is the right-hand side, and *C* is the coefficient matrix of the constraints. The vectors $b_l$, $b_u$, $x_l$ and $x_u$ are the lower and upper bounds on the constraints and the variables. This general problem is solved with `imsl_f_lin_lsq_lin_constraints`.

For the special case of where there are only non-negative constraints, $x \ge 0$, solve the problem with `imsl_f_nonneg_least_squares`.

## Non-Negative Matrix Factorization

If the matrix $A_{m \times n} \geq 0$, factor it as a product of two matrices, $A_{m \times n} = F_{m \times k} G_{k \times n}$. The matrices *F* and *G* are both non-negative and $k \leq \min(m, n)$. The factors are computed so that the residual matrix $E = A - F G$ has a sum of squares norm that is minimized. There are normalizations of $F_{m \times k}$ and $G_{k \times n}$ described in the documentation of `imsl_f_nonneg_matrix_factorization`.

## Singular Value Decompositions and Generalized Inverses

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

## Ill-Conditioning and Singularity

An $m \times n$ matrix *A* is mathematically singular if there is an $x \neq 0$ such that $Ax = 0$. In this case, the system of linear equations $Ax = b$ does not have a unique solution. On the other hand, a matrix *A* is ***numerically*** singular if it is "close" to a mathematically singular matrix. Such problems are called ***ill-conditioned***. If the numerical results with an ill-conditioned problem are unacceptable, users can either use more accuracy if it is available (for type ***float*** accuracy switch to ***double***) or they can obtain an ***approximate*** solution to the system. One form of approximation can be obtained using the SVD of *A*: If $q = \min(m, n)$ and

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

then the approximate solution is given by the following:

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

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

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

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

# lin_sol_gen

more...

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

## Synopsis

*#include* `<imsl.h>`

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

The type *double* function is `imsl_d_lin_sol_gen`.

## Required Arguments

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

*float* `a[]` (Input)
Array of size *n* × *n* containing the matrix.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

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

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

`IMSL_TRANSPOSE`,

IMSL_RETURN_USER, *float* x[],

IMSL_FACTOR, *int* **p_pvt, *float* **p_factor,

IMSL_FACTOR_USER, *int* pvt[], *float* factor[],

IMSL_FAC_COL_DIM, *int* fac_col_dim,

IMSL_INVERSE, *float* **p_inva,

IMSL_INVERSE_USER, *float* inva[],

IMSL_INV_COL_DIM, *int* inva_col_dim,

IMSL_CONDITION, *float* *cond,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_INVERSE_ONLY,

0)

## Optional Arguments

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

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

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

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

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

*float* **p_factor (Output)
The address of a pointer to an array of size $n \times n$ containing the $LU$ factorization of $A$ with column pivoting. On return, the necessary space is allocated by imsl_f_lin_sol_gen. The lower-triangular part of this array contains information necessary to construct $L$, and the upper-triangular part contains $U$ (see Example 2). Typically, *float* *p_factor is declared, and &p_factor is used as an argument.

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

*int* pvt[] (Input/Output)
A user-allocated array of size $n$ containing the pivot sequence for the factorization.

*float* factor[] (Input/Output)
A user-allocated array of size $n \times n$ containing the $LU$ factorization of $A$. The strictly lower-triangular part of this array contains information necessary to construct $L$, and the upper-triangular part contains $U$ (see Example 2). If $A$ is not needed, factor and a can share the same storage.

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

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

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

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

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

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

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

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

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

## Description

The function `imsl_f_lin_sol_gen` solves a system of linear algebraic equations with a real coefficient matrix *A*. It first computes the *LU* factorization of *A* with partial pivoting such that $L^{-1}A = U$. Let *F* be the matrix `p_factor` returned by optional argument `IMSL_FACTOR`. The triangular matrix *U* is stored in the upper triangle of *F*. The strict lower triangle of *F* contains the information needed to reconstruct $L^{-1}$ using

$$L^{-1} = L_{n-1}P_{n-1} \ldots L_1P_1$$

The factors $P_i$ and $L_i$ are defined by partial pivoting. $P_i$ is the identity matrix with rows *i* and `p_pvt[i-1]` interchanged. $L_i$ is the identity matrix with $F_{ji}$, for $j = i + 1, \ldots, n$, inserted below the diagonal in column *i*.

The factorization efficiency is based on a technique of "loop unrolling and jamming" by Dr. Leonard J. Harding of the University of Michigan, Ann Arbor, Michigan. The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the $L_1$ condition number of *A* is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1/\epsilon$ (where $\epsilon$ is the machine precision), a warning message is issued. This indicates that very small changes in *A* may produce large changes in the solution *x*. The function `imsl_f_lin_sol_gen` fails if *U*, the upper triangular part of the factorization, has a zero diagonal element.

## Examples

### Example 1

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

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

```
#include <imsl.h>

int main()
{
   int         n = 3;
   float       *x;
   float       a[] = {1.0, 3.0, 3.0,
                      1.0, 3.0, 4.0,
                      1.0, 4.0, 3.0};
   float       b[] = {1.0, 4.0, -1.0};
                               /* Solve Ax = b for x */
   x = imsl_f_lin_sol_gen (n, a, b, 0);
                               /* Print x */
   imsl_f_write_matrix ("Solution, x, of Ax = b", 1, 3, x, 0);
```

```
}
```

### Output

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

## Example 2

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

```
#include <imsl.h>

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

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

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

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

### Output

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

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

```
2          -1          1          0
3          -1          0          1

Pivot sequence
  1  2  3
  1  3  3
```

### Reconstruction of $L^{-1}$ and *U* from `factor`:

$$L^{-1} = L_2 P_2 L_1 P_1$$

$P_i$ is the identity matrix with row *i* and row `pvt`[*i*-1] interchanged.

| `pvt` **= 1, 3, 3** | |
|---|---|
| $P_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ | row 1 and row `pvt`[0], or row 1, are interchanged, which is still the identity matrix. |
| $P_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ | row 2 and row `pvt[1]`, or row 3, are interchanged. |

$L_i$ is the identity matrix with $F_{ji}$, for $j = i + 1$, *n*, inserted below the diagonal in column *i*, where *F* is `factor`:

| $\text{factor} = \begin{bmatrix} 1 & 3 & 3 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$ | |
|---|---|
| $L_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$ | second and third elements of column 1 of `factor` are inserted below the diagonal in column 1. |
| $L_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ | third element of column 2 of `factor` is inserted below the diagonal in column 2. |

$$L^{-1} = L_2 P_2 L_1 P_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix}$$

*U* is the upper triangle of `factor`:

$$U = \begin{bmatrix} 1 & 3 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

## Example 3

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

```
#include <imsl.h>

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

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

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

                                   /* Print solution */

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

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

## Output

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

```
        -2          -2          3

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

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

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

## Example 4

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

```
#include <imsl.h>

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

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

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

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

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

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

}
```

### Output

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

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

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# lin_sol_gen (complex)

more...

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

## Synopsis

*#include* <imsl.h>

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

The type *d_complex* function is imsl_z_lin_sol_gen.

## Required Arguments

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_lin_sol_gen (*int* n, *f_complex* a[], *f_complex* b[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_TRANSPOSE,

IMSL_RETURN_USER, *f_complex* x[],

IMSL_FACTOR, *int* **p_pvt, *f_complex* **p_factor,

IMSL_FACTOR_USER, *int* pvt[], *f_complex* factor[],

IMSL_FAC_COL_DIM, *int* fac_col_dim,

IMSL_INVERSE, *f_complex* **p_inva,

IMSL_INVERSE_USER, *f_complex* inva[],

IMSL_INV_COL_DIM, *int* inva_col_dim,

IMSL_CONDITION, *float* *cond,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_INVERSE_ONLY,

0)

## Optional Arguments

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

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

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

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

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

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

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

*int* `pvt[]` (Input/Output)
A user-allocated array of size *n* containing the pivot sequence for the factorization.

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

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

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

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

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

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

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

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

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

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

# Description

The function `imsl_c_lin_sol_gen` solves a system of linear algebraic equations with a complex coefficient matrix *A*. It first computes the *LU* factorization of *A* with partial pivoting such that $L^{-1}A = U$. Let *F* be the matrix `p_factor` returned by optional argument `IMSL_FACTOR`. The triangular matrix *U* is stored in the upper triangle of *F*. The strict lower triangle of *F* contains the information needed to reconstruct
$L^{-1}$ using

$$L^{-1} = L_{n-1}P_{n-1} \ldots L_1P_1$$

The factors $P_i$ and $L_i$ are defined by partial pivoting. $P_i$ is the identity matrix with rows *i* and `p_pvt[i-1]` interchanged. $L_i$ is the identity matrix with $F_{ji}$, for $j = i + 1,\ldots, n$, inserted below the diagonal in column *i*.

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

# Examples

## Example 1

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

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

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

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

```
#include <imsl.h>

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

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

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

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

### Output

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

## Example 2

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

```
#include <imsl.h>

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

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

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

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

### Output

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

                              LU factors of A
                      1                      2                      3
```

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

### Example 3

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

```
#include <imsl.h>

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

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

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

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

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

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

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

#### Output

```
                        Solution, x, of Ax = b
```

```
                     1                       2                       3
(        1,        -1)  (        2,        4)  (        3,        -0)

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

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

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| IMSL_SINGULAR_MATRIX | The input matrix is singular. |

# lin_sol_posdef

more...

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

## Synopsis

*#include* `<imsl.h>`

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

The type *double* function is `imsl_d_lin_sol_posdef`.

## Required Arguments

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

*float* `a[]` (Input)
Array of size *n* × *n* containing the matrix.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

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

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

`IMSL_RETURN_USER`, *float* `x[]`,

IMSL_FACTOR, *float* **p_factor,

IMSL_FACTOR_USER, *float* factor[],

IMSL_FAC_COL_DIM, *int* fac_col_dim,

IMSL_INVERSE, *float* **p_inva,

IMSL_INVERSE_USER, *float* inva[],

IMSL_INV_COL_DIM, *int* inv_col_dim,

IMSL_CONDITION, *float* *cond,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_INVERSE_ONLY,

0)

## Optional Arguments

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

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

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

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

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

IMSL_INVERSE, *float* **p_inva (Output)

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

IMSL_INVERSE_USER, *float* inva[] (Output)

A user-allocated array of size $n \times n$ containing the inverse of $A$.

IMSL_INV_COL_DIM, *int* inva_col_dim (Input)

The column dimension of the array containing the inverse of $A$.
Default: inva_col_dim = $n$

IMSL_CONDITION, *float* *cond (Output)

A pointer to a scalar containing an estimate of the $L_1$ norm condition number of the matrix $A$. Do not use this option with IMSL_SOLVE_ONLY.

IMSL_FACTOR_ONLY

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

IMSL_SOLVE_ONLY

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

IMSL_INVERSE_ONLY

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

# Description

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

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

# Examples

## Example 1

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

$$x_1 - 3x_2 + 2x_3 = 27$$
$$-3x_1 + 10x_2 - 5x_3 = -78$$
$$2x_1 - 5x_2 + 6x_3 = 64$$

```
#include <imsl.h>

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

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

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

### Output

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

## Example 2

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

```
#include <imsl.h>

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

```
   float      b[] = {27.0, -78.0, 64.0};

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

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

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

### Output

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

Cholesky factor L, and trans(L), of A
           1          2          3
1          1         -3          2
2         -3          1          1
3          2          1          1
```

## Example 3

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

```
#include <imsl.h>

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

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

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

```
}
```

### Output

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

## Warning Errors

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

## Fatal Errors

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

# lin_sol_posdef (complex)

more...

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

## Synopsis

*#include* <imsl.h>

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

The type *d_complex* function is `imsl_z_lin_sol_posdef`.

## Required Arguments

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

*f_complex* `a[]` (Input)
Array of size *n* × *n* containing the matrix.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

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

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

`IMSL_RETURN_USER`, *f_complex* *x* `[]`, IMSL_FACTOR, *f_complex* **p_factor,

IMSL_FACTOR_USER, *f_complex* factor[],

IMSL_FAC_COL_DIM, *int* fac_col_dim,

IMSL_CONDITION, *float* *cond,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

0)

## Optional Arguments

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

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

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

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

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

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

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

`IMSL_SOLVE_ONLY`

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

## Description

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

## Examples

### Example 1

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

$$2x_1 + (-1 + i)x_2 = 1 + 5i$$
$$(-1 - i)x_1 + 4x_2 + (1 + 2i)x_3 = 12 - 6i$$
$$(1 - 2i)x_2 + 10x_3 + 4ix_4 = 1 - 16i$$
$$-4ix_3 + 6x_4 + (1 + i)x_5 = -3 - 3i$$
$$(1 - i)x_4 + 9x_5 = 25 + 16i$$

```
#include <imsl.h>

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

```
                    };

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

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

### Output

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

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

## Example 2

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

```
#include <imsl.h>

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

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

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

### Output

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

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


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

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

## Warning Errors

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

## Fatal Errors

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

# lin_sol_gen_band

more...

Solves a real general band system of linear equations, $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the *LU* factorization of *A* using partial pivoting, solving $A^T x = b$, or computing the solution of $Ax = b$ given the *LU* factorization of *A*.

## Synopsis

*#include* <imsl.h>

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

The type *double* function is `imsl_d_lin_sol_gen_band`.

## Required Arguments

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

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_lin_sol_gen_band (*int* n, *float* a[], *int* nlca, *int* nuca, *float* b[],

IMSL_TRANSPOSE,

IMSL_RETURN_USER, *float* x[],

IMSL_FACTOR, *int* **p_pvt, *float* **p_factor,

IMSL_FACTOR_USER, *int* pvt[], *float* factor[],

IMSL_CONDITION, *float* *condition,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_BLOCKING_FACTOR, *int* block_factor,

0)

## Optional Arguments

IMSL_TRANSPOSE

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

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length $n$ containing the solution $x$.

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

*int* **p_pvt (Input/Output)

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

*float* **p_factor (Input/Output)

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

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

*int* pvt[] (Input/Output)

A user-allocated array of size $n$ containing the pivot sequence for the factorization.

*float* factor[] (Input/Output)

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

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

IMSL_CONDITION, *float* *condition (Output)

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

IMSL_FACTOR_ONLY

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

IMSL_SOLVE_ONLY

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

IMSL_BLOCKING_FACTOR, *int* block_factor (Input)

The blocking factor. block_factor must be set no larger than 32.

Default: block_factor = 1

# Description

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

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

# Examples

## Example 1

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

$$2x_1 - x_2 = 3$$
$$-3x_1 + x_2 - 2x_3 = 1$$
$$-x_3 + 2x_4 = 11$$
$$2x_3 + x_4 = -2$$

```
#include <imsl.h>

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

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

   float a[] = {0.0, -1.0, -2.0, 2.0,
           2.0, 1.0, -1.0, 1.0,
           -3.0, 0.0, 2.0, 0.0};
   float b[] = {3.0, 1.0, 11.0, -2.0};

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

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

### Output

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

## Example 2

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

```
#include <imsl.h>

int main()
{
   int        n = 4;
```

```
    int         nlca = 1;
    int         nuca = 1;
    int         *pivot;
    f_complex *x;
    f_complex *factor;

    /* Note that a is in band storage mode */
    f_complex     a[] =
        {{0.0, 0.0}, {4.0, 0.0}, {-2.0, 2.0}, {-4.0, -1.0},
        {-2.0, -3.0}, {-0.5, 3.0}, {3.0, -3.0}, {1.0, -1.0},
        {6.0, 1.0}, {1.0, 1.0}, {0.0, 2.0}, {0.0, 0.0}};
    f_complex     b[] =
        {{-10.0, -5.0}, {9.5, 5.5}, {12.0, -12.0}, {0.0, 8.0}};

    /* Solve Ax = b and return LU */
    x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
        IMSL_FACTOR, &pivot, &factor,
        0);

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

    /* Use precomputed LU to solve ctrans(A)x = b */
    x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
        IMSL_FACTOR_USER, pivot, factor,
        IMSL_TRANSPOSE,
        0);

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

## Output

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

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

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# lin_sol_gen_band (complex)

more...

Solves a complex general band system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the *LU* factorization of *A* using partial pivoting, solving $A^H x = b$, or computing the solution of $Ax = b$ given the *LU* factorization of *A*.

## Synopsis

*#include* <imsl.h>

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

The type ***double*** function is `imsl_z_lin_sol_gen_band`.

## Required Arguments

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

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_lin_sol_gen_band (*int* n, *f_complex* a[], *int* nlca, *int* nuca, *f_complex* b[],

IMSL_TRANSPOSE,

IMSL_RETURN_USER, *f_complex* x[],

IMSL_FACTOR, *int* **p_pvt, *f_complex* **p_factor,

IMSL_FACTOR_USER, *int* pvt[], *f_complex* factor[],

IMSL_CONDITION, *float* *condition,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

0)

## Optional Arguments

IMSL_TRANSPOSE

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

IMSL_RETURN_USER, *f_complex* x[] (Output)

A user-allocated array of length *n* containing the solution *x*.

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

*int* **p_pvt (Input/Output)

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

*f_complex* **p_factor (Input/Output)

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

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

*int* pvt[] (Input/Output)

A user-allocated array of size *n* containing the pivot sequence for the factorization.

*f_complex* `factor[]` (Input/Output)

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

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

`IMSL_CONDITION`, *float* `*condition` (Output)

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

`IMSL_FACTOR_ONLY`

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

`IMSL_SOLVE_ONLY`

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

# Description

The function `imsl_c_lin_sol_gen_band` solves a system of linear algebraic equations with a complex band matrix *A*. It first computes the *LU* factorization of *A* using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same $L_\infty$ norm. The factorization fails if *U* has a zero diagonal element. This can occur only if *A* is singular or very close to a singular matrix.

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

# Examples

## Example 1

The following linear system is solved:

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

```
#include <imsl.h>

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

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

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

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

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

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

### Output

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

## Example 2

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

```
#include <imsl.h>

int main()
{
   int            n = 4;
   int            nlca = 1;
   int            nuca = 1;
```

```
    int          *pivot;
    f_complex    *x;
    f_complex    *factor;

    /* Note that a is in band storage mode */
    f_complex     a[] =
        {{0.0, 0.0}, {4.0, 0.0}, {-2.0, 2.0}, {-4.0, -1.0},
        {-2.0, -3.0}, {-0.5, 3.0}, {3.0, -3.0}, {1.0, -1.0},
        {6.0, 1.0}, {1.0, 1.0}, {0.0, 2.0}, {0.0, 0.0}};
    f_complex     b[] =
        {{-10.0, -5.0}, {9.5, 5.5}, {12.0, -12.0}, {0.0, 8.0}};

    /* Solve Ax = b and return LU */
    x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
        IMSL_FACTOR, &pivot, &factor,
        0);
    imsl_c_write_matrix ("solution of Ax = b", n, 1, x,
        0);
    imsl_free (x);

    /* Use precomputed LU to solve ctrans(A)x = b */
    x = imsl_c_lin_sol_gen_band (n, a, nlca, nuca, b,
        IMSL_FACTOR_USER, pivot, factor,
        IMSL_TRANSPOSE,
        0);
    imsl_c_write_matrix ("solution of ctrans(A)x = b", n, 1, x,
        0);
}
```

## Output

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

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

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# lin_sol_posdef_band

more...

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

## Synopsis

*#include* <imsl.h>

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

The type *double* function is imsl_d_lin_sol_posdef_band.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

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

IMSL_RETURN_USER, *float* x[],

IMSL_FACTOR, *float* **p_factor,

IMSL_FACTOR_USER, *float* factor[],

IMSL_CONDITION, *float* *cond,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length *n* containing the solution *x*.

IMSL_FACTOR, *float* **p_factor (Output)

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

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

A user-allocated array of size (*ncoda* + 1) × *n* containing the $LL^T$ factorization of *A* in band symmetric form. If *A* is not needed, factor and a can share the same storage. These parameters are “Input” if IMSL_SOLVE is specified. They are “Output” otherwise.

IMSL_CONDITION, *float* *cond (Output)

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

IMSL_FACTOR_ONLY

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

IMSL_SOLVE_ONLY

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

## Description

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

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

The function `imsl_f_lin_sol_posdef_band` fails if any submatrix of *R* is not positive definite or if *R* has a zero diagonal element. These errors occur only if *A* is very close to a singular matrix or to a matrix which is not positive definite.

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

## Example 1

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

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

```
#include <imsl.h>

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

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

   float       a[] = {0.0, 0.0, -1.0, 1.0,
                      0.0, 0.0, 2.0, -1.0,
                      2.0, 4.0, 7.0, 3.0};
   float       b[] = {6.0, -11.0, -11.0, 19.0};

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

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

## Output

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

## Example 2

This example solves the same problem $Ax = b$ given in the first example. The solution is returned in user-allocated space and an estimate of $\kappa_1(A)$ is computed. Additionally, the $R^T R$ factorization is returned. Then, knowing that $\kappa_1(A) = ||A|| \ ||A^{-1}||$, the condition number is computed directly and compared to the estimate from Higham's method.

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

int main()
{
   int        n = 4;
   int        ncoda = 2;
   float      a[] =
       {0.0, 0.0, -1.0, 1.0,
        0.0, 0.0, 2.0, -1.0,
        2.0, 4.0, 7.0, 3.0};
   float      b[] = {6.0, -11.0, -11.0, 19.0};
   float      x[4];
   float      e_i[4];
   float     *factor;
   float      condition;
   float      column_norm;
   float      inverse_norm;
   int        i;
   int        j;

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

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

   /* find one norm of inverse */
   inverse_norm = 0.0;

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

```
        e_i[i] = 1.0;

        /* determine one norm of each column of inverse */
        imsl_f_lin_sol_posdef_band (n, a, ncoda, e_i,
            IMSL_FACTOR_USER, factor,
            IMSL_SOLVE_ONLY,
            IMSL_RETURN_USER, x,
            0);

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

        /* the max of the column norms is the norm of
        inv(A) */
        if (inverse_norm < column_norm)
            inverse_norm = column_norm;
    }

    /* by observation, one norm of A is 11 */
    printf ("\nHigham's condition estimate = %f\n", condition);
    printf ("Direct condition estimate  = %f\n",
        11.0*inverse_norm);
}
```

# Output

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

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

# Warning Errors

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

# Fatal Errors

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

# lin_sol_posdef_band (complex)

more...

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

## Synopsis

*#include* `<imsl.h>`

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

The type *double* function is `imsl_z_lin_sol_posdef_band`.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*f_complex* *imsl_c_lin_sol_posdef_band (*int* n, *f_complex* a[], *int* ncoda, *f_complex* b[],

IMSL_RETURN_USER, *f_complex* x[],

IMSL_FACTOR, *f_complex* **p_factor,

IMSL_FACTOR_USER, *f_complex* factor[],

IMSL_CONDITION, *float* *condition,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

0)

## Optional Arguments

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

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

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

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

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

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

## Description

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

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

The function `imsl_c_lin_sol_posdef_band` fails if any submatrix of *R* is not positive definite or if *R* has a zero diagonal element. These errors occur only if *A* is very close to a singular matrix or to a matrix which is not positive definite.

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

## Examples

### Example 1

Solve a linear system $Ax = b$ where

$$A = \begin{bmatrix} 2 & -1+i & 0 & 0 & 0 \\ -1-i & 4 & 1+2i & 0 & 0 \\ 0 & 1-2i & 10 & 4i & 0 \\ 0 & 0 & -4i & 6 & 1+i \\ 0 & 0 & 0 & 1-i & 9 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1+5i \\ 12-6i \\ 1-16i \\ -3-3i \\ 25+16i \end{bmatrix}$$

```
#include <imsl.h>

int main()
{
   int        n = 5;
   int        ncoda = 1;
   f_complex *x;

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

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

```
                    {25.0, 16.0}};

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

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

### Output

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

## Example 2

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

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

int main()
{
   int         n = 5, ncoda = 1, i, j;

   /* Note that a is in band storage mode */
   f_complex  a[] =
       {{0.0, 0.0}, {-1.0, 1.0}, {1.0, 2.0}, {0.0, 4.0},
        {1.0, 1.0},
        {2.0, 0.0}, {4.0, 0.0}, {10.0, 0.0}, {6.0, 0.0},
        {9.0, 0.0}};
   f_complex  b[] =
        {{1.0, 5.0}, {12.0, -6.0}, {1.0, -16.0},{-3.0, -3.0},
         {25.0, 16.0}};
   f_complex  x[5], e_i[5], *factor;
   float      condition, column_norm, inverse_norm;

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

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

   /* Find one norm of inverse */
   inverse_norm = 0.0;

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

       /* Determine one norm of each column of inverse */
       imsl_c_lin_sol_posdef_band (n, a, ncoda, e_i,
           IMSL_FACTOR_USER, factor,
           IMSL_SOLVE_ONLY,
           IMSL_RETURN_USER, x,
           0);

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

       /* The max of the column norms is the norm of inv(A) */
       if (inverse_norm < column_norm)
           inverse_norm = column_norm;
   }

   /* By observation, one norm of A is 14+sqrt(5) */
   printf ("\nHigham’s condition estimate = %7.4f\n", condition);
   printf ("Direct condition estimate  = %7.4f\n",
       (14.0+sqrt(5.0))*inverse_norm);
}
```

## Output

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

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

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

## Warning Errors

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

## Fatal Errors

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

# lin_sol_gen_coordinate

more...

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

## Synopsis

*#include* `<imsl.h>`

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

The type *double* function is `imsl_d_lin_sol_gen_coordinate`.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

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

`IMSL_RETURN_SPARSE_LU_FACTOR`, *Imsl_f_sparse_lu_factor* `*lu_factor`, IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_sparse_lu_factor* *lu_factor,

IMSL_FREE_SPARSE_LU_FACTOR,

IMSL_RETURN_SPARSE_LU_IN_COORD, *Imsl_f_sparse_elem* **lu_coordinate, *int* **row_pivots, *int* **col_pivots,

IMSL_SUPPLY_SPARSE_LU_IN_COORD, *int* nzlu, *Imsl_f_sparse_elem* *lu_coordinate, *int* *row_pivots, *int* *col_pivots,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_RETURN_USER, *float* x[],

IMSL_TRANSPOSE,

IMSL_CONDITION, *float* *condition,

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method,

IMSL_NUMBER_OF_SEARCH_ROWS, *int* num_search_row,

IMSL_ITERATIVE_REFINEMENT,

IMSL_DROP_TOLERANCE, *float* tolerance,

IMSL_HYBRID_FACTORIZATION, *float* density, *int* order_bound,

IMSL_STABILITY_FACTOR, *float* s_factor,

IMSL_GROWTH_FACTOR_LIMIT, *float* gf_limit,

IMSL_GROWTH_FACTOR, *float* *gf,

IMSL_SMALLEST_PIVOT, *float* *small_pivot

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values,

IMSL_MEMORY_BLOCK_SIZE, *int* block_size,

0)

## Optional Arguments

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

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_sparse_lu_factor* *lu_factor (Input)

The address of a structure of type *Imsl_f_sparse_lu_factor*. This structure contains the *LU* factorization of the input matrix computed by imsl_f_lin_sol_gen_coordinate with the IMSL_RETURN_SPARSE_LU_FACTOR option.

IMSL_FREE_SPARSE_LU_FACTOR,

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

IMSL_RETURN_SPARSE_LU_IN_COORD, *Imsl_f_sparse_elem* **lu_coordinate, *int* **row_pivots, *int* **col_pivots (Output)

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

IMSL_SUPPLY_SPARSE_LU_IN_COORD, *int* nzlu, *Imsl_f_sparse_elem* *lu_coordinate, *int* *row_pivots, *int* *col_pivots (Input)

Supply the *LU* factorization in coordinate form. See IMSL_RETURN_SPARSE_LU_IN_COORD for a description.

IMSL_FACTOR_ONLY,

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

IMSL_SOLVE_ONLY,

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

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length *n* containing the solution *x*.

IMSL_TRANSPOSE,

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

IMSL_CONDITION, *float* *condition,

Estimate the $L_1$ condition number of *A* and return in the variable condition.

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method (Input)

Select the pivoting strategy by setting method to one of the following: IMSL_ROW_MARKOWITZ, IMSL_COLUMN_MARKOWITZ, or IMSL_SYMMETRIC_MARKOWITZ.
Default: IMSL_SYMMETRIC_MARKOWITZ.

`IMSL_NUMBER_OF_SEARCH_ROWS`, *int* `num_search_row` (Input)

The number of rows which have the least number of nonzero elements that will be searched for a pivot element.

Default: `num_search_row` = 3.

`IMSL_ITERATIVE_REFINEMENT`,

Select this option if iterative refinement is desired.

`IMSL_DROP_TOLERANCE`, *float* `tolerance` (Input)

Possible fill-in is checked against tolerance. If the absolute value of the new element is less than `tolerance`, it will be discarded.

Default: `tolerance` = 0.0.

`IMSL_HYBRID_FACTORIZATION`, *float* `density`, *int* `order_bound`,

Enable the function to switch to a dense factorization method when the density of the active submatrix reaches 0.0 ≤ `density` ≤ 1.0 and the order of the active submatrix is less than or equal to `order_bound`.

`IMSL_STABILITY_FACTOR`, *float* `s_factor` (Input)

The absolute value of the pivot element must be bigger than the largest element in absolute value in its row divided by `s_factor`.

Default: `s_factor` = 10.0.

`IMSL_GROWTH_FACTOR_LIMIT`, *float* `gf_limit` (Input)

The computation stops if the growth factor exceeds `gf_limit`.

Default: `gf_limit` = 1.0e16.

`IMSL_GROWTH_FACTOR`, *float* `*gf` (Output)

Argument `gf` is calculated as the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in *A*.

`IMSL_SMALLEST_PIVOT`, *float* `*small_pivot` (Output)

A pointer to the value of the pivot element of smallest magnitude that occurred during the factorization.

`IMSL_NUM_NONZEROS_IN_FACTOR`, *int* `*num_nonzeros` (Output)

A pointer to a scalar containing the total number of nonzeros in the factor.

`IMSL_CSC_FORMAT`, *int* `*col_ptr`, *int* `*row_ind`, *float* `*values` (Input)

Accept the coefficient matrix in Compressed Sparse Column (CSC) Format. See the main "Introduction" chapter of this manual for a discussion of this storage scheme.

`IMSL_MEMORY_BLOCK_SIZE`, *int* `blocksize` (Input)

If space must be allocated for fill-in, allocate enough space for `blocksize` new nonzero elements.

Default: `blocksize` = `nz`.

## Description

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

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

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

The option `IMSL_CONDITION` is used to calculate and return an estimation of the $L_1$ condition number of $A$. The algorithm used is due to Higham. Specification of `IMSL_CONDITION` causes a complete $L$ to be computed and stored, even if a one off problem is being solved. This is due to the fact that Higham's method requires solution to problems of the form $Az = r$ and $A^Tz = r$.

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

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

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

# Examples

## Example 1

As an example, consider the following matrix:

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

Let $x^T = (1, 2, 3, 4, 5, 6)$ so that $Ax = (10, 7, 45, 33, -34, 31)^T$. The number of nonzeros in $A$ is `nz` = 15.

```
#include <imsl.h>

int main()
{
   Imsl_f_sparse_elem a[] =
       {0, 0, 10.0,
        1, 1, 10.0,
        1, 2, -3.0,
        1, 3, -1.0,
        2, 2, 15.0,
        3, 0, -2.0,
        3, 3, 10.0,
        3, 4, -1.0,
        4, 0, -1.0,
        4, 3, -5.0,
        4, 4, 1.0,
        4, 5, -3.0,
        5, 0, -1.0,
        5, 1, -2.0,
        5, 5, 6.0};

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

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

```
    imsl_f_write_matrix ("solution", 1, n, x,
        0);
    imsl_free (x);
}
```

### Output

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

## Example 2

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

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

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

    /* Get the coefficient matrix */
    a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

    /* Set two different predetermined solutions */
    mod_five = (float*) malloc (n*sizeof(*mod_five));
    mod_ten = (float*) malloc (n*sizeof(*mod_ten));

    for (i=0; i<n; i++) {
```

```
        mod_five[i] = (float) (i % 5);
        mod_ten[i] = (float) (i % 10);
    }

    /* Choose b so that x will approximate mod_five */
    b = (float *) imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_five,
        0);

    /* Time the factor/solve */
    time_factor_solve = imsl_ctime();

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

    time_factor_solve = imsl_ctime() - time_factor_solve;

    /* Compute max absolute error */
    error_factor_solve = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);

    free (mod_five);
    imsl_free (b);
    imsl_free (x);

    /* Get new right hand side -- b = A * mod_ten */
    b = (float *) imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_ten,
        0);

    /* Use the previously computed factorization
    to solve Ax = b */
    time_solve = imsl_ctime();

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

    time_solve = imsl_ctime() - time_solve;

    error_solve = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_ten,
        IMSL_INF_NORM, &index,
        0);
```

```
    free (mod_ten);
    imsl_free (b);
    imsl_free (x);

    /* Print errors and ratio of execution times */
    printf ("absolute error (factor/solve) = %e\n",
        error_factor_solve);
    printf ("absolute error (solve)       = %e\n", error_solve);
    printf ("time_solve/time_factor_solve = %f\n",
        time_solve/time_factor_solve);
}
```

### Output

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

## Example 3

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

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

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

```
/* Get the coefficient matrix */
a = imsl_f_generate_test_coordinate (n, c, &nz, 0);
for (i=0; i<nz; i++) a[i].val *= 0.05;

/* Set a predetermined solution */
mod_five = (float*) malloc (n*sizeof(*mod_five));
for (i=0; i<n; i++)
    mod_five[i] = (float) (i % 5);

/* Choose b so that x will approximate mod_five */
b = imsl_f_mat_mul_rect_coordinate ("A*x",
    IMSL_A_MATRIX, n, n, nz, a,
    IMSL_X_VECTOR, n, mod_five,
    0);

/* Time the factor/solve */
time_zero_drop_tol = imsl_ctime();

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

time_zero_drop_tol = imsl_ctime() - time_zero_drop_tol;

/* Compute max abolute error */
error_zero_drop_tol = imsl_f_vector_norm (n, x,
    IMSL_SECOND_VECTOR, mod_five,
    IMSL_INF_NORM, &index,
    0);

imsl_free (x);

/* Solve the same problem, with drop
tolerance = 0.005 */
time_nonzero_drop_tol = imsl_ctime();

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

time_nonzero_drop_tol = imsl_ctime() - time_nonzero_drop_tol;

/* Compute max abolute error */
error_nonzero_drop_tol = imsl_f_vector_norm (n, x,
    IMSL_SECOND_VECTOR, mod_five,
    IMSL_INF_NORM, &index,
    0);

imsl_free (x);
```

```
    /* Solve the same problem with IR, use last
    factorization */
    time_nonzero_drop_tol_IR = imsl_ctime();

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

    time_nonzero_drop_tol_IR = imsl_ctime() - time_nonzero_drop_tol_IR;

    /* Compute max abolute error */
    error_nonzero_drop_tol_IR = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);

    imsl_free (x);
    imsl_free (b);

    /* Print errors and ratio of execution times */
    printf ("drop tolerance = 0.0\n");
    printf ("\tabsolute error = %e\n", error_zero_drop_tol);
    printf ("\tfillin         = %d\n\n", nz_zero_drop_tol);
    printf ("drop tolerance = 0.005\n");
    printf ("\tabsolute error = %e\n", error_nonzero_drop_tol);
    printf ("\tfillin         = %d\n\n", nz_nonzero_drop_tol);
    printf ("drop tolerance = 0.005 (with IR)\n");
    printf ("\tabsolute error = %e\n", error_nonzero_drop_tol_IR);
    printf ("\tfillin         = %d\n\n", nz_nonzero_drop_tol);
    printf ("time_nonzero_drop_tol/time_zero_drop_tol = %f\n",
        time_nonzero_drop_tol/time_zero_drop_tol);
    printf ("time_nonzero_drop_tol_IR/time_zero_drop_tol = %f\n",
        time_nonzero_drop_tol_IR/time_zero_drop_tol);
}
```

### Output

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

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

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

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

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

# lin_sol_gen_coordinate (complex)

more...

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

## Synopsis

*#include* `<imsl.h>`

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

The type *double* function is `imsl_z_lin_sol_gen_coordinate`.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*f_complex* *imsl_c_lin_sol_gen_coordinate (*int* n, *int* nz, *Imsl_c_sparse_elem* *a,
*f_complex* *b,

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_c_sparse_lu_factor* *lu_factor,

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_c_sparse_lu_factor* *lu_factor,

IMSL_FREE_SPARSE_LU_FACTOR,

IMSL_RETURN_SPARSE_LU_IN_COORD, *Imsl_c_sparse_elem* **lu_coordinate,
*int* **row_pivots, *int* **col_pivots,

IMSL_SUPPLY_SPARSE_LU_IN_COORD, *int* nzlu, *Imsl_c_sparse_elem* *lu_coordinate,
*int* *row_pivots, *int* *col_pivots,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_RETURN_USER, *f_complex* x[],

IMSL_TRANSPOSE,

IMSL_CONDITION, *float* *condition,

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method,

IMSL_NUMBER_OF_SEARCH_ROWS, *int* num_search_row,

IMSL_ITERATIVE_REFINEMENT,

IMSL_DROP_TOLERANCE, *float* tolerance,

IMSL_HYBRID_FACTORIZATION, *float* density, *int* order_bound,

IMSL_GROWTH_FACTOR_LIMIT, *float* gf_limit,

IMSL_GROWTH_FACTOR, *float* *gf,

IMSL_SMALLEST_PIVOT, *float* *small_pivot

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *f_complex* *values,

IMSL_MEMORY_BLOCK_SIZE, *int* block_size,

0)

## Optional Arguments

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

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

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

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

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

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

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

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

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

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

IMSL_PIVOTING_STRATEGY, *Imsl_pivot* method (Input)

Select the pivoting strategy by setting method to one of the following: IMSL_ROW_MARKOWITZ, IMSL_COLUMN_MARKOWITZ, or IMSL_SYMMETRIC_MARKOWITZ.

Default: IMSL_SYMMETRIC_MARKOWITZ.

IMSL_NUMBER_OF_SEARCH_ROWS, *int* num_search_row (Input)

The number of rows which have the least number of nonzero elements that will be searched for a pivot element.

Default: num_search_row = 3

IMSL_ITERATIVE_REFINEMENT,

Select this option if iterative refinement is desired.

IMSL_DROP_TOLERANCE, *float* tolerance (Input)

Possible fill-in is checked against tolerance. If the absolute value of the new element is less than tolerance, it will be discarded.

Default: tolerance = 0.0

IMSL_HYBRID_FACTORIZATION, *float* density, *int* order_bound, (Input)

Enable the code to switch to a dense factorization method when the density of the active submatrix reaches 0.0 ≤ density ≤ 1.0 and the order of the active submatrix is less than or equal to order_bound.

IMSL_GROWTH_FACTOR_LIMIT, *float* gf_limit (Input)

The computation stops if the growth factor exceeds gf_limit.

Default: gf_limit = 1.e16

IMSL_GROWTH_FACTOR, *float* *gf (Output)

gf is calculated as the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in *A*.

IMSL_SMALLEST_PIVOT, *float* *small_pivot (Output)

A pointer to the value of the pivot element of smallest magnitude.

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros (Output)

A pointer to a scalar containing the total number of nonzeros in the factor.

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *f_complex* *values (Input)

Accept the coefficient matrix in Compressed Sparse Column (CSC) Format. See the main ***Introduction*** chapter at the beginning of this manual for a discussion of this storage scheme.

IMSL_MEMORY_BLOCK_SIZE, *int* blocksize (Input)

If space must be allocated for fill-in, allocate enough space for blocksize new nonzero elements.

Default: blocksize = nz

## Description

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

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

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

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

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

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

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

# Examples

## Example 1

As an example, consider the following matrix:

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

Let

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

so that

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

```
#include <imsl.h>

int main()
{
   static Imsl_c_sparse_elem a[] =
       {0, 0, {10.0, 7.0},
        1, 1, {3.0, 2.0},
        1, 2, {-3.0, 0.0},
        1, 3, {-1.0, 2.0},
        2, 2, {4.0, 2.0},
        3, 0, {-2.0, -4.0},
        3, 3, {1.0, 6.0},
        3, 4, {-1.0, 3.0},
        4, 0, {-5.0, 4.0},
        4, 3, {-5.0, 0.0},
        4, 4, {12.0, 2.0},
        4, 5, {-7.0, 7.0},
        5, 0, {-1.0, 12.0},
        5, 1, {-2.0, 8.0},
        5, 5, {3.0, 7.0}};

   static f_complex b[] =
       {{3.0, 17.0}, {-19.0, 5.0}, {6.0, 18.0},
        {-38.0, 32.0}, {-63.0, 49.0}, {-57.0, 83.0}};
```

```
   int        n = 6;
   int        nz = 15;
   f_complex *x;

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

   imsl_c_write_matrix ("solution", n, 1, x,
       0);
   imsl_free (x);
}
```

### Output

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

## Example 2

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

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

int main()
{
   Imsl_c_sparse_elem     *a;
   Imsl_c_sparse_lu_factor lu_factor;
   f_complex              *b, *x, *mod_five, *mod_ten;
   float                   error_factor_solve, error_solve;
   double                  time_factor_solve, time_solve;
   int                     n = 1000, c = 10, i, nz, index;

   /* Get the coefficient matrix */
   a = imsl_c_generate_test_coordinate (n, c, &nz, 0);

   /* Set two different predetermined solutions */
   mod_five = (f_complex*) malloc (n*sizeof(*mod_five));
   mod_ten = (f_complex*) malloc (n*sizeof(*mod_ten));
```

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

/* Choose b so that x will approximate mod_five */
b = imsl_c_mat_mul_rect_coordinate ("A*x",
    IMSL_A_MATRIX, n, n, nz, a,
    IMSL_X_VECTOR, n, mod_five,
    0);

/* Time the factor/solve */
time_factor_solve = imsl_ctime();

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

time_factor_solve = imsl_ctime() - time_factor_solve;

/* Compute max abolute error */
error_factor_solve = imsl_c_vector_norm (n, x,
    IMSL_SECOND_VECTOR, mod_five,
    IMSL_INF_NORM, &index,
    0);

imsl_free (b);
imsl_free (x);

/* Get new right hand side -- b = A * mod_ten */
b = imsl_c_mat_mul_rect_coordinate ("A*x",
    IMSL_A_MATRIX, n, n, nz, a,
    IMSL_X_VECTOR, n, mod_ten,
    0);

/* Use the previously computed factorization to solve Ax = b */
time_solve = imsl_ctime();

x = imsl_c_lin_sol_gen_coordinate (n, nz, a, b,
    IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
    IMSL_SOLVE_ONLY,
    0);

time_solve = imsl_ctime() - time_solve;

error_solve = imsl_c_vector_norm (n, x,
    IMSL_SECOND_VECTOR, mod_ten,
    IMSL_INF_NORM, &index,
    0);
```

```
    imsl_free (b);
    imsl_free (x);

    /* Print errors and ratio of execution times */
    printf ("absolute error (factor/solve) = %e\n",
        error_factor_solve);
    printf ("absolute error (solve)        = %e\n", error_solve);
    printf ("time_solve/time_factor_solve = %f\n",
        time_solve/time_factor_solve);
}
```

### Output

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

# superlu

more...

Computes the *LU* factorization of a general sparse matrix by a column method and solves the real sparse linear system of equations $Ax = b$.

## Synopsis

*#include* <imsl.h>

*float* `*imsl_f_superlu` (*int* `n`, *int* `nz`, *Imsl_f_sparse_elem* `a[]`, *float* `b[]`, …, 0)

*void* `imsl_f_superlu_factor_free` *(Imsl_f_super_lu_factor* `*factor`)

The type ***double*** functions are `imsl_d_superlu` and `imsl_d_superlu_factor_free`.

## Required Arguments

*int* `n` (Input)
The order of the input matrix.

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

*Imsl_f_sparse_elem* `a[]` (Input)
Array of length `nz` containing the location and value of each nonzero entry in the matrix. See the explanation of the ***Imsl_f_sparse_elem*** structure in the section Matrix Storage Modes in the “Introduction” chapter of this manual.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_superlu (*int* n, *int* nz, *Imsl_f_sparse_elem* a[], *float* b[],

IMSL_EQUILIBRATE, *int* equilibrate,

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method,

IMSL_COLPERM_VECTOR, *int* permc[],

IMSL_TRANSPOSE, *int* transpose,

IMSL_ITERATIVE_REFINEMENT, *int* refine,

IMSL_FACTOR_SOLVE, *int* factsol,

IMSL_DIAG_PIVOT_THRESH, *double* diag_pivot_thresh,

IMSL_SYMMETRIC_MODE, *int* symm_mode,

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[],

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind [], *float* HB_values[],

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *float* HB_values[],

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_super_lu_factor* lu_factor_supplied,

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_f_super_lu_factor* *lu_factor_returned,

IMSL_CONDITION, *float* *condition,

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth,

IMSL_FORWARD_ERROR_BOUND, *float* *ferr,

IMSL_BACKWARD_ERROR, *float* *berr,

IMSL_RETURN_USER, *float* x[],

0)

# Optional Arguments

IMSL_EQUILIBRATE, *int* equilibrate (Input)
Specifies if the input matrix *A* should be equilibrated before factorization.

| equilibrate | Description |
|---|---|
| 0 | Do not equilibrate *A* before factorization |
| 1 | Equilibrate *A* before factorization. |

Default: equilibrate = 0

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method (Input)

The column ordering method used to preserve sparsity prior to the factorization process. Select the ordering method by setting method to one of the following:

| **method** | **Description** |
|---|---|
| IMSL_NATURAL | Natural ordering, i.e.the column ordering of the input matrix. |
| IMSL_MMD_ATA | Minimum degree ordering on the structure of $A^TA$. |
| IMSL_MMD_AT_PLUS_A | Minimum degree ordering on the structure of $A^T + A$. |
| IMSL_COLAMD | Column approximate minimum degree ordering. |
| IMSL_PERMC | Use ordering given in permutation vector permc, which is input by the user through optional argument IMSL_COLPERM_VECTOR. Vector permc is a permutation of the numbers 0,1,...,n-1. |

Default: method = IMSL_COLAMD

IMSL_COLPERM_VECTOR, *int* permc[] (Input)

Array of length n which defines the permutation matrix $P_c$ before postordering. This argument is required if IMSL_COLUMN_ORDERING_METHOD with method = IMSL_PERMC is used. Otherwise, it is ignored.

IMSL_TRANSPOSE, *int* transpose (Input)

Indicates if the transposed problem $A^Tx = b$ is to be solved. This option can be used in conjunction with either of the options that supply the factorization.

| **transpose** | **Description** |
|---|---|
| 0 | Solve $Ax = b$. |
| 1 | Solve $A^Tx = b$. |

Default: transpose = 0

IMSL_ITERATIVE_REFINEMENT, *int* refine (Input)

Indicates if iterative refinement is desired.

| **refine** | **Description** |
|---|---|
| 0 | No iterative refinement. |
| 1 | Do iterative refinement. |

Default: `refine` = 1

`IMSL_FACTOR_SOLVE`, *int* `factsol` (Input)

Indicates if the *LU* factorization, the solution of a linear system or both are to be computed.

| `factsol` | **Description** |
|---|---|
| 0 | Compute the *LU* factorization of the input matrix *A* and solve the system $Ax = b$. |
| 1 | Only compute the *LU* factorization of the input matrix and return.<br>The *LU* factorization is returned via optional argument `IMSL_RETURN_SPARSE_LU_FACTOR`.<br>Input argument `b` is ignored. |
| 2 | Only solve $Ax = b$ given the *LU* factorization of $A$.<br>The *LU* factorization of $A$ must be supplied via optional argument `IMSL_SUPPLY_SPARSE_LU_FACTOR`.<br>Input argument `a` is ignored unless iterative refinement, computation of the condition number or computation of the reciprocal pivot growth factor is required. |

Default: `factsol` = 0

`IMSL_DIAG_PIVOT_THRESH`, *double* `diag_pivot_thresh` (Input)

Specifies the threshold used for a diagonal entry to be an acceptable pivot,

0.0 ≤ `diag_pivot_thresh` ≤ 1.0.

Default: `diag_pivot_thresh` = 1.0

`IMSL_SYMMETRIC_MODE`, *int* `symm_mode` (Input)

Indicates if the symmetric mode option is to be used. This mode should be applied if the input matrix $A$ is diagonally dominant or nearly so. The user should then define a small diagonal pivot threshold (e.g. 0.0 or 0.01) via option `IMSL_DIAG_PIVOT_THRESH` and choose an ($A^T$+$A$)-based column permutation algorithm (e.g. column permutation method `IMSL_MMD_AT_PLUS_A`).

| `symm_mode` | **Description** |
|---|---|
| 0 | Do not use symmetric mode option. |
| 1 | Use symmetric mode option. |

Default: `symm_mode` = 0

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[] (Input)

Array of length 6 containing positive parameters that allow the user to tune the performance of the matrix factorization algorithm.

| i | **Description of** sp_ienv[i] |
|---|---|
| 0 | The panel size.<br>Default: sp_ienv[0] = 10 |
| 1 | The relaxation parameter to control supernode amalgamation.<br>Default: sp_ienv[1] = 5 |
| 2 | The maximum allowable size for a supernode.<br>Default: sp_ienv[2] = 100 |
| 3 | The minimum row dimension to be used for 2D blocking.<br>Default: sp_ienv[3] = 200 |
| 4 | The minimum column dimension to be used for 2D blocking.<br>Default: sp_ienv[4] = 40 |
| 5 | The estimated fill factor for ***L*** and ***U***, compared to ***A***.<br>Default: sp_ienv[5] = 20 |

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *float* HB_values[] (Input)

Accept the coefficient matrix in Compressed Sparse Column (CSC) Format in the ***Introduction*** chapter of this manual for a discussion of this storage scheme.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_super_lu_factor* lu_factor_supplied (Input)

A structure of type *Imsl_f_super_lu_factor* containing the *LU* factorization of the input matrix computed with the IMSL_RETURN_SPARSE_LU_FACTOR option. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function *imsl_f_superlu_factor_free.*

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_f_super_lu_factor* *lu_factor_returned (Output)

The address of a structure of type *Imsl_f_super_lu_factor* containing the *LU* factorization of the input matrix. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function *imsl_f_superlu_factor_free.*

IMSL_CONDITION, *float* *condition (Output)

The estimate of the reciprocal condition number of matrix a after equilibration (if done).

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth (Output)

The reciprocal pivot growth factor

$$\min_j \left\{ \left\| \left( P_r D_r A D_c P_c \right)_j \right\|_\infty / \left\| U_j \right\|_\infty \right\}$$

If recip_pivot_growth is much less than 1, the stability of the LU factorization could be poor.

`IMSL_FORWARD_ERROR_BOUND`, *float* `*ferr` (Output)

The estimated forward error bound for the solution vector *x*. This option requires argument `IMSL_ITERATIVE_REFINEMENT` set to 1.

`IMSL_BACKWARD_ERROR`, *float* `*berr` (Output)

The componentwise relative backward error of the solution vector *x*. This option requires argument `IMSL_ITERATIVE_REFINEMENT` set to 1.

`IMSL_RETURN_USER`, *float* `x[]` (Output)

A user-allocated array of length `n` containing the solution `x` of the linear system.

## Description

Consider the sparse linear system of equations

$$Ax = b$$

Here, $A$ is a general square, nonsingular $n$ by $n$ sparse matrix, and $x$ and $b$ are vectors of length $n$. All entries in $A$, $x$ and $b$ are of real type.

Gaussian elimination, applied to the system above, can be shortly described as follows:

1. Compute a triangular factorization $P_rD_rAD_cP_c = LU$. Here, $D_r$ and $D_c$ are positive definite diagonal matrices to equilibrate the system and $P_r$ and $P_c$ are permutation matrices to ensure numerical stability and preserve sparsity. $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix.
2. Solve $Ax = b$ by evaluating

$$x = A^{-1}b = D_c\left(P_c\left(U^{-1}\left(L^{-1}\left(P_r\left(D_rb\right)\right)\right)\right)\right)$$

This is done efficiently by multiplying from right to left in the last expression: Scale the rows of $b$ by $D_r$. Multiplying $P_r\left(D_rb\right)$ means permuting the rows of $D_rb$.

Multiplying $L^{-1}\left(P_rD_rb\right)$ means solving the triangular system of equations with matrix $L$ by substitution. Similarly, multiplying $U^{-1}\left(L^{-1}\left(P_rD_rb\right)\right)$ means solving the triangular system with $U$.

Function `imsl_f_superlu` handles step 1 above by default or if optional argument `IMSL_FACTOR_SOLVE` is used and set to 1. More precisely, before $Ax = b$ is solved, the following steps are performed:

1. Equilibrate matrix $A$, i.e. compute diagonal matrices $D_r$ and $D_c$ so that $\hat{A} = D_rAD_c$ is "better conditioned" than $A$, i.e. $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than $A^{-1}$ is to perturbations in $A$.
2. Order the columns of $\hat{A}$ to increase the sparsity of the computed $L$ and $U$ factors, i.e. replace $\hat{A}$ by $\hat{A}P_c$ where $P_c$ is a column permutation matrix.

3. Compute the *LU* factorization of $\hat{A}P_c$. For numerical stability, the rows of $\hat{A}P_c$ are eventually permuted through the factorization process by scaled partial pivoting, leading to the decomposition $\tilde{A} := P_r\hat{A}P_c = LU$. The *LU* factorization is done by a left looking supernode-panel algorithm with 2-D blocking. See Demmel, Eisenstat, Gilbert et al. (1999) for further information on this technique.

4. Compute the reciprocal pivot growth factor

$$\min_{1 \le j \le n} \frac{\|\tilde{A}_j\|_\infty}{\|U_j\|_\infty}$$

   where $\tilde{A}_j$ and $U_j$ denote the $j$-th column of matrices $\tilde{A}$ and $U$, respectively.

5. Estimate the reciprocal of the condition number of matrix $\tilde{A}$.

During the solution process, this information is used to perform the following steps:

1. Solve the system $Ax = b$ using the computed triangular *L* and *U* factors.

2. Iteratively refine the solution, again using the computed triangular factors. This is equivalent to Newton's method.

3. Compute forward and backward error bounds for the solution vector $x$.

Some of the steps mentioned above are optional. Their settings can be controlled by the appropriate optional arguments of function `imsl_f_superlu`.

Function `imsl_f_superlu` uses a supernodal storage scheme for the *LU* factorization of matrix *A*. The factorization is contained in structure *Imsl_f_super_lu_factor* and two sub-structures. Following is a short description of these structures:

```
typedef struct{
  int nnz;                 /* Number of nonzeros in the matrix */
  float *nzval;            /* Array of nonzero values packed by column
                             */
  int *rowind;             /* Array of row indices of the nonzeros */
  int *colptr;             /* colptr[j] stores the location in nzval[]
                              and rowind[] which starts column j. It
                              has ncol+1 entries, and colptr[ncol]
                              points to the first free location in
                              arrays nzval[] and rowind[]. */
} Imsl_f_hb_format;

typedef struct{
  int nnz;                 /* Number of nonzeros in the supernodal
                              matrix */
  int nsuper;              /* Index of the last supernode */
  float *nzval;            /* Array of nonzero values packed by column
                             */
  int *nzval_colptr;       /* Array of length ncol+1; nzval_colptr[j]
                              stores the location in nzval which starts
                              column j. nzval_colptr[ncol] points to
                              the first free location in arrays
```

```
                              nzval[] and nzval_colptr[]. */
  int *rowind;              /* Array of compressed row indices of
                               rectangular supernodes */
  int *rowind_colptr;       /* Array of length ncol+1;
                               rowind_colptr[sup_to_col[s]] stores the
                               location in rowind[] which starts
                               all columns in supernode s, and
                               rowind_colptr[ncol] points to the first
                               free location in rowind[]. */
  int *col_to_sup;          /* Array of length ncol+1; col_to_sup[j] is
                               the supernode number to which column j
                               belongs. Only the first ncol entries in
                               col_to_sup[] are defined. */
  int *sup_to_col;          /* Array of length ncol+1; sup_to_col[s]
                               points to the starting column of the s-th
                               supernode. Only the first nsuper+2
                               entries in sup_to_col[] are defined, and
                               sup_to_col[nsuper+1] = ncol+1. */
} Imsl_f_sc_format;

typedef struct{
  int nrow;                 /* number of rows of matrix A */
  int ncol;                 /* number of columns of matrix A */
  int equilibration_method; /* The method used to equilibrate A:
                               0 - No equilibration
                               1 - Row equilibration.
                               2 - Column equilibration
                               3 - Both row and column equilibration */
  float *rowscale;          /* Array of length nrow containing the row
                               scale factors for A */
  float *columnscale;       /* Array of length ncol containing the
                               column scale factors for A */
  int *rowperm;             /* Row permutation array of length nrow
                               describing the row permutation matrix Pr
                             */
  int *colperm;             /* Column permutation array of length ncol
                               describing the column permutation matrix
                               Pc */
  Imsl_f_hb_format *U;      /* The part of the U factor of A outside the
                               supernodal blocks, stored in Harwell-
                               Boeing format */
  Imsl_f_sc_format *L;      /* The L factor of A, stored in supernodal
                               format as block lower triangular matrix
                             */
} Imsl_f_super_lu_factor;
```

Structure *Imsl_d_super_lu_factor* and its two sub-structures are defined similarly by replacing *float* by *double*, *Imsl_f_hb_format* by *Imsl_d_hb_format* and *Imsl_f_sc_format* by *Imsl_d_sc_format* in their definitions.

For a definition of supernodes and its use in sparse LU factorization, see the SuperLU Users' guide (1999) and J.W. Demmel, S. C. Eisenstat et al. (1999).

As an example, consider the matrix

$$A = \begin{bmatrix} 19 & 0 & 21 & 21 & 0 \\ 12 & 21 & 0 & 0 & 0 \\ 0 & 12 & 16 & 0 & 0 \\ 0 & 0 & 0 & 5 & 21 \\ 12 & 12 & 0 & 0 & 18 \end{bmatrix}$$

taken from the SuperLU Users' guide (1999).

Factorization of this matrix via `imsl_f_superlu` using natural column ordering, no equilibration and setting `sp_ienv`[1] from its default value 5 to 1 results in the following *LU* decomposition:

$$A = LU = \begin{bmatrix} 1.00 & & & & \\ 0.63 & 1.00 & & & \\ & 0.57 & 1.00 & & \\ & & & 1.00 & \\ 0.63 & 0.57 & -0.24 & -0.77 & 1.00 \end{bmatrix} \begin{bmatrix} 19.00 & & 21.00 & 21.00 & \\ & 21.00 & -13.26 & -13.26 & \\ & & 23.58 & 7.58 & \\ & & & 5.00 & 21.00 \\ & & & & 34.20 \end{bmatrix}.$$

Considering the filled matrix *F* (*I* denoting the identity matrix)

$$F = L + U - I = \begin{bmatrix} 19.00 & & 21.00 & 21.00 & \\ 0.63 & 21.00 & -13.26 & -13.26 & \\ & 0.57 & 23.58 & 7.58 & \\ & & & 5.00 & 21.00 \\ 0.63 & 0.57 & -0.24 & -0.77 & 34.20 \end{bmatrix}$$

the supernodal structure of the factors of matrix *A* can be described by

$$\begin{bmatrix} s_1 & & u_3 & u_4 & \\ s_1 & s_2 & s_2 & u_4 & \\ & s_2 & s_2 & u_4 & \\ & & & s_3 & s_3 \\ s_1 & s_2 & s_2 & s_3 & s_3 \end{bmatrix}$$

where $s_i$ denotes a nonzero entry in the $i$ th supernode and $u_i$ denotes a nonzero entry in the $i$ th column of $U$ outside the supernodal block.

Therefore, in a supernodal storage scheme the supernodal part of matrix *F* is stored as the lower block-diagonal matrix

$$L_{snode} = \begin{bmatrix} 19.00 & & & & \\ 0.63 & 21.00 & -13.26 & & \\ & 0.57 & 23.58 & & \\ & & & 5.00 & 21.00 \\ 0.63 & 0.57 & -0.24 & -077 & 34.20 \end{bmatrix}$$

and the part outside the supernodes as the upper triangular matrix

$$U_{snode} = \begin{bmatrix} * & & 21.00 & 21.00 & \\ & * & & -13.26 & \\ & & * & 7.58 & \\ & & & * & \\ & & & & * \end{bmatrix}$$

This is in accordance with the output for structure *Imsl_f_super_lu_factor*:

```
Equilibration method: 0

Scale vectors:
rowscale:   1.000000 1.000000 1.000000 1.000000 1.000000
columnscale: 1.000000 1.000000 1.000000 1.000000 1.000000
Permutation vectors:
colperm: 0 1 2 3 4
rowperm: 0 1 2 3 4
Harwell-Boeing matrix U:
nrow 5, ncol 5, nnz 11
nzval: 21.000000 -13.263157 7.578947 21.000000
rowind: 0 1 2 0
colptr: 0 0 0 1 4 4

Supernodal matrix L:
nrow 5, ncol 5, nnz 11, nsuper 2
nzval:
0  0  1.900000e+001
1  0  6.315789e-001
4  0  6.315789e-001
1  1  2.100000e+001
2  1  5.714286e-001
4  1  5.714286e-001
1  2  -1.326316e+001
2  2  2.357895e+001
4  2  -2.410714e-001
3  3  5.000000e+000
4  3  -7.714285e-001
3  4  2.100000e+001
4  4  3.420000e+001
```

```
nzval_colptr: 0 3 6 9 11 13

rowind: 0 1 4 1 2 4 3 4

rowind_colptr: 0 3 6 6 8 8

col_to_sup: 0 1 1 2 2
sup_to_col: 0 1 3 5
```

Function `imsl_f_superlu` is based on the SuperLU code written by Demmel, Gilbert, Li et al. For more detailed explanations of the factorization and solve steps, see the SuperLU User's Guide (1999).



## Examples

### Example 1

The LU factorization of the sparse 6×6 matrix

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

is computed.

Let $y = (1, 2, 3, 4, 5, 6)^T$, so that $b_1 := Ay = (10, 7, 45, 33, -34, 31)^T$ and $b_2 := A^T y = (-9, 8, 39, 13, 1, 21^T)$

The *LU* factorization of *A* is used to solve the sparse linear systems $Ax = b_1$ and $A^T x = b_2$.

```
#include <imsl.h>

int 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 b1[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
   float b2[] = { -9.0, 8.0, 39.0, 13.0, 1.0, 21.0 };
   int n = 6, nz = 15;
   float *x = NULL;

   x = imsl_f_superlu (n, nz, a, b1, 0);
   imsl_f_write_matrix ("solution to A*x = b1", 1, n, x, 0);
   imsl_free (x);

   x = imsl_f_superlu (n, nz, a, b2, IMSL_TRANSPOSE, 1, 0);
   imsl_f_write_matrix ("solution to A^T*x = b2", 1, n, x, 0);
   imsl_free (x);
}
```

## Output

```
                          solution to A*x = b
        1          2          3          4          5          6
```

```
        1           2           3           4           5           6

                        solution to A^T*x = b2
        1           2           3           4           5           6
        1           2           3           4           5           6
```

## Example 2

This example uses the matrix *A* = *E*(1000,10) to show how the *LU* factorization of *A* can be used to solve a linear system with the same coefficient matrix *A* but different right-hand side. Maximum absolute errors are printed. After the computations, the space allocated for the *LU* factorization is freed via function `imsl_f_superlu_factor_free`.

```
#include <imsl.h>

int main(){

    Imsl_f_sparse_elem *a;
    Imsl_f_super_lu_factor lu_factor;
    float *b, *x, *mod_five, *mod_ten;
    float error_factor_solve, error_solve;
    int n = 1000, c = 10;
    int i, nz, index;

    /* Get the coefficient matrix */
    a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

    /* Set two different predetermined solutions */
    mod_five = (float*) malloc (n*sizeof(*mod_five));
    mod_ten = (float*) malloc (n*sizeof(*mod_ten));
    for (i=0; i<n; i++) {
      mod_five[i] = (float) (i % 5);
      mod_ten[i] = (float) (i % 10);
    }

    /* Choose b so that x will approximate mod_five */
    b = imsl_f_mat_mul_rect_coordinate ("A*x",
            IMSL_A_MATRIX, n, n, nz, a,
            IMSL_X_VECTOR, n, mod_five, 0);

    /* Solve Ax = b */
    x = imsl_f_superlu (n, nz, a, b,
            IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor, 0);

    /* Compute max absolute error */
    error_factor_solve = imsl_f_vector_norm (n, x,
            IMSL_SECOND_VECTOR, mod_five,
            IMSL_INF_NORM, &index,
            0);
```

```
    imsl_free (mod_five);
    imsl_free (b);
    imsl_free (x);

    /* Get new right hand side -- b = A * mod_ten */
    b = imsl_f_mat_mul_rect_coordinate ("A*x",
            IMSL_A_MATRIX, n, n, nz, a,
            IMSL_X_VECTOR, n, mod_ten,
            0);

    /* Use the previously computed factorization
       to solve Ax = b */
    x = imsl_f_superlu (n, nz, a, b,
            IMSL_SUPPLY_SPARSE_LU_FACTOR, lu_factor,
            IMSL_FACTOR_SOLVE, 2,
            0);
    error_solve = imsl_f_vector_norm (n, x,
            IMSL_SECOND_VECTOR, mod_ten,
            IMSL_INF_NORM, &index,
            0);

    imsl_free (mod_ten);
    imsl_free (b);
    imsl_free (x);
    imsl_free (a);

    /* Free sparse LU structure */
    imsl_f_superlu_factor_free (&lu_factor);

    /* Print errors */
    printf ("absolute error (factor/solve) = %e\n",
             error_factor_solve);
    printf ("absolute error (solve) = %e\n", error_solve);
}
```

### Output

```
absolute error (factor/solve) = 1.502037e-005
absolute error (solve) = 1.621246e-005
```

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# superlu (complex)

more...

Computes the *LU* factorization of a general complex sparse matrix by a column method and solves the complex sparse linear system of equations $Ax = b$ .

## Synopsis

*#include* <imsl.h>

*f_complex* `*imsl_c_superlu` (*int* `n`, *int* `nz`, *Imsl_c_sparse_elem* `a[]`, *f_complex* `b[]`, ..., 0)

*void* `imsl_c_superlu_factor_free` *(Imsl_c_super_lu_factor* `*factor`)

The type ***double*** functions are `imsl_z_superlu` and `imsl_z_superlu_factor_free`.

## Required Arguments

*int* `n` (Input)
The order of the input matrix.

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

*Imsl_c_sparse_elem* `a[]` (Input)
Array of length `nz` containing the location and value of each nonzero entry in the matrix. See the explanation of the ***Imsl_c_sparse_elem*** structure in the section Matrix Storage Modes in the “Introduction” chapter of this manual.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* `*imsl_c_superlu` (*int* `n`, *int* `nz`, *Imsl_c_sparse_elem* `a[]`, *f_complex* `b[]`,

`IMSL_EQUILIBRATE`, *int equilibrate*,

`IMSL_COLUMN_ORDERING_METHOD`, *Imsl_col_ordering* `method`,

`IMSL_COLPERM_VECTOR`, *int* `permc[]`,

`IMSL_TRANSPOSE`, *int* `transpose`,

`IMSL_ITERATIVE_REFINEMENT`, *int* `refine`,

`IMSL_FACTOR_SOLVE`, *int* `factsol`,

`IMSL_DIAG_PIVOT_THRESH`, *double* `diag_pivot_thresh`,

`IMSL_SYMMETRIC_MODE`, *int* `symm_mode`,

`IMSL_PERFORMANCE_TUNING`, *int* `sp_ienv[]`,

`IMSL_CSC_FORMAT`, *int* `HB_col_ptr[]`, *int* `HB_row_ind[]`, *f_complex* `HB_values[]`,

`IMSL_SUPPLY_SPARSE_LU_FACTOR`, *Imsl_c_super_lu_factor* `lu_factor_supplied`,

`IMSL_RETURN_SPARSE_LU_FACTOR`, *Imsl_c_super_lu_factor* `*lu_factor_returned`,

`IMSL_CONDITION`, *float* `*condition`,

`IMSL_PIVOT_GROWTH_FACTOR`, *float* `*recip_pivot_growth`,

`IMSL_FORWARD_ERROR_BOUND`, *float* `*ferr`,

`IMSL_BACKWARD_ERROR`, *float* `*berr`,

`IMSL_RETURN_USER`, *f_complex* `x[]`,

0)

## Optional Arguments

`IMSL_EQUILIBRATE`, *int* `equilibrate` (Inputs)

Specifies if the input matrix *A* should be equilibrated before factorization.

| **equilibrate** | **Description** |
|---|---|
| 0 | Do not equilibrate *A* before factorization |
| 1 | Equilibrate *A* before factorization. |

Default: `equilibrate` = 0

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method (Input)

The column ordering method used to preserve sparsity prior to the factorization process. Select the ordering method by setting method to one of the following:

| method | Description |
|---|---|
| IMSL_NATURAL | Natural ordering, i.e.the column ordering of the input matrix.. |
| IMSL_MMD_ATA | Minimum degree ordering on the structure of $A^T A$ . |
| IMSL_MMD_AT_PLUS_A | Minimum degree ordering on the structure of $A^T + A$ . |
| IMSL_COLAMD | Column approximate minimum degree ordering. |
| IMSL_PERMC | Use ordering given in permutation vector permc, which is input by the user through optional argument IMSL_COLPERM_VECTOR. Vector permc is a permutation of the numbers 0,1,...,n−1. |

Default: method = IMSL_COLAMD

IMSL_COLPERM_VECTOR, *int* permc[] (Input)

Array of length n which defines the permutation matrix $P_c$ before postordering. This argument is required if IMSL_COLUMN_ORDERING_METHOD with method = IMSL_PERMC is used. Otherwise, it is ignored.

IMSL_TRANSPOSE, *int* transpose (Input)

Indicates if the problem $Ax = b$ or one of the transposed problems $A^T x = b$ or $A^H x = b$ is to be solved.

| transpose | Description |
|---|---|
| 0 | Solve $Ax = b$ . |
| 1 | Solve $A^T x = b$ .<br>This option can be used in conjunction with either of the options that supply the factorization. |
| 2 | Solve $A^H x = b$ .<br>This option can be used in conjunction with either of the options that supply the factorization. |

Default: transpose = 0

IMSL_ITERATIVE_REFINEMENT, *int* refine (Input)

Indicates if iterative refinement is desired.

| refine | Description |
|---|---|
| 0 | No iterative refinement. |
| 1 | Do iterative refinement. |

Default: `refine` = 1

`IMSL_FACTOR_SOLVE`, *int* `factsol` (Input)

Indicates if the *LU* factorization, the solution of a linear system or both are to be computed.

| factsol | Description |
|---|---|
| 0 | Compute the *LU* factorization of the input matrix *A* and solve the system $Ax = b$. |
| 1 | Only compute the *LU* factorization of the input matrix and return.<br>The *LU* factorization is returned via optional argument `IMSL_RETURN_SPARSE_LU_FACTOR`.<br>Input argument `b` is ignored. |
| 2 | Only solve $Ax = b$ given the *LU* factorization of $A$.<br>The *LU* factorization of $A$ must be supplied via optional argument `IMSL_SUPPLY_SPARSE_LU_FACTOR`.<br>Input argument `a` is ignored unless iterative refinement, computation of the condition number or computation of the reciprocal pivot growth factor is required. |

Default: `factsol` = 0

`IMSL_DIAG_PIVOT_THRESH`, *double* `diag_pivot_thresh` (Input)

Specifies the threshold used for a diagonal entry to be an acceptable pivot, 0.0 ≤ `diag_pivot_thresh` ≤ 1.0.

Default: `diag_pivot_thresh` = 1.0.

`IMSL_SYMMETRIC_MODE`, *int* `symm_mode` (Input)

Indicates if the symmetric mode option is to be used. This mode should be applied if the input matrix $A$ is diagonally dominant or nearly so. The user should then define a small diagonal pivot threshold (e.g. 0.0 or 0.01) via optional argument `IMSL_DIAG_PIVOT_THRESH` and choose an $\left(A^T + A\right)$-based column permutation algorithm (e.g. column permutation method `IMSL_MMD_AT_PLUS_A`).

| symm_mode | Description |
|---|---|
| 0 | Do not use symmetric mode option. |
| 1 | Use symmetric mode option. |

Default: symm_mode = 0

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[] (Input)

Vector of length 6 containing positive parameters that allow the user to tune the performance of the matrix factorization algorithm.

| i | Description of sp_ienv[i] |
|---|---|
| 0 | The panel size.<br>Default: sp_ienv[0] = 10 |
| 1 | The relaxation parameter to control supernode amalgamation.<br>Default: sp_ienv[1] = 5 |
| 2 | The maximum allowable size for a supernode.<br>Default: sp_ienv[2] = 100 |
| 3 | The minimum row dimension to be used for 2D blocking.<br>Default: sp_ienv[3] = 200 |
| 4 | The minimum column dimension to be used for 2D blocking.<br>Default: sp_ienv[4] = 40 |
| 5 | The estimated fill factor for *L* and *U*, compared to *A*.<br>Default: sp_ienv[5] = 20 |

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *f_complex* HB_values[] (Input)

Accept the coefficient matrix in Compressed Sparse Column (CSC) Format in the main *Introduction* chapter of this manual for a discussion of this storage scheme.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_c_super_lu_factor* lu_factor_supplied (Input)

A structure of type *Imsl_c_super_lu_factor* containing the *LU* factorization of the input matrix computed with the IMSL_RETURN_SPARSE_LU_FACTOR option. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function *imsl_c_superlu_factor_free.*

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_c_super_lu_factor* *lu_factor_returned (Output)

The address of a structure of type *Imsl_c_super_lu_factor* containing the *LU* factorization of the input matrix. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function *imsl_c_superlu_factor_free.*

IMSL_CONDITION, *float* *condition (Output)

The estimate of the reciprocal condition number of matrix *A* after equilibration (if done).

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth (Output)

The reciprocal pivot growth factor

$$\min_j \left\{ \left\| \left( P_r D_r A D_c P_c \right)_j \right\|_\infty / \left\| U_j \right\|_\infty \right\}$$

If `recip_pivot_growth` is much less than 1, the stability of the *LU* factorization could be poor.

`IMSL_FORWARD_ERROR_BOUND`, *float* `*ferr` (Output)
The estimated forward error bound for the solution vector *x*. This option requires argument `IMSL_ITERATIVE_REFINEMENT` set to 1.

`IMSL_BACKWARD_ERROR`, *float* `*berr` (Output)
The componentwise relative backward error of the solution vector *x*. This option requires argument `IMSL_ITERATIVE_REFINEMENT` set to 1.

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

## Description

Consider the sparse linear system of equations

$$Ax = b$$

Here, $A$ is a general square, nonsingular $n$ by $n$ sparse matrix, and $x$ and $b$ are vectors of length $n$. All entries in $A$, $x$ and $b$ are of complex type.

Gaussian elimination, applied to the system above, can be shortly described as follows:

1. Compute a triangular factorization $P_r D_r A D_c P_c = LU$. Here, $D_r$ and $D_c$ are positive definite diagonal matrices to equilibrate the system and $P_r$ and $P_c$ are permutation matrices to ensure numerical stability and preserve sparsity. $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix.
2. Solve $Ax = b$ by evaluating

$$x = A^{-1}b = D_\mathrm{c}\left(P_\mathrm{c}\left(U^{-1}\left(L^{-1}\left(P_\mathrm{r}\left(D_\mathrm{r}b\right)\right)\right)\right)\right)$$

This is done efficiently by multiplying from right to left in the last expression: Scale the rows of $b$ by $D_r$. Multiplying $P_r\left(D_r b\right)$ means permuting the rows of $D_r b$.

Multiplying $L^{-1}\left(P_r D_r b\right)$ means solving the triangular system of equations with matrix $L$ by substitution. Similarly, multiplying $U^{-1}\left(L^{-1}\left(P_\mathrm{r} D_\mathrm{r} b\right)\right)$ means solving the triangular system with $U$.

Function `imsl_c_superlu` handles step 1 above by default or if optional argument `IMSL_FACTOR_SOLVE` is used and set to 1. More precisely, before $Ax = b$ is solved, the following steps are performed:

1. Equilibrate matrix $A$, i.e. compute diagonal matrices $D_r$ and $D_c$ so that $\hat{A} = D_r A D_c$ is "better conditioned" than $A$, i.e. $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than $A^{-1}$ is to perturbations in $A$.

2. Order the columns of $\hat{A}$ to increase the sparsity of the computed $L$ and $U$ factors, i.e. replace $\hat{A}$ by $\hat{A}P_c$ where $P_c$ is a column permutation matrix.

3. Compute the *LU* factorization of $\hat{A}P_c$. For numerical stability, the rows of $\hat{A}P_c$ are eventually permuted through the factorization process by scaled partial pivoting, leading to the decomposition $\tilde{A} := P_r\hat{A}P_c = LU$. The *LU* factorization is done by a left looking supernode-panel algorithm with 2-D blocking. See Demmel, Eisenstat, Gilbert et al. (1999) for further information on this technique.

4. Compute the reciprocal pivot growth factor

$$\min_{1 \le j \le n} \frac{\|\tilde{A}_j\|_\infty}{\|U_j\|_\infty}$$

where $\tilde{A}_j$ and $U_j$ denote the $j$-th column of matrices $\tilde{A}$ and $U$, respectively.

5. Estimate the reciprocal of the condition number of matrix $\tilde{A}$.

During the solution process, this information is used to perform the following steps:

1. Solve the system $Ax = b$ using the computed triangular *L* and *U* factors.

2. Iteratively refine the solution, again using the computed triangular factors. This is equivalent to Newton's method.

3. Compute forward and backward error bounds for the solution vector $x$.

Some of the steps mentioned above are optional. Their settings can be controlled by the appropriate optional arguments of function `imsl_c_superlu`.

Function `imsl_c_superlu` uses a supernodal storage scheme for the *LU* factorization of matrix *A*. The factorization is contained in structure *Imsl_c_super_lu_factor* and two sub-structures. Following is a short description of these structures:

```
typedef struct{
  int nnz;              /* Number of nonzeros in the matrix */
  f_complex *nzval;     /* Array of nonzero values packed by column
                           */
  int *rowind;          /* Array of row indices of the nonzeros */
  int *colptr;          /* colptr[j] stores the location in nzval[]
                             and rowind[] which starts column j. It has
                             ncol+1 entries, and colptr[ncol] points to
                             the first free location in arrays nzval[]
                             and rowind[]. */
} Imsl_c_hb_format;

typedef struct{
  int nnz;              /* Number of nonzeros in the supernodal
                             matrix */
  int nsuper;           /* Index of the last supernode */
  f_complex *nzval;     /* Array of nonzero values packed by column
                           */
```

```
  int *nzval_colptr;      /* Array of length ncol+1; nzval_colptr[j]
                             stores the location in nzval which starts
                             column j. nzval_colptr[ncol] points to the
                             first free location in arrays nzval[] and
                             nzval_colptr[]. */
  int *rowind;            /* Array of compressed row indices of
                             rectangular supernodes */
  int *rowind_colptr;     /* Array of length ncol+1;
                             rowind_colptr[sup_to_col[s]] stores the
                             location in rowind[] which starts
                             all columns in supernode s, and
                             rowind_colptr[ncol] points to the first
                             free location in rowind[]. */
  int *col_to_sup;        /* Array of length ncol+1; col_to_sup[j] is
                             the supernode number to which column j
                             belongs. Only the first ncol entries in
                             col_to_sup[] are defined. */
  int *sup_to_col;        /* Array of length ncol+1; sup_to_col[s]
                             points to the starting column of the s-th
                             supernode. Only the first nsuper+2 entries
                             in sup_to_col[] are defined, and
                             sup_to_col[nsuper+1] = ncol+1. */
} Imsl_c_sc_format;

typedef struct{
  int nrow;                /* number of rows of matrix A */
  int ncol;                /* number of columns of matrix A */
  int equilibration_method; /* The method used to equilibrate A:
                              0 - No equilibration
                              1 - Row equilibration.
                              2 - Column equilibration
                              3 - Both row and column equilibration */
  float *rowscale;        /* Array of length nrow containing the row
                             scale factors for A */
  float *columnscale;     /* Array of length ncol containing the
                             column scale factors for A */
  int *rowperm;           /* Row permutation array of length nrow
                             describing the row permutation matrix Pr
                           */
  int *colperm;           /* Column permutation array of length ncol
                             describing the column permutation matrix
                             Pc */
  Imsl_c_hb_format *U;    /* The part of the U factor of A outside the
                             supernodal blocks, stored in Harwell-
                             Boeing format */
  Imsl_c_sc_format *L;    /* The L factor of A, stored in supernodal
                             format as block lower triangular matrix */
} Imsl_c_super_lu_factor;
```

Structure *Imsl_z_super_lu_factor* and its two sub-structures are defined similarly by replacing *float* by *double*, *f_complex* by *d_complex*, *Imsl_c_hb_format* by *Imsl_z_hb_format* and *Imsl_c_sc_format* by *Imsl_z_sc_format* in their definitions.

For a definition of supernodes and its use in sparse unsymmetric LU factorization, see the SuperLU Users' guide (1999) and J.W. Demmel, S. C. Eisenstat et al. (1999).

As an example, consider the matrix

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

Factorization of this matrix via `imsl_c_superlu` using natural column ordering, no equilibration, setting `sp_ienv`[1] from its default value 5 to 1 and reducing the diagonal pivot thresh factor to 0.5 results in the following *LU* decomposition:

$$A = LU = \begin{bmatrix} 1 & & & & \\ 1+i & 1 & & & \\ & i & 1 & & \\ & & & 1 & \\ 1+i & i & 2i & 2 & 1 \end{bmatrix} \begin{bmatrix} 1-i & & 1-i & 1-i & \\ & 1-i & -2 & -2 & \\ & & 1+i & 2i & \\ & & & 1+i & 1-i \\ & & & & i \end{bmatrix}$$

Considering the filled matrix $F$ ( $I$ denoting the identity matrix),

$$F = L + U - I = \begin{bmatrix} 1-i & & 1-i & 1-i & \\ 1+i & 1-i & -2 & -2 & \\ & i & 1+i & 2i & \\ & & & 1+i & 1-i \\ 1+i & i & 2i & 2 & i \end{bmatrix}$$

the supernodal structure of the factors of matrix *A* can be described by

$$\begin{bmatrix} s_1 & & u_3 & u_4 & \\ s_1 & s_2 & s_2 & u_4 & \\ & s_2 & s_2 & u_4 & \\ & & & s_3 & s_3 \\ s_1 & s_2 & s_2 & s_3 & s_3 \end{bmatrix}$$

where $s_i$ denotes a nonzero entry in the $i$ th supernode and $u_i$ denotes a nonzero entry in the *i*–th column of $U$ outside the supernodal block.

Therefore, in a supernodal storage scheme the supernodal part of matrix *F* is stored as the lower block-diagonal matrix

$$L_{snode} = \begin{bmatrix} 1-i & & & & \\ 1+i & 1-i & -2 & & \\ & i & 1+i & & \\ & & & 1+i & 1-i \\ 1+i & i & 2i & 2 & i \end{bmatrix}$$

and the part outside the supernodes as the upper triangular matrix

$$U_{snode} = \begin{bmatrix} * & & 1-i & 1-i & \\ & * & & -2 & \\ & & * & 2i & \\ & & & * & \\ & & & & * \end{bmatrix}$$

This is in accordance with the output for structure *Imsl_c_super_lu_factor*:

```
Equilibration method: 0

Scale vectors:
rowscale:   1.000000 1.000000 1.000000 1.000000 1.000000
columnscale: 1.000000 1.000000 1.000000 1.000000 1.000000
Permutation vectors:
colperm: 0 1 2 3 4
rowperm: 0 1 2 3 4
Harwell-Boeing matrix U:
nrow 5, ncol 5, nnz 11
nzval: (1.000000,-1.000000) (-2.000000,0.000000) (0.000000,2.000000)
      (1.000000,-1.000000)
rowind: 0 1 2 0
colptr: 0 0 0 1 4 4

Supernodal matrix L:
nrow 5, ncol 5, nnz 11, nsuper 2
nzval:
0      0      (1.000000,-1.000000)
1      0      (1.000000,1.000000)
4      0      (1.000000,1.000000)
1      1      (1.000000,-1.000000)
2      1      (0.000000,1.000000)
4      1      (0.000000,1.000000)
1      2      (-2.000000,0.000000)
2      2      (1.000000,1.000000)
4      2      (0.000000,2.000000)
3      3      (1.000000,1.000000)
4      3      (2.000000,0.000000)
3      4      (1.000000,-1.000000)
4      4      (0.000000,1.000000)
```

```
nzval_colptr: 0 3 6 9 11 13
rowind: 0 1 4 1 2 4 3 4
rowind_colptr: 0 3 6 6 8 8
col_to_sup: 0 1 1 2 2
sup_to_col: 0 1 3 5
```

Function `imsl_c_superlu` is based on the SuperLU code written by Demmel, Gilbert, Li et al. For more detailed explanations of the factorization and solve steps, see the SuperLU User's Guide (1999).



## Examples

### Example 1

The *LU* factorization of the sparse complex 6×6 matrix

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

is computed. Let

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

so that

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

$$b_1: = A^T y = (-112+54i, -58+46i, 12i, -51+5i, 34+78i, -94+60i)^T$$

and

$$b_2: = A^H y = (54-112i,\ 46-58i, 12,\ 5-51i, 78+34i, 60-94i^T)$$

The *LU* factorization of *A* is used to solve the sparse complex linear systems $Ax = b$, $A^T x = b_1$ and $A^H x = b_2$.

```
#include <imsl.h>

int 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[] = {{3.0, 17.0}, {-19.0, 5.0}, {6.0, 18.0},
                    {-38.0, 32.0}, {-63.0, 49.0}, {-57.0, 83.0}};

   f_complex b1[] = {{-112.0,54.0}, {-58.0,46.0}, {0.0,12.0},
                     {-51.0,5.0}, {34.0,78.0}, {-94.0,60.0}};

   f_complex b2[] = {{54.0,-112.0}, {46.0, -58.0}, {12.0, 0.0},
                     {5.0, -51.0}, {78.0, 34.0}, {60.0, -94.0}};
```

```
   int n = 6, nz = 15;
   f_complex *x = NULL;

   x = imsl_c_superlu (n, nz, a, b, 0);
   imsl_c_write_matrix ("solution to A*x = b", n, 1, x, 0);
   imsl_free (x);

   x = imsl_c_superlu (n, nz, a, b1, IMSL_TRANSPOSE, 1, 0);
   imsl_c_write_matrix ("solution to A^T*x = b1", n, 1, x, 0);
   imsl_free (x);

   x = imsl_c_superlu (n, nz, a, b2, IMSL_TRANSPOSE, 2, 0);
   imsl_c_write_matrix ("solution to A^H*x = b2", n, 1, x, 0);
   imsl_free (x);
}
```

### Output

```
    solution to A*x = b
1 (         1,         1)
2 (         2,         2)
3 (         3,         3)
4 (         4,         4)
5 (         5,         5)
6 (         6,         6)

  solution to A^T*x = b1
1 (         1,         1)
2 (         2,         2)
3 (         3,         3)
4 (         4,         4)
5 (         5,         5)
6 (         6,         6)

  solution to A^H*x = b2
1 (         1,         1)
2 (         2,         2)
3 (         3,         3)
4 (         4,         4)
5 (         5,         5)
6 (         6,         6)
```

## Example 2

This example uses the matrix $A = E(1000,10)$ to show how the $LU$ factorization of $A$ can be used to solve a linear system with the same coefficient matrix $A$ but different right-hand side. Maximum absolute errors are printed. After the computations, the space allocated for the $LU$ factorization is freed via function `imsl_c_superlu_factor_free`.

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

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

int main()
{
    Imsl_c_sparse_elem *a;
    Imsl_c_super_lu_factor lu_factor;
    f_complex *b, *x, *mod_five, *mod_ten;
    float error_factor_solve, error_solve;
    int n = 1000, c = 10;
    int i, nz, index;

    /* Get the coefficient matrix */
    a = imsl_c_generate_test_coordinate (n, c, &nz, 0);

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

    /* Choose b so that x will approximate mod_five */
    b = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_five,
        0);

    /* Solve Ax = b */
    x = imsl_c_superlu (n, nz, a, b,
        IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor,
        0);

    /* Compute max absolute error */
    error_factor_solve = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);

    free (mod_five);
    imsl_free (b);
    imsl_free (x);

    /* Get new right hand side -- b = A * mod_ten */
    b = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_ten,
        0);

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

```
    x = imsl_c_superlu (n, nz, a, b,
        IMSL_SUPPLY_SPARSE_LU_FACTOR, lu_factor,
        IMSL_FACTOR_SOLVE, 2,
        0);
    error_solve = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_ten,
        IMSL_INF_NORM, &index,
        0);

    free (mod_ten);
    imsl_free (b);
    imsl_free (x);
    imsl_free (a);

    /* Free sparse LU structure */
    imsl_c_superlu_factor_free (&lu_factor);

    /* Print errors */
    printf ("absolute error (factor/solve) = %e\n",
        error_factor_solve);
    printf ("absolute error (solve) = %e\n", error_solve);
}
```

### Output

```
absolute error (factor/solve) = 9.581565e-007
absolute error (solve) = 2.017575e-006
```

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# superlu_smp

more...

more...

Computes the *LU* factorization of a general sparse matrix by a left-looking column method using OpenMP parallelism, and solves the real sparse linear system of equations *Ax* = *b* .

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_superlu_smp (*int* n, *int* nz, *Imsl_f_sparse_elem* a [], *float* b [],…,0)

*void* imsl_f_superlu_smp_factor_free (*Imsl_f_super_lu_smp_factor* *factor)

The type ***double*** functions are imsl_d_superlu_smp and imsl_d_superlu_smp_factor_free.

## Required Arguments

*int* n (Input)
The order of the input matrix.

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

*Imsl_f_sparse_elem* a [] (Input)
An array of length nz containing the location and value of each nonzero entry in the matrix. See the explanation of the Imsl_f_sparse_elem structure in the section Matrix Storage Modes in the "Introduction" chapter of this manual.

*float* b [] (Input)
An array of length n containing the right-hand side.

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_superlu_smp (*int* n, *int* nz, *Imsl_f_sparse_elem* a[], *float* b[],

IMSL_EQUILIBRATE, *int* equilibrate,

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method,

IMSL_COLPERM_VECTOR, *int* permc[],

IMSL_TRANSPOSE, *int* transpose,

IMSL_ITERATIVE_REFINEMENT, *int* refine,

IMSL_FACTOR_SOLVE, *int* factsol,

IMSL_DIAG_PIVOT_THRESH, *float* diag_pivot_thresh,

IMSL_SNODE_PREDICTION, *int* snode_prediction,

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[],

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind, *float* HB_values[],

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_super_lu_smp_factor* *lu_factor_supplied,

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_f_super_lu_smp_factor* *lu_factor_returned,

IMSL_CONDITION, *float* *condition,

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth,

IMSL_FORWARD_ERROR_BOUND, *float* *ferr,

IMSL_BACKWARD_ERROR, *float* *berr,

IMSL_RETURN_USER, *float* x[],

0)

## Optional Arguments

IMSL_EQUILIBRATE, *int* equilibrate (Input)

Specifies if the input matrix *A* should be equilibrated before factorization.

| **equilibrate** | **Description** |
|---|---|
| 0 | Do not equilibrate *A* before factorization |
| 1 | Equilibrate *A* before factorization. |

Default: equilibrate = 0.

`IMSL_COLUMN_ORDERING_METHOD`, *Imsl_col_ordering* `method` (Input)

The column ordering method used to preserve sparsity prior to the factorization process. Select the ordering method by setting `method` to one of the following:

| `method` | Description |
|---|---|
| `IMSL_NATURAL` | Natural ordering, i.e.the column ordering of the input matrix. |
| `IMSL_MMD_ATA` | Minimum degree ordering on the structure of $A^T A$. |
| `IMSL_MMD_AT_PLUS_A` | Minimum degree ordering on the structure of $A^T + A$. |
| `IMSL_COLAMD` | Column approximate minimum degree ordering. |
| `IMSL_PERMC` | Use ordering given in permutation vector `permc`, which is input by the user through the optional argument `IMSL_COLPERM_VECTOR`. Vector `permc` is a permutation of the numbers 0,1,..., `n`-1. |

Default: `method` = `IMSL_COLAMD`.

`IMSL_COLPERM_VECTOR`, *int* `permc[]` (Input)

Array of length `n` that defines the permutation matrix $P_c$ before postordering. This argument is required if `IMSL_COLUMN_ORDERING_METHOD` with `method` = `IMSL_PERMC` is used. Otherwise, it is ignored.

`IMSL_TRANSPOSE`, *int* `transpose` (Input)

Indicates if the transposed problem $A^T x = b$ is to be solved. This option can be used in conjunction with either of the options that supply the factorization.

| `transpose` | Description |
|---|---|
| 0 | Solve $Ax = b$. |
| 1 | Solve $A^T x = b$. |

Default: `transpose` = 0.

`IMSL_ITERATIVE_REFINEMENT`, *int* `refine` (Input)

Indicates if iterative refinement is desired.

| `refine` | Description |
|---|---|
| 0 | No iterative refinement. |
| 1 | Do iterative refinement. |

Default: `refine` = 1.

`IMSL_FACTOR_SOLVE`, *int* `factsol` (Input)

Indicates if the *LU* factorization, the solution of a linear system, or both are to be computed.

| factsol | Description |
|---|---|
| 0 | Compute the *LU* factorization of the input matrix *A* and solve the system $Ax = b$. |
| 1 | Only compute the *LU* factorization of the input matrix and return.<br>The *LU* factorization is returned via the optional argument IMSL_RETURN_SPARSE_LU_FACTOR.<br>Input argument b is ignored. |
| 2 | Only solve $Ax = b$ given the *LU* factorization of *A*.<br>The *LU* factorization of *A* must be supplied via the optional argument IMSL_SUPPLY_SPARSE_LU_FACTOR.<br>Input argument a is ignored unless iterative refinement, computation of the condition number, or computation of the reciprocal pivot growth factor is required. |

Default: factsol = 0.

IMSL_DIAG_PIVOT_THRESH, *float* diag_pivot_thresh (Input)

Specifies the threshold used for a diagonal entry to be an acceptable pivot,

0.0 ≤ diag_pivot_thresh ≤ 1.0.

Default: diag_pivot_thresh = 1.0.

IMSL_SNODE_PREDICTION, *int* snode_prediction (Input)

Indicates which scheme is used to predict the number of nonzeros in the *L* supernodes.

| snode_prediction | Description |
|---|---|
| 0 | Use static scheme for the prediction of the number of nonzeros in the *L* supernodes. |
| 1 | Use dynamic scheme for the prediction of the number of nonzeros in the *L* supernodes. |

Default: snode_prediction = 0.

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[] (Input)

Array of length 8 containing parameters that allow the user to tune the performance of the matrix factorization algorithm. The elements sp_ienv[i] must be positive for i = 0,…,4 and different from zero for i = 5,6,7.

| i | Description of sp_ienv[i] |
|---|---|
| 0 | The panel size.<br>Default: sp_ienv[0] = 10. |
| 1 | The relaxation parameter to control supernode amalgamation.<br>Default: sp_ienv[1] = 5. |

| i | Description of sp_ienv[i] |
|---|---|
| 2 | The maximum allowable size for a supernode.<br>Default: sp_ienv[2] = 100. |
| 3 | The minimum row dimension to be used for 2D blocking.<br>Default: sp_ienv[3] = 200. |
| 4 | The minimum column dimension to be used for 2D blocking.<br>Default: sp_ienv[4] = 40. |
| 5 | The size of the array nzval to store the values of the *L* supernodes. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>This element of array sp_ienv is used only if a dynamic scheme for the prediction of the sizes of the *L* supernodes is used, i.e. if snode_prediction = 1.<br>Default: sp_ienv[5] = -20. |
| 6 | The size of the arrays rowind and nzval to store the columns in *U*. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>Default: sp_ienv[6] = -20. |
| 7 | The size of the array rowind to store the subscripts of the *L* supernodes. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>Default: sp_ienv[7] = -10. |

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *float* HB_values[] (Input)

Accept the coefficient matrix in compressed sparse column (CSC) format, as described in the Compressed Sparse Column (CSC) Format section of the “Introduction” chapter of this manual.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_f_super_lu_smp_factor* *lu_factor_supplied (Input)

The address of a structure of type *Imsl_f_super_lu_smp_factor* containing the *LU* factors of the input matrix computed with the IMSL_RETURN_SPARSE_LU_FACTOR option. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function imsl_f_superlu_smp_factor_free.

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_f_super_lu_smp_factor* *lu_factor_returned (Output)

The address of a structure of type *Imsl_f_super_lu_smp_factor* containing the *LU* factorization of the input matrix. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function imsl_f_superlu_smp_factor_free.

IMSL_CONDITION, *float* *condition (Output)

The estimate of the reciprocal condition number of matrix a after equilibration (if done).

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth (Output)

The reciprocal pivot growth factor

$$\min_j \left\{ \left\| \left( P_r D_r A D_c P_c \right)_j \right\|_\infty / \left\| U_j \right\|_\infty \right\}.$$

If recip_pivot_growth is much less than 1, the stability of the *LU* factorization could be poor.

IMSL_FORWARD_ERROR_BOUND, *float* *ferr (Output)

The estimated forward error bound for the solution vector *x*. This option requires argument IMSL_ITERATIVE_REFINEMENT set to 1.

IMSL_BACKWARD_ERROR, *float* *berr (Output)

The componentwise relative backward error of the solution vector *x*. This option requires argument IMSL_ITERATIVE_REFINEMENT set to 1.

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length n containing the solution *x* of the linear system.

## Description

The steps imsl_f_superlu_smp uses to solve linear systems are identical to the steps described in the documentation of the serial version imsl_f_superlu.

Function imsl_f_superlu_smp uses a supernodal storage scheme for the *LU* factorization of matrix *A*. In contrast to the sequential version, the consecutive columns and supernodes of the *L* and *U* factors might not be stored contiguously in memory. Thus, in addition to the pointers to the beginning of each column or supernode, also pointers to the end of each column or supernode are needed. The factorization is contained in structure *Imsl_f_super_lu_smp_factor* and its two sub-structures *Imsl_f_hbp_format* and *Imsl_f_scp_format*. Following is a short description of these structures:

Table 1.1 — Structure Imsl_f_hbp_format

| **Parameter** | **Data Type** | **Description** |
|---|---|---|
| nnz | *int* | The number of nonzeros in the matrix. |
| nzval | *float* * | Array of nonzero values packed by column. |
| rowind | *int* * | Array of row indices of the nonzeros. |
| colbeg | *int* * | Array of size ncol+1; colbeg[j] stores the location in nzval[] and rowind[], which starts column j. Element colbeg[ncol] points to the first free location in arrays nzval[] and rowind[]. |
| colend | *int* * | Array of size ncol; colend[j] stores the location in nzval[] and rowind[] which is one past the last element of column j. |

Table 1.2 — Structure Imsl_f_scp_format

| Parameter | Data Type | Description |
|---|---|---|
| nnz | *int* | The number of nonzeros in the supernodal matrix. |
| nsuper | *int* | The number of supernodes minus one. |
| nzval | *float* * | Array of nonzero values packed by column. |
| nzval_colbeg | *int* * | Array of size ncol+1; nzval_colbeg[j] points to the beginning of column j in nzval[]. Entry nzval_colbeg[ncol] points to the first free location in nzval[]. |
| nzval_colend | *int* * | Array of size ncol; nzval_colend[j] points to one past the last element of column j in nzval[]. |
| rowind | *int* * | Array of compressed row indices of the rectangular supernodes. |
| rowind_colbeg | *int* * | Array of size ncol+1; rowind_colbeg[j] points to the beginning of column j in rowind[]. Element rowind_colbeg[ncol] points to the first free location in rowind[]. |
| rowind_colend | *int* * | Array of size ncol; rowind_colend[j] points to one past the last element of column j in rowind[]. |
| col_to_sup | *int* * | Array of size ncol+1; col_to_sup[j] is the supernode number to which column j belongs. Only the first ncol entries in col_to_sup[] are defined. |
| sup_to_colbeg | *int* * | Array of size ncol+1; sup_to_colbeg[s] points to the first column of the *s*-th supernode; only the first nsuper+1 locations of this array are used. |
| sup_to_colend | *int* * | Array of size ncol; sup_to_colend[s] points to one past the last column of the *s*-th supernode. Only the first nsuper+1 locations of this array are used. |

Table 1.3 — Structure Imsl_f_super_lu_smp_factor

| Parameter | Data Type | Description |
|---|---|---|
| nrow | *int* | The number of rows of matrix *A*. |
| ncol | *int* | The number of columns of matrix *A*. |
| equilibration_method | *int* | The method used to equilibrate *A*:<br>0 – No equilibration.<br>1 – Row equilibration.<br>2 – Column equilibration.<br>3 – Both row and column equilibration. |
| rowscale | *float* * | Array of size nrow containing the row scale factors for *A*. |

Table 1.3 — Structure Imsl_f_super_lu_smp_factor

| | | |
|---|---|---|
| columnscale | *float* * | Array of size ncol containing the column scale factors for *A*. |
| rowperm | *int* * | Row permutation array of size nrow describing the row permutation matrix $P_r$. |
| colperm | *int* * | Column permutation array of size ncol describing the column permutation matrix $P_c$. |
| U | *Imsl_f_hbp_format* * | The part of the *U* factor of *A* outside the supernodal blocks, stored in Harwell-Boeing format. |
| L | *Imsl_f_scp_format* * | The *L* factor of *A*, stored in supernodal format as block lower triangular matrix. |

Structure *Imsl_d_super_lu_smp_factor* and its two sub-structures are defined similarly by replacing *float* with *double*, *Imsl_f_hbp_format* with *Imsl_d_hbp_format*, and *Imsl_f_scp_format* with *Imsl_d_scp_format* in their respective definitions.

In contrast to the sequential version, the numerical factorization phase of the *LU* decomposition is parallelized. Since a dynamic memory expansion as in the serial case is difficult to implement for the parallel code, the estimated sizes of array rowind for the *L* and of arrays rowind and nzval for the *U* factor (see structures *Imsl_f_scp_format* and *Imsl_f_hbp_format* above) must be predetermined by the user via elements 6 and 7 of the performance tuning array sp_ienv.

In order to ensure that the columns of each *L* supernode are stored contiguously in memory, a static or dynamic prediction scheme for the size of the *L* supernodes can be used. The static version, which function imsl_f_superlu_smp uses by default, exploits the observation that for any row permutation *P* in *PA* = *LU*, the nonzero structure of *L* is contained in that of the Householder matrix *H* from the Householder sparse *QR* factorization *A* = *QR*. Furthermore, it can be shown that each fundamental supernode in *L* is always contained in a fundamental supernode of *H*. Therefore, the storage requirement for the *L* supernodes and array nzval in the *L* factor respectively can be estimated and allocated prior to the factorization based on the size of the *H* supernodes. The algorithm used to compute the supernode partition and the size of the supernodes in *H* is almost linear in the number of nonzeros of matrix *A*.

In practice, the above static prediction scheme is quite tight for most problems. However, if the number of nonzeros in *H* greatly exceeds the number of nonzeros in *L*, the user can try a dynamic prediction scheme by setting optional argument IMSL_SNODE_PREDICTION to 1. This scheme still uses the supernode partition in *H*, but dynamically searches the supernodal graph of *L* to obtain a much tighter upper bound for the required storage. Use of the dynamic scheme requires the user to define the size of array nzval in the *L* factor via element 5 of the performance tuning array sp_ienv.

For a complete description of the parallel algorithm, see Demmel et al. (1999c).

# Examples

## Example 1

The LU factorization of the sparse 6×6 matrix

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

is computed.

Let $y = (1,2,3,4,5,6)^T$, so that $b_1 := Ay = (10,7,45,33,-34,31)^T$ and $b_2 := A^T y = (-9,8,39,13,1,21)^T$.

The *LU* factorization of *A* is used to solve the sparse linear systems $Ax = b_1$ and $A^T x = b_2$.

```
#include <imsl.h>

int 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 b1[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
    float b2[] = { -9.0, 8.0, 39.0, 13.0, 1.0, 21.0 };
    int n = 6, nz = 15;
    float *x = NULL;

    x = imsl_f_superlu_smp (n, nz, a, b1, 0);
    imsl_f_write_matrix ("solution to A*x = b1", 1, n, x, 0);
    imsl_free (x);

    x = imsl_f_superlu_smp (n, nz, a, b2, IMSL_TRANSPOSE, 1, 0);
    imsl_f_write_matrix ("solution to A^T*x = b2", 1, n, x, 0);
    imsl_free (x);
}
```

### Output

```
                    solution to A*x = b1
     1          2          3          4          5          6
     1          2          3          4          5          6

                   solution to A^T*x = b2
     1          2          3          4          5          6
     1          2          3          4          5          6
```

## Example 2

This example uses the matrix $A = E(1000,10)$ to show how the *LU* factorization of *A* can be used to solve a linear system with the same coefficient matrix *A* but different right-hand side. Maximum absolute errors are printed. After the computations, the space allocated for the *LU* factorization is freed via function `imsl_f_superlu_smp_factor_free`.

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

```
int main(){

    Imsl_f_sparse_elem *a = NULL;
    Imsl_f_super_lu_smp_factor lu_factor;
    float *b = NULL, *x = NULL, *mod_five = NULL, *mod_ten = NULL;
    float error_factor_solve, error_solve;
    int n = 1000, c = 10;
    int i, nz, index;

    /* Get the coefficient matrix */
    a = imsl_f_generate_test_coordinate (n, c, &nz, 0);

    /* Set two different predetermined solutions */
    mod_five = (float*) malloc (n*sizeof(*mod_five));
    mod_ten = (float*) malloc (n*sizeof(*mod_ten));
    for (i=0; i<n; i++) {
        mod_five[i] = (float) (i % 5);
        mod_ten[i] = (float) (i % 10);
    }

    /* Choose b so that x will approximate mod_five */
    b = (float *) imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_five, 0);

    /* Solve Ax = b */
    x = imsl_f_superlu_smp (n, nz, a, b,
        IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor, 0);

    /* Compute max absolute error */
    error_factor_solve = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);

    free (mod_five);
    imsl_free (b);
    imsl_free (x);

    /* Get new right hand side -- b = A * mod_ten */
    b = (float *) imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_ten,
        0);

    /* Use the previously computed factorization
    to solve Ax = b */
    x = imsl_f_superlu_smp (n, nz, a, b,
        IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
        IMSL_FACTOR_SOLVE, 2,
```

```
            0);
    error_solve = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_ten,
        IMSL_INF_NORM, &index,
        0);

    free (mod_ten);
    imsl_free (b);
    imsl_free (x);
    imsl_free (a);

    /* Free sparse LU structure */
    imsl_f_superlu_smp_factor_free (&lu_factor);

    /* Print errors */
    printf ("absolute error (factor/solve) = %e\n",
        error_factor_solve);
    printf ("absolute error (solve) = %e\n", error_solve);
}
```

### Output

```
absolute error (factor/solve) = 1.096725e-005
absolute error (solve) = 5.435944e-005
```

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| IMSL_SINGULAR_MATRIX | The input matrix is singular. |

# superlu_smp (complex)

more...

more...

Computes the *LU* factorization of a general complex sparse matrix by a left-looking column method using OpenMP parallelism and solves the complex sparse linear system of equations *Ax* = *b*.

## Synopsis

*#include* `<imsl.h>`

*f_complex* `*imsl_c_superlu_smp` (*int* `n`, *int* `nz`, *Imsl_c_sparse_elem* `a[]`, *f_complex* b[],…,0)

*void* `imsl_c_superlu_smp_factor_free` (*Imsl_c_super_lu_smp_factor* `*factor`)

The type *d_complex* functions are `imsl_z_superlu_smp` and `imsl_z_superlu_smp_factor_free`.

## Required Arguments

*int* `n` (Input)
The order of the input matrix.

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

*Imsl_c_sparse_elem* `a[]` (Input)
An array of length `nz` containing the location and value of each nonzero entry in the matrix. See the main "Introduction" chapter of this manual for an explanation of the Imsl_c_sparse_elem structure.

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*f_complex* *imsl_c_superlu_smp (*int* n, *int* nz, *Imsl_c_sparse_elem* a[], *f_complex* b[],

IMSL_EQUILIBRATE, *int* equilibrate,

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method,

IMSL_COLPERM_VECTOR, *int* permc[],

IMSL_TRANSPOSE, *int* transpose,

IMSL_ITERATIVE_REFINEMENT, *int* refine,

IMSL_FACTOR_SOLVE, *int* factsol,

IMSL_DIAG_PIVOT_THRESH, *float* diag_pivot_thresh,

IMSL_SNODE_PREDICTION, *int* snode_prediction,

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[],

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *f_complex* HB_values[],

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_c_super_lu_smp_factor* *lu_factor_supplied,

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_c_super_lu_smp_factor* *lu_factor_returned,

IMSL_CONDITION, *float* *condition,

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth,

IMSL_FORWARD_ERROR_BOUND, *float* *ferr,

IMSL_BACKWARD_ERROR, *float* *berr,

IMSL_RETURN_USER, *f_complex* x[],

0)

# Optional Arguments

IMSL_EQUILIBRATE, *int* equilibrate (Inputs)
Specifies if the input matrix *A* should be equilibrated before factorization.

| **equilibrate** | Description |
|---|---|
| 0 | Do not equilibrate *A* before factorization. |
| 1 | Equilibrate *A* before factorization. |

Default: equilibrate = 0

IMSL_COLUMN_ORDERING_METHOD, *Imsl_col_ordering* method (Input)

The column ordering method used to preserve sparsity prior to the factorization process. Select the ordering method by setting method to one of the following:

| method | Description |
|---|---|
| IMSL_NATURAL | Natural ordering, i.e.the column ordering of the input matrix. |
| IMSL_MMD_ATA | Minimum degree ordering on the structure of $A^T A$. |
| IMSL_MMD_AT_PLUS_A | Minimum degree ordering on the structure of $A^T + A$. |
| IMSL_COLAMD | Column approximate minimum degree ordering. |
| IMSL_PERMC | Use ordering given in permutation vector permc, which is input by the user through optional argument IMSL_COLPERM_VECTOR. Vector permc is a permutation of the numbers 0,1,...,n−1. |

Default: method = IMSL_COLAMD

IMSL_COLPERM_VECTOR, *int* permc[] (Input)

An array of length n which defines the permutation matrix $P_c$ before postordering. This argument is required if IMSL_COLUMN_ORDERING_METHOD with method = IMSL_PERMC is used. Otherwise, it is ignored.

IMSL_TRANSPOSE, *int* transpose (Input)

Indicates if the problem $Ax = b$ or one of the transposed problems $A^Tx = b$ or $A^Hx = b$ is to be solved.

| transpose | Description |
|---|---|
| 0 | Solve $Ax = b$. |
| 1 | Solve $A^Tx = b$.<br>This option can be used in conjunction with either of the options that supply the factorization. |
| 2 | Solve $A^Hx = b$.<br>This option can be used in conjunction with either of the options that supply the factorization. |

Default: transpose = 0.

IMSL_ITERATIVE_REFINEMENT, *int* refine (Input)

Indicates if iterative refinement is desired.

| refine | Description |
|---|---|
| 0 | No iterative refinement. |
| 1 | Do iterative refinement. |

Default: refine = 1.

IMSL_FACTOR_SOLVE, *int* factsol (Input)

Indicates if the *LU* factorization, the solution of a linear system, or both are to be computed.

| factsol | Description |
|---|---|
| 0 | Compute the *LU* factorization of the input matrix *A* and solve the system $Ax = b$. |
| 1 | Only compute the *LU* factorization of the input matrix and return.<br>The *LU* factorization is returned via optional argument IMSL_RETURN_SPARSE_LU_FACTOR.<br>Input argument b is ignored. |
| 2 | Only solve $Ax = b$ given the *LU* factorization of *A*.<br>The *LU* factorization of *A* must be supplied via optional argument IMSL_SUPPLY_SPARSE_LU_FACTOR.<br>Input argument a is ignored unless iterative refinement, computation of the condition number, or computation of the reciprocal pivot growth factor is required. |

Default: factsol = 0.

IMSL_DIAG_PIVOT_THRESH, *float* diag_pivot_thresh (Input)

Specifies the threshold used for a diagonal entry to be an acceptable pivot,

0.0 ≤ diag_pivot_thresh ≤ 1.0.

Default: diag_pivot_thresh = 1.0.

IMSL_SNODE_PREDICTION, *int* snode_prediction (Input)

Indicates which scheme is used to predict the number of nonzeros in the *L* supernodes.

| snode_prediction | Description |
|---|---|
| 0 | Use static scheme for the prediction of the number of nonzeros in the *L* supernodes. |
| 1 | Use dynamic scheme for the prediction of the number of nonzeros in the *L* supernodes. |

Default: snode_prediction = 0.

IMSL_PERFORMANCE_TUNING, *int* sp_ienv[] (Input)

An array of length 8 containing parameters that allow the user to tune the performance of the matrix factorization algorithm. The elements sp_ienv[i] must be positive for i = 0,…,4 and different from zero for i = 5,6,7.

| i | **Description of sp_ienv[i]** |
|---|---|
| 0 | The panel size.<br>Default: sp_ienv[0] = 10. |
| 1 | The relaxation parameter to control supernode amalgamation.<br>Default: sp_ienv[1] = 5. |
| 2 | The maximum allowable size for a supernode.<br>Default: sp_ienv[2] = 100. |
| 3 | The minimum row dimension to be used for 2D blocking.<br>*Default:* sp_ienv[3] = 200. |
| 4 | The minimum column dimension to be used for 2D blocking.<br>Default: sp_ienv[4] = 40. |
| 5 | The size of the array nzval to store the values of the *L* supernodes. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>This element of array sp_ienv is used only if a dynamic scheme for the prediction of the sizes of the *L* supernodes is used, i.e. if snode_prediction = 1.<br>Default: sp_ienv[5] = -20. |
| 6 | The size of the arrays rowind and nzval to store the columns in *U*. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>Default: sp_ienv[6] = -20. |
| 7 | The size of the array rowind to store the subscripts of the *L* supernodes. A negative number represents the fills growth factor, i.e. the product of its absolute magnitude and the number of nonzeros in the original matrix *A* will be used to allocate storage. A positive number represents the number of nonzeros for which storage will be allocated.<br>Default: sp_ienv[7] = -10. |

IMSL_CSC_FORMAT, *int* HB_col_ptr[], *int* HB_row_ind[], *f_complex* HB_values[] (Input)

Accept the coefficient matrix in compressed sparse column (CSC) format, as described in the Compressed Sparse Column (CSC) Format section of the “Introduction” chapter of this manual.

IMSL_SUPPLY_SPARSE_LU_FACTOR, *Imsl_c_super_lu_smp_factor* *lu_factor_supplied (Input)

The address of a structure of type *Imsl_c_super_lu_smp_factor* containing the *LU* factors of the input matrix computed with the IMSL_RETURN_SPARSE_LU_FACTOR option. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function imsl_c_superlu_smp_factor_free.

IMSL_RETURN_SPARSE_LU_FACTOR, *Imsl_c_super_lu_smp_factor* *lu_factor_returned (Output)
The address of a structure of type *Imsl_c_super_lu_smp_factor* containing the *LU* factorization of the input matrix. See the Description section for a definition of this structure. To free the memory allocated within this structure, use function imsl_c_superlu_smp_factor_free.

IMSL_CONDITION, *float* *condition (Output)
The estimate of the reciprocal condition number of matrix *A* after equilibration (if done).

IMSL_PIVOT_GROWTH_FACTOR, *float* *recip_pivot_growth (Output)
The reciprocal pivot growth factor:

$$\min_j \left\{ \left\| \left( P_r D_r A D_c P_c \right)_j \right\|_\infty / \left\| U_j \right\|_\infty \right\}.$$

If recip_pivot_growth is much less than 1, the stability of the *LU* factorization could be poor.

IMSL_FORWARD_ERROR_BOUND, *float* *ferr (Output)
The estimated forward error bound for the solution vector *x*. This option requires argument IMSL_ITERATIVE_REFINEMENT set to 1.

IMSL_BACKWARD_ERROR, *float* *berr (Output)
The componentwise relative backward error of the solution vector *x*. This option requires argument IMSL_ITERATIVE_REFINEMENT set to 1.

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

## Description

The steps imsl_c_superlu_smp uses to solve linear systems are identical to the steps described in the documentation of the serial version imsl_c_superlu.

Function imsl_c_superlu_smp uses a supernodal storage scheme for the *LU* factorization of matrix *A*. In contrast to the sequential version, the consecutive columns and supernodes of the *L* and *U* factors might not be stored contiguously in memory. Thus, in addition to the pointers to the beginning of each column or supernode, also pointers to the end of each column or supernode are needed. The factorization is contained in structure *Imsl_c_super_lu_smp_factor* and its two sub-structures *Imsl_c_hbp_format* and *Imsl_c_scp_format*. Following is a short description of these structures:

Table 1.4 — Structure `Imsl_c_hbp_format`

| Parameter | Data Type | Description |
|---|---|---|
| `nnz` | *int* | The number of nonzeros in the matrix. |
| `nzval` | *f_complex* * | Array of nonzero values packed by column. |
| `rowind` | *int* * | Array of row indices of the nonzeros. |
| `colbeg` | *int* * | Array of size `ncol+1`; `colbeg[j]` stores the location in `nzval[]` and `rowind[]`, which starts column `j`. Element `colbeg[ncol]` points to the first free location in arrays `nzval[]` and `rowind[]`. |
| `colend` | *int* * | Array of size `ncol`; `colend[j]` stores the location in `nzval[]` and `rowind[]`, which is one past the last element of column `j`. |

Table 1.5 — Structure `Imsl_c_scp_format`

| Parameter | Data Type | Description |
|---|---|---|
| `nnz` | *int* | The number of nonzeros in the supernodal matrix. |
| `nsuper` | *int* | The number of supernodes minus one. |
| `nzval` | *f_complex* * | Array of nonzero values packed by column. |
| `nzval_colbeg` | *int* * | Array of size `ncol+1`; `nzval_colbeg[j]` points to the beginning of column j in `nzval[]`. Entry `nzval_colbeg[ncol]` points to the first free location in `nzval[]`. |
| `nzval_colend` | *int* * | Array of size `ncol`; `nzval_colend[j]` points to one past the last element of column `j` in `nzval[]`. |
| `rowind` | *int* * | Array of compressed row indices of the rect-angular supernodes. |
| `rowind_colbeg` | *int* * | Array of size `ncol+1`; `rowind_colbeg[j]` points to the beginning of column j in `rowind[]`. Element `rowind_colbeg[ncol]` points to the first free location in `rowind[]`. |
| `rowind_colend` | *int* * | Array of size `ncol`; `rowind_colend[j]` points to one past the last element of column `j` in `rowind[]`. |
| `col_to_sup` | *int* * | Array of size `ncol+1`; `col_to_sup[j]` is the supernode number to which column `j` belongs. Only the first `ncol` entries in `col_to_sup[]` are defined. |

Table 1.5 — Structure Imsl_c_scp_format

| Parameter | Data Type | Description |
|---|---|---|
| sup_to_colbeg | *int* * | Array of size ncol+1; sup_to_colbeg[s] points to the first column of the ***s***-th supernode; only the first nsuper+1 locations of this array are used. |
| sup_to_colend | *int* * | Array of size ncol; sup_to_colend[s] points to one past the last column of the ***s***-th supernode. Only the first nsuper+1 locations of this array are used. |

Table 1.6 — Structure Imsl_c_super_lu_smp_factor

| Parameter | Data Type | Description |
|---|---|---|
| nrow | *int* | The number of rows of matrix ***A***. |
| ncol | *int* | The number of columns of matrix ***A***. |
| equilibration_method | *int* | The method used to equilibrate ***A***:<br>0 – No equilibration.<br>1 – Row equilibration.<br>2 – Column equilibration.<br>3 – Both row and column equilibration. |
| rowscale | *float* * | Array of size nrow containing the row scale factors for ***A***. |
| columnscale | *float* * | Array of size ncol containing the column scale factors for ***A***. |
| rowperm | *int* * | Row permutation array of size nrow describing the row permutation matrix $P_r$. |
| colperm | *int* * | Column permutation array of size ncol describing the column permutation matrix $P_c$. |
| U | *Imsl_c_hbp_format* * | The part of the ***U*** factor of ***A*** outside the supernodal blocks, stored in Harwell-Boeing format. |
| L | *Imsl_c_scp_format* * | The ***L*** factor of ***A***, stored in supernodal format as block lower triangular matrix. |

Structure *Imsl_z_super_lu_smp_factor* and its two sub-structures are defined similarly by replacing *float* with *double*, *f_complex* with *d_complex*, *Imsl_c_hbp_format* with *Imsl_z_hbp_format*, and *Imsl_c_scp_format* with *Imsl_z_scp_format* in their respective definitions.

In contrast to the sequential version, the numerical factorization phase of the *LU* decomposition is parallelized. Since a dynamic memory expansion as in the serial case is difficult to implement for the parallel code, the estimated sizes of array rowind for the *L* and of arrays rowind and nzval for the *U* factor (see structures *Imsl_c_scp_format* and *Imsl_c_hbp_format* above) must be predetermined by the user via elements 6 and 7 of the performance tuning array sp_ienv.

In order to ensure that the columns of each *L* supernode are stored contiguously in memory, a static or dynamic prediction scheme for the size of the *L* supernodes can be used. The static version, which function `imsl_c_superlu_smp` uses by default, exploits the observation that for any row permutation *P* in *PA* = *LU*, the nonzero structure of *L* is contained in that of the Householder matrix *H* from the Householder sparse *QR* factorization *A* = *QR*. Furthermore, it can be shown that each fundamental supernode in *L* is always contained in a fundamental supernode of *H*. Therefore, the storage requirement for the *L* supernodes and array `nzval` in the *L* factor respectively can be estimated and allocated prior to the factorization based on the size of the *H* supernodes. The algorithm used to compute the supernode partition and the size of the supernodes in *H* is almost linear in the number of nonzeros of matrix *A*.

In practice, the above static prediction scheme is quite tight for most problems. However, if the number of nonzeros in *H* greatly exceeds the number of nonzeros in *L*, the user can try a dynamic prediction scheme by setting optional argument `IMSL_SNODE_PREDICTION` to 1. This scheme still uses the supernode partition in *H*, but dynamically searches the supernodal graph of *L* to obtain a much tighter upper bound for the required storage. Use of the dynamic scheme requires the user to define the size of array `nzval` in the *L* factor via element 5 of the performance tuning array `sp_ienv`.

For a complete description of the parallel algorithm, see Demmel et al. (1999c).

# Examples

## Example 1

The *LU* factorization of the sparse complex 6×6 matrix

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

is computed. Let

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

so that

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

$$b_1 := A^T y = (-112+54i, -58+46i, 12i, -51+5i, 34+78i, -94+60i)^T$$

and

$$b_2 := A^H y = (54-112i, 46-58i, 12, 5-51i, 78+34i, 60-94i)^T$$

The *LU* factorization of $A$ is used to solve the sparse complex linear systems $Ax=b$, $A^Tx=b_1$, and $A^Hx=b_2$.

```
#include <imsl.h>

int 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[] = {{3.0, 17.0}, {-19.0, 5.0}, {6.0, 18.0},
       {-38.0, 32.0}, {-63.0, 49.0}, {-57.0, 83.0}};
```

```
    f_complex b1[] = {{-112.0,54.0}, {-58.0,46.0}, {0.0,12.0},
        {-51.0,5.0}, {34.0,78.0}, {-94.0,60.0}};

    f_complex b2[] = {{54.0,-112.0}, {46.0, -58.0}, {12.0, 0.0},
        {5.0, -51.0}, {78.0, 34.0}, {60.0, -94.0}};

    int n = 6, nz = 15;
    f_complex *x = NULL;

    x = imsl_c_superlu_smp (n, nz, a, b, 0);
    imsl_c_write_matrix ("solution to A*x = b", n, 1, x, 0);
    imsl_free (x);

    x = imsl_c_superlu_smp (n, nz, a, b1, IMSL_TRANSPOSE, 1, 0);
    imsl_c_write_matrix ("solution to A^T*x = b1", n, 1, x, 0);
    imsl_free (x);

    x = imsl_c_superlu_smp (n, nz, a, b2, IMSL_TRANSPOSE, 2, 0);
    imsl_c_write_matrix ("solution to A^H*x = b2", n, 1, x, 0);
    imsl_free (x);
}
```

### Output

```
    solution to A*x = b
1 (          1,          1)
2 (          2,          2)
3 (          3,          3)
4 (          4,          4)
5 (          5,          5)
6 (          6,          6)

  solution to A^T*x = b1
1 (          1,          1)
2 (          2,          2)
3 (          3,          3)
4 (          4,          4)
5 (          5,          5)
6 (          6,          6)

  solution to A^H*x = b2
1 (          1,          1)
2 (          2,          2)
3 (          3,          3)
4 (          4,          4)
5 (          5,          5)
6 (          6,          6)
```

## Example 2

This example uses the matrix $A = E(1000,10)$ to show how the *LU* factorization of *A* can be used to solve a linear system with the same coefficient matrix *A* but different right-hand side. Maximum absolute errors are printed. After the computations, the space allocated for the *LU* factorization is freed via function `imsl_c_superlu_smp_factor_free`.

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

int main()
{
    Imsl_c_sparse_elem *a = NULL;
    Imsl_c_super_lu_smp_factor lu_factor;
    f_complex *b = NULL, *x = NULL, *mod_five = NULL, *mod_ten = NULL;
    float error_factor_solve, error_solve;
    int n = 1000, c = 10;
    int i, nz, index;
    /* Get the coefficient matrix */
    a = imsl_c_generate_test_coordinate (n, c, &nz, 0);
    /* Set two different predetermined solutions */
    mod_five = (f_complex*) malloc (n*sizeof(*mod_five));
    mod_ten = (f_complex*) malloc (n*sizeof(*mod_ten));
    for (i=0; i<n; i++) {
        mod_five[i] = imsl_cf_convert ((float)(i % 5), 0.0);
        mod_ten[i] = imsl_cf_convert ((float)(i % 10), 0.0);
    }
    /* Choose b so that x will approximate mod_five */
    b = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_five,
        0);
    /* Solve Ax = b */
    x = imsl_c_superlu_smp (n, nz, a, b,
        IMSL_RETURN_SPARSE_LU_FACTOR, &lu_factor,
        0);
    /* Compute max absolute error */
    error_factor_solve = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
    free (mod_five);
    imsl_free (b);
    imsl_free (x);
    /* Get new right hand side -- b = A * mod_ten */
    b = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_ten,
        0);
```

```
    /* Use the previously computed factorization to solve Ax = b */
    x = imsl_c_superlu_smp (n, nz, a, b,
        IMSL_SUPPLY_SPARSE_LU_FACTOR, &lu_factor,
        IMSL_FACTOR_SOLVE, 2,
        0);
    error_solve = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_ten,
        IMSL_INF_NORM, &index,
        0);
    free (mod_ten);
    imsl_free (b);
    imsl_free (x);
    imsl_free (a);
    /* Free sparse LU structure */
    imsl_c_superlu_smp_factor_free (&lu_factor);
    /* Print errors */
    printf ("absolute error (factor/solve) = %e\n",
        error_factor_solve);
    printf ("absolute error (solve) = %e\n", error_solve);
}
```

### Output

```
absolute error (factor/solve) = 9.581556e-007
absolute error (solve) = 2.017572e-006
```

## Warning Errors

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

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_MATRIX` | The input matrix is singular. |

# lin_sol_posdef_coordinate

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

## Synopsis

*#include* <imsl.h>

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

*void* imsl_free_symbolic_factor (*Imsl_symbolic_factor* *sym_factor)

*void* imsl_f_free_numeric_factor (*Imsl_f_numeric_factor* *num_factor)

The type ***double*** functions are imsl_d_lin_sol_posdef_coordinate and imsl_d_free_numeric_factor.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_lin_sol_posdef_coordinate (*int* n, *int* nz, *Imsl_f_sparse_elem* *a, *float* *b, IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor,

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor,

IMSL_SYMBOLIC_FACTOR_ONLY,

IMSL_RETURN_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor,

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor,

IMSL_NUMERIC_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_MULTIFRONTAL_FACTORIZATION,

IMSL_RETURN_USER, *float* x[],

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *small_element,

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *largest_element,

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values,

0)

## Optional Arguments

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor (Output)

A pointer to a structure of type *Imsl_symbolic_factor* containing, on return, the symbolic factorization of the input matrix. A detailed description of the *Imsl_symbolic_factor* structure is given in the following table:

| Parameter | Data Type | Description |
|---|---|---|
| nzsub | *int* ** | A pointer to an array containing the compressed row subscripts of the non-zero off-diagonal elements of the Cholesky factor. |
| xnzsub | *int* ** | A pointer to an array of length n + 1 containing indices for *nzsub. The row subscripts for the non-zeros in column *j* of the Cholesky factor are stored consecutively beginning with (*nzsub)[(*xnzsub)[j]]. |
| maxsub | *int* | The number of elements in array *nzsub that are used as subscripts. Note that the size of *nzsub can be larger than maxsub. |
| xlnz | *int* ** | A pointer to an array of length n + 1 containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array *alnz (For a description of alnz, see the description section of optional argument IMSL_RETURN_NUMERIC_FACTOR). For column *j* of the factor matrix, the starting and stopping indices of *alnz are stored in (*xlnz)[j] and (*xlnz)[j + 1] respectively. |
| maxlnz | *int* | The number of non-zero off-diagonal elements in the Cholesky factor. |
| perm | *int* ** | A pointer to an array of length n containing the permutation vector. |
| invp | *int* ** | A pointer to an array of length n containing the inverse permutation vector. |
| multifrontal_space | *int* | The required size of working storage for the stack of frontal matrices. If no multifrontal factorization is used, then this variable is set to zero. |

To free the memory allocated within this structure, use function imsl_free_symbolic_factor.

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor (Input)

A pointer to a structure of type *Imsl_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by imsl_f_lin_sol_posdef_coordinate with the IMSL_RETURN_SYMBOLIC_FACTOR option. The structure is described in the IMSL_RETURN_SYMBOLIC_FACTOR optional argument description. To free the memory allocated within this structure, use function imsl_free_symbolic_factor.

IMSL_SYMBOLIC_FACTOR_ONLY,

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

`IMSL_RETURN_NUMERIC_FACTOR`, *Imsl_f_numeric_factor* `*num_factor` (Output)

A pointer to a structure of type *Imsl_f_numeric_factor* containing, on return, the numeric factorization of the input matrix. A detailed description of the *Imsl_f_numeric_factor* structure is given in the following table:

| **Parameter** | **Data Type** | **Description** |
|---|---|---|
| `nzsub` | *int* `**` | A pointer to an array containing the row subscripts for the non-zero off-diagonal elements of the Cholesky factor. This array is allocated to be of length `nz` but all elements of the array may not be used. |
| `xnzsub` | *int* `**` | A pointer to an array of length `n` + 1 containing indices for `nzsub`. The row subscripts for the non-zeros in column *j* of the cholesky factor are stored consecutively beginning with `nzsub[xnzsub[j]]`. |
| `xlnz` | *int* `**` | A pointer to an array of length `n` + 1 containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array `alnz`. For column *j* of the factor matrix, the starting and stopping indices of `alnz` are stored in `xlnz[j]` and `xlnz[j + 1]` respectively. |
| `alnz` | *float* `**` | A pointer to an array containing the non-zero off-diagonal elements of the Cholesky factor. |
| `perm` | *int* `**` | A pointer to an array of length `n` containing the permutation vector. |
| `diag` | *float* `**` | A pointer to an array of length `n` containing the diagonal elements of the Cholesky factor. |

Let *L* be the Cholesky factor of *a* and `num_nonzeros` be the number of nonzeros in *L*. In the structure described above, the diagonal elements of *L* are stored in `diag`. The off-diagonal non-zero elements of *L* are stored in `alnz`. The starting and stopping indices to use to extract the non-zero elements of *L* from `alnz` for column *j* are stored in `xlnz[`*j*`]` and `xlnz[`*j + 1*`]` respectively. The row indices of the non–zero elements of *L* are contained in `nzsub`. `xnzsub[`*i*`]` contains the index of `nzsub` from which one should start to extract the row indices for *L* for column *i*. This is best illustrated by the following code fragment which reconstructs the lower triangle of the factor matrix *L* from the components of the above structure:

```
Imsl_f_numeric_factor numfctr;
.
.
.
for (i = 0; i < n; i++){
    L[i][i] = (*numfctr.diag)[i];
    if ((*numfctr.xlnz)[i] > (num_nonzeros-n)) continue;
    start = (*numfctr.xlnz)[i]-1;
    stop = (*numfctr.xlnz)[i+1]-1;
    k = (*numfctr.xnzsub)[i]-1;
    for (j = start; j < stop; j++){
```

```
            L[(*numfctr.nzsub)[k]-1][i] = (*numfctr.alnz)[j];
            k++;
        }
    }
```

To free the memory allocated within this structure, use function `imsl_f_free_numeric_factor`.

`IMSL_SUPPLY_NUMERIC_FACTOR`, *Imsl_f_numeric_factor* `*num_factor` (Input)

A pointer to a structure of type *Imsl_f_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by `imsl_f_lin_sol_posdef_coordinate` with the `IMSL_RETURN_NUMERIC_FACTOR` option. The structure is described in the `IMSL_RETURN_NUMERIC_FACTOR` optional argument description.

To free the memory allocated within this structure, use function `imsl_f_free_numeric_factor`.

`IMSL_NUMERIC_FACTOR_ONLY`,

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

`IMSL_SOLVE_ONLY`,

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

`IMSL_MULTIFRONTAL_FACTORIZATION`,

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

`IMSL_RETURN_USER`, *float* `x[]` (Output)

A user-allocated array of length *n* containing the solution *x*.

`IMSL_SMALLEST_DIAGONAL_ELEMENT`, *float* `*small_element` (Output)

A pointer to a scalar containing the smallest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to `imsl_f_lin_sol_posdef_coordinate`.

`IMSL_LARGEST_DIAGONAL_ELEMENT`, *float* `*large_element` (Output)

A pointer to a scalar containing the largest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to `imsl_f_lin_sol_posdef_coordinate`.

`IMSL_NUM_NONZEROS_IN_FACTOR`, *int* `*num_nonzeros` (Output)

A pointer to a scalar containing the total number of nonzeros in the factor.

`IMSL_CSC_FORMAT`, *int* `*col_ptr`, *int* `*row_ind`, *float* `*values` (Input)

Accept the coefficient matrix in Compressed Sparse Column (CSC) Format. See the “Matrix Storage Modes” section of the “Introduction” at the beginning of this manual for a discussion of this storage scheme.

## Description

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

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

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

$$PAP^{\mathrm{T}} = LL^{\mathrm{T}}$$

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

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

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

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

$$Ly_1 = Pb$$

$$L^{\mathrm{T}}y_2 = y_1$$

$$x = P^{\mathrm{T}}y_2$$

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

# Examples

## Example 1

As an example consider the 5 × 5 coefficient matrix:

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

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

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

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

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

```
#include <imsl.h>

int main()
{
   Imsl_f_sparse_elem a[] =
       {0, 0, 10.0,
        1, 1, 20.0,
        2, 0, 1.0,
        2, 2, 30.0,
        3, 2, 4.0,
        3, 3, 40.0,
        4, 0, 2.0,
        4, 1, 3.0,
        4, 3, 5.0,
        4, 4, 50.0};

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

   x = imsl_f_lin_sol_posdef_coordinate (n, nz, a, b,
       0);
```

```
    imsl_f_write_matrix ("solution", 1, n, x,
        0);
    imsl_free (x);
}
```

### Output

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

## Example 2

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

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

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

    ic = 50;
    n = ic*ic;

    /* Generate two right hand sides */
    b_1 = imsl_f_random_uniform (n*sizeof(*b_1),
        0);
    b_2 = imsl_f_random_uniform (n*sizeof(*b_2),
        0);

    /* Build coefficient matrix a */
    a = imsl_f_generate_test_coordinate (n, ic, &nz,
        IMSL_SYMMETRIC_STORAGE,
        0);

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

```
    time_1 = imsl_ctime ();

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

    time_1 = imsl_ctime () - time_1;

    /* Now solve Ax_2 = b_2 given the numeric
    factorization */
    time_2 = imsl_ctime ();

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

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

### Output

```
time_2/time_1 = 0.037037
```

# lin_sol_posdef_coordinate (complex)

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

## Synopsis

*#include* <imsl.h>

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

*void* imsl_free_symbolic_factor (*Imsl_symbolic_factor* *sym_factor)

*void* imsl_c_free_numeric_factor (*Imsl_c_numeric_factor* *num_factor)

The type *d_complex* functions are imsl_z_lin_sol_posdef_coordinate and imsl_z_free_numeric_factor.

## Required Arguments

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

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_lin_sol_posdef_coordinate (*int* n, *int* nz, *Imsl_c_sparse_elem* *a,
*f_complex* *b,

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor,

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor,

IMSL_SYMBOLIC_FACTOR_ONLY,

IMSL_RETURN_NUMERIC_FACTOR, *Imsl_c_numeric_factor* *num_factor,

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_c_numeric_factor* *num_factor,

IMSL_NUMERIC_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_MULTIFRONTAL_FACTORIZATION,

IMSL_RETURN_USER, *f_complex* x[],

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *small_element,

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *largest_element,

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_CSC_FORMAT, *int* *col_ptr, *int* *row_ind, *float* *values,

0)

## Optional Arguments

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor (Output)

A pointer to a structure of type *Imsl_symbolic_factor* containing, on return, the symbolic factorization of the input matrix. A detailed description of the *Imsl_symbolic_factor* structure is given in the following table:

| Parameter | Data Type | Description |
|---|---|---|
| nzsub | *int* ** | A pointer to an array containing the compressed row subscripts of the non-zero off-diagonal elements of the Cholesky factor. |
| xnzsub | *int* ** | A pointer to an array of length n + 1 containing indices for *nzsub. The row subscripts for the non-zeros in column *j* of the Cholesky factor are stored consecutively beginning with (*nzsub)[(*xnzsub)[j]]. |
| maxsub | *int* | The number of elements in array *nzsub that are used as subscripts. Note that the size of *nzsub can be larger than maxsub. |
| xlnz | *int* ** | A pointer to an array of length n + 1 containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array *alnz (For a description of alnz, see the description section of optional argument IMSL_RETURN_NUMERIC_FACTOR). For column *j* of the factor matrix, the starting and stopping indices of *alnz are stored in (*xlnz)[j] and (*xlnz)[j+1] respectively. |
| maxlnz | *int* | The number of non-zero off-diagonal elements in the Cholesky factor. |
| perm | *int* ** | A pointer to an array of length n containing the permutation vector. |
| invp | *int* ** | A pointer to an array of length n containing the inverse permutation vector. |
| multifrontal_space | *int* | The required size of working storage for the stack of frontal matrices. If no multifrontal factorization is used, then this variable is set to zero. |

To free the memory allocated within this structure, use function imsl_free_symbolic_factor.

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_symbolic_factor* *sym_factor (Input)

A pointer to a structure of type *Imsl_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by imsl_c_lin_sol_posdef_coordinate with the IMSL_RETURN_SYMBOLIC_FACTOR option. The structure is described in the IMSL_RETURN_SYMBOLIC_FACTOR optional argument description. To free the memory allocated within this structure, use function imsl_free_symbolic_factor.

IMSL_SYMBOLIC_FACTOR_ONLY,

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

`IMSL_RETURN_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Output)

A pointer to a structure of type *Imsl_c_numeric_factor* containing, on return, the numeric factorization of the input matrix. A detailed description of the *Imsl_c_numeric_factor* structure is given in the following table:

| **Parameter** | **Data Type** | **Description** |
|---|---|---|
| `nzsub` | `int **` | A pointer to an array containing the row subscripts for the non-zero off-diagonal elements of the Cholesky factor. This array is allocated to be of length `nz` but all elements of the array may not be used. |
| `xnzsub` | `int **` | A pointer to an array of length `n` + 1 containing indices for `nzsub`. The row subscripts for the non-zeros in column *j* of the Cholesky factor are stored consecutively beginning with `nzsub[xnzsub[j]]`. |
| `xlnz` | `int **` | A pointer to an array of length `n` + 1 containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array `alnz`. For column *j* of the factor matrix, the starting and stopping indices of `alnz` are stored in `xlnz[j]` and `xlnz[j + 1]` respectively. |
| `alnz` | `f_complex **` | A pointer to an array containing the non-zero off-diagonal elements of the Cholesky factor. |
| `perm` | `int **` | A pointer to an array of length `n` containing the permutation vector. |
| `diag` | `f_complex **` | A pointer to an array of length `n` containing the diagonal elements of the Cholesky factor. |

Let *L* be the Cholesky factor of *a* and `num_nonzeros` be the number of nonzeros in *L*. In the structure described above, the diagonal elements of *L* are stored in `diag`. The off-diagonal non-zero elements of *L* are stored in `alnz`. The starting and stopping indices to use to extract the non-zero elements of *L* from `alnz` for column *j* are stored in `xlnz`[*j*] and `xlnz`[*j + 1*] respectively. The row indices of the elements of *L* which are non-zero are contained in `nzsub`. `xnzsub`[*i*] contains the index of `nzsub` from which one should start to extract the row indices for *L* for column *i*. This is best illustrated by the following code fragment which reconstructs the lower triangle of the factor matrix *L* from the components of the above structure:

```
Imsl_c_numeric_factor numfctr;
.
.
.
for (i = 0; i < n; i++){
```

```
        L[i][i] = (*numfctr.diag)[i];
        if ((*numfctr.xlnz)[i] > (num_nonzeros-n)) continue;
        start = (*numfctr.xlnz)[i]-1;
        stop = (*numfctr.xlnz)[i+1]-1;
        k = (*numfctr.xnzsub)[i]-1;
        for (j = start; j < stop; j++){
            L[(*numfctr.nzsub)[k]-1][i] = (*numfctr.alnz)[j];
            k++;
        }
    }
```

To free the memory allocated within this structure, use function `imsl_c_free_numeric_factor`.

`IMSL_SUPPLY_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Input)

A pointer to a structure of type *Imsl_c_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by `imsl_c_lin_sol_posdef_coordinate` with the `IMSL_RETURN_NUMERIC_FACTOR` option. The structure is described in the `IMSL_RETURN_NUMERIC_FACTOR` optional argument desription.

To free the memory allocated within this structure, use function `imsl_c_free_numeric_factor`.

`IMSL_NUMERIC_FACTOR_ONLY`,

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

`IMSL_SOLVE_ONLY`,

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

`IMSL_MULTIFRONTAL_FACTORIZATION`,

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

`IMSL_RETURN_USER`, *f_complex* `x[]` (Output)

A user-allocated array of length $n$ containing the solution $x$.

`IMSL_SMALLEST_DIAGONAL_ELEMENT`, *float* `*small_element` (Output)

A pointer to a scalar containing the smallest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to `imsl_c_lin_sol_posdef_coordinate`.

`IMSL_LARGEST_DIAGONAL_ELEMENT`, *float* `*large_element` (Output)

A pointer to a scalar containing the largest diagonal element that occurred during the numeric factorization. This option is valid only if the numeric factorization is computed during this call to `imsl_c_lin_sol_posdef_coordinate`.

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

`IMSL_CSC_FORMAT`, *int* `*col_ptr`, *int* `*row_ind`, *float* `*values` (Input)
Accept the coefficient matrix in Compressed Sparse Column (CSC) Format. See the "Matrix Storage Modes" section of the "Introduction" at the beginning of this manual for a discussion of this storage scheme.

## Description

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

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

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

$$PAP^{\mathrm{T}} = LL^{\mathrm{H}}$$

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

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

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

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

$$Ly_1 = Pb$$

$$L^{\mathrm{H}}y_2 = y_1$$

$$x = P^{\mathrm{T}}y_2$$

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

# Examples

## Example 1

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

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

Let $x^T = (1 + i, 2 + 2i, 3 + 3i)$ so that $Ax = (-2 + 2i, 5 + 15i, 36 + 28i)^T$. The number of nonzeros in the lower triangle is nz = 5.

```
#include <imsl.h>

int main()
{
   Imsl_c_sparse_elem a[] = {0, 0, {2.0, 0.0},
                            1, 1, {4.0, 0.0},
                            2, 2, {10.0, 0.0},
                            1, 0, {-1.0, -1.0},
                            2, 1, {1.0, -2.0}};

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

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

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

   imsl_free (x);
}
```

### Output

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

### Example 2

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

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

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

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

   /* Generate two right hand sides */
   rand_vec = imsl_f_random_uniform (4*n*sizeof(*rand_vec),
       0);

   for (i=0; i<n; i++) {
       b_1[i].re = rand_vec[index++];
       b_1[i].im = rand_vec[index++];
       b_2[i].re = rand_vec[index++];
       b_2[i].im = rand_vec[index++];
   }

   /* Build coefficient matrix a */
   a = imsl_c_generate_test_coordinate (n, ic, &nz,
       IMSL_SYMMETRIC_STORAGE,
       0);

   /* Now solve Ax_1 = b_1 and return the numeric factorization */
   time_1 = imsl_ctime ();

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

   time_1 = imsl_ctime () - time_1;

   /* Now solve Ax_2 = b_2 given the numeric factorization */
   time_2 = imsl_ctime ();
```

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

   time_2 = imsl_ctime () - time_2;

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

### Output

```
time_2/time_1 = 0.096386
```

# sparse_cholesky_smp

more...

more...

Computes the Cholesky factorization of a sparse real symmetric positive definite matrix *A* by an OpenMP parallelized supernodal algorithm and solves the sparse real positive definite system of linear equations *Ax* = *b*.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_sparse_cholesky_smp (*int* n, *int* nz, *Imsl_f_sparse_elem* a[], *float* b[], …, 0)

*void* imsl_free_snodal_symbolic_factor (*Imsl_snodal_symbolic_factor* *sym_factor)

*void* imsl_f_free_numeric_factor (*Imsl_f_numeric_factor* *num_factor)

The type ***double*** functions are imsl_d_sparse_cholesky_smp and imsl_d_free_numeric_factor.

## Required Arguments

*int* n (Input)
The order of the input matrix.

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

*Imsl_f_sparse_elem* a[] (Input)
An array of length nz containing the location and value of each nonzero entry in the lower triangle of the matrix.

*float* b[] (Input)
An array of length n containing the right-hand side.

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_sparse_cholesky_smp (*int* n, *int* nz, *Imsl_f_sparse_elem* a[], *float* b[],

IMSL_CSC_FORMAT, *int* col_ptr[], *int* row_ind[], *float* values[],

IMSL_PREORDERING, *int* preorder,

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor,

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor,

IMSL_SYMBOLIC_FACTOR_ONLY,

IMSL_RETURN_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor,

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor,

IMSL_NUMERIC_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *smallest_element,

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *largest_element,

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_RETURN_USER, *float* x[],

0)

## Optional Arguments

IMSL_CSC_FORMAT, *int* col_ptr[], *int* row_ind[], *float* values[] (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format, as described in the Compressed Sparse Column (CSC) Format section of the "Introduction" chapter of this manual.

IMSL_PREORDERING, *int* preorder (Input)
The variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of matrix *A*:

| preorder | Method |
|---|---|
| 0 | George and Liu's Quotient Minimum Degree algorithm. |
| 1 | A variant of George and Liu's Quotient Minimum Degree algorithm using a preprocessing phase and external degrees. |

Default: preorder = 0.

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor (Output)

A pointer to a structure of type *Imsl_snodal_symbolic_factor* containing, on return, the supernodal symbolic factorization of the input matrix. A detailed description of the *Imsl_snodal_symbolic_factor* structure is given in the following table:

Table 1.7 — Structure Imsl_snodal_symbolic_factor

| **Parameter** | **Data Type** | **Description** |
|---|---|---|
| nzsub | *int* ** | A pointer to an array containing the compressed row subscripts of the non-zero off-diagonal elements of the Cholesky factor. |
| xnzsub | *int* ** | A pointer to an array of length n+1 containing indices for *nzsub. The row subscripts for the non-zeros in column j of the Cholesky factor are stored consecutively beginning with (*nzsub)[(*xnzsub)[j]]. |
| maxsub | *int* | The number of elements in array *nzsub that are used as subscripts. Note that the size of *nzsub can be larger than maxsub. |
| xlnz | *int* ** | A pointer to an array of length n+1 containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array *alnz (For a description of alnz, see the description section of optional argument IMSL_RETURN_NUMERIC_FACTOR). For column j of the factor matrix, the starting and stopping indices of *alnz are stored in (*xlnz)[j] and (*xlnz)[j+1] respectively. |
| maxlnz | *int* | The number of non-zero off-diagonal elements in the Cholesky factor. |
| perm | *int* ** | A pointer to an array of length n containing the permutation vector. |
| invp | *int* ** | A pointer to an array of length n containing the inverse permutation vector. |
| multifrontal_space | *int* | This variable is not used in the current implementation. |
| nsuper | *int* | The number of supernodes in the Cholesky factor. |
| snode | *int* ** | A pointer to an array of length n. Element (*snode)[j] contains the number of the fundamental supernode to which column j belongs. |
| snode_ptr | *int* ** | A pointer to an array of length nsuper+1 containing the start column of each supernode. |
| nleaves | *int* | The number of leaves in the postordered elimination tree of the symmetrically permuted input matrix *A*. |
| etree_leaves | *int* ** | A pointer to an array of length nleaves+1 containing the leaves of the elimination tree. |

To free the memory allocated within this structure, use function imsl_free_snodal_symbolic_factor.

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor (Input)
A pointer to a structure of type *Imsl_snodal_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by imsl_f_sparse_cholesky_smp with the IMSL_RETURN_SYMBOLIC_FACTOR option. The structure is described in the IMSL_RETURN_SYMBOLIC_FACTOR optional argument description.
To free the memory allocated within this structure, use function imsl_free_snodal_symbolic_factor.

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

IMSL_RETURN_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor (Output)
A pointer to a structure of type *Imsl_f_numeric_factor* containing, on return, the numeric factorization of the input matrix. A detailed description of the *Imsl_f_numeric_factor* structure is given in the IMSL_RETURN_NUMERIC_FACTOR optional argument description of function imsl_f_lin_sol_posdef_coordinate. To free the memory allocated within this structure, use function imsl_f_free_numeric_factor.

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_f_numeric_factor* *num_factor (Input)
A pointer to a structure of type *Imsl_f_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by imsl_f_sparse_cholesky_smp with the IMSL_RETURN_NUMERIC_FACTOR option. The structure is described in the IMSL_RETURN_NUMERIC_FACTOR optional argument description of function imsl_f_lin_sol_posdef_coordinate.
To free the memory allocated within this structure, use function imsl_f_free_numeric_factor.

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

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

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

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

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

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

# Description

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

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

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

$$PAP^{\mathrm{T}} = LL^{\mathrm{T}}$$

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

The numerical factorization is an implementation of a parallel supernodal algorithm that combines a left-looking and a right-looking column computation scheme. This algorithm is described in detail in Rauber et al. (1999).

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

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

$$Ly_1 = Pb$$

$$L^{\mathrm{T}}y_2 = y_1$$

$$x = P^{\mathrm{T}}y_2$$

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

# Examples

## Example 1

Consider the 5 × 5 coefficient matrix *A*,

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

The number of nonzeros in the lower triangle of *A* is `nz` = 10. We construct the solution $x^T = (5, 4, 3, 2, 1)$ to the system $Ax = b$ by setting $b := Ax = (55, 83, 103, 97, 82)^T$. The solution is computed and printed.

```
#include <imsl.h>

int main()
{
   Imsl_f_sparse_elem a[] =
       {0, 0, 10.0,
        1, 1, 20.0,
        2, 0, 1.0,
        2, 2, 30.0,
        3, 2, 4.0,
        3, 3, 40.0,
        4, 0, 2.0,
        4, 1, 3.0,
        4, 3, 5.0,
        4, 4, 50.0};

   float b[] = {55.0, 83.0, 103.0, 97.0, 82.0};
   int   n = 5, nz = 10;
   float *x = NULL;

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

   imsl_f_write_matrix ("solution", 1, n, x, 0);
   imsl_free (x);
}
```

### Output

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

## Example 2

This example shows how a symbolic factor can be re-used. At first, the system *Ax* = *b* with *A* = *E*(2500, 50) is solved and the symbolic factorization of *A* is returned. Then, the system *Cy* = *d* with *C = A*+2***I*, *I* the identity matrix, is solved using the symbolic factorization just computed. This is possible because *A* and *C* have the same nonzero structure and therefore also the same symbolic factorization. The solution errors are printed.

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

int main()
{
    Imsl_f_sparse_elem *a = NULL, *c = NULL;
    Imsl_snodal_symbolic_factor symbolic_factor;
    float *b = NULL, *d = NULL, *x = NULL, *y = NULL;
    float *mod_vector = NULL;
    int n, ic, nz, i, index;
    float error_1, error_2;

    ic = 50;
    n = ic * ic;
    mod_vector = (float*) malloc (n * sizeof(float));

    /* Build coefficient matrix A */
    a = (Imsl_f_sparse_elem *) imsl_f_generate_test_coordinate (n, ic,
        &nz,
        IMSL_SYMMETRIC_STORAGE,
        0);

    /* Build coefficient matrix C */
    c = (Imsl_f_sparse_elem*) malloc (nz * sizeof(Imsl_f_sparse_elem));
    for (i = 0; i < nz; i++) c[i] = a[i];
    for (i = 0; i < n; i++)
        c[i].val = 6.0;

    /* Form right hand side b */
    for (i = 0; i < n; i++)
        mod_vector[i] = (float) (i % 5);

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

    /* Form right hand side d */
    d = (float *) imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, c,
        IMSL_X_VECTOR, n, mod_vector,
        IMSL_SYMMETRIC_STORAGE,
```

```
        0);

    /* Solve Ax = b and return the symbolic factorization */
    x = imsl_f_sparse_cholesky_smp (n, nz, a, b,
        IMSL_RETURN_SYMBOLIC_FACTOR, &symbolic_factor,
        0);

    /* Compute solution error |x - mod_vector| */
    error_1 = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_vector,
        IMSL_INF_NORM, &index,
        0);

    /* Solve Cy = d given the symbolic factorization */
    y = imsl_f_sparse_cholesky_smp (n, nz, c, d,
        IMSL_SUPPLY_SYMBOLIC_FACTOR, &symbolic_factor,
        0);

    /* Compute solution error |y - mod_vector| */
    error_2 = imsl_f_vector_norm (n, y,
        IMSL_SECOND_VECTOR, mod_vector,
        IMSL_INF_NORM, &index,
        0);

    printf ("Solution error |x - mod_vector| = %e\n", error_1);
    printf ("Solution error |y - mod_vector| = %e\n", error_2);

    /* Free allocated memory */
    if (b) imsl_free(b);
    if (d) imsl_free(d);
    if (x) imsl_free(x);
    if (y) imsl_free(y);
    if (mod_vector) free(mod_vector);
    if (a) imsl_free(a);
    if (c) free(c);
    imsl_free_snodal_symbolic_factor (&symbolic_factor);
}
```

### Output

```
Solution error |x - mod_vector| = 4.529953e-005
Solution error |y - mod_vector| = 2.861023e-006
```

## Example 3

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

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

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

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

   ic = 50;
   n = ic * ic;

   /* Generate two right hand sides */
   imsl_random_seed_set (1234567);
   b_1 = imsl_f_random_uniform (n, 0);
   b_2 = imsl_f_random_uniform (n, 0);

   /* Build coefficient matrix a */
   a = imsl_f_generate_test_coordinate (n, ic, &nz,
       IMSL_SYMMETRIC_STORAGE,
       0);

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

   time_1 = omp_get_wtime();

   x_1 = imsl_f_sparse_cholesky_smp (n, nz, a, b_1,
       IMSL_RETURN_NUMERIC_FACTOR, &numeric_factor,
       0);

   time_1 = omp_get_wtime() - time_1;

   /* Now solve Ax_2 = b_2 given the numeric
       factorization */
   time_2 = omp_get_wtime();

   x_2 = imsl_f_sparse_cholesky_smp (n, nz, a, b_2,
       IMSL_SUPPLY_NUMERIC_FACTOR, &numeric_factor,
       IMSL_SOLVE_ONLY,
       0);

   time_2 = omp_get_wtime() - time_2;
   printf("time_2/time_1 = %lf\n", time_2/time_1);

   /* Free allocated memory */
   if (x_1) imsl_free(x_1);
   if (x_2) imsl_free(x_2);
   if (b_1) imsl_free(b_1);
```

```
    if (b_2) imsl_free(b_2);
    if (a) imsl_free(a);
    imsl_f_free_numeric_factor (&numeric_factor);
}
```

### Output

```
time_2/time_1 = 0.029411
```

## Fatal Errors

| | |
|---|---|
| `IMSL_BAD_SQUARE_ROOT` | A zero or negative square root has occurred during the factorization. The coefficient matrix is not positive definite. |

# sparse_cholesky_smp (complex)

more...

more...

Computes the Cholesky factorization of a sparse Hermitian positive definite matrix *A* by an OpenMP parallelized supernodal algorithm and solves the sparse Hermitian positive definite system of linear equations *Ax* = *b*.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_sparse_cholesky_smp (*int* n, *int* nz, *Imsl_c_sparse_elem* a[], *f_complex* b[], …, 0)

*void* imsl_free_snodal_symbolic_factor (*Imsl_snodal_symbolic_factor* *sym_factor)

*void* imsl_c_free_numeric_factor (*Imsl_c_numeric_factor* *num_factor)

The type *d_complex* functions are imsl_z_sparse_cholesky_smp and imsl_z_free_numeric_factor.

## Required Arguments

*int* n (Input)
The order of the input matrix.

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

*Imsl_c_sparse_elem* a[] (Input)
An array of length nz containing the location and value of each nonzero entry in the lower triangle of the matrix.

*f_complex* b[] (Input)
An array of length n containing the right-hand side.

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_sparse_cholesky_smp (*int* n, *int* nz, *Imsl_c_sparse_elem* a[],
*f_complex* b[],

IMSL_CSC_FORMAT, *int* col_ptr[], *int* row_ind[], *f_complex* values[],

IMSL_PREORDERING, *int* preorder,

IMSL_RETURN_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor,

IMSL_SUPPLY_SYMBOLIC_FACTOR, *Imsl_snodal_symbolic_factor* *sym_factor,

IMSL_SYMBOLIC_FACTOR_ONLY,

IMSL_RETURN_NUMERIC_FACTOR, *Imsl_c_numeric_factor* *num_factor,

IMSL_SUPPLY_NUMERIC_FACTOR, *Imsl_c_numeric_factor* *num_factor,

IMSL_NUMERIC_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_SMALLEST_DIAGONAL_ELEMENT, *float* *smallest_element,

IMSL_LARGEST_DIAGONAL_ELEMENT, *float* *largest_element,

IMSL_NUM_NONZEROS_IN_FACTOR, *int* *num_nonzeros,

IMSL_RETURN_USER, *f_complex* x[],

0)

## Optional Arguments

IMSL_CSC_FORMAT, *int* col_ptr[], *int* row_ind[], *f_complex* values[] (Input)
Accept the coefficient matrix in compressed sparse column (CSC) format, as describedin the Compressed Sparse Column (CSC) Format section of the "Introduction" chapter of this manual.

IMSL_PREORDERING, *int* preorder (Input)
The variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of matrix *A*:

| preorder | Method |
|---|---|
| 0 | George and Liu's Quotient Minimum Degree algorithm. |
| 1 | A variant of George and Liu's Quotient Minimum Degree algorithm using a preprocessing phase and external degrees. |

Default: `preorder` = 0.

`IMSL_RETURN_SYMBOLIC_FACTOR`, *Imsl_snodal_symbolic_factor* `*sym_factor` (Output)

A pointer to a structure of type *Imsl_snodal_symbolic_factor* containing, on return, the supernodal symbolic factorization of the input matrix. A detailed description of the *Imsl_snodal_symbolic_factor* structure is given in the following table:

Table 1.8 — Strucuture `Imsl_snodal_symbolic_factor`

| **Parameter** | **Data Type** | **Description** |
|---|---|---|
| `nzsub` | *int* `**` | A pointer to an array containing the compressed row subscripts of the non-zero off-diagonal elements of the Cholesky factor. |
| `xnzsub` | *int* `**` | A pointer to an array of length `n+1` containing indices for `*nzsub`. The row subscripts for the non-zeros in column `j` of the Cholesky factor are stored consecutively beginning with `(*nzsub)[(*xnzsub)[j]]`. |
| `maxsub` | *int* | The number of elements in array `*nzsub` that are used as subscripts. Note that the size of `*nzsub` can be larger than `maxsub`. |
| `xlnz` | *int* `**` | A pointer to an array of length `n+1` containing the starting and stopping indices to use to extract the non-zero off-diagonal elements from array `*alnz` (For a description of `alnz`, see the description section of optional argument `IMSL_RETURN_NUMERIC_FACTOR`). For column `j` of the factor matrix, the starting and stopping indices of `*alnz` are stored in `(*xlnz)[j]` and `(*xlnz)[j+1]` respectively. |
| `maxlnz` | *int* | The number of non-zero off-diagonal elements in the Cholesky factor. |
| `perm` | *int* `**` | A pointer to an array of length `n` containing the permutation vector. |
| `invp` | *int* `**` | A pointer to an array of length `n` containing the inverse permutation vector. |
| `multifrontal_space` | *int* | This variable is not used in the current implementation. |
| `nsuper` | *int* | The number of supernodes in the Cholesky factor. |
| `snode` | *int* `**` | A pointer to an array of length `n`. Element `(*snode)[j]` contains the number of the fundamental supernode to which column `j` belongs. |
| `snode_ptr` | *int* `**` | A pointer to an array of length `nsuper+1` containing the start column of each supernode. |
| `nleaves` | *int* | The number of leaves in the postordered elimination tree of the symmetrically permuted input matrix *A*. |
| `etree_leaves` | *int* `**` | A pointer to an array of length `nleaves+1` containing the leaves of the elimination tree. |

To free the memory allocated within this structure, use function
`imsl_free_snodal_symbolic_factor`.

`IMSL_SUPPLY_SYMBOLIC_FACTOR`, *Imsl_snodal_symbolic_factor* `*sym_factor` (Input)
A pointer to a structure of type *Imsl_snodal_symbolic_factor*. This structure contains the symbolic factorization of the input matrix computed by `imsl_c_sparse_cholesky_smp` with the `IMSL_RETURN_SYMBOLIC_FACTOR` option. The structure is described in the `IMSL_RETURN_SYMBOLIC_FACTOR` optional argument description.
To free the memory allocated within this structure, use function
`imsl_free_snodal_symbolic_factor`.

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

`IMSL_RETURN_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Output)
A pointer to a structure of type *Imsl_c_numeric_factor* containing, on return, the numeric factorization of the input matrix. A detailed description of the *Imsl_c_numeric_factor* structure is given in the `IMSL_RETURN_NUMERIC_FACTOR` optional argument description of function `imsl_c_lin_sol_posdef_coordinate (complex)`. To free the memory allocated within this structure, use function `imsl_c_free_numeric_factor`.

`IMSL_SUPPLY_NUMERIC_FACTOR`, *Imsl_c_numeric_factor* `*num_factor` (Input)
A pointer to a structure of type *Imsl_c_numeric_factor*. This structure contains the numeric factorization of the input matrix computed by `imsl_c_sparse_cholesky_smp` with the `IMSL_RETURN_NUMERIC_FACTOR` option. The structure is described in the `IMSL_RETURN_NUMERIC_FACTOR` optional argument description of function `imsl_lin_sol_posdef_coordinate (complex)`.
To free the memory allocated within this structure, use function
`imsl_c_free_numeric_factor`.

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

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

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

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

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

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

## Description

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

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

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

$$PAP^{\mathrm{T}} = LL^{\mathrm{H}}$$

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

The numerical factorization is an implementation of a parallel supernodal algorithm that combines a left-looking and a right-looking column computation scheme. This algorithm is described in detail in Rauber et al. (1999).

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

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

$$Ly_1 = Pb$$

$$L^{\mathrm{H}}y_2 = y_1$$

$$x = P^{\mathrm{T}}y_2$$

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

# Examples

## Example 1

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

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

We construct the solution $x^T = (1 + i, 2 + 2i, 3 + 3i)$ to the system $Ax = b$ by setting

$b := Ax = (-2 + 2i, 5 + 15i, 36 + 28i)^T$. The number of nonzeros in the lower triangle is `nz` = 5. The solution is computed and printed.

```
#include <imsl.h>

int main()
{
   int n = 3, nz = 5;
   f_complex b[] = {{-2.0, 2.0}, {5.0, 15.0}, {36.0, 28.0}};
   f_complex *x = NULL;
   Imsl_c_sparse_elem a[] = {0, 0, {2.0, 0.0},
       1, 1, {4.0, 0.0},
       2, 2, {10.0, 0.0},
       1, 0, {-1.0, -1.0},
       2, 1, {1.0, -2.0}};

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

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

   imsl_free (x);
}
```

### Output

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

## Example 2

This example shows how a symbolic factor can be re-used. Consider matrix $A$, a Hermitian positive definite matrix with value 6 on the diagonal and value $-1-i$ on its lower codiagonal and the lower band at distance 50 from the diagonal. At first, the system $Ax = b$ is solved and the symbolic factorization of $A$ is returned. Then, the system $Cy = d$ with $C = A+4*I$, $I$ the identity matrix, is solved using the symbolic factorization just computed. This is possible because $A$ and $C$ have the same nonzero structure and therefore also the same symbolic factorization. The solution errors are printed.

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

int main()
{
    int n, ic, nz, i, index;
    float error_1, error_2;
    f_complex *b = NULL, *d = NULL, *x = NULL, *y = NULL;
    f_complex *mod_vector = NULL;
    Imsl_c_sparse_elem *a = NULL, *c = NULL;
    Imsl_snodal_symbolic_factor symbolic_factor;

    ic = 50;
    n = ic * ic;

    mod_vector = (f_complex*) malloc (n * sizeof(f_complex));

    /* Build coefficient matrix A */
    a = imsl_c_generate_test_coordinate (n, ic,
        &nz,
        IMSL_SYMMETRIC_STORAGE,
        0);

    /* Build coefficient matrix C */
    c = (Imsl_c_sparse_elem *) malloc (nz * sizeof (Imsl_c_sparse_elem));

    for (i=0; i<nz; i++)
        c[i] = a[i];

    for (i=0; i<n; i++)
    {
        c[i].val.re = 10.0;
        c[i].val.im = 0.0;
    }

    /* Form right hand side b */
    for (i = 0; i < n; i++)
    {
        mod_vector[i].re = (float) (i % 5);
        mod_vector[i].im = 0.0;
    }
    b = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, mod_vector,
        IMSL_SYMMETRIC_STORAGE,
        0);

    /* Form right hand side d */
    d = (f_complex *) imsl_c_mat_mul_rect_coordinate ("A*x",
```

```
        IMSL_A_MATRIX, n, n, nz, c,
        IMSL_X_VECTOR, n, mod_vector,
        IMSL_SYMMETRIC_STORAGE,
        0);

    /* Solve Ax = b and return the symbolic factorization */
    x = imsl_c_sparse_cholesky_smp (n, nz, a, b,
        IMSL_RETURN_SYMBOLIC_FACTOR, &symbolic_factor,
        0);

    /* Compute error |x-mod_vector| */
    error_1 = imsl_c_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_vector,
        IMSL_INF_NORM, &index,
        0);

    /* Solve Cy = d given the symbolic factorization */
    y = imsl_c_sparse_cholesky_smp (n, nz, c, d,
        IMSL_SUPPLY_SYMBOLIC_FACTOR, &symbolic_factor,
        0);

    /* Compute error |y-mod_vector| */
    error_2 = imsl_c_vector_norm (n, y,
        IMSL_SECOND_VECTOR, mod_vector,
        IMSL_INF_NORM, &index,
        0);

    printf ("Solution error |x - mod_vector| = %e\n", error_1);
    printf ("Solution error |y - mod_vector| = %e\n", error_2);

    /* Free allocated memory */
    if (mod_vector) free(mod_vector);
    if (a) imsl_free (a);
    if (c) free (c);
    if (b) imsl_free (b);
    if (d) imsl_free (d);
    if (y) imsl_free (y);
    if (x) imsl_free (x);
    imsl_free_snodal_symbolic_factor(&symbolic_factor);
}
```

## Output

```
Solution error |x - mod_vector| = 2.885221e-006
Solution error |y - mod_vector| = 2.386146e-006
```

### Example 3

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

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

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

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

   /* Generate two right hand sides */
   imsl_random_seed_set (1234567);
   rand_vec = imsl_f_random_uniform (4 * n, 0);

   for (i = 0; i < n; i++) {
       b_1[i].re = rand_vec[index++];
       b_1[i].im = rand_vec[index++];
       b_2[i].re = rand_vec[index++];
       b_2[i].im = rand_vec[index++];
   }

   /* Build coefficient matrix a */
   a = imsl_c_generate_test_coordinate (n, ic, &nz,
       IMSL_SYMMETRIC_STORAGE,
       0);

   /* Now solve Ax_1 = b_1 and return the numeric factorization */
   time_1 = omp_get_wtime();

   x_1 = imsl_c_sparse_cholesky_smp (n, nz, a, b_1,
       IMSL_RETURN_NUMERIC_FACTOR, &numeric_factor,
       0);

   time_1 = omp_get_wtime() - time_1;

   /* Now solve Ax_2 = b_2 given the numeric factorization */
   time_2 = omp_get_wtime();
```

```
   x_2 = imsl_c_sparse_cholesky_smp (n, nz, a, b_2,
       IMSL_SUPPLY_NUMERIC_FACTOR, &numeric_factor,
       IMSL_SOLVE_ONLY,
       0);

   time_2 = omp_get_wtime() - time_2;

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

   /* Free memory */
   if (rand_vec) imsl_free(rand_vec);
   if (x_1) imsl_free(x_1);
   if (x_2) imsl_free(x_2);
   if (a) imsl_free(a);
   imsl_c_free_numeric_factor(&numeric_factor);
}
```

### Output

```
time_2/time_1 = 0.025771
```

## Fatal Errors

| | |
|---|---|
| `IMSL_BAD_SQUARE_ROOT` | A zero or negative square root has occurred during the factorization. The coefficient matrix is not positive definite. |

# lin_sol_gen_min_residual

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

## Synopsis

*#include* <imsl.h>

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

The type *double* function is imsl_d_lin_sol_gen_min_residual.

## Required Arguments

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_lin_sol_gen_min_residual (*int* n, *void* amultp (), *float* *b,

IMSL_RETURN_USER, *float* x[],

IMSL_MAX_ITER, *int* *maxit,

IMSL_REL_ERR, *float* tolerance,

IMSL_PRECOND, *void* precond (),

IMSL_MAX_KRYLOV_SUBSPACE_DIM, *int* kdmax,

IMSL_HOUSEHOLDER_REORTHOG,

IMSL_FCN_W_DATA, *void* amultp(), *void* *data,

IMSL_PRECOND_W_DATA, *void* precond(), *void* *data,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length n containing the solution *x*.

IMSL_MAX_ITER, *int* *maxit (Input/Output)

A pointer to an integer, initially set to the maximum number of GMRES iterations allowed. On exit, the number of iterations used is returned.

Default: maxit = 1000

IMSL_REL_ERR, *float* tolerance (Input)

The algorithm attempts to generate *x* such that $\|b - Ax\|_2 \le \tau \|b\|_2$, where $\tau$ = tolerance.

Default: tolerance = sqrt(imsl_f_machine(4))

IMSL_PRECOND, *void* precond (*float* *r, *float* *z) (Input)

User supplied function which sets $z = M^{-1}r$, where *M* is the preconditioning matrix.

IMSL_MAX_KRYLOV_SUBSPACE_DIM, *int* kdmax, (Input)

The maximum Krylov subspace dimension, i.e., the maximum allowable number of GMRES iterations allowed before restarting.

Default: kdmax = imsl_i_min(n, 20)

IMSL_HOUSEHOLDER_REORTHOG,

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

IMSL_FCN_W_DATA, *void* amultp (*float* *p, *float* *z, *void* *data), *void* *data, (Input)

User supplied function which computes $z = Ap$, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

IMSL_PRECOND_W_DATA, *void* precond (*float* *r, *float* *z, *void* *data), *void* *data (Input)

User supplied function which sets $z = M^{-1}r$, where *M* is the preconditioning matrix, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions section in the introduction to this manual for more details.

# Description

The function `imsl_f_lin_sol_gen_min_residual`, based on the FORTRAN subroutine GMRES by H.F. Walker, solves the linear system $Ax = b$ using the GMRES method. This method is described in detail by Saad and Schultz (1986) and Walker (1988).

The GMRES method begins with an approximate solution $x_0$ and an initial residual $r_0 = b - Ax_0$. At iteration $m$, a correction $z_m$ is determined in the Krylov subspace

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

$v = r_0$ which solves the least-squares problem

$$\left(z \in \overset{\min}{\kappa_m}(r_0)\right) \quad \left\| b - A\left(x_0 + z\right) \right\|_2$$

Then at iteration $m$, $x_m = x_0 + z_m$.

Orthogonalization by Householder transformations requires less storage but more arithmetic than Gram-Schmidt. However, Walker (1988) reports numerical experiments which suggest the Householder approach is more stable, especially as the limits of residual reduction are reached.

# Examples

## Example 1

As an example, consider the following matrix:

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

Let $x^T = (1, 2, 3, 4, 5, 6)$ so that $Ax = (10, 7, 45, 33, -34, 31)^T$. The function `imsl_f_mat_mul_rect_coordinate` is used to form the product $Ax$.

```
#include <imsl.h>

void amultp (float*, float*);

int main()
{
   float b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
   int n = 6;
   float *x;
```

```
   x = imsl_f_lin_sol_gen_min_residual (n, amultp, b,
           0);

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

void amultp (float *p, float *z)
{
   Imsl_f_sparse_elem a[] = {0, 0, 10.0,
                             1, 1, 10.0,
                             1, 2, -3.0,
                             1, 3, -1.0,
                             2, 2, 15.0,
                             3, 0, -2.0,
                             3, 3, 10.0,
                             3, 4, -1.0,
                             4, 0, -1.0,
                             4, 3, -5.0,
                             4, 4, 1.0,
                             4, 5, -3.0,
                             5, 0, -1.0,
                             5, 1, -2.0,
                             5, 5, 6.0};
   int n = 6;
   int nz = 15;

   imsl_f_mat_mul_rect_coordinate ("A*x",
           IMSL_A_MATRIX, n, n, nz, a,
           IMSL_X_VECTOR, n, p,
           IMSL_RETURN_USER_VECTOR, z,
           0);
}
```

### Output

```
                       Solution, x, to Ax = b
        1           2           3           4           5           6
        1           2           3           4           5           6
```

## Example 2

In this example, the same system given in the first example is solved. This time a preconditioner is provided. The preconditioned matrix is chosen as the diagonal of *A*.

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

void amultp (float*, float*);
void precond (float*, float*);
```

```
int main()
{
   float b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
   int n = 6;
   float *x;
   int maxit = 1000;

   x = imsl_f_lin_sol_gen_min_residual (n, amultp, b,
       IMSL_MAX_ITER, &maxit,
       IMSL_PRECOND, precond,
       0);

   imsl_f_write_matrix ("Solution, x, to Ax = b", 1, n, x, 0);
   printf ("\nNumber of iterations taken = %d\n", maxit);
}

/* Set z = Ap */
void amultp (float *p, float *z)
{
   static Imsl_f_sparse_elem a[] =
       {0, 0, 10.0,
        1, 1, 10.0,
        1, 2, -3.0,
        1, 3, -1.0,
        2, 2, 15.0,
        3, 0, -2.0,
        3, 3, 10.0,
        3, 4, -1.0,
        4, 0, -1.0,
        4, 3, -5.0,
        4, 4, 1.0,
        4, 5, -3.0,
        5, 0, -1.0,
        5, 1, -2.0,
        5, 5, 6.0};
   int n = 6;
   int nz = 15;

   imsl_f_mat_mul_rect_coordinate ("A*x",
       IMSL_A_MATRIX, n, n, nz, a,
       IMSL_X_VECTOR, n, p,
       IMSL_RETURN_USER_VECTOR, z,
       0);
}

/* Solve Mz = r */
void precond (float *r, float *z)
{
   static float diagonal_inverse[] =
   {0.1, 0.1, 1.0/15.0, 0.1, 1.0, 1.0/6.0};
   int n = 6;
```

```
    int i;

    for (i=0; i<n; i++)
        z[i] = diagonal_inverse[i]*r[i];
}
```

### Output

```
                       Solution, x, to Ax = b
        1           2           3           4           5           6
        1           2           3           4           5           6

Number of iterations taken =
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# lin_sol_def_cg

Solves a real symmetric definite linear system using a conjugate gradient method. Using optional arguments, a preconditioner can be supplied.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_lin_sol_def_cg (*int* n, *void* amultp (), *float* *b, …, 0)

The type *double* function is imsl_d_lin_sol_def_cg.

## Required Arguments

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

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

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

## Return Value

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_lin_sol_def_cg (*int* n, *void* amultp (), *float* *b,

IMSL_RETURN_USER, *float* x [],

IMSL_MAX_ITER, *int* *maxit,

IMSL_REL_ERR, *float* relative_error,

IMSL_PRECOND, *void* precond (),

IMSL_JACOBI, *float* *diagonal,

IMSL_FCN_W_DATA, *void* amultp (), *void* data, IMSL_PRECOND_W_DATA, *void* precond(), *void* *data,

0)

## Optional Arguments

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

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

IMSL_REL_ERR, *float* relative_error (Input)
The relative error desired.
Default: relative_error = sqrt(imsl_f_machine(4))

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

IMSL_JACOBI, *float* diagonal[] (Input)
Use the Jacobi preconditioner, i.e. $M$ = diag($A$). The user-supplied vector diagonal should be set so that diagonal[i] = $A_{ii}$.

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

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

## Description

The function imsl_f_lin_sol_def_cg solves the symmetric definite linear system $Ax = b$ using the conjugate gradient method with optional preconditioning. This method is described in detail by Golub and Van Loan (1983, Chapter 10), and in Hageman and Young (1981, Chapter 7).

The preconditioning matrix *M* is a matrix that approximates *A*, and for which the linear system $Mz = r$ is easy to solve. These two properties are in conflict; balancing them is a topic of much current research. In the default use of imsl_f_lin_sol_def_cg, $M = I$. If the option IMSL_JACOBI is selected, *M* is set to the diagonal of *A*.

The number of iterations needed depends on the matrix and the error tolerance. As a rough guide,

$$\text{maxit} = \sqrt{n} \quad \text{for} \quad n >> 1$$

See the references mentioned above for details.

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

$$\begin{aligned}
&\lambda = -1\\
&p_0 = x_0\\
&r_1 = b - Ap\\
&\text{for } k = 1, \ldots, \text{maxit}\\
&\quad z_k = M^{-1} r_k\\
&\quad \text{if } k = 1, \text{ then}\\
&\quad\quad \beta_k = 1\\
&\quad\quad p_k = z_k\\
&\quad \text{else}\\
&\quad\quad \beta_k = \left(z_k^T r_k\right) / \left(z_{k-1}^T r_{k-1}\right)\\
&\quad\quad p_k = z_k + \beta_k p_k\\
&\quad \text{endif}\\
&\quad z_k = Ap\\
&\quad \alpha_k = \left(z_{k-1}^T z_{k-1}\right) / \left(z_k^T p_k\right)\\
&\quad x_k = x_k + \alpha_k p_k\\
&\quad r_k = r_k - \alpha_k z_k\\
&\quad \text{if } \left(\|z_k\|_2 \le \tau\left(1 - \lambda\right)\|x_k\|_2\right) \text{ then}\\
&\quad\quad \text{recompute } \lambda\\
&\quad\quad \text{if } \left(\|z_k\|_2 \le \tau\left(1 - \lambda\right)\|x_k\|_2\right) \text{ exit}\\
&\quad \text{endif}\\
&\text{endfor}
\end{aligned}$$

Here $\lambda$ is an estimate of $\lambda_{max}(G)$, the largest eigenvalue of the iteration matrix $G = I - M^{-1} A$. The stopping criterion is based on the result (Hageman and Young 1981, pp. 148-151)

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

where

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

It is also known that

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

where the $T_n$ are the symmetric, tridiagonal matrices

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

with $\mu_k = 1 - \beta_k/\alpha_{k-1} - 1/\alpha_k$, $\mu_1 = 1 - 1/\alpha_1$ and

$$\omega_k = \sqrt{B_k} / \alpha_{k-1}$$

Usually the eigenvalue computation is needed for only a few of the iterations.

## Examples

### Example 1

In this example, the solution to a linear system is found. The coefficient matrix is stored as a full matrix.

```
#include <imsl.h>

static void amultp (float*, float*);

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

   x = imsl_f_lin_sol_def_cg (n, amultp, b, 0);

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

static void amultp (float *p, float *z)
{
   static float a[] = {1.0, -3.0, 2.0,

                       -3.0, 10.0, -5.0,
                       2.0, -5.0, 6.0};
   int n = 3;

   imsl_f_mat_mul_rect ("A*x",
            IMSL_A_MATRIX, n, n, a,
            IMSL_X_VECTOR, n, p,
            IMSL_RETURN_USER, z,
            0);
```

```
}
```

### Output

```
                x
        1           2           3
        1          -4           7
```

## Example 2

In this example, two different preconditioners are used to find the solution of a linear system which occurs in a finite difference solution of Laplace's equation on a regular $c \times c$ grid, $c = 100$. The matrix is $A = E\,(c^2, c)$. For the first solution, select Jacobi preconditioning and supply the diagonal, so $M =$ `diag` ($A$). The number of iterations performed and the maximum absolute error are printed. Next, use a more complicated preconditioning matrix, $M$, consisting of the symmetric tridiagonal part of $A$.

Notice that the symmetric positive definite band solver is used to factor $M$ once, and subsequently just perform forward and back solves. Again, the number of iterations performed and the maximum absolute error are printed. Note the substantial reduction in iterations.

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

static void amultp (float*, float*);
static void precond (float*, float*);
static Imsl_f_sparse_elem *a;
static int n = 2500;
static int c = 50;
static int nz;

int main()
{
   int maxit = 1000;
   int i;
   int index;
   float *b;
   float *x;
   float *mod_five;
   float *diagonal;
   float norm;

   n = c*c;
   mod_five = (float*) malloc (n*sizeof(*mod_five));
   diagonal = (float*) malloc (n*sizeof(*diagonal));
   b = (float*) malloc (n*sizeof(*b));

   /* Generate coefficient matrix */
   a = imsl_f_generate_test_coordinate (n, c, &nz,
       0);
```

```
    /* Set a predetermined answer and diagonal */
    for (i=0; i<n; i++) {
        mod_five[i] = (float) (i % 5);
        diagonal[i] = 4.0;
    }

    /* Get right hand side */
    amultp (mod_five, b);

    /* Solve with jacobi preconditioning */
    x = imsl_f_lin_sol_def_cg (n, amultp, b,
        IMSL_MAX_ITER, &maxit,
        IMSL_JACOBI, diagonal,
        0);

    /* Find max absolute error, print results */
    norm = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
    printf ("iterations = %d, norm = %e\n", maxit, norm);
    imsl_free (x);

    /* Solve same system, with different preconditioner */
    x = imsl_f_lin_sol_def_cg (n, amultp, b,
        IMSL_MAX_ITER, &maxit,
        IMSL_PRECOND, precond,
        0);

    norm = imsl_f_vector_norm (n, x,
        IMSL_SECOND_VECTOR, mod_five,
        IMSL_INF_NORM, &index,
        0);
    printf ("iterations = %d, norm = %e\n", maxit, norm);
}

/* Set z = Ap */
static void amultp (float *p, float *z)
{
    imsl_f_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, p,
        IMSL_RETURN_USER_VECTOR, z,
        0);
}

/* Solve Mz = r */
static void precond (float *r, float *z)
{
    static float *m;
```

```
    static float *factor;
    static int first = 1;
    float *null = (float*) 0;

    if (first) {
        /* Factor the first time through */
        m = imsl_f_generate_test_band (n, 1,
            IMSL_SYMMETRIC_STORAGE,
            0);

        imsl_f_lin_sol_posdef_band (n, m, 1, null,
            IMSL_FACTOR, &factor,
            IMSL_FACTOR_ONLY,
            0);
        first = 1;
    }

    /* Perform the forward and back solves */
    imsl_f_lin_sol_posdef_band (n, m, 1, r,
        IMSL_FACTOR_USER, factor,
        IMSL_SOLVE_ONLY,
        IMSL_RETURN_USER, z,
        0);
}
```

### Output

```
iterations = 115, norm = 1.382828e-05
iterations = 75, norm = 7.319450e-05
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# lin_least_squares_gen

more...

Solves a linear least-squares problem *Ax* = *b*. Using optional arguments, the *QR* factorization of *A*, *AP* = *QR*, and the solve step based on this factorization can be computed.

## Synopsis

*#include* <imsl.h>

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

The type *double* procedure is `imsl_d_lin_least_squares_gen`.

## Required Arguments

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

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

*float* `a[]` (Input)
Array of size *m* × *n* containing the matrix.

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

## Return Value

If no optional arguments are used, function `imsl_f_lin_least_squares_gen` returns a pointer to the solution *x* of the linear least-squares problem *Ax* = *b*. To release this space, use `imsl_free`. If no value can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* `*imsl_f_lin_least_squares_gen` (*int* `m`, *int* `n`, *float* `a[]`, *float* `b[]`,

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

IMSL_RETURN_USER, *float* x[],

IMSL_BASIS, *float* tol, *int* *kbasis,

IMSL_RESIDUAL, *float* **p_res,

IMSL_RESIDUAL_USER, *float* res[],

IMSL_FACTOR, *float* **p_qraux, *float* **p_qr,

IMSL_FACTOR_USER, *float* qraux[], *float* qr[],

IMSL_FAC_COL_DIM, *int* qr_col_dim,

IMSL_Q, *float* **p_q,

IMSL_Q_USER, *float* q[],

IMSL_Q_COL_DIM, *int* q_col_dim,

IMSL_PIVOT, *int* pvt[],

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

0)

## Optional Arguments

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

IMSL_RETURN_USER, *float* x[] (Output)
A user-allocated array of size *n* containing the least-squares solution ***x***. If IMSL_RETURN_USER is used, the return value of the function is a pointer to the array x.

IMSL_BASIS, *float* tol, *int* *kbasis (Input, Input/Output)

*float* tol (Input)
Nonnegative tolerance used to determine the subset of columns of ***A*** to be included in the solution.
Default: tol = sqrt (imsl_amach(4))

*int* *kbasis (Input/Output)
Integer containing the number of columns used in the solution. kbasis = *k* if $|r_{k+1,k+1}| < |tol| * |r_{1,1}|$. For more information on the use of this option, see Description section.
Default: kbasis = min (*m*, *n*)

IMSL_RESIDUAL, *float* **p_res (Output)

The address of a pointer to an array of size $m$ containing the residual vector $b - Ax$. On return, the necessary space is allocated by the function. Typically, *float* *p_res is declared, and &p_res is used as an argument.

IMSL_RESIDUAL_USER, *float* res[] (Output)

A user-allocated array of size $m$ containing the residual vector $b - Ax$.

IMSL_FACTOR, *float* **p_qraux, *float* **p_qr (Output)

*float* **p_qraux (Input/Output)

The address of a pointer qraux to an array of size $n$ containing the scalars $\tau_k$ of the Householder transformations in the first min ($m$, $n$) positions. On return, the necessary space is allocated by the function. Typically, *float* *qraux is declared, and &qraux is used as an argument.

*float* **p_qr (Input/Output)

The address of a pointer to an array of size $m \times n$ containing the Householder transformations that define the decomposition. The strictly lower-triangular part of this array contains the information to construct $Q$, and the upper-triangular part contains $R$. On return, the necessary space is allocated by the function. Typically, *float* *qr is declared, and &qr is used as an argument.

IMSL_FACTOR_USER, *float* qraux[], *float* qr[] (Input /Output)

*float* qraux[] (Input/Output)

A user-allocated array of size $n$ containing the scalars $\tau_k$ of the Householder transformations in the first min ($m$, $n$) positions.

*float* qr[] (Input/Output)

A user-allocated array of size $m \times n$ containing the Householder transformations that define the decomposition. The strictly lower-triangular part of this array contains the information to construct $Q$. The upper-triangular part contains $R$. If the data in a is not needed, qr can share the same storage locations as a by using a instead of the separate argument qr.

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

IMSL_FAC_COL_DIM, *int* qr_col_dim (Input)

The column dimension of the array containing $QR$ factorization.

Default: qr_col_dim = $n$

IMSL_Q, *float* **p_q (Output)

The address of a pointer to an array of size $m \times m$ containing the orthogonal matrix of the factorization. On return, the necessary space is allocated by the function. Typically, *float* *q is declared, and &q is used as an argument.

IMSL_Q_USER, *float* q[] (Output)

A user-allocated array of size $m \times m$ containing the orthogonal matrix $Q$ of the $QR$ factorization.

`IMSL_Q_COL_DIM`, *int* `q_col_dim` (Input)

The column dimension of the array containing the *Q* matrix of the factorization.

Default: `q_col_dim` = *m*

`IMSL_PIVOT`, *int* `pvt[]` (Input/Output)

Array of size *n* containing the desired variable order and usage information. The argument is used with `IMSL_FACTOR_ONLY` or `IMSL_SOLVE_ONLY`.

On input, if `pvt` [$k - 1$] > 0, then column *k* of *A* is an initial column. If `pvt` [$k - 1$] = 0, then the column of *A* is a free column and can be interchanged in the column pivoting. If `pvt` [$k - 1$] < 0, then column *k* of *A* is a final column. If all columns are specified as initial (or final) columns, then no pivoting is performed. (The permutation matrix *P* is the identity matrix in this case.)

On output, `pvt` [$k - 1$] contains the index of the column of the original matrix that has been interchanged into column *k*.

Default: `pvt` [$k - 1$] = 0, $k = 1, \ldots, n$

`IMSL_FACTOR_ONLY`

Compute just the *QR* factorization of the matrix *AP* with the permutation matrix *P* defined by `pvt` and by further pivoting involving free columns. If `IMSL_FACTOR_ONLY` is used, the additional arguments `IMSL_PIVOT` and `IMSL_FACTOR` are required. In that case, the required argument `b` is ignored, and the returned value of the function is `NULL`.

`IMSL_SOLVE_ONLY`

Compute the solution to the least-squares problem $Ax = b$ given the *QR* factorization previously computed by this function. If `IMSL_SOLVE_ONLY` is used, arguments `IMSL_FACTOR_USER`, `IMSL_PIVOT`, and `IMSL_BASIS` are required, and the required argument `a` is ignored.

## Description

The function `imsl_f_lin_least_squares_gen` solves a system of linear least-squares problems $Ax = b$ with column pivoting. It computes a *QR* factorization of the matrix *AP*, where *P* is the permutation matrix defined by the pivoting, and computes the smallest integer *k* satisfying $|r_{k+1,k+1}| < |tol| * |r_{1,1}|$ to the output variable `kbasis`. Householder transformations

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

$k = 1, \ldots, \min(m - 1, n)$are used to compute the factorization. The decomposition is computed in the form $Q_{\min(m-1,\, n)} \ldots Q_1 AP = R$, so $AP = QR$ where $Q = Q_1 \ldots Q_{\min(m-1,\, n)}$. Since each Householder vector $u_k$ has zeros in the first $k - 1$ entries, it is stored as part of column *k* of `qr`. The upper-trapezoidal matrix *R* is stored in the upper-trapezoidal part of the first min (*m*, *n*) rows of `qr`. The solution *x* to the least-squares problem is computed by solving the upper-triangular system of linear equations $R(1{:}k, 1{:}k)\, y(1{:}k) = (Q^T b)(1{:}k)$ with *k* = `kbasis`. The solution is completed by setting $y(k + 1 : n)$ to zero and rearranging the variables, $x = Py$.

When IMSL_FACTOR_ONLY is specified, the function computes the *QR* factorization of *AP* with *P* defined by the input pvt and by column pivoting among "free" columns. Before the factorization, initial columns are moved to the beginning of the array a and the final columns to the end. Both initial and final columns are not permuted further during the computation. Just the free columns are moved.

If IMSL_SOLVE_ONLY is specified, then the function computes the least-squares solution to $Ax = b$ given the *QR* factorization previously defined. There are kbasis columns used in the solution. Hence, in the case that all columns are free, $x$ is computed as described in the default case.

# Examples

## Example 1

This example illustrates the least-squares solution of four linear equations in three unknowns using column pivoting. The problem is equivalent to least-squares quadratic polynomial fitting to four data values. Write the polynomial as $p(t) = x_1 + tx_2 + t^2x_3$ and the data pairs $(t_i, b_i)$, $t_i = 2i$, $i = 1, 2, 3, 4$. A pointer to the solution to $Ax = b$ is returned by the function imsl_f_lin_least_squares_gen.

```
#include <imsl.h>

float   a[] = {1.0, 2.0, 4.0,
               1.0, 4.0, 16.0,
               1.0, 6.0, 36.0,
               1.0, 8.0, 64.0};

float   b[] = {4.999, 9.001, 12.999, 17.001};

int main()
{
   int           m = 4, n = 3;
   float         *x;
                                   /* Solve Ax = b for x */

   x = imsl_f_lin_least_squares_gen (m, n, a, b, 0);

                                   /* Print x */
   imsl_f_write_matrix ("Solution vector", 1, n, x, 0);
}
```

### Output

```
Solution vector
    1          2          3
0.999      2.000      0.000
```

## Example 2

This example uses the same coefficient matrix *A* as in the initial example. It computes the *QR* factorization of *A* with column pivoting. The final and free columns are specified by `pvt` and the column pivoting is done only among the free columns.

```
#include <imsl.h>

float   a[] =  {1.0, 2.0, 4.0,
                 1.0, 4.0, 16.0,
                 1.0, 6.0, 36.0,
                 1.0, 8.0, 64.0};

int     pvt[] = {0, 0, -1};

int main()
{
   int            m = 4, n = 3;
   float          *x, *b;
   float          *p_qraux, *p_qr;
   float          *p_q;
                                  /* Compute the QR factorization */
                                  /* of A with partial column */
                                  /* pivoting */
   x = imsl_f_lin_least_squares_gen (m, n, a, b,
                                    IMSL_PIVOT, pvt,
                                    IMSL_FACTOR, &p_qraux, &p_qr,
                                    IMSL_Q, &p_q,
                                    IMSL_FACTOR_ONLY,
                                    0);

                                  /* Print Q */
   imsl_f_write_matrix ("The matrix Q", m, m, p_q, 0);

                                  /* Print R */
   imsl_f_write_matrix ("The matrix R", m, n, p_qr,
                        IMSL_PRINT_UPPER,
                        0);

                                  /* Print pivots */
   imsl_i_write_matrix ("The Pivot Sequence", 1, n, pvt, 0);

}
```

### Output

```
                 The matrix Q
           1          2          3          4
1    -0.1826    -0.8165     0.5000    -0.2236
2    -0.3651    -0.4082    -0.5000     0.6708
3    -0.5477     0.0000    -0.5000    -0.6708
```

```
4    -0.7303     0.4082     0.5000     0.2236

           The matrix R
           1           2           3
1      -10.95       -1.83      -73.03
2                   -0.82       16.33
3                                8.00

The Pivot Sequence
    1   2   3
    2   1   3
```

## Example 3

This example computes the *QR* factorization with column pivoting for the matrix *A* of the initial example. It computes the least-squares solutions to $Ax = b_i$ for $i = 1, 2, 3$.

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

float   a[]  = {1.0, 2.0, 4.0,
                1.0, 4.0, 16.0,
                1.0, 6.0, 36.0,
                1.0, 8.0, 64.0};

float   b[]  = {4.999, 9.001, 12.999, 17.001,
                2.0,   3.142, 5.11,   0.0,
                1.34, 8.112, 3.76, 10.99};

int     pvt[] = {0, 0, 0};

int main()
{
   int    m = 4, n = 3;
   int    i, k = 3;
   float *p_qraux, *p_qr;
   float  tol = 1.e-4;
   int   *kbasis;
   float *x, *p_res;

   /* Factor A with the given pvt */
   /* setting all variables to */
   /* be imsl_free */
   imsl_f_lin_least_squares_gen (m, n, a, b,
       IMSL_BASIS, tol, &kbasis,
       IMSL_PIVOT, pvt,
       IMSL_FACTOR, &p_qraux, &p_qr,
       IMSL_FACTOR_ONLY,
       0);
```

```
    /* Print some factorization */
    /* information*/
    printf("Number of Columns in the base\n%2d", kbasis);

    imsl_f_write_matrix ("Upper triangular R Matrix", m, n, p_qr,
        IMSL_PRINT_UPPER,
        0);
    imsl_i_write_matrix ("The output column order ", 1, n, pvt,
        0);

    /* Solve Ax = b for each x */
    /* given the factorization */
    for ( i = 0; i < k; i++) {
        x = imsl_f_lin_least_squares_gen (m, n, a, &b[i*m],
            IMSL_BASIS, tol, &kbasis,
            IMSL_PIVOT, pvt,
            IMSL_FACTOR_USER, p_qraux, p_qr,
            IMSL_RESIDUAL, &p_res,
            IMSL_SOLVE_ONLY,
            0);

        /* Print right-hand side, b */
        /* and solution, x */
        imsl_f_write_matrix ("Right-hand side, b ", 1, m, &b[i*m],
            0);
        imsl_f_write_matrix ("Solution, x ", 1, n, x, 0);

        /* Print residuals, b - Ax */
        imsl_f_write_matrix ("Residual, b - Ax ", 1, m, p_res,
            0);
    }
}
```

### Output

```
Number of Columns in the base
3
     Upper triangular R Matrix
           1           2           3
1     -75.26     -10.63      -1.59
2                 -2.65      -1.15
3                             0.36

The output column order
       1  2  3
       3  2  1

             Right-hand side, b
        1          2          3          4
        5          9         13         17
```

```
          Solution, x
     1           2           3
 0.999       2.000       0.000

             Residual, b - Ax
     1           2           3           4
-0.0004     0.0012     -0.0012      0.0004

            Right-hand side, b
     1           2           3           4
 2.000       3.142       5.110       0.000

          Solution, x
     1           2           3
-4.244       3.706      -0.391

             Residual, b - Ax
     1           2           3           4
 0.395      -1.186       1.186      -0.395

            Right-hand side, b
     1           2           3           4
  1.34        8.11        3.76       10.99


          Solution, x
     1           2           3
0.4735      0.9437      0.0286

             Residual, b - Ax
     1           2           3           4
-1.135       3.406      -3.406       1.135
```

## Fatal Errors

| | |
|---|---|
| `IMSL_SINGULAR_TRI_MATRIX` | The input triangular matrix is singular. The index of the first zero diagonal term is #. |

# nonneg_least_squares

Compute the non-negative least squares (NNLS) solution of an *m* × *n* real linear least squares system, $Ax \cong b$, $x \geq 0$.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_nonneg_least_squares (*int* m, *int* n, *float* a[], *float* b[],…, 0)

The type *double* function is imsl_d_nonneg_least_squares.

## Required Arguments

*int* m (Input)
The number of rows in the matrix.

*int* n (Input)
The number of columns in the matrix.

*float* a[] (Input)
An array of length m × n containing the matrix.

*float* b[] (Input)
An array of length m containing the right-hand side vector.

## Return Value

An array of length n containing the approximate solution vector, $x \geq 0$.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_nonneg_least_squares (*int* m, *int* n, *float* a[], *float* b[],

IMSL_ITMAX, *int* itmax,

IMSL_DROP_MAX_POS_DUAL, *int* maxdual,

IMSL_DROP_TOLERANCE, *float* tol,

IMSL_SUPPLY_WORK_ARRAYS, *int* lwork, *float* work[], *int* liwork, *int* iwork[],

IMSL_OPTIMIZED, *int* *iflag,

IMSL_DUAL_SOLUTION, *float* **dual,

IMSL_DUAL_SOLUTION_USER, *float* udual[],

IMSL_RESIDUAL_NORM, *float* *rnorm,

IMSL_RETURN_USER, *float* x[],

0)

## Optional Arguments

IMSL_ITMAX, *int* itmax (Input)

The number of times a constraint is added or dropped should not exceed this maximum value. An approximate solution $x \geq 0$ is returned when the maximum number is reached.
Default: itmax = 3 × n.

IMSL_DROP_MAX_POS_DUAL, *int* maxdual (Input)

Indicates how a variable is moved from its constraint to a positive value, or dropped, when its current dual value is positive. By dropping the variable corresponding to the first computed positive dual value, instead of the maximum, better runtime efficiency usually results by avoiding work in the early stages of the algorithm.
If maxdual = 0, the first encountered positive dual is used. Otherwise, the maximum positive dual, is used. The results for $x \geq 0$ will usually vary slightly depending on the choice.
Default: maxdual = 0

IMSL_DROP_TOLERANCE, *float* tol (Input)

This is a rank-determination tolerance. A candidate column

$$a = \begin{bmatrix} c \\ d \end{bmatrix}$$

has values eliminated below the first entry of $d$. The resulting value must satisfy the relative condition

$$\|d\|_2 > tol \times \|c\|_2$$

Otherwise the constraint remains satisfied because the column $a$ is linearly dependent on previously dropped columns.
Default: tol = sqrt(imsl_f_machine(3));

IMSL_SUPPLY_WORK_ARRAYS , *int* lwork, *float* work[], *int* liwork, *int* iwork[] (Input/Output)

The use of this optional argument will increase efficiency and avoid memory fragmentation run-time failures for large problems by allowing the user to provide the sizes and locations of the working arrays work and iwork. With maxt as the maximum number of threads that will be active, it is required that:

lwork ≥ maxt*(m*(n+2) + n), and liwork ≥ maxt*n.

Without the use of OpenMP and parallel threading, maxt=1.

IMSL_OPTIMIZED, *int* *flag (Output)

A 0-1 flag noting whether or not the optimum residual norm was obtained. A value of 1 indicates the optimum residual norm was obtained. A value of 0 occurs if the maximum number of iterations was reached.

| flag | Description |
|---|---|
| 0 | the maximum number of iterations was reached. |
| 1 | the optimum residual norm was obtained. |

IMSL_DUAL_SOLUTION, *float* **dual (Output)

An array of length n containing the dual vector, $w = A^T(Ax - b)$. This may not be optimal (all components may not satisfy $w \le 0$), if the maximum number of iterations occurs first.

IMSL_DUAL_SOLUTION_USER, *float* dual[] (Output)

Storage for dual provided by the user. See IMSL_DUAL_SOLUTION.

IMSL_RESIDUAL_NORM, *float* *rnorm (Output)

The value of the residual vector norm, $\|Ax-b\|_2$.

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length n containing the approximate solution vector, $x \ge 0$.

# Description

Function imsl_f_nonneg_least_squares computes the constrained least squares solution of $Ax \cong b$, by minimizing $\|Ax-b\|_2$ subject to $x \ge 0$. It uses the algorithm NNLS found in Charles L. Lawson and Richard J. Hanson, Solving Least Squares Problems, SIAM Publications, Chap. 23, (1995). The functionality for multiple threads and the constraint dropping strategy are new features. The original NNLS algorithm was silent about multiple threads; all dual components were computed when only one was used. Using the first encountered eligible variable to make non-active usually improves performance. An optimum solution is obtained in either approach. There is no restriction on the relative sizes of m and n.

# Examples

## Example 1

A model function of exponentials is

$$f(t) = c_1 + c_2\exp(-\lambda_2 t) + c_3\exp(-\lambda_3 t),\ t \ge 0$$

The exponential function argument parameters

$$\lambda_2 = 1,\ \lambda_3 = 5$$

are fixed. The coefficients

$$c_j \geq 0, \; j = 1, 2, 3$$

are estimated by sampling data values,

$$f\left(t_i\right), i = 1,...21$$

using non-negative least squares. The values used for the data are

$$t_i = 0.25i, \; i = 0,...20$$

with

$$c_1 = 1, \; c_2 = 0.2, \; c_3 = 0.3$$

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

#define M 21
#define N 3

int main() {
   int i;
   float a[M][N], b[M], *c;

   for (i = 0; i < M; i++) {
       /* Generate exponential values. This model is
          y(t) = c_0 + c_1*exp(-t) + c_2*exp(-5*t) */

       a[i][0] = 1.0;
       a[i][1] = exp(-(i*0.25));
       a[i][2] = exp(-(i*0.25)*5.0);

       /* Compute sample values */
       b[i] = a[i][0] + 0.2*a[i][1] + 0.3*a[i][2];
   }

   /* Solve for coefficients, constraining values
      to be non-negative. */
   c = imsl_f_nonneg_least_squares(M, N, &a[0][0], b, 0);

   /* With noise level = 0, solution should be (1, 0.2, 0.3) */
   imsl_f_write_matrix("Coefficients", 1, N, c, 0);
}
```

## Output

```
           Coefficients
         1            2            3
       1.0          0.2          0.3
```

## Example 2

The model function of exponentials is

$$f(t) = c_1 + c_2\exp(-\lambda_2 t) + c_3\exp(-\lambda_3 t) + n(t), t \geq 0$$

The values $\lambda_2, \lambda_3$ are the same as in Example 1. The function *n* (*t*) represents normally distributed random noise with a standard deviation $\sigma = 10^{-3}$ . A simulation is done with *n*s = 10001 samples for *n* (*t*). The resulting problem is solved using OpenMP. To check that the OpenMP results are correct, a loop computes the solutions without OpenMP followed by the same loop using OpenMP. The residual norms agree, showing that the routine returns the same values using OpenMP as without using OpenMP.

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

#define M 21
#define N 3
#define NS 10001

int main() {
#define BS(i_,j_)   bs[(i_)*M + (j_)]
#define X(i_,j_)   x[(i_)*N + (j_)]
   int thread_safe=1, seed=123457, i, *iwork, j, lwork, liwork, maxt;
   float b[M], *work, sigma=1.0e-3, a[M][N], rseq[NS], rpar[NS],
       *bs, *x;

   /* Allocate work memory for all threads that are
      used in the loops below. */
   maxt = omp_get_max_threads();
   lwork = maxt*(M*(N+2)+N);
   liwork = maxt*N;

   work = (float *) malloc(lwork * sizeof(float));
   iwork = (int *) malloc(liwork * sizeof(int));
   x = (float *) malloc(NS*N * sizeof(float));
   bs = (float *) malloc(NS*M * sizeof(float));

   for (i = 0; i < M; i++) {
       /* Generate matrix values.
          This model is y(t) =
          c_0 + c_1*exp(-t) + c_2*exp(-5*t) + n(t) */
       a[i][0] = 1.0;
       a[i][1] = exp(-(i*0.25));
       a[i][2] = exp(-(i*0.25)*5.0);
   }
```

```
    /* Solve for coefficients, constraining values to be non-negative.
       First use a sequential for loop. Then a parallel for loop.
       Record the residual norms and compare them. */

    imsl_random_seed_set(seed);
    /* First the sequential loop.
       Working memory is not included as an argument. */
    for (j = 0; j < NS; j++) {
        imsl_f_random_normal(M, IMSL_RETURN_USER, b, 0);

        /* Add normal pdf noise at the level sigma. */
        for (i=0; i<M; i++) {
            b[i] = sigma*b[i] + a[i][0] + 0.2*a[i][1] + 0.3*a[i][2];
            BS(j,i) = b[i];
        }

        imsl_f_nonneg_least_squares(M, N, &a[0][0], &BS(j,0),
            IMSL_RETURN_USER, &X(j,0),
            IMSL_RESIDUAL_NORM, &rseq[j],
            0);
    }

    /* Then the parallel for loop using OpenMP.
       Working memory is an optional argument. This is not required
       but helps prevent memory fragmentation. */

    /* Reset x for output for the OpenMP loop. */
    for (i = 0; i < NS*N; i++)
        x[i] = 0.0;

#pragma omp parallel for private(j)
    for (j = 0; j < NS; j++) {
        imsl_f_nonneg_least_squares(M, N, &a[0][0], &BS(j,0),
            IMSL_RETURN_USER, &X(j,0),
            IMSL_RESIDUAL_NORM, &rpar[j],
            IMSL_SUPPLY_WORK_ARRAYS, lwork, work, liwork, iwork,
            0);
    }

    /* Check that residual norms agree exactly for both loops. They
       should because the same problems are solved - one set
       sequentially and the next set in parallel. */
    for (j = 0; j < NS; j++) {
        /* Since the two loops solve the same set of problems, the
           residual norms must agree exactly. */
        if (rpar[j] != rseq[j]) {
            thread_safe = 0;
            break;
        }
    }
```

```
    if(thread_safe)
        printf("imsl_f_nonneg_least_squares is thread-safe.\n");
    else
        printf("imsl_f_nonneg_least_squares is not thread-safe.\n");

    system("pause");
}
```

### Output

```
imsl_f_nonneg_least_squares is thread-safe.
```

## Warning Errors

| | |
|---|---|
| IMSL_MAX_NNLS_ITER_REACHED | The maximum number of iterations was reached. The best answer will be returned. "itmax" = # was used. A larger value may help the algorithm complete. |

# lin_lsq_lin_constraints

Solves a linear least-squares problem with linear constraints.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_lin_lsq_lin_constraints (*int* nra, *int* nca, *int* ncon, *float* a[], *float* b[], *float* c[], *float* bl[], *float* bu[], *int* con_type[], *float* xlb[], *float* xub[], …, 0)

The type double function is imsl_d_lin_lsq_lin_constraints.

## Required Arguments

*int* nra (Input)
Number of least-squares equations.

*int* nca (Input)
Number of variables.

*int* ncon (Input)
Number of constraints.

*float* a[] (Input)
Array of size nra × nca containing the coefficients of the nra least-squares equations.

*float* b[] (Input)
Array of length nra containing the right-hand sides of the least-squares equations.

*float* c[] (Input)
Array of size ncon × nca containing the coefficients of the ncon constraints.

*float* bl[] (Input)
Array of length ncon containing the lower limit of the general constraints. If there is no lower limit on the *i*–th constraint, then bl[i] will not be referenced.

*float* bu[] (Input)
Array of length ncon containing the upper limit of the general constraints. If there is no upper limit on the *i*–th constraint, then bu[i] will not be referenced. If there is no range constraint, bl and bu can share the same storage.

*int* con_type[] (Input)
Array of length ncon indicating the type of constraints exclusive of simple bounds, where con_type[i] = 0, 1, 2, 3 indicates =, <=, >= and range constraints, respectively.

*float* `xlb[]` (Input)

Array of length `nca` containing the lower bound on the variables. If there is no lower bound on the *i*–th variable, then `xlb[i]` should be set to 1.0e30.

*float* `xub[]` (Input)

Array of length `nca` containing the upper bound on the variables. If there is no lower bound on the *i*–th variable, then `xub[i]` should be set to −1.0e30.

# Return Value

A pointer to the to a vector of length `nca` containing the approximate solution. To release this space, use `imsl_free`. If no solution was computed, then `NULL` is returned.

# Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `*imsl_f_lin_lsq_lin_constraints` (*int* `nra`, *int* `nca`, *int* `ncon`, *float* `a[]`, *float* `b[]`, *float* `c[]`, *float* `bl[]`, *float* `bu[]`, *int* `con_type[]`, *float* `xlb[]`, *float* `xub[]`,

`IMSL_RETURN_USER`, *float* `x[]`,

`IMSL_RESIDUAL`, *float* `**residual`,

`IMSL_RESIDUAL_USER`, *float* `residual_user[]`,

`IMSL_PRINT`,

`IMSL_ITMAX`, *int* `max_iter`,

`IMSL_REL_FCN_TOL`, *float* `rel_tol`,

`IMSL_ABS_FCN_TOL`, *float* `abs_tol`,

0)

# Optional Arguments

`IMSL_RETURN_USER`, *float* `x[]` (Output)

Store the solution in the user supplied vector `x` of length `nca`.

`IMSL_RESIDUAL`, *float* `**residual` (Output)

The address of a pointer to an array containing the residuals $b - Ax$ of the least-squares equations at the approximate solution.

`IMSL_RESIDUAL_USER`, *float* `residual_user[]` (Output)

Store the residuals in the user-supplied vector of length `nra`.

`IMSL_PRINT`,

Debug output flag. Choose this option if more detailed output is desired.

`IMSL_ITMAX`, *int* `max_iter` (Input)

Set the maximum number of add/drop iterations.

Default: `max_iter = 5*max(nra, nca)`

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

Relative rank determination tolerance to be used.

Default: `rel_tol = sqrt(imsl_f_machine(4))`

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

Absolute rank determination tolerance to be used.

Default: `abs_tol = sqrt(imsl_f_machine(4))`

# Description

The function `imsl_f_lin_lsq_lin_constraints` solves linear least-squares problems with linear constraints. These are systems of least-squares equations of the form

$$Ax \cong b$$

subject to

$$b_l \leq Cx \leq b_u$$

$$x_l \leq x \leq x_u$$

Here $A$ is the coefficient matrix of the least-squares equations, $b$ is the right-hand side, and $C$ is the coefficient matrix of the constraints. The vectors $b_l$, $b_u$, $x_l$ and $x_u$ are the lower and upper bounds on the constraints and the variables, respectively. The system is solved by defining dependent variables $y \equiv Cx$ and then solving the least-squares system with the lower and upper bounds on $x$ and $y$. The equation $Cx - y = 0$ is a set of equality constraints. These constraints are realized by heavy weighting, i.e., a penalty method, Hanson (1986, pp. 826-834).

# Examples

## Example 1

In this example, the following problem is solved in the least-squares sense:

$$3x_1 + 2x_2 + x_3 = 3.3$$

$$4x_1 + 2x_2 + x_3 = 2.2$$

$$2x_1 + 2x_2 + x_3 = 1.3$$

$$x_1 + x_2 + x_3 = 1.0$$

Subject to

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

$$0 \leq x_1 \leq 0.5$$

$$0 \leq x_2 \leq 0.5$$

$$0 \leq x_3 \leq 0.5$$

```
#include <imsl.h>

int main()
{
   int    nra = 4;
   int    nca = 3;
   int    ncon = 1;
   float *x;
   float  a[] = {3.0, 2.0, 1.0,
                  4.0, 2.0, 1.0,
                  2.0, 2.0, 1.0,
                  1.0, 1.0, 1.0};
   float  b[] = {3.3, 2.3, 1.3, 1.0};
   float  c[] = {1.0, 1.0, 1.0};
   float  xlb[] = {0.0, 0.0, 0.0};
   float  xub[] = {0.5, 0.5, 0.5};
   int    con_type[] = {1};
   float  bc[] = {1.0};

   x = imsl_f_lin_lsq_lin_constraints (nra, nca, ncon, a, b, c,
       bc, bc, con_type, xlb, xub,
       0);
   imsl_f_write_matrix ("Solution", 1, nca, x,
       0);
}
```

### Output

```
            Solution
        1           2           3
      0.5         0.3         0.2
```

## Example 2

The same problem solved in the first example is solved again. This time residuals of the least-squares equations at the approximate solution are returned, and the norm of the residual vector is printed. Both the solution and residuals are returned in user-supplied space.

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

int main()
{
   int    nra = 4;
   int    nca = 3;
```

```
    int    ncon = 1;
    float  x[3];
    float  residual[4];
    float  a[] = {3.0, 2.0, 1.0,
        4.0, 2.0, 1.0,
        2.0, 2.0, 1.0,
        1.0, 1.0, 1.0};
    float  b[] = {3.3, 2.3, 1.3, 1.0};
    float  c[] = {1.0, 1.0, 1.0};
    float  xlb[] = {0.0, 0.0, 0.0};
    float  xub[] = {0.5, 0.5, 0.5};
    int    con_type[] = {1};
    float  bc[] = {1.0};

    imsl_f_lin_lsq_lin_constraints (nra, nca, ncon, a, b, c,
        bc, bc, con_type, xlb, xub,
        IMSL_RETURN_USER, x,
        IMSL_RESIDUAL_USER, residual,
        0);

    imsl_f_write_matrix ("Solution", 1, nca, x, 0);
    imsl_f_write_matrix ("Residual", 1, nra, residual, 0);
    printf ("\n\nNorm of residual = %f\n",
        imsl_f_vector_norm (nra, residual, 0));
}
```

### Output

```
            Solution
      1           2           3
    0.5         0.3         0.2

                  Residual
      1           2           3           4
   -1.0         0.5         0.5        -0.0

Norm of residual = 1.224745
```

## Fatal Errors

| | |
|---|---|
| `IMSL_BAD_COLUMN_ORDER` | The input order of columns must be between 1 and "`nvar`" while input order = # and "`nvar`" = # are given. |
| `IMSL_BAD_POLARITY_FLAGS` | The bound polarity flags must be positive while component # flag "ibb[#]". |
| `IMSL_TOO_MANY_ITN` | Maximum numbers of iterations exceeded. |

# nonneg_matrix_factorization

OpenMP
more...

Given an $m \times n$ real matrix $A \geq 0$, and an integer $k \leq \min(m, n)$, compute a factorization $A \cong FG$. The matrix factors $F_{m \times k} \geq 0, G_{k \times n} \geq 0$ are computed to minimize the Frobenius, or sum of squares, norm of the error matrix: $E = \{e_{i,\ j}\} = A - FG$.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_nonneg_matrix_factorization (*int* m, *int* n, *int* k, *float* a[], *float* f[], *float* g[], …, 0)

The type ***double*** function is imsl_d_nonneg_matrix_factorization.

## Required Arguments

*int* m (Input)
The number of rows in the matrix.

*int* n (Input)
The number of columns in the matrix.

*int* k (Input)
The number of columns in the matrix ***F*** and rows in the matrix ***G***.

*float* a[] (Input)
An array of length m × n containing the ***A*** matrix.

*float* f[] (Input/Output)
An array of length m × k containing the ***F*** matrix. If IMSL_INITIAL_FACTORS is used, the sweeps begin using the input values for $F_{m \times k} \geq 0$.

*float* g[] (Output)
An array of length k × n containing the ***G*** matrix.

## Return Value

A scalar containing the Frobenius norm of the error matrix

$$E{:}error = \left(\sum_{i,\,j} e_{i,\,j}^2\right)^{1/2}$$

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_nonneg_matrix_factorization (*int* m, *int* n, *int* k, *float* a[], *float* f[], *float* g[],

IMSL_WEIGHT, *float* w[],

IMSL_INITIAL_FACTORS, *int* factors,

IMSL_ITMAX, *int* itmax,

IMSL_RESIDUAL_ERROR, *float* err,

IMSL_RELATIVE_ERROR, *float* rerr,

IMSL_STOPPING_CRITERION, *int* *reason,

IMSL_NSTEPS_TAKEN, *int* *nsteps,

0)

## Optional Arguments

IMSL_WEIGHT, *float* w[] (Input)

An array of length m × n containing the matrix $W \geq 0$ of weights that will be applied to the entries of $A \geq 0$ during the solution sweeps. The factorization obtained is $FG \cong W \circ A$, where the weights are applied element-wise.
Default: Weights are not applied, or equivalently, the weights all have value 1.

IMSL_INITIAL_FACTORS, *int* factors (Input)

A flag that signifies if the matrix *F* is given an input estimate. If factors = 0, start sweeps using

$$F = \begin{bmatrix} I_k \\ 0 \end{bmatrix}$$

Otherwise, use initial values in f as the matrix *F* to start the sweeps.
Default: factors = 0

IMSL_ITMAX, *int* itmax (Input)

The maximum number of sweeps allowed for alternately solving for $G \geq 0$, then $F \geq 0$.
Default: itmax = 2 * (m + n + 1)

IMSL_RESIDUAL_ERROR, *float* err (Input)

A scalar that will stop the sweeps at the first one satisfying *error* ≤ *err*.

Default: err = 0

IMSL_RELATIVE_ERROR, *float* rerr (Input)

A scalar that will stop the sweeps at the first one satisfying

$$error_{\text{iter}-2} - error_{\text{iter}-1} \le rerr \times error_{\text{iter}},\ iter > 2.$$

This test is made after three values of the error matrix norm have been computed. The sequence $\{error_{\text{iter}}\}$ is decreasing with increasing values of the iteration counter, *iter*. If $error_{\text{iter}} \ge error_{\text{iter}-1}$ occurs, the sweeps stop.

Default: rerr = (imsl_f_machine(3))$^{0.4}$.

IMSL_STOPPING_CRITERION, *int* *reason (Output)

This flag has the value 0,1,2 or 3 depending on which of the following conditions stopped the sweeps:

| reason | Description |
|---|---|
| 0 | Errors in user input occurred |
| 1 | Reached maximum iterations |
| 2 | Residual norm is small |
| 3 | Relative error convergence |

IMSL_NSTEPS_TAKEN, *int* *nsteps (Output)

The last value of the iteration count, $iter$, that gives the number of sweeps.

## Description

Function imsl_f_nonneg_matrix_factorization computes an approximation $A \cong FG$, or with weights, $W \circ A \cong FG$; the factors are constrained: $F_{m\times k} \ge 0,\ G_{k\times n} \ge 0$. The matrix factors $F_{m\times k} \ge 0,\ G_{k\times n} \ge 0$ are computed to minimize the Frobenius or sum of squares, norm of the error matrix: $E = \left\{e_{i,\ j}\right\} = A - FG$ .

The algorithm is based on Alternating Least Squares, presented by P. Paatero and U. Tapper, "Positive Matrix Factorization, etc." *Environmetrics*, (**5**), p. 111-126 (1994).

Each constrained least squares problem is solved using imsl_f_nonneg_least_squares. This process alternates between computing the batch of $n$ columns of $G$ and then the batch of $m$ rows of $F$. This constitutes a "sweep."

There is no restriction on the relative sizes of $m$ and $n$. The restrictions on the integer $k$ are $0 < k \le \min(m, n)$. When an initial matrix $G$ is to be used, instead of an initial $F$, repose the factorization in transposed form $A^T \cong G^T F^T$, or with weights, $A^T \circ W^T \cong G^T F^T$.

The matrix factors $F, G$ are not unique. In the function, the output rows of $G$ are scaled to have sum equal to the value 1. The scaled columns of $F$ are sorted so the column sums are non-increasing. This sort order is then applied to the rows of $G$.

## Example

Five customers, Beth, Dick, Fred, Joe and Kay make purchases at a convenience store.

| | Flour | Balloons | Beer | Sugar | Chips |
|---|---|---|---|---|---|
| Beth | | 3 | 8 | | 1 |
| Dick | | 2 | 5 | 1 | |
| Fred | 5 | | 1 | 10 | |
| Joe | | 20 | 40 | 2 | 1 |
| Kay | 10 | | 1 | 10 | 1 |

This matrix $A_{5\times 5}$ of customers versus items purchased is approximated by a non-negative matrix factorization, using $k = 2$: $A \cong FG$. The example is taken from one due to H. Jin and M. Saunders, "***Exploring Nonnegative Matrix Factorization***," A Workshop on Algorithms for Massive Data Sets, Stanford University, June 25-28, (2008).

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

#define M 5
#define N 5
#define K 2

int main() {
   float a[M][N]= {
       { 0, 3, 8, 0, 1},
       { 0, 2, 5, 1, 0},
       { 5, 0, 1, 10, 0},
       { 0, 20, 40, 2, 1},
       {10, 0, 1, 10, 1}
   };
   float error, f[M*K], g[K*N];
   int nsteps, reason;

   /* Solve for factors, constraining values to be non-negative.
      Get reason for stopping and number of sweeps. */
   error = imsl_f_nonneg_matrix_factorization(M, N, K, &a[0][0], f, g,
       IMSL_STOPPING_CRITERION, &reason,
       IMSL_NSTEPS_TAKEN, &nsteps,
       0);

   imsl_f_write_matrix("Matrix Factor F", M, K, f, 0);
   imsl_f_write_matrix("Matrix Factor G", K, N, g, 0);
```

```
   printf("\nFrobenius Norm of E=A-F*G is %e\n", error);
   printf("Reason for stopping sweeps: %d\n", reason);
   printf("Number of sweeps taken: %d\n", nsteps);
}
```

## Output

```
     Matrix Factor F
           1           2
1      11.96        0.00
2       7.51        0.94
3       0.33       16.61
4      62.90        0.13
5       0.00       21.35

                         Matrix Factor G
           1           2           3           4           5
1     0.0000      0.3150      0.6373      0.0298      0.0178
2     0.4048      0.0000      0.0473      0.5190      0.0288

Frobenius Norm of E=A-F*G is 3.195350e+000
Reason for stopping sweeps: 3
Number of sweeps taken: 10
```

# lin_svd_gen

more...

Computes the SVD, $A = USV^T$, of a real rectangular matrix $A$. An approximate generalized inverse and rank of $A$ also can be computed.

## Synopsis

*#include* `<imsl.h>`

*float* `*imsl_f_lin_svd_gen` (*int* `m`, *int* `n`, *float* `a[]`, …, 0)

The type *double* procedure is `imsl_d_lin_svd_gen`.

## Required Arguments

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

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

*float* `a[]` (Input)
Array of size `m` × `n` containing the matrix.

## Return Value

If no optional arguments are used, `imsl_f_lin_svd_gen` returns a pointer to an array of size min (`m`, `n`) containing the ordered singular values of the matrix. To release this space, use `imsl_free`. If no value can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `*imsl_f_lin_svd_gen` (*int* `m`, *int* `n`, *float* `a[]`,

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

`IMSL_RETURN_USER`, *float* `s[]`,

IMSL_RANK, *float* tol, *int* *rank,

IMSL_U, *float* **p_u,

IMSL_U_USER, *float* u[],

IMSL_U_COL_DIM, *int* u_col_dim,

IMSL_V, *float* **p_v,

IMSL_V_USER, *float* v[],

IMSL_V_COL_DIM, *int* v_col_dim,

IMSL_INVERSE, *float* **p_gen_inva,

IMSL_INVERSE_USER, *float* gen_inva[],

IMSL_INV_COL_DIM, *int* gen_inva_col_dim,

0)

## Optional Arguments

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

IMSL_RETURN_USER, *float* s[] (Output)
A user-allocated array of size min (m+*1*, n) containing the singular values of *A* in its first min (m, n) positions in nonincreasing order. If IMSL_RETURN_USER is used, the return value of imsl_f_lin_svd_gen is s.

IMSL_RANK, *float* tol, *int* *rank (Input/Output)

*float* tol (Input)
Scalar containing the tolerance used to determine when a singular value is negligible and replaced by the value zero. If tol > 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i}$ ≤ tol. If tol < 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i}$ ≤ |tol| * $\|A\|_\infty$. In this case, |tol| should be an estimate of relative error or uncertainty in the data.

*int* *rank (Input/Output)
Integer containing an estimate of the rank of *A*.

IMSL_U, *float* **p_u (Output)
**p_u: The address of a pointer to an array of size m × min (m, n) containing the min (m, n) left-singular vectors of *A*. On return, the necessary space is allocated by imsl_f_lin_svd_gen. Typically, *float* *p_u is declared, and &p_u is used as an argument.

IMSL_U_USER, *float* u[] (Output)

u[]: The address of a pointer to an array of size m × min (m, n) containing the min (m, n) left-singular vectors of *A*. If m ≥ n, the left-singular vectors can be returned using the storage locations of the array a.

IMSL_U_COL_DIM, *int* u_col_dim (Input)

The column dimension of the array containing the left-singular vectors.

Default: u_col_dim = min (m, n)

IMSL_V, *float* **p_v (Output)

**p_v: The address of a pointer to an array of size n × n containing the right singular vectors of *A*. On return, the necessary space is allocated by imsl_f_lin_svd_gen. Typically, *float* *p_v is declared, and &p_v is used as an argument.

IMSL_V_USER, *float* v[] (Output)

v[]: The address of a pointer to an array of size n × n containing the right singular vectors of *A*. The right-singular vectors can be returned using the storage locations of the array a. Note that the return of the left- and right-singular vectors cannot use the storage locations of a simultaneously.

IMSL_V_COL_DIM, *int* v_col_dim (Input)

The column dimension of the array containing the right-singular vectors.

Default: v_col_dim = n

IMSL_INVERSE, *float* **p_gen_inva (Output)

The address of a pointer to an array of size n × m containing the generalized inverse of the matrix *A*. On return, the necessary space is allocated by imsl_f_lin_svd_gen. Typically, *float* *p_gen_inva is declared, and &p_gen_inva is used as an argument.

IMSL_INVERSE_USER, *float* gen_inva[] (Output)

A user-allocated array of size n × m containing the general inverse of the matrix *A*.

IMSL_INV_COL_DIM, *int* gen_inva_col_dim (Input)

The column dimension of the array containing the general inverse of the matrix *A*.

Default: gen_inva_col_dim = m

## Description

The function imsl_f_lin_svd_gen computes the singular value decomposition of a real matrix *A*. It first reduces the matrix *A* to a bidiagonal matrix *B* by pre- and post-multiplying Householder transformations. Then, the singular value decomposition of *B* is computed using the implicit-shifted *QR* algorithm. An estimate of the rank of the matrix *A* is obtained by finding the smallest integer *k* such that $s_{k,k}$ ≤ tol or $s_{k,k}$ ≤ |tol|*$\|A\|_\infty$. Since $s_{i+1,i+1} \le s_{i,i}$, it follows that all the $s_{i,i}$ satisfy the same inequality for $i = k, \ldots,$ min (m, n) − 1. The rank is set to the value $k - 1$. If $A = USV^T$, its generalized inverse is $A^+ = VS^+ U^T$. Here,

$$S^{+} = diag\left(s_{1,1}^{-1}, \dots, s_{i,i}^{-1}, 0, \dots, 0\right)$$

Only singular values that are not negligible are reciprocated. If `IMSL_INVERSE` or `IMSL_INVERSE_USER` is specified, the function first computes the singular value decomposition of the matrix *A*. The generalized inverse is then computed. The function `imsl_f_lin_svd_gen` fails if the *QR* algorithm does not converge after 30 iterations isolating an individual singular value.

# Examples

## Example 1

This example computes the singular values of a real 6 × 4 matrix.

```
#include <imsl.h>

float a[] =  {1.0, 2.0, 1.0, 4.0,
              3.0, 2.0, 1.0, 3.0,
              4.0, 3.0, 1.0, 4.0,
              2.0, 1.0, 3.0, 1.0,
              1.0, 5.0, 2.0, 2.0,
              1.0, 2.0, 2.0, 3.0};

int main()
{
   int          m = 6, n = 4;
   float        *s;
                             /* Compute singular values */
   s = imsl_f_lin_svd_gen (m, n, a, 0);
                             /* Print singular values */
   imsl_f_write_matrix ("Singular values", 1, n, s, 0);
}
```

### Output

```
         Singular values
    1          2          3          4
11.49       3.27       2.65       2.09
```

## Example 2

This example computes the singular value decomposition of the 6 × 4 real matrix *A*. The singular values are returned in the user-provided array. The matrices *U* and *V* are returned in the space provided by the function `imsl_f_lin_svd_gen`.

```
#include <imsl.h>

float a[] = {1.0, 2.0, 1.0, 4.0,
```

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

int main()
{
   int           m = 6, n = 4;
   float        s[4], *p_u, *p_v;
                                   /* Compute SVD */
   imsl_f_lin_svd_gen (m, n, a,
                      IMSL_RETURN_USER, s,
                      IMSL_U, &p_u,
                      IMSL_V, &p_v,
                      0);
                                   /* Print decomposition*/

   imsl_f_write_matrix ("Singular values, S", 1, n, s, 0);
   imsl_f_write_matrix ("Left singular vectors, U", m, n, p_u, 0);
   imsl_f_write_matrix ("Right singular vectors, V", n, n, p_v, 0);
}
```

### Output

```
             Singular values, S
          1           2           3           4
      11.49        3.27        2.65        2.09

           Left singular vectors, U
           1           2           3           4
1    -0.3805      0.1197      0.4391     -0.5654
2    -0.4038      0.3451     -0.0566      0.2148
3    -0.5451      0.4293      0.0514      0.4321
4    -0.2648     -0.0683     -0.8839     -0.2153
5    -0.4463     -0.8168      0.1419      0.3213
6    -0.3546     -0.1021     -0.0043     -0.5458

           Right singular vectors, V
           1           2           3           4
1    -0.4443      0.5555     -0.4354      0.5518
2    -0.5581     -0.6543      0.2775      0.4283
3    -0.3244     -0.3514     -0.7321     -0.4851
4    -0.6212      0.3739      0.4444     -0.5261
```

## Example 3

This example computes the rank and generalized inverse of a 3 × 2 matrix *A*. The rank and the 2 × 3 generalized inverse matrix $A^+$ are printed.

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

float a[] =
   {1.0,  0.0,
    1.0,  1.0,
    100.0, -50.0};

int main()
{
   int          m = 3, n = 2;
   float        tol;
   float        gen_inva[6];
   float       *s;
   int         rank;

   /* Compute generalized inverse */
   tol = 1.e-4;

   s = imsl_f_lin_svd_gen (m, n, a,
       IMSL_RANK, tol, &rank,
       IMSL_INVERSE_USER, gen_inva,
       IMSL_INV_COL_DIM, m,
       0);

   /* Print rank, singular values and */
   /* generalized inverse. */
   printf ("Rank of matrix = %2d", rank);
   imsl_f_write_matrix ("Singular values", 1, n, s, 0);
   imsl_f_write_matrix ("Generalized inverse", n, m, gen_inva,
       IMSL_A_COL_DIM, m,
       0);
}
```

### Output

```
Rank of matrix = 2
   Singular values
        1          2
    111.8        1.4

        Generalized inverse
           1          2          3
1      0.100      0.300      0.006
2      0.200      0.600     -0.008
```

## Warning Errors

| | |
|---|---|
| `IMSL_SLOWCONVERGENT_MATRIX` | Convergence cannot be reached after 30 iterations. |

# lin_svd_gen (complex)

more...

Computes the SVD, ***A*** = *USV*^H^, of a complex rectangular matrix ***A***. An approximate generalized inverse and rank of *A* also can be computed.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_lin_svd_gen (*int* m, *int* n, *f_complex* a[], ..., 0)

The type *d_complex function* is imsl_z_lin_svd_gen.

## Required Arguments

*int* m (Input)
Number of rows in the matrix.

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

*f_complex* a[] (Input)
Array of size m × n containing the matrix.

## Return Value

Using only required arguments, imsl_c_lin_svd_gen returns a pointer to a complex array of length min (m, n) containing the singular values of the matrix. To release this space, use imsl_free. If no value can be computed then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_lin_svd_gen (*int* m, *int* n, *f_complex* a[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_RETURN_USER, *f_complex* s[],

IMSL_RANK, *float* tol, *int* *rank,

IMSL_U, *f_complex* **p_u,

IMSL_U_USER, *f_complex* u[],

IMSL_U_COL_DIM, *int* u_col_dim,

IMSL_V, *f_complex* **p_v,

IMSL_V_USER, *f_complex* v[],

IMSL_V_COL_DIM, *int* v_col_dim,

IMSL_INVERSE, *f_complex* **p_gen_inva,

IMSL_INVERSE_USER, *f_complex* gen_inva[],

IMSL_INV_COL_DIM, *int* gen_inva_col_dim,

0)

## Optional Arguments

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

IMSL_RETURN_USER, *f_complex* s[] (Output)
A user-allocated array of length min (m, n) containing the singular values of *A* in its first min (m, *n*) positions in nonincreasing order. The complex entries are all real. If IMSL_RETURN_USER is used, the return value of imsl_c_lin_svd_gen is s.

IMSL_RANK, *float* tol, *int* *rank (Input/Output)

*float* tol (Input)
Scalar containing the tolerance used to determine when a singular value is negligible and replaced by the value zero. If tol > 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i}$ ≤ tol. If tol < 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i}$ ≤ |tol|*$\|A\|_\infty$. In this case, should be an estimate of relative error or uncertainty in the data.

*int* *rank (Input/Output)
Integer containing an estimate of the rank of *A*.

`IMSL_U`, *f_complex* `**p_u` (Output)
: The address of a pointer to an array of size `m` × min (`m`, `n`) containing the min (`m`, `n`) left-singular vectors of *A*. On return, the necessary space is allocated by `imsl_c_lin_svd_gen`. Typically, *f_complex* `*p_u` is declared, and `&p_u` is used as an argument.

`IMSL_U_USER`, *f_complex* `u[]` (Output)
: The address of a pointer to an array of size `m` × min (`m`, `n`) containing the min (`m`, `n`) left-singular vectors of *A*. If `m` ≥ `n`, the left-singular vectors can be returned using the storage locations of the array `a`.

`IMSL_U_COL_DIM`, *int* `u_col_dim` (Input)
: The column dimension of the array containing the left-singular vectors.
Default: `u_col_dim` = min (`m`, `n`)

`IMSL_V`, *f_complex* `**p_v` (Output)
: The address of a pointer to an array of size `n` × `n` containing the right singular vectors of *A*. On return, the necessary space is allocated by `imsl_c_lin_svd_gen`. Typically, *f_complex* `*p_v` is declared, and `&p_v` is used as an argument.

`IMSL_V_USER`, *f_complex* `v[]` (Output)
: The address of a pointer to an array of size `n` × `n` containing the right singular vectors of *A*. The right-singular vectors can be returned using the storage locations of the array `a`. Note that the return of the left and right-singular vectors cannot use the storage locations of `a` simultaneously.

`IMSL_V_COL_DIM`, *int* `v_col_dim` (Input)
: The column dimension of the array containing the right-singular vectors.
Default: `v_col_dim` = `n`

`IMSL_INVERSE`, *f_complex* `**p_gen_inva` (Output)
: The address of a pointer to an array of size `n` × `m` containing the generalized inverse of the matrix *A*. On return, the necessary space is allocated by `imsl_c_lin_svd_gen`. Typically, *f_complex* `*p_gen_inva` is declared, and `&p_gen_inva` is used as an argument.

`IMSL_INVERSE_USER`, *f_complex* `gen_inva[]` (Output)
: A user-allocated array of size `n` × `m` containing the general inverse of the matrix *A*.

`IMSL_INV_COL_DIM`, *int* `gen_inva_col_dim` (Input)
: The column dimension of the array containing the general inverse of the matrix *A*.
Default: `gen_inva_col_dim` = `m`

## Description

The function `imsl_c_lin_svd_gen` computes the singular value decomposition of a complex matrix *A*. It first reduces the matrix *A* to a bidiagonal matrix *B* by pre- and post-multiplying Householder transformations. Then, the singular value decomposition of *B* is computed using the implicit-shifted *QR* algorithm. An estimate of the rank of the matrix $A$ is obtained by finding the smallest integer $k$ such that $s_{k,k} \le$ `tol` or $s_{k,k} \le |\mathtt{tol}| * \|A\|_\infty$. Since $s_{i+1,i+1} \le s_{i,i}$, it follows that all the $s_{i,i}$ satisfy the same inequality for $i = k, \ldots, \min(m, n) - 1$. The rank is set to the value $k - 1$. If $A = USV^H$, its generalized inverse is $A^+ = VS^+ U^T$.

Here,

$$S^+ = diag\left(s_{1,1}^{-1}, \ldots, s_{i,i}^{-1}, 0, \ldots, 0\right)$$

Only singular values that are not negligible are reciprocated. If `IMSL_INVERSE` or `IMSL_INVERSE_USER` is specified, the function first computes the singular value decomposition of the matrix $A$. The generalized inverse is then computed. The function `imsl_c_lin_svd_gen` fails if the $QR$ algorithm does not converge after 30 iterations isolating an individual singular value.

# Examples

## Example 1

This example computes the singular values of a 6 × 3 complex matrix.

```
#include <imsl.h>
int main()
{
   int              m = 6, n = 3;
   f_complex        *s;
   f_complex a[] = {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0},
                       {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0},
                       {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0},
                       {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0},
                       {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0},
                       {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}};
                                    /* Compute singular values */
   s = imsl_c_lin_svd_gen (m, n, a, 0);
                                    /* Print singular values */
   imsl_c_write_matrix ("Singular values", 1, n, s, 0);
}
```

### Output

```
                           Singular values
                   1                      2                      3
(    11.77,      0.00) (     9.30,      0.00) (     4.99,      0.00)
```

## Example 2

This example computes the singular value decomposition of the 6 × 3 complex matrix $A$. The singular values are returned in the user-provided array. The matrices $U$ and $V$ are returned in the space provided by the function `imsl_c_lin_svd_gen`.

```
#include <imsl.h>

int main()
{
   int          m = 6, n = 3;
   f_complex    s[3], *p_u, *p_v;
   f_complex a[] = {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0},
                    {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0},

                    {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0},
                    {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0},
                    {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0},
                    {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}};
                                    /* Compute SVD of a */
       imsl_c_lin_svd_gen (m, n, a,
                          IMSL_RETURN_USER, s,
                          IMSL_U, &p_u,
                          IMSL_V, &p_v,
                          0);
                                    /* Print decomposition factors */
       imsl_c_write_matrix ("Singular values, S", 1, n, s, 0);
       imsl_c_write_matrix ("Left singular vectors, U", m, n, p_u, 0);
       imsl_c_write_matrix ("Right singular vectors, V", n, n, p_v, 0);
}
```

## Output

```
                          Singular values, S
                    1                      2                      3
(    11.77,      0.00)  (     9.30,      0.00)  (     4.99,      0.00)

                         Left singular vectors, U
                      1                      2                      3
1 (   0.1968,    0.2186) (   0.5011,    0.0217) (  -0.2007,   -0.1003)
2 (   0.3443,   -0.3542) (  -0.2933,    0.0248) (   0.1155,   -0.2338)
3 (   0.1457,    0.2307) (  -0.5424,    0.1381) (  -0.4361,   -0.4407)
4 (   0.3016,   -0.0844) (   0.2157,    0.2659) (  -0.0523,   -0.0894)
5 (   0.2283,   -0.6008) (  -0.1325,    0.1433) (   0.3152,   -0.0090)
6 (   0.2876,   -0.0350) (   0.4377,   -0.0400) (   0.0458,   -0.6205)

                        Right singular vectors, V
                      1                      2                      3
1 (   0.6616,    0.0000) (  -0.2651,    0.0000) (  -0.7014,    0.0000)
2 (   0.7355,    0.0379) (   0.3850,   -0.0707) (   0.5482,    0.0624)
3 (   0.0507,   -0.1317) (   0.1724,    0.8642) (  -0.0173,   -0.4509)
```

## Example 3

This example computes the rank and generalized inverse of a 6 × 4 matrix *A*. The rank and the 4 × 6 generalized inverse matrix $A^+$ are printed.

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

int main()
{
   int          m = 6, n = 4, rank;
   float        tol;
   f_complex    gen_inv[24], *s;
   f_complex a[] = {{1.0, 2.0}, {3.0, 2.0}, {1.0,-4.0}, {1.0,0.0},
                      {3.0,-2.0}, {2.0,-4.0}, {1.0, 3.0}, {0.0,1.0},
                      {4.0, 3.0}, {-2.0,1.0}, {1.0, 4.0}, {0.0,0.0},
                      {2.0,-1.0}, {3.0, 0.0}, {3.0,-1.0}, {2.0,1.0},
                      {1.0,-5.0}, {2.0,-5.0}, {2.0, 2.0}, {1.0,3.1},
                      {1.0, 2.0}, {4.0,-2.0}, {2.0,-3.0}, {1.4,1.9}};

   /* Factor a */
   tol = 1.e-4;

   s = imsl_c_lin_svd_gen (m, n, a,
       IMSL_RANK, tol, &rank,
       IMSL_INVERSE_USER, gen_inv,
       IMSL_INV_COL_DIM, m,
       0);

   /* Print rank and generalized inverse matrix */
   printf ("Rank = %2d", rank);
   imsl_c_write_matrix ("Singular values", 1, n, s,
       0);
   imsl_c_write_matrix ("Generalized inverse", n, m, gen_inv,
       IMSL_A_COL_DIM, m,
       0);
}
```

### Output

```
Rank = 4
                           Singular values
                    1                      2                      3
(    12.13,      0.00)  (     9.53,      0.00)  (     5.67,      0.00)

                    4
(     1.74,      0.00)

                          Generalized inverse
                      1                      2                      3
1 (   0.0266,    0.0164) (  -0.0185,    0.0453) (   0.0720,    0.0700)
```

```
2 (   0.0061,   0.0280) (   0.0820,  -0.1156) (  -0.0410,  -0.0242)
3 (  -0.0019,  -0.0572) (   0.1174,   0.0812) (   0.0499,   0.0463)
4 (   0.0380,   0.0298) (  -0.0758,  -0.2158) (   0.0356,  -0.0557)

                      4                     5                     6
1 (  -0.0220,  -0.0428) (  -0.0003,  -0.0709) (   0.0254,   0.1050)
2 (   0.0959,   0.0885) (  -0.0187,   0.0287) (  -0.0218,  -0.1109)
3 (  -0.0234,   0.1033) (  -0.0769,   0.0103) (   0.0810,  -0.1074)
4 (   0.2918,  -0.0763) (   0.0881,   0.2070) (  -0.1531,   0.0814)
```

## Warning Errors

| | |
|---|---|
| IMSL_SLOWCONVERGENT_MATRIX | Convergence cannot be reached after 30 iterations. |

# lin_sol_nonnegdef

Solves a real symmetric nonnegative definite system of linear equations $Ax = b$. Using options, computes a Cholesky factorization of the matrix $A$, such that $A = R^T R = LL^T$. Computes the solution to $Ax = b$ given the Cholesky factor.

## Synopsis

*#include* <imsl.h>

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

The type *double* function is imsl_d_lin_sol_nonnegdef.

## Required Arguments

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

*float* a[] (Input)
Array of size n × n containing the matrix.

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

## Return Value

Using required arguments, imsl_f_lin_sol_nonnegdef returns a pointer to a solution *x* of the linear system. To release this space, use imsl_free. If no value can be computed, NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_lin_sol_nonnegdef (*int* n, *float* a[], *float* b[],

IMSL_RETURN_USER, *float* x[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_FACTOR, *float* **p_factor,

IMSL_FACTOR_USER, *float* factor[],

IMSL_FAC_COL_DIM, *int* fac_col_dim,

IMSL_INVERSE, *float* **p_inva,

IMSL_INVERSE_USER, *float* inva[],

IMSL_INV_COL_DIM, *int* inv_col_dim,

IMSL_TOLERANCE, *float* tol,

IMSL_FACTOR_ONLY,

IMSL_SOLVE_ONLY,

IMSL_INVERSE_ONLY,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* x[] (Output)

A user-allocated array of length n containing the solution *x*. When this option is specified, no storage is allocated for the solution, and imsl_f_lin_sol_nonnegdef returns a pointer to the array x.

IMSL_A_COL_DIM, *int* a_col_dim (Input)

The column dimension of the array a.
Default: a_col_dim = n

IMSL_FACTOR, *float* **p_factor (Output)

The address of a pointer to an array of size n × n containing the $LL^T$ factorization of *A*. When this option is specified, the space for the factor matrix is allocated by imsl_f_lin_sol_nonnegdef. The lower-triangular part of the factor array contains *L*, and the upper-triangular part contains $L^TR$. Typically, *float* *p_factor is declared, and &p_factor is used as an argument.

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

A user-allocated array of size n × n containing the $LL^T$ factorization of *A*. The lower-triangular part of factor contains *L*, and the upper-triangular part contains $L^T$. If a is not needed, a and factor can be the same storage locations. If IMSL_SOLVE is specified, this parameter is ***input***; otherwise, it is ***output***.

IMSL_FAC_COL_DIM, *int* fac_col_dim (Input)

The column dimension of the array containing the $LL^T$ factorization.
Default: fac_col_dim = n

IMSL_INVERSE, *float* **p_inva (Output)

The address of a pointer to an array of size n × n containing the inverse of *A*. The space for this array is allocated by imsl_f_lin_sol_nonnegdef. Typically, *float* *p_inva is declared, and &p_inva is used as an argument.

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

A user-allocated array of size `n` × `n` containing the inverse of *A*. If `a` is not needed, `a` and `factor` can be the same storage locations. The storage locations for *A* cannot be the factorization and the inverse of *A* at the same time.

`IMSL_INV_COL_DIM`, *int* `inva_col_dim` (Input)

The column dimension of the array containing the inverse of *A*.

Default: `inva_col_dim` = `n`

`IMSL_TOLERANCE`, *float* `tol` (Input)

Tolerance used in determining linear dependence. See the documentation for `-imsl_f_machine` (`imsl_f_machine(float)`) in Chapter 12, "Utilities."

Default: `tol` = 100* `imsl_f_machine(4)`

`IMSL_FACTOR_ONLY`

Compute the $LL^T$ factorization of *A* only. The argument `b` is ignored, and either the optional argument `IMSL_FACTOR` or `IMSL_FACTOR_USER` is required.

`IMSL_SOLVE_ONLY`

Solve $Ax = b$ using the factorization previously computed by this function. The argument `a` is ignored, and the optional argument `IMSL_FACTOR_USER` is required.

`IMSL_INVERSE_ONLY`

Compute the inverse of *A* only. The argument `b` is ignored, and either the optional argument `IMSL_INVERSE` or `IMSL_INVERSE_USER` is required.

## Description

The function `imsl_f_lin_sol_nonnegdef` solves a system of linear algebraic equations having a symmetric nonnegative definite (positive semidefinite) coefficient matrix. It first computes a Cholesky ($LL^T$ or $R^TR$) factorization of the coefficient matrix *A*.

The factorization algorithm is based on the work of Healy (1968) and proceeds sequentially by columns. The *i*-th column is declared to be linearly dependent on the first $i - 1$ columns if

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

where ε (specified in `tol`) may be set by the user. When a linear dependence is declared, all elements in the *i*-th row of *R* (column of *L*) are set to zero.

Modifications due to Farebrother and Berry (1974) and Barrett and Healy (1978) for checking for matrices that are not nonnegative definite also are incorporated. The function `imsl_f_lin_sol_nonnegdef` declares *A* to not be nonnegative definite and issues an error message if either of the following conditions are satisfied:

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

2. $r_{ii} = 0$ and $|a_{ik} - \sum_{j=1}^{i-1} r_{ji} r_{jk}| > \varepsilon \sqrt{a_{ii} a_{kk}}, k > i$

Healy's (1968) algorithm and the function `imsl_f_lin_sol_nonnegdef` permit the matrices *A* and *R* to occupy the same storage. Barrett and Healy (1978) in their remark neglect this fact. The function `imsl_f_lin_sol_nonnegdef` uses

$$\sum_{j=1}^{i-1} r_{ij}^2$$

for $a_{ii}$ in the above condition 2 to remedy this problem.

If an inverse of the matrix *A* is required and the matrix is not (numerically) positive definite, then the resulting inverse is a symmetric $g_2$ inverse of *A*. For a matrix *G* to be a $g_2$ inverse of a matrix *A*, *G* must satisfy conditions 1 and 2 for the Moore-Penrose inverse, but generally fail conditions 3 and 4. The four conditions for *G* to be a Moore-Penrose inverse of *A* are as follows:

1. *AGA* = *A*
2. *GAG* = *G*
3. *AG* is symmetric
4. *GA* is symmetric

The solution of the linear system *Ax* = *b* is computed by solving the factored version of the linear system $R^T Rx = b$ as two successive triangular linear systems. In solving the triangular linear systems, if the elements of a row of *R* are all zero, the corresponding element of the solution vector is set to zero. For a detailed description of the algorithm, see Section 2 in Sallas and Lionti (1988).

# Examples

## Example 1

A solution to a system of four linear equations is obtained. Maindonald (1984, pp. 83-86 and 104-105) discusses the computations for the factorization and solution to this problem.

```
#include <imsl.h>

int main()
{
   int         n = 4;
   float       *x;
   float       a[] = {36.0, 12.0, 30.0, 6.0,
                      12.0, 20.0, 2.0, 10.0,
                      30.0, 2.0, 29.0, 1.0,
```

```
                        6.0, 10.0, 1.0, 14.0};
    float       b[] = {18.0, 22.0, 7.0, 20.0};

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

### Output

```
            Solution, x
   1           2           3           4
0.167       0.500       0.000       1.000
```

## Example 2

The symmetric nonnegative definite matrix in the initial example is used to compute the factorization only in the first call to `lin_sol_nonnegdef`. The space needed for the factor is provided by the user. On the second call, both the $LL^T$ factorization and the right-hand side vector in the first example are used as the input to compute a solution *x*. It also illustrates another way to obtain the solution array *x*.

```
#include <imsl.h>

int main()
{
    int         n = 4, a_col_dim = 6;
    float       factor[36], x[5];
    float       a[] = {36.0, 12.0, 30.0, 6.0,
                       12.0, 20.0, 2.0, 10.0,
                       30.0, 2.0, 29.0, 1.0,
                        6.0, 10.0, 1.0, 14.0};
    float       b[] = {18.0, 22.0, 7.0, 20.0};
                                /* Factor A */
    imsl_f_lin_sol_nonnegdef(n, a, b,
                             IMSL_FACTOR_USER, factor,
                             IMSL_FAC_COL_DIM, a_col_dim,
                             IMSL_FACTOR_ONLY,
                             0);
                                /* NULL is returned in */
                                /* this case. Another */
                                /* way to obtain the   */
                                /* factor is to use the */
                                /* IMSL_FACTOR option. */
    imsl_f_write_matrix("factor", n, n, factor,
                        IMSL_A_COL_DIM, a_col_dim,
                        0);
                                /* Get the solution using */
                                /* the factorized matrix. */
    imsl_f_lin_sol_nonnegdef(n, a, b,
```

```
                                IMSL_FACTOR_USER, factor,
                                IMSL_FAC_COL_DIM, a_col_dim,
                                IMSL_RETURN_USER, x,
                                IMSL_SOLVE_ONLY,
                                0);
   imsl_f_write_matrix("Solution, x, of Ax = b", 1, n, x, 0);
}
```

### Output

```
                    Factor
           1           2           3           4
1          6           2           5           1
2          2           4          -2           2
3          5          -2           0           0
4          1           2           0           3

           Solution, x, of Ax = b
        1           2           3           4
    0.167       0.500       0.000       1.000
```

## Example 3

This example uses the `IMSL_INVERSE` option to compute the symmetric ***g*** inverse of the symmetric nonnegative matrix in the first example. Maindonald (1984, p. 106) discusses the computations for this problem.

```
#include <imsl.h>

int main()
{
   int         n = 4;
   float       *p_a_inva, *p_a_inva_a, *p_inva;
   float       a[] =
       {36.0, 12.0, 30.0, 6.0,
        12.0, 20.0, 2.0, 10.0,
        30.0, 2.0, 29.0, 1.0,
        6.0, 10.0, 1.0, 14.0};

   /* Get g2_inverse(a) */
   imsl_f_lin_sol_nonnegdef(n, a, NULL,
       IMSL_INVERSE, &p_inva,
       IMSL_INVERSE_ONLY,
       0);

   /* Form a*g2_inverse(a) */
   p_a_inva = imsl_f_mat_mul_rect("A*B",
       IMSL_A_MATRIX, n, n, a,
       IMSL_B_MATRIX, n, n, p_inva,
       0);
```

```
   /* Form a*g2_inverse(a)*a */
   p_a_inva_a = imsl_f_mat_mul_rect("A*B",
       IMSL_A_MATRIX, n, n, p_a_inva,
       IMSL_B_MATRIX, n, n, a,
       0);

   imsl_f_write_matrix("The g2 inverse of a", n, n, p_inva,
       0);
   imsl_f_write_matrix("a*g2_inverse(a)\nviolates condition 3 of"
       " the M-P inverse", n, n, p_a_inva,
       0);
   imsl_f_write_matrix("a = a*g2_inverse(a)*a\ncondition 1 of"
       " the M-P inverse", n, n, p_a_inva_a,
       0);
}
```

### Output

```
             The g2 inverse of a
          1           2           3           4
1    0.0347     -0.0208      0.0000      0.0000
2   -0.0208      0.0903      0.0000     -0.0556
3    0.0000      0.0000      0.0000      0.0000
4    0.0000     -0.0556      0.0000      0.1111

              a*g2_inverse(a)
   violates condition 3 of the M-P inverse
          1           2           3           4
1       1.0        -0.0         0.0         0.0
2       0.0         1.0         0.0         0.0
3       1.0        -0.5         0.0         0.0
4       0.0        -0.0         0.0         1.0

             a = a*g2_inverse(a)*a
       condition 1 of the M-P inverse
          1           2           3           4
1        36          12          30           6
2        12          20           2          10
3        30           2          29           1
4         6          10           1          14
```

## Warning Errors

| | |
|---|---|
| IMSL_INCONSISTENT_EQUATIONS_2 | The linear system of equations is inconsistent.. |
| IMSL_NOT_NONNEG_DEFINITE | The matrix *A* is not nonnegative definite. |

# Chapter 2 Eigensystem Analysis

## Functions

# Usage Notes

An ordinary linear eigensystem problem is represented by the equation $Ax = \lambda x$ where $A$ denotes an $n \times n$ matrix. The value $\lambda$ is an ***eigenvalue*** and $x \neq 0$ is the corresponding ***eigenvector***. The eigenvector is determined up to a scalar factor. In all functions, we have chosen this factor so that $x$ has Euclidean length one, and the component of $x$ of largest magnitude is positive. The eigenvalues and corresponding eigenvectors are sorted then returned in the order of largest to smallest complex magnitude. If $x$ is a complex vector, this component of largest magnitude is scaled to be real and positive. The entry where this component occurs can be arbitrary for eigenvectors having nonunique maximum magnitude values.

A generalized linear eigensystem problem is represented by $Ax = \lambda Bx$ where $A$ and $B$ are $n \times n$ matrices. The value $\lambda$ is a generalized eigenvalue, and $x$ is the corresponding generalized eigenvector. The generalized eigenvectors are normalized in the same manner as the ordinary eigensystem problem.

## Error Analysis and Accuracy

The remarks in this section are for ordinary eigenvalue problems. Except in special cases, functions will not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem $Ax = \lambda x$. Typically, the computed pair

$$\tilde{x}, \tilde{\lambda}$$

are an exact eigenvector-eigenvalue pair for a "nearby" matrix $A + E$. Information about $E$ is known only in terms of bounds of the form $\|E\|_2 \leq f(n)\ \|A\|_2 \varepsilon$. The value of $f(n)$ depends on the algorithm, but is typically a small fractional power of $n$. The parameter $\varepsilon$ is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan 1989, p. 342),

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

where $\sigma(A)$ is the set of all eigenvalues of $A$ (called the ***spectrum*** of $A$), $X$ is the matrix of eigenvectors, $\|\cdot\|_2$ is Euclidean length, and $\kappa(X)$ is the condition number of $X$ defined as $\kappa(X) = \|X\|_2\|X^{-1}\|_2$. If $A$ is a real symmetric or complex Hermitian matrix, then its eigenvector matrix $X$ is respectively orthogonal or unitary. For these matrices, $\kappa(X) = 1$.

The accuracy of the computed eigenvalues

$$\tilde{\lambda}_j$$

and eigenvectors

$$\tilde{x}_j$$

can be checked by computing their performance index $\tau$. The performance index is defined to be

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

where $\varepsilon$ is again the machine precision.

The performance index $\tau$ is related to the error analysis because

$$\|E\tilde{x}_j\|_2 = \|A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j\|_2$$

where $E$ is the "nearby" matrix discussed above.

While the exact value of $\tau$ is precision and data dependent, the performance of an eigensystem analysis function is defined as excellent if $\tau < 1$, good if $1 \le \tau \le 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of $\tau$ can serve as a warning that there is a significant error in the calculation.

If the condition number $\kappa(X)$ of the eigenvector matrix $X$ is large, there can be large errors in the eigenvalues even if $\tau$ is small. In particular, it is often difficult to recognize near multiple eigenvalues or unstable mathematical problems from numerical results. This facet of the eigenvalue problem is often difficult for users to understand. Suppose the accuracy of an individual eigenvalue is desired. This can be answered approximately by computing the ***condition number of an individual eigenvalue*** (see Golub and Van Loan 1989, pp. 344 - 345). For matrices $A$, such that the computed array of normalized eigenvectors $X$ is invertible, the condition number of $\lambda_j$ is

$$\kappa_j = \|e_j^T X^{-1}\|$$

the Euclidean length of the $j$-th row of $X^{-1}$. Users can choose to compute this matrix using function `imsl_c_lin_sol_gen` in Chapter 1, "Linear Systems." An approximate bound for the accuracy of a computed eigenvalue is then given by $\kappa_j \varepsilon \|A\|$. To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by $|\lambda_j|$.

# Reformulating Generalized Eigenvalue Problems

The eigenvalue problem $Ax = \lambda Bx$ is often difficult for users to analyze because it is frequently ill-conditioned. Occasionally, changes of variables can be performed on the given problem to ease this ill-conditioning. Suppose that $B$ is singular, but $A$ is nonsingular. Define the reciprocal $\mu = \lambda\text{-}1$. Then assuming $A$ is definite, the roles of $A$ and $B$ are interchanged so that the reformulated problem $Bx = \mu Ax$ is solved. Those generalized eigenvalues $\mu_j = 0$ correspond to eigenvalues $\lambda_j = \infty$. The remaining $\lambda_j = \mu_j^{-1}$. The generalized eigenvectors for $\lambda_j$ correspond to those for $\mu_j$.

Now suppose that $B$ is nonsingular. The user can solve the ordinary eigenvalue problem $Cx = \lambda x$ where $C = B^{-1}A$. The matrix $C$ is subject to perturbations due to ill-conditioning and rounding errors when computing $B^{-1}A$. Computing the condition numbers of the eigenvalues for $C$ may, however, be helpful for analyzing the accuracy of results for the generalized problem.

There is another method that users can consider to reduce the generalized problem to an alternate ordinary problem. This technique is based on first computing a matrix decomposition $B = PQ$ where both $P$ and $Q$ are matrices that are "simple" to invert. Then, the given generalized problem is equivalent to the ordinary eigenvalue problem $Fy = \lambda y$. The matrix $F = P^{-1}AQ^{-1}$ and the unnormalized eigenvectors of the generalized problem are given by $x = Q^{-1}y$. An example of this reformulation is used in the case where $A$ and $B$ are real and symmetric, with $B$ positive definite. The function `imsl_f_eig_symgen` uses $P = R^T$ and $Q = R$ where $R$ is an upper-triangular matrix obtained from a Cholesky decomposition, $B = R^TR$. The matrix $F = R^{-T}AR^{-1}$ is symmetric and real. Computation of the eigenvalue-eigenvector expansion for $F$ is based on function `imsl_f_eig_sym`.

# eig_gen

more...

Computes the eigenexpansion of a real matrix *A*.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_f_eig_gen (*int* n, *float* *a, ..., 0)

The type *d_complex* function is imsl_d_eig_gen.

## Required Arguments

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

*float* *a (Input)
An array of size n × n containing the matrix.

## Return Value

A pointer to the n complex eigenvalues of the matrix. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_f_eig_gen (*int* n, *float* *a,

IMSL_VECTORS, *f_complex* **evec,

IMSL_VECTORS_USER, *f_complex* evecu[],

IMSL_RETURN_USER, *f_complex* evalu[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_EVECU_COL_DIM, *int* evecu_col_dim,

0)

## Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size `n` × `n` containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size `n` × `n`containing the matrix of eigenvectors is returned in the space `evecu`.

`IMSL_RETURN_USER`, *f_complex* `evalu[]` (Output)
Store the `n`eigenvalues in the space `evalu`.

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

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = `n`

## Description

Function `imsl_f_eig_gen` computes the eigenvalues of a real matrix by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary orthogonal or Gauss similarity transformations. Then, eigenvalues are computed using a *QR* or combined *LR-QR* algorithm (Golub and Van Loan 1989, pp. 373 - 382, and Watkins and Elsner 1990). The combined *LR-QR* algorithm is based on an implementation by Jeff Haag and David Watkins. Eigenvectors are then calculated as required. When eigenvectors are computed, the *QR* algorithm is used to compute the eigenexpansion. When only eigenvalues are required, the combined *LR-QR* algorithm is used.

## Examples

### Example 1

```
#include <imsl.h>

int main()
{
   int         n = 3;
   float       a[] = {8.0, -1.0, -5.0,
                      -4.0, 4.0, -2.0,
```

```
                     18.0, -5.0, -7.0};
   f_complex  *eval;
                             /* Compute eigenvalues of A */
   eval = imsl_f_eig_gen (n, a, 0);
                             /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

### Output

```
                              Eigenvalues
                    1                      2                      3
(        2,         4)  (        2,        -4)  (        1,         0)
```

## Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

int main()
{
   int         n = 3;
   float       a[] = {8.0, -1.0, -5.0,
                      -4.0, 4.0, -2.0,
                      18.0, -5.0, -7.0};
   f_complex  *eval;
   f_complex  *evec;
                             /* Compute eigenvalues of A */
   eval = imsl_f_eig_gen (n, a,
                      IMSL_VECTORS, &evec,
                      0);
                             /* Print eigenvalues and eigenvectors */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
   imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

### Output

```
                              Eigenvalues
                    1                      2                      3
(        2,         4)  (        2,        -4)  (        1,         0)

                              Eigenvectors
                    1                     2                     3
1 (  0.3162,  0.3162)  (  0.3162,  -0.3162)  (  0.4082,  0.0000)
2 (  0.0000,  0.6325)  (  0.0000,  -0.6325)  (  0.8165,  0.0000)
3 (  0.6325,  0.0000)  (  0.6325,   0.0000)  (  0.4082,  0.0000)
```

## Warning Errors

| | |
|---|---|
| `IMSL_SLOW_CONVERGENCE_GEN` | The iteration for an eigenvalue did not converge after # iterations. |

# eig_gen (complex)

more...

Computes the eigenexpansion of a complex matrix *A*.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_eig_gen *(int* n, *f_complex* *a, …, *0)*

The type *d_complex* procedure is imsl_z_eig_gen.

## Required Arguments

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

*f_complex* *a (Input)
Array of size n ×n containing the matrix.

## Return Value

A pointer to the n complex eigenvalues of the matrix. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_eig_gen (*int* n, *f_complex* *a

IMSL_VECTORS, *f_complex* **evec,

IMSL_VECTORS_USER, *f_complex* evecu[],

IMSL_RETURN_USER, *f_complex* evalu[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_EVECU_COL_DIM, *int* evecu_col_dim,

0)

## Optional Arguments

IMSL_VECTORS, *f_complex* **evec (Output)
The address of a pointer to an array of size n ×n containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* *evecu is declared, and &evecu is used as an argument.

IMSL_VECTORS_USER, *f_complex* evecu[] (Output)
Compute eigenvectors of the matrix. An array of size n × n containing the matrix of eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *f_complex* evalu[] (Output)
Store the *n* eigenvalues in the space evalu.

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

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of evecu.
Default: evecu_col_dim = n

## Description

The function imsl_c_eig_gen computes the eigenvalues of a complex matrix by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary Gauss transformations. Then, the eigenvalues are computed using an explicitly shifted *LR* algorithm. Eigenvectors are calculated during the iterations for the eigenvalues (Martin and Wilkinson 1971).

## Examples

### Example 1

```
#include <imsl.h>

int main()
{
   int          n = 4;
   f_complex   a[] =  { {5,9}, {5,5}, {-6,-6}, {-7,-7},
                        {3,3}, {6,10}, {-5,-5}, {-6,-6},
                        {2,2}, {3,3}, {-1, 3}, {-5,-5},
                        {1,1}, {2,2}, {-3,-3}, { 0, 4} };
   f_complex   *eval;
```

```
                                /* Compute eigenvalues */
   eval = imsl_c_eig_gen (n, a, 0);
                                /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

### Output

```
                             Eigenvalues
                    1                      2                      3
(         4,         8)  (         3,         7)  (         2,         6)

                    4
(         1,         5)
```

## Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

int main()
{
   int         n = 4;
   f_complex   a[] =  { {5,9}, {5,5}, {-6,-6}, {-7,-7},
                       {3,3}, {6,10}, {-5,-5}, {-6,-6},
                       {2,2}, {3,3}, {-1, 3}, {-5,-5},
                       {1,1}, {2,2}, {-3,-3}, { 0, 4} };
   f_complex   *eval;
   f_complex   *evec;
                           /* Compute eigenvalues and eigenvectors */
   eval = imsl_c_eig_gen (n, a,
                           IMSL_VECTORS, &evec,
                           0);
                           /* Print eigenvalues and eigenvectors */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval, 0);
   imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

### Output

```
                             Eigenvalues
                    1                      2                      3
(         4,         8)  (         3,         7)  (         2,         6)

                    4
(         1,         5)

                             Eigenvectors
                     1                    2                     3
1 (   0.5773,   -0.0000) (   0.5774    0.0000) (   0.3780,   -0.0000)
```

```
2 (   0.5773,  -0.0000) (   0.5773, -0.0000) (   0.7559,   0.0000)
3 (   0.5774,   0.0000) ( -0.0000, -0.0000) (   0.3780,   0.0000)
4 ( -0.0000,  -0.0000) (   0.5774,   0.0000) (   0.3780,  -0.0000)

                      4
1 (    0.7559,    0.0000)
2 (    0.3780,    0.0000)
3 (    0.3780,    0.0000)
4 (    0.3780,    0.0000)
```

# Fatal Errors

| | |
|---|---|
| IMSL_SLOW_CONVERGENCE_GEN | The iteration for an eigenvalue did not converge after # iterations. |

# eig_sym

more...

Computes the eigenexpansion of a real symmetric matrix *A*.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_eig_sym (*int* n, *float* *a, …, 0)

The type *double* procedure is imsl_d_eig_sym.

## Required Arguments

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

*float* *a (Input)
Array of size n × n containing the symmetric matrix.

## Return Value

A pointer to the n eigenvalues of the symmetric matrix. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_eig_sym (*int* n, *float* *a,

IMSL_VECTORS, *float* **evec,

IMSL_VECTORS_USER, *float* evecu[],

IMSL_RETURN_USER, *float* evalu[],

IMSL_RANGE, *float* elow, *float* ehigh,

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_EVECU_COL_DIM, *int* evecu_col_dim,

IMSL_RETURN_NUMBER, *int* *n_eval,

0)

## Optional Arguments

IMSL_VECTORS, *float* **evec (Output)

The address of a pointer to an array of size n × n containing the eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *float* *evec is declared, and &evec is used as an argument.

IMSL_VECTORS_USER, *float* evecu[] (Output)

Compute eigenvectors of the matrix. An array of size n × n containing the orthogonal matrix of eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *float* evalu[] (Output)

Store the *n* eigenvalues in the space evalu.

IMSL_RANGE, *float* elow, *float* ehigh (Input)

Return eigenvalues and optionally eigenvectors that lie in the interval with lower limit elow and upper limit ehigh.
Default: (elow, ehigh) = $(-\infty, +\infty)$

IMSL_A_COL_DIM, *int* a_col_dim (Input)

The column dimension of a.
Default: a_col_dim = n

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)

The column dimension of evecu.
Default: evecu_col_dim = n

IMSL_RETURN_NUMBER, *int* *n_eval (Output)

The number of output eigenvalues and eigenvectors in the range elow, ehigh.

## Description

The function imsl_f_eig_sym computes the eigenvalues of a symmetric real matrix by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational *QR* or bisection algorithm. Eigenvectors are calculated as required (Parlett 1980, pp. 169 - 173).

# Examples

## Example 1

```
#include <imsl.h>

int main()
{
   int          n = 3;
   float        a[] =  {7.0, -8.0, -8.0,
                       -8.0, -16.0, -18.0,
                       -8.0, -18.0, 13.0};
   float        *eval;
                                 /* Compute eigenvalues */
   eval = imsl_f_eig_sym(n, a, 0);
                                 /* Print eigenvalues */
   imsl_f_write_matrix ("Eigenvalues", 1, 3, eval, 0);
}
```

## Output

```
           Eigenvalues
        1          2          3
   -27.90      22.68       9.22
```

## Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

int main()
{
   int          n = 3;
   float        a[] =  {7.0, -8.0, -8.0,
                       -8.0, -16.0, -18.0,
                       -8.0, -18.0, 13.0};
   float        *eval;
   float        *evec;
                                 /* Compute eigenvalues and eigenvectors */
   eval = imsl_f_eig_sym(n, a,
                           IMSL_VECTORS, &evec,
                           0);
                                 /* Print eigenvalues and eigenvectors */
   imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
   imsl_f_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

### Output

```
          Eigenvalues
       1           2           3
  -27.90       22.68        9.22

          Eigenvectors
          1           2           3
1      0.2945     -0.2722      0.9161
2      0.8521     -0.3591     -0.3806
3      0.4326      0.8927      0.1262
```

## Warning Errors

| | |
|---|---|
| IMSL_SLOW_CONVERGENCE_SYM | The iteration for the eigenvalue failed to converge in 100 iterations before deflating. |
| IMSL_SLOW_CONVERGENCE_2 | Inverse iteration did not converge. Eigenvector is not correct for the specified eigenvalue. |
| IMSL_LOST_ORTHOGONALITY_2 | The eigenvectors have lost orthogonality. |
| IMSL_NO_EIGENVALUES_RETURNED | The number of eigenvalues in the specified interval exceeds mxeval. The argument n_eval contains the number of eigenvalues in the interval. No eigenvalues will be returned. |

# eig_herm (complex)

more...

Computes the eigenexpansion of a complex Hermitian matrix *A*.

## Synopsis

*#include* <imsl.h>

*float* *imsl_c_eig_herm (*int* n, *f_complex* *a, …, 0)

The type *double* procedure is imsl_d_eig_herm.

## Required Arguments

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

*f_complex* *a (Input)
Array of size n × n containing the matrix.

## Return Value

A pointer to the n eigenvalues of the matrix. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_c_eig_herm (*int* n, *f_complex* *a,

IMSL_VECTORS, *f_complex* **evec,

IMSL_VECTORS_USER, *f_complex* evecu[],

IMSL_RETURN_USER, *float* evalu[],

IMSL_RANGE, *float* elow, *float* ehigh,

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_EVECU_COL_DIM, *int* evecu_col_dim,

IMSL_RETURN_NUMBER, *int* *n_eval,

0)

## Optional Arguments

IMSL_VECTORS, *f_complex* **evec (Output)
The address of a pointer to an array of size n × n containing eigenvectors of the matrix. On return, the necessary space is allocated by the function. Typically, *f_complex* *evec is declared, and &evec is used as an argument.

IMSL_VECTORS_USER, *f_complex* evecu[] (Output)
Compute eigenvectors of the matrix. An array of size n × n containing the unitary matrix of eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *float* evalu[] (Output)
Store the n eigenvalues in the space evalu.

IMSL_RANGE, *float* elow, *float* ehigh (Input)
Return eigenvalues and optionally eigenvectors that lie in the interval with lower limit elow and upper limit ehigh.
Default: (elow, ehigh) = $(-\infty, +\infty)$.

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

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of *X*.
Default: evecu_col_dim = n

IMSL_RETURN_NUMBER, *int* *n_eval (Output)
The number of output eigenvalues and eigenvectors in the range elow, ehigh.

## Description

The function imsl_c_eig_herm computes the eigenvalues of a complex Hermitian matrix by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational *QR* or bisection algorithm. Eigenvectors are calculated as required.

# Examples

## Example 1

```
#include <imsl.h>

int main()
{
   int         n = 3;
   f_complex   a[] =  { {1,0}, {1,-7}, {0,-1},
                        {1,7}, {5,0}, {10,-3},
                        {0,1}, {10,3}, {-2,0} };
   float       *eval;
                                    /* Compute eigenvalues */
   eval = imsl_c_eig_herm(n, a, 0);
                                    /* Print eigenvalues */
   imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

### Output

```
           Eigenvalues
        1           2           3
    15.38      -10.63       -0.75
```

## Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

int main()
{
   int         n = 3;
   f_complex   a[] =  { {1,0}, {1,-7}, {0,-1},
                        {1,7}, {5,0}, {10,-3},
                        {0,1}, {10,3}, {-2,0} };
   float       *eval;
   f_complex   *evec;
                                 /* Compute eigenvalues and eigenvectors */
   eval = imsl_c_eig_herm(n, a,
                           IMSL_VECTORS, &evec,
                           0);
                                 /* Print eigenvalues and eigenvectors */
   imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
   imsl_c_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

### Output

```
           Eigenvalues
        1           2           3
    15.38      -10.63       -0.75

                                        Eigenvectors
                         1                      2                      3
1 (  0.0631,  -0.4075) (  -0.0598,  -0.3117) (   0.8539,   0.0000)
2 (  0.7703,   0.0000) (  -0.5939,   0.1841) (  -0.0313,  -0.1380)
3 (  0.4668,   0.1366) (   0.7160,   0.0000) (   0.0808,  -0.4942)
```

## Warning Errors

| | |
|---|---|
| `IMSL_LOST_ORTHOGONALITY` | The iteration for at least one eigenvector failed to converge. Some of the eigenvectors may be inaccurate. |
| `IMSL_NEVAL_MXEVAL_MISMATCH` | The determined number of eigenvalues in the interval (#, #) is #. However, the input value for the maximum number of eigenvalues in this interval is #. |

## Fatal Errors

| | |
|---|---|
| `IMSL_SLOW_CONVERGENCE_GEN` | The iteration for the eigenvalues did not converge. |
| `IMSL_HERMITIAN_DIAG_REAL` | The matrix element *A* (#, #) = #. The diagonal of a Hermitian matrix must be real. |

# eig_symgen

more...

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. The matrices *A* and *B* are real and symmetric, and *B* is positive definite.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_eig_symgen *(int* n, *float* *a, *float* *b, …, *0)*

The type *double* procedure is imsl_d_eig_symgen.

## Required Arguments

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

*float* *a (Input)
Array of size n ×n containing the symmetric coefficient matrix *A*.

*float* *b (Input)
Array of size n ×n containing the positive definite symmetric coefficient matrix *B*.

## Return Value

A pointer to the n eigenvalues of the symmetric matrix. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_eig_symgen (*int* n, *float* *a, *float* *b,

IMSL_VECTORS, *float* **evec,

IMSL_VECTORS_USER, *float* evecu[],

IMSL_RETURN_USER, *float* evalu[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_B_COL_DIM, *int* b_col_dim,

IMSL_EVECU_COL_DIM, *int* evecu_col_dim,

0)

## Optional Arguments

IMSL_VECTORS, *float* **evec (Output)
The address of a pointer to an array of size n ×n containing eigenvectors of the problem. On return, the necessary space is allocated by the function. Typically, *float* *evec is declared, and &evec is used as an argument.

IMSL_VECTORS_USER, *float* evecu[] (Output)
Compute eigenvectors of the matrix. An array of size n ×n containing the matrix of generalized eigenvectors is returned in the space evecu.

IMSL_RETURN_USER, *float* evalu[] (Output)
Store the *n* eigenvalues in the space evalu.

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

IMSL_B_COL_DIM, *int* b_col_dim (Input)
The column dimension of *B*.
Default: b_col_dim = n

IMSL_EVECU_COL_DIM, *int* evecu_col_dim (Input)
The column dimension of evecu.
Default: evecu_col_dim = n

## Description

The function imsl_f_eig_symgen computes the eigenvalues of a symmetric, positive definite eigenvalue problem by a three-phase process (Martin and Wilkinson 1971). The matrix *B* is reduced to factored form using the Cholesky decomposition. These factors are used to form a congruence transformation that yields a symmetric real matrix whose eigenexpansion is obtained. The problem is then transformed back to the original coordinates. Eigenvectors are calculated and transformed as required.

# Examples

## Example 1

```
#include <imsl.h>

int main()
{
   int        n = 3;
   float      a[] =  {1.1, 1.2, 1.4,
                       1.2, 1.3, 1.5,
                       1.4, 1.5, 1.6};
   float      b[] =  {2.0, 1.0, 0.0,
                       1.0, 2.0, 1.0,
                       0.0, 1.0, 2.0};
   float      *eval;
                              /* Solve for eigenvalues */
   eval = imsl_f_eig_symgen (n, a, b, 0);
                              /* Print eigenvalues */
   imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
}
```

### Output

```
          Eigenvalues
       1          2          3
   1.386     -0.058     -0.003
```

## Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
#include <imsl.h>

int main()
{
   int        n = 3;
   float      a[] =  {1.1, 1.2, 1.4,
                       1.2, 1.3, 1.5,
                       1.4, 1.5, 1.6};
   float      b[] =  {2.0, 1.0, 0.0,
                       1.0, 2.0, 1.0,
                       0.0, 1.0, 2.0};
   float      *eval;
   float      *evec;
                             /* Solve for eigenvalues and eigenvectors */
   eval = imsl_f_eig_symgen (n, a, b,
                            IMSL_VECTORS, &evec,
                            0);
                             /* Print eigenvalues and eigenvectors */
```

```
   imsl_f_write_matrix ("Eigenvalues", 1, n, eval, 0);
   imsl_f_write_matrix ("Eigenvectors", n, n, evec, 0);
}
```

### Output

```
          Eigenvalues
        1           2           3
    1.386      -0.058      -0.003

          Eigenvectors
           1           2           3
1     0.6431     -0.1147     -0.6817
2    -0.0224     -0.6872      0.7266
3     0.7655      0.7174     -0.0858
```

## Warning Errors

| | |
|---|---|
| IMSL_SLOW_CONVERGENCE_SYM | The iteration for an eigenvalue failed to converge in 100 iterations before deflating. |

## Fatal Errors

| | |
|---|---|
| IMSL_SUBMATRIX_NOT_POS_DEFINITE | The leading # by # submatrix of the input matrix is not positive definite. |
| IMSL_MATRIX_B_NOT_POS_DEFINITE | Matrix B is not positive definite. |

# geneig

more...

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with $A$ and $B$ real.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_geneig (*int* n, *float* *a, *float* *b, *f_complex* *alpha, *float* *beta, …, 0)

The *double* analogue is imsl_d_geneig.

## Required Arguments

*int* n (Input)
Number of rows and columns in $A$ and $B$.

*float* *a (Input)
Array of size n × n containing the coefficient matrix $A$.

*float* *b (Input)
Array of size n × n containing the coefficient matrix $B$.

*f_complex* *alpha (Output)
Vector of size n containing scalars $\alpha_i$. If $\beta_i \neq 0$, $\lambda_i = \alpha_i/\beta_i$ for $i = 0, \ldots,$ n − 1 are the eigenvalues of the system.

*float* *beta (Output)
Vector of size n .

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_f_geneig (*int* n, *float* *a, *float* *b,

IMSL_VECTORS, *f_complex* **evec,

IMSL_VECTORS_USER, *f_complex* evecu[],

IMSL_A_COL_DIM, *int* a_col_dim,

`IMSL_B_COL_DIM`, *int* `b_col_dim`,

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim`,

0)

## Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size `n` × `n` containing eigenvectors of the problem. Each vector is normalized to have Euclidean length equal to the value one. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size `n` × `n` containing the matrix of generalized eigenvectors is returned in the space `evecu`. Each vector is normalized to have Euclidean length equal to the value one.

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

`IMSL_B_COL_DIM`, *int* `b_col_dim` (Input)
The column dimension of *B*.
Default: `b_col_dim` = `n` .

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = `n`

## Description

The function `imsl_f_geneig` uses the QZ algorithm to compute the eigenvalues and eigenvectors of the generalized eigensystem $Ax = \lambda Bx$, where *A* and *B* are real matrices of order *n*. The eigenvalues for this problem can be infinite, so $\alpha$ and $\beta$ are returned instead of $\lambda$. If $\beta$ is nonzero, $\lambda = \alpha/\beta$.

The first step of the QZ algorithm is to simultaneously reduce *A* to upper-Hessenberg form and *B* to upper-triangular form. Then, orthogonal transformations are used to reduce *A* to quasi-upper-triangular form while keeping *B* upper triangular. The generalized eigenvalues and eigenvectors for the reduced problem are then computed.

The function `imsl_f_geneig` is based on the QZ algorithm due to Moler and Stewart (1973), as implemented by the EISPACK routines QZHES, QZIT and QZVAL; see Garbow et al. (1977).

# Examples

## Example 1

In this example, the eigenvalue, $\lambda$, of system $Ax = \lambda Bx$ is computed, where

$$A = \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ -10.0 & 2.0 & 0.0 \\ 5.0 & 1.0 & 0.5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 3.0 & 3.0 & 0.0 \\ 4.0 & 0.5 & 1.0 \end{bmatrix}$$

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

int main()
{
   int       n = 3;
   f_complex alpha[3];
   float     beta[3];
   int       i;
   f_complex eval[3];

   float     a[] =
       {1.0, 0.5, 0.0,
       -10.0, 2.0, 0.0,
        5.0, 1.0, 0.5};

   float     b[] =
       {0.5, 0.0, 0.0,
        3.0, 3.0, 0.0,
        4.0, 0.5, 1.0};

   /* Compute eigenvalues */
   imsl_f_geneig (n, a, b, alpha, beta,
       0);

   for (i=0; i<n; i++)
       if (beta[i] != 0.0)
           eval[i] = imsl_c_div(alpha[i],
           imsl_cf_convert(beta[i], 0.0));
       else
           printf ("Infinite eigenvalue\n");

   /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval,
       0);
}
```

### Output

```
                            Eigenvalues
                   1                   2                    3
(    0.833,    1.993) (    0.833,   -1.993) (    0.500,    0.000)
```

## Example 2

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

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

```
int main()
{
   int       n = 3;
   f_complex alpha[3];
   float     beta[3];
   int       i;
   f_complex eval[3];
   f_complex *evec;

   float     a[] =
       {1.0, 0.5, 0.0,
       -10.0, 2.0, 0.0,
        5.0, 1.0, 0.5};

   float          b[] =
       {0.5, 0.0, 0.0,
        3.0, 3.0, 0.0,
        4.0, 0.5, 1.0};

   imsl_f_geneig (n, a, b, alpha, beta,
       IMSL_VECTORS, &evec,
       0);

   for (i=0; i<n; i++)
       if (beta[i] != 0.0)
           eval[i] = imsl_c_div(alpha[i],
           imsl_cf_convert(beta[i], 0.0));
       else
           printf ("Infinite eigenvalue\n");

   /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval,
       0);

   /* Print eigenvectors */
   imsl_c_write_matrix ("Eigenvectors", n, n, evec,
       0);
}
```

### Output

```
                             Eigenvalues
                     1                     2                       3
(    0.833,    1.993) (    0.833,   -1.993) (    0.500,    -0.000)

                              Eigenvectors
                      1                    2                     3
1 (   -0.197,    0.150) (   -0.197,   -0.150) (   -0.000,     0.000)
2 (   -0.069,   -0.568) (   -0.069,    0.568) (   -0.000,     0.000)
3 (    0.782,    0.000) (    0.782,    0.000) (    1.000,     0.000)
```

# geneig (complex)

more...

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with *A* and *B* complex.

## Synopsis

*#include* <imsl.h>

*void* imsl_c_geneig (*int* n, *f_complex* *a, *f_complex* *b, *f_complex* *alpha, *float* *beta, ..., 0)

The *double* analogue is imsl_z_geneig.

## Required Arguments

*int* n (Input)
Number of rows and columns in *A* and *B*.

*f_complex* *a (Input)
Array of size n × n containing the coefficient matrix *A*.

*f_complex* *b (Input)
Array of size n × n containing the coefficient matrix *B.*

*f_complex* *alpha (Output)
Vector of size *n* containing scalars $\alpha_i$. If $\beta_i \neq 0$, $\lambda_i = \alpha_i/\beta_i$ for *i* = 0, …, n − 1 are the eigenvalues of the system.

*f_complex* *beta (Output)
Vector of size n .

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_c_geneig (*int* n, *f_complex* *a, *f_complex* *b, *f_complex* *alpha, *f_complex* *beta,

IMSL_VECTORS, *f_complex* **evec,

IMSL_VECTORS_USER, *f_complex* evecu[],

IMSL_A_COL_DIM, *int* a_col_dim,

`IMSL_B_COL_DIM`, *int* `b_col_dim`,

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim`,

0)

## Optional Arguments

`IMSL_VECTORS`, *f_complex* `**evec` (Output)
The address of a pointer to an array of size `n` $\times$ `n` containing eigenvectors of the problem. Each vector is normalized to have Euclidean length equal to the value one. On return, the necessary space is allocated by the function. Typically, *f_complex* `*evec` is declared, and `&evec` is used as an argument.

`IMSL_VECTORS_USER`, *f_complex* `evecu[]` (Output)
Compute eigenvectors of the matrix. An array of size `n` $\times$ `n` containing the matrix of generalized eigenvectors is returned in the space `evecu`. Each vector is normalized to have Euclidean length equal to the value one.

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

`IMSL_B_COL_DIM`, *int* `b_col_dim` (Input)
The column dimension of *B*.
Default: `b_col_dim` = `n`.

`IMSL_EVECU_COL_DIM`, *int* `evecu_col_dim` (Input)
The column dimension of `evecu`.
Default: `evecu_col_dim` = `n`.

## Description

The function `imsl_c_geneig` uses the QZ algorithm to compute the eigenvalues and eigenvectors of the generalized eigensystem $Ax = \lambda Bx$, where *A* and *B* are complex matrices of order *n*. The eigenvalues for this problem can be infinite, so $\alpha$ and $\beta$ are returned instead of $\lambda$. If $\beta$ is nonzero, $\lambda = \alpha/\beta$.

The first step of the QZ algorithm is to simultaneously reduce *A* to upper-Hessenberg form and *B* to upper-triangular form. Then, orthogonal transformations are used to reduce *A* to quasi-upper-triangular form while keeping *B* upper triangular. The generalized eigenvalues and eigenvectors for the reduced problem are then computed.

The function `imsl_c_geneig` is based on the QZ algorithm due to Moler and Stewart (1973).

# Examples

## Example 1

In this example, the eigenvalue, λ, of system *Ax* = λ*Bx* is solved, where

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

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

int main()
{
   int       n = 3, i;
   f_complex alpha[3], beta[3], eval[3];
   f_complex zero = {0.0, 0.0};
   f_complex a[] = {{1.0, 0.0}, {0.5, 1.0}, {0.0, 5.0},
                    {-10.0, 0.0}, {2.0, 1.0}, {0.0, 0.0},
                    {5.0, 1.0}, {1.0, 0.0}, {0.5, 3.0}};
   f_complex b[] = {{0.5, 0.0}, {0.0, 0.0}, {0.0, 0.0},
                    {3.0, 3.0}, {3.0, 3.0}, {0.0, 1.0},
                    {4.0, 2.0}, {0.5, 1.0}, {1.0, 1.0}};

   /* Compute eigenvalues */
   imsl_c_geneig (n, a, b, alpha, beta,
       0);

   for (i=0; i<n; i++)
       if (!imsl_c_eq(beta[i], zero))
           eval[i] = imsl_c_div(alpha[i], beta[i]);
       else
           printf ("Infinite eigenvalue\n");

   /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval,
       0);
}
```

### Output

```
                              Eigenvalues
                    1                      2                      3
(    -8.18,    -25.38) (     2.18,      0.61) (     0.12,     -0.39)
```

## Example 2

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

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

int main()
{
   int       n = 3, i;
   f_complex alpha[3], beta[3], eval[3], *evec;
   f_complex zero = {0.0, 0.0};
   f_complex a[] = {{1.0, 0.0}, {0.5, 1.0}, {0.0, 5.0},
                    {-10.0, 0.0}, {2.0, 1.0}, {0.0, 0.0},
                    {5.0, 1.0}, {1.0, 0.0}, {0.5, 3.0}};
   f_complex b[] = {{0.5, 0.0}, {0.0, 0.0}, {0.0, 0.0},
                    {3.0, 3.0}, {3.0, 3.0}, {0.0, 1.0},
                    {4.0, 2.0}, {0.5, 1.0}, {1.0, 1.0}};

   /* Compute eigenvalues and eigenvectors */
   imsl_c_geneig (n, a, b, alpha, beta,
       IMSL_VECTORS, & evec,
       0);

   for (i=0; i<n; i++)
       if (!imsl_c_eq(beta[i], zero))
           eval[i] = imsl_c_div(alpha[i], beta[i]);
       else
           printf ("Infinite eigenvalue\n");

   /* Print eigenvalues */
   imsl_c_write_matrix ("Eigenvalues", 1, n, eval,
       0);

   /*Print eigenvectors */
   imsl_c_write_matrix ("Eigenvectors", n, n, evec,
       0);
}
```

### Output

```
                              Eigenvalues
                   1                      2                      3
(    -8.18,   -25.38)  (      2.18,     0.61) (      0.12,    -0.39)

                               Eigenvectors
                     1                      2                      3
1 (  -0.3267, -0.1245) (  -0.3007, -0.2444) (   0.0371,   0.1518)
2 (   0.1767,  0.0054) (   0.8959,  0.0000) (   0.9577,   0.0000)
3 (   0.9201,  0.0000) (  -0.2019,  0.0801) (  -0.2215,   0.0968)
```

# Chapter 3 Interpolation and Approximation

## Functions

# Usage Notes

The majority of the functions in this chapter produce cubic piecewise polynomial or general spline functions that either interpolate or approximate given data or support the evaluation and integration of these functions. Two major subdivisions of functions are provided. The cubic spline functions begin with the prefix "`cub_spline_`" and use the piecewise polynomial representation described below. The spline functions begin with the prefix "`spline_`" and use the B-spline representation described below. Most of the spline functions are based on routines in the book by de Boor (1978).

We provide a few general purpose routines for general least-squares fit to data and a routine that produces an interpolant to two-dimensional scattered data.

## Piecewise Polynomials

A univariate piecewise polynomial (function) ***p*** is specified by giving its breakpoint sequence $\xi \in \mathfrak{R}$ , the order ***k*** (degree ***k*** − 1) of its polynomial pieces, and the ***k*** × (***n*** − 1) matrix ***c*** of its local polynomial coefficients. In terms of this information, the piecewise polynomial (ppoly) function is given by

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

The breakpoint sequence $\xi$ is assumed to be strictly increasing, and we extend the ppoly function to the entire real axis by extrapolation from the first and last intervals. This representation is redundant when the ppoly function is known to be smooth. For example, if ***p*** is known to be continuous, then we can compute $c_{1,i+1}$ from the $c_{ji}$ as follows:

$$c_{1,\, i+1} = p(\xi_{i+1}) = \sum_{j=1}^{k} c_{ji} \frac{(\xi_{i+1} - \xi_i)^{j-1}}{(j-1)!}$$

For smooth ppoly, we prefer to use the nonredundant representation in terms of the "basis" or B-splines, at least when such a function is first to be determined.

## Splines and B-Splines

B-splines provide a particularly convenient and suitable basis for a given class of smooth ppoly functions. Such a class is specified by giving its breakpoint sequence, its order ***k***, and the required smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence $\mathbf{t} \in \mathfrak{R}^M$. The specification rule is as follows: If the class is to have all derivatives up to and including the ***j***-th derivative continuous across the interior breakpoint $\xi_i$, then the number $\xi_i$ should occur ***k*** − ***j*** − 1 times in the knot sequence. Assuming that $\xi_1$ and $\xi_n$ are the endpoints of the interval of interest, choose the first ***k*** knots equal to $\xi_1$ and the last ***k*** knots equal to $\xi_n$. This can be done because the B-splines are defined to be right continuous near $\xi_1$ and left continuous near $\xi_n$.

When the above construction is completed, a knot sequence **t** of length $M$ is generated, and there are $m: = M - k$ B-splines of order $k$, for example $B_0, \ldots, B_{m-1}$, spanning the ppoly functions on the interval with the indicated smoothness. That is, each ppoly function in this class has a unique representation

$$p = a_0 B_0 + a_1 B_1 + \ \ldots \ + a_{m-1} B_{m-1}$$

as a linear combination of B-splines. A B-spline is a particularly compact ppoly function. $B_i$ is a nonnegative function that is nonzero only on the interval $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. More precisely, the support of the $i$-th B-spline is $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. No ppoly function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals. When it is necessary to emphasize the dependence of the B-spline on its parameters, we will use the notation $B_{i,k,t}$ to denote the $i$-th B-spline of order $k$ for the knot sequence **t**.

# Cubic Splines

Cubic splines are smooth (i.e., $C^0$, $C^1$ or $C^2$), fourth-order ppoly functions. For historical and other reasons, cubic splines are the most heavily used ppoly functions. Therefore, we provide special functions for their construction and evaluation. These routines use the ppoly representation as described above for general ppoly functions (with $k = 4$).

We provide three cubic spline interpolation functions: `imsl_f_cub_spline_interp_e_cnd`, `imsl_f_cub_spline_interp_shape`, and `imsl_f_cub_spline_tcb`. The function `imsl_f_cub_spline_interp_e_cnd` allows the user to specify various endpoint conditions (such as the value of the first or second derivative at the right and left points). The natural cubic spline, for example, can be obtained using this function by setting the second derivative to zero at both endpoints. The function `imsl_f_cub_spline_interp_shape` is designed so that the shape of the curve matches the shape of the data. In particular, one option of this function preserves the convexity of the data while the default attempts to minimize oscillations. The function `imsl_f_cub_spline_tcb` allows the user to specify tension, continuity and bias parameters at each data point.

It is possible that the cubic spline interpolation functions will produce unsatisfactory results. For example, the interpolant may not have the shape required by the user, or the data may be noisy and require a least-squares fit. The interpolation function `imsl_f_spline_interp` is more flexible, as it allows you to choose the knots and order of the spline interpolant. We encourage the user to use this routine and exploit the flexibility provided.

# Tensor Product Splines

The simplest method of obtaining multivariate interpolation and approximation functions is to take univariate methods and form a multivariate method via tensor products. In the case of two-dimensional spline interpolation, the derivation proceeds as follows. Let $\mathbf{t}_x$ be a knot sequence for splines of order $k_x$, and $\mathbf{t}_y$ be a knot sequence for splines of order $k_y$. Let $N_x + k_x$ be the length of $\mathbf{t}_x$, and $N_y + k_x$ be the length of $\mathbf{t}_y$. Then, the tensor-product spline has the following form.

$$\sum_{m=0}^{N_y-1}\sum_{n=0}^{N_x-1} c_{nm} B_{n,\,k_x,\,t_x}(x) B_{m,\,k_y,\,t_y}(y)$$

Given two sets of points

$$\{x_i\}_{i=1}^{N_x}$$

and

$$\{y_i\}_{i=1}^{N_y}$$

for which the corresponding univariate interpolation problem can be solved, the tensor-product interpolation problem finds the coefficients $c_{\mathrm{nm}}$ so that

$$\sum_{m=0}^{N_y-1}\sum_{n=0}^{\mathrm{N_x}-1} c_{nm} B_{n,k_x,t_x}(x_\mathrm{i}) B_{m,k_y,t_y}(y_j) = f_{ij}$$

This problem can be solved efficiently by repeatedly solving univariate interpolation problems as described in de Boor (1978, p. 347). Three-dimensional interpolation can be handled in an analogous manner. This chapter provides functions that compute the two-dimensional, tensor-product spline coefficients given two-dimensional interpolation data (`imsl_f_spline_2d_interp`) and that compute the two-dimensional, tensor-product spline coefficients for a tensor-product, least-squares problem (`imsl_f_spline_2d_least_squares`). In addition, we provide evaluation, differentiation, and integration functions for the two-dimensional, tensor-product spline functions. The relevant functions are `imsl_f_spline_2d_value` and `imsl_f_spline_2d_integral`.

## Scattered Data Interpolation

The IMSL C Math Library provides one function, `imsl_f_scattered_2d_interp`, that returns values of an interpolant to scattered data in the plane. This function is based on work by Akima (1978), which uses $C^1$ piecewise quintics on a triangular mesh.

## Multi-dimensional Interpolation

imsl_f_`spline_nd_interp` computes a piecewise polynomial interpolant, of up to 15-th degree, to a function of up to 7 variables, defined on a multi-dimensional grid.

## Least Squares

The IMSL C Math Library includes functions for smoothing noisy data. The function `imsl_f_user_fcn_least_squares` computes regressions with user-supplied functions. The function `imsl_f_spline_least_squares` computes a least-squares fit using splines with fixed knots or variable knots. These functions produce cubic spline, least-squares fit by default. Optional arguments allow the user to choose the order and the knot sequence. IMSL C Math Library also includes a tensor-product spline regression function (`imsl_f_spline_2d_least_squares`), mentioned above. The function `imsl_f_radial_scattered_fit` computes an approximation to scattered data in $\mathfrak{R}^N$ using radial-basis functions.

In addition to the functions listed above, several functions in Chapter 10, "Statistics and Random Number Generation", provide for polynomial regression and general linear regression.

# Smoothing by Cubic Splines

One "smoothing spline" function is provided. The default action of `imsl_f_cub_spline_smooth` estimates a smoothing parameter by cross-validation and then returns the cubic spline that smooths the data. If the user wishes to supply a smoothing parameter, then this function returns the appropriate cubic spline.

# Structures for Splines and Piecewise Polynomials

This optional section includes more details concerning the structures for splines and piecewise polynomials.

## B-Splines

A spline may be viewed as a mapping with domain $\mathfrak{R}^d$ and target $\mathfrak{R}^r$, where $d$ and $r$ are positive integers. For this version of the IMSL C Math Library, only $r = 1$ is supported. Thus, if $s$ is a spline, then for some $d$ and $r$

$$s : \mathfrak{R}^d \rightarrow \mathfrak{R}^r$$

This implies that such a spline $s$ must have $d$ knot sequences and orders (one for each domain dimension). Thus, associated with $s$, we have knots and orders

$$\mathbf{t}^0, \ldots, \mathbf{t}^{d-1}$$
$$k_0, \ldots, k_{d-1}$$

The precise form of the spline follows:

$$s(x) = (s_0(x), \ldots, s_{r-1}(x)) \qquad x = (x_1, \ldots, x_d) \in \mathfrak{R}^d$$

where the following equation is true.

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

Note that $n_i$ is the number of knots in $\mathbf{t}^i$ minus the order $k_i$.

We store all the information for a spline in one structure called *Imsl_f_spline*. (If the type is double, then the structure name is *Imsl_d_spline*, and the *float* becomes *double*.) The specification for this structure follows:

```
typedef struct {
   int   domain_dim;
   int   target_dim;
```

```
    int   *order;
    int   *num_coef;
    int   *num_knots;
    float **knots;
    float **coef;
} Imsl_f_spline;
```

The following function demonstrates how the contents of an *Imsl_f_spline* can be viewed:

```
#include <imsl.h>
#include <stdio.h>
void sp_print(Imsl_f_spline *sp)
{
   int i, j;

   printf("Domain dimension:     %d\n", sp->domain_dim);
   printf("Target dimension:     %d\n\n", sp->target_dim);
   for (i = 0; i < sp->domain_dim; i++) {
       printf("Domain #%d\n", (i + 1));
       printf("   Order             %d\n", sp->order[i]);
       printf("   # of coefficients %d\n", sp->num_coef[i]);
       printf("   # of knots        %d\n",sp->num_knots[i]);
       printf("   Knots:\n");
       for (j = 0; j < (sp->num_knots[i]); j++)
           printf("                     %8.3f\n", sp->knots[i][j]);
   }
   /*
   * Handle printing of 1D and 2D B-spline coefficients separately.
   */
   if (sp->domain_dim==1) {
       imsl_f_write_matrix("Spline Coefficients",
           sp->num_coef[0], 1, sp->coef[0], 0);
   }
   if (sp->domain_dim==2) {
       /*
       * Coefficients of 2D B-splines are stored in column-major order.
       * To view the coefficients correctly we reverse the dimensions and
       * use optional argument IMSL_TRANSPOSE when calling
       * imsl_f_write_matrix() .
       */
       imsl_f_write_matrix("Spline Coefficients",
           sp->num_coef[1], sp->num_coef[0],
           sp->coef[0], IMSL_TRANSPOSE, 0);
   }
}
```

## Example

The data for this example comes from the function $e^x \sin(x + y)$ on the rectangle [0, 3] × [0, 5]. This function is sampled on a 50 × 25 grid and a tensor-product spline approximation is computed using `imsl_f_spline_2d_least_squares()`. The contents of the spline structure are then printed using the function `sp_print()` provided above.

```
#include <imsl.h>
void sp_print(Imsl_f_spline *sp);

int main()
{
#define NXDATA          50
#define NYDATA          25
                                /* Define function */
#define F(x,y)          (float)(exp(x)*sin(x+y))
   int                 i, j;
   float               fdata[NXDATA][NYDATA];
   float               xdata[NXDATA], ydata[NYDATA];
   Imsl_f_spline       *sp;
                                /* Set up grid */
   for (i = 0; i < NXDATA; i++)
       xdata[i] = 3.*(float) i / ((float)(NXDATA-1));
   for (i = 0; i < NYDATA; i++)
       ydata[i] = 5.*(float) i / ((float)(NYDATA-1));
                                /* Compute function values on grid */
   for (i = 0; i < NXDATA; i++)
       for (j = 0; j < NYDATA; j++)
           fdata[i][j] = F(xdata[i], ydata[j]);
                               /* Compute tensor-product fit */
   sp = imsl_f_spline_2d_least_squares(NXDATA, &xdata[0], NYDATA,
                                       &ydata[0], &fdata[0][0], 5, 7, 0);
                               /* Print contents of spline structure. */
   sp_print(sp);
}
```

## Output

```
Domain dimension:      2
Target dimension:      1

Domain #1
    Order              4
    # of coefficients  5
    # of knots         9
    Knots:
                     0.000
                     0.000
                     0.000
                     0.000
```

```
                         1.500
                         3.000
                         3.000
                         3.000
                         3.000
Domain #2
    Order                 4
    # of coefficients 7
    # of knots            11
    Knots:
                         0.000
                         0.000
                         0.000
                         0.000
                         1.250
                         2.500
                         3.750
                         5.000
                         5.000
                         5.000
                         5.000

                            Spline Coefficients
             1              2              3              4              5
1        -0.02           0.43           1.34           0.87          -0.78
2         0.52           0.99           1.62           0.35          -1.40
3         3.35           4.99           6.16          -0.46          -6.45
4        10.43           7.44          -5.11         -16.78          -5.56
5         2.98          -5.24         -23.55         -18.74          11.62

             6              7
1        -1.18          -1.05
2        -1.30          -0.95
3        -4.60          -2.79
4         7.10          10.21
5        21.49          20.07
```

## Piecewise Polynomials

For `ppoly` functions, we view a `ppoly` as a mapping with domain $\Re^d$ and target $\Re^r$ where ***d*** and ***r*** are positive integers. Thus, if ***p*** is a `ppoly`, then for some ***d*** and ***r*** the following is true.

$$p : \Re^{d} \rightarrow \Re^{r}$$

For this version of the C MathLibrary, only ***r*** = ***d*** = 1 is supported. See the section Piecewise Polynomials near the beginning of this chapter for a detailed description of ppoly construction.

We store all the information for a ppoly in one structure called ***Imsl_f_ppoly***. (If the type is ***double***, then the structure name is ***Imsl_d_ppoly***, and the ***float*** becomes ***double***.) The following is the specification for this structure.

```
typedef struct {
```

```
    int   domain_dim;
    int   target_dim;
    int   *order;
    int   *num_coef;
    int   *num_breakpoints;
    float **breakpoints;
    float **coef;
} Imsl_f_ppoly;
```

The following function demonstrates how the contents of an *Imsl_f_ppoly* can be viewed.

```
#include <imsl.h>
#include <stdio.h>
void pp_print(Imsl_f_ppoly *pp)
{
   int i, j, k;
   printf("Domain dimension:     %d\n", pp->domain_dim);
   printf("Target dimension:     %d\n\n", pp->target_dim);
   for (i = 0; i < pp->domain_dim; i++) {
       printf("Domain #%d\n", (i + 1));
       printf("   Order              %d\n", pp->order[i]);
       printf("   # of coefficients %d\n", pp->num_coef[i]);
       printf("   # of breakpoints  %d\n",pp->num_breakpoints[i]);
       printf("   Breakpoints:\n");
       for (j = 0; j < (pp->num_breakpoints[i]); j++)
           printf("                   %8.3f\n", pp->breakpoints[i][j]);
   }

   printf("\nCoefficients:\n");
   for (j = 0; j < ((pp->num_breakpoints[0]) - 1); j++)
   {
       printf("     ppoly piece %4d", j + 1);
       for (k = 0; k < (pp->order[0]); k++)
           printf(" %9.3f ", pp->coef[0][j * (pp->order[0]) + k]);
       printf("\n");
   }

}
```

## Example

In this example, a cubic spline interpolant to a function *f* is computed. The contents of the ppoly structure are then printed using the sample code `pp_print()` provided above.

```
#include <imsl.h>
void pp_print(Imsl_f_ppoly *pp);
```

```
int main()
{
#define NDATA 11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))
   int                 i;
   float               fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_ppoly        *ppoly;
                                /* Compute xdata and fdata */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                                /* Compute cubic spline interpolant */
   ppoly = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);
                                /* Print contents of ppoly structure. */
   pp_print(ppoly);
}
```

## Output

```
Domain dimension:      1
Target dimension:      1

Domain #1
    Order   4
    # of coefficients 40
    # of breakpoints  11
    Breakpoints:
                     0.000
                     0.100
                     0.200
                     0.300
                     0.400
                     0.500
                     0.600
                     0.700
                     0.800
                     0.900
                     1.000
Coefficients:
      ppoly piece    1      0.000       23.414    -310.479    1250.916
      ppoly piece    2      0.997       -1.379    -185.387    1250.917
      ppoly piece    3      0.141      -13.663     -60.295    3294.986
      ppoly piece    4     -0.978       -3.218     269.203   -1956.621
      ppoly piece    5     -0.279       13.919      73.541   -3253.285
      ppoly piece    6      0.938        5.007    -251.787    1394.176
      ppoly piece    7      0.412      -13.201    -112.370    3540.767
      ppoly piece    8     -0.880       -6.734     241.707   -1152.020
      ppoly piece    9     -0.537       11.677     126.505   -2758.902
      ppoly piece   10      0.804       10.532    -149.385   -2758.903
```

# cub_spline_interp_e_cnd

Computes a cubic spline interpolant, specifying various endpoint conditions. The default interpolant satisfies the "not-a-knot" condition.

## Synopsis

*#include* `<imsl.h>`

*Imsl_f_ppoly* `*imsl_f_cub_spline_interp_e_cnd` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`, …, 0)

The type *Imsl_d_ppoly* function is `imsl_d_cub_spline_interp_e_cnd`.

## Required Arguments

*int* `ndata` (Input)
Number of data points.

*float* `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the interpolation problem.

*float* `fdata[]` (Input)
Array with `ndata` components containing the ordinates for the interpolation problem.

## Return Value

A pointer to the structure that represents the cubic spline interpolant. If an interpolant cannot be computed, then `NULL` is returned. To release this space, use `imsl_free`.

## Synopsis with Optional Arguments

#*include* `<imsl.h>`

*Imsl_f_ppoly* `*imsl_f_cub_spline_interp_e_cnd` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`,

`IMSL_LEFT`, *int* `ileft`, *float* `left`,

`IMSL_RIGHT`, *int* `iright`, *float* `right`,

`IMSL_PERIODIC`,

0)

## Optional Arguments

`IMSL_LEFT`, *int* `ileft`, *float* `left` (Input)

Set the value for the first or second derivative of the interpolant at the left endpoint. If `ileft` = *i*, then the interpolant ***s*** satisfies

$$s^{(i)}(x_L) = \texttt{left}$$

where $x_L$ is the leftmost abscissa. The only valid values for `ileft` are 1 or 2.

`IMSL_RIGHT`, *int* `iright`, *float* `right` (Input)

Set the value for the first or second derivative of the interpolant at the right endpoint. If `iright` = *i*, then the interpolant ***s*** satisfies

$$s^{(i)}(x_R) = \texttt{right}$$

where $x_R$ is the rightmost abscissa. The only valid values for `iright` are 1 or 2.

`IMSL_PERIODIC`

Compute the $C^2$ periodic interpolant to the data. That is, we require

$$s^{(i)}(x_L) = s^{(i)}(x_R) \qquad i = 0, 1, 2$$

where $s$, $x_L$, and $x_R$ are defined above.

## Description

The function `imsl_f_cub_spline_interp_e_cnd` computes a $C^2$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots,$ `ndata` $- 1 = n$. The breakpoints of the spline are the abscissas. We emphasize here that for all the univariate interpolation functions, the abscissas need not be sorted. Endpoint conditions are to be selected by the user. The user may specify "not-a-knot" or first derivative or second derivative at each endpoint, or $C^2$ periodicity may be requested (see de Boor 1978, Chapter 4). If no defaults are selected, then the "not-a-knot" spline interpolant is computed. If the `IMSL_PERIODIC` keyword is selected, then all other keywords are ignored; and a $C^2$ periodic interpolant is computed. In this case, if the `fdata` values at the left and right endpoints are not the same, then a warning message is issued; and we set the right value equal to the left. If `IMSL_LEFT` or `IMSL_RIGHT` are selected (in the absence of `IMSL_PERIODIC`), then the user has the ability to select the values of the first or second derivative at either endpoint. The default case (when the keyword is not used) is the "not-a-knot" condition on that endpoint. Thus, when no optional arguments are chosen, this function produces the "not-a-knot" interpolant.

If the data (including the endpoint conditions) arise from the values of a smooth (say $C^4$) function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{\left[\xi_0, \xi_n\right]} \le C \left\| f^{(4)} \right\|_{\left[\xi_0, \xi_n\right]} |\xi|^4$$

where

$$|\xi| := \max_{i=0,\ldots,n-1} |\xi_{i+1} - \xi_i|$$

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

The return value for this function is a pointer to the structure *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*ppoly`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y*

```
y = imsl_f_cub_spline_value (x, ppoly, 0)
```

The difference between the default ("not-a-knot") spline and the interpolating cubic spline, which has first derivative set to 1 at the left end and the second derivative set to −90 at the right end, is illustrated in the following figure.

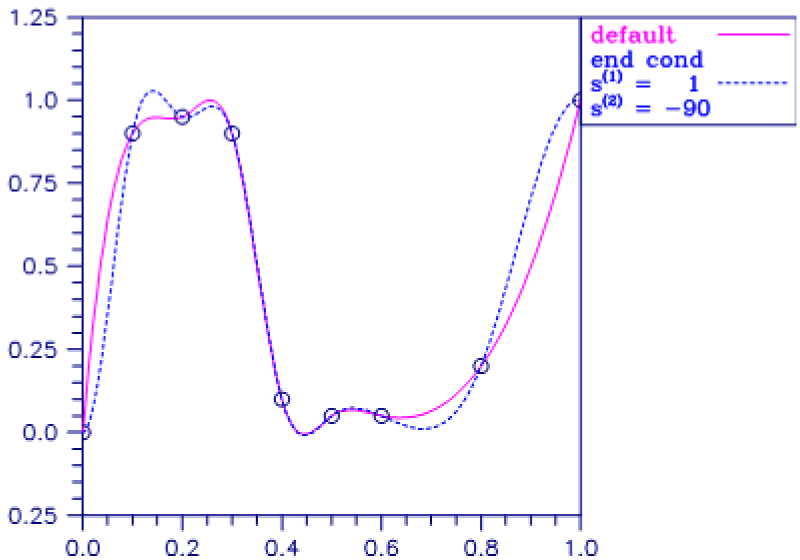



Figure 3.1 — Two Interpolating Splines

# Examples

## Example 1

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. Since we are using the default settings, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

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

#define NDATA 11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

int main()
{
   int                 i;
   float               fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_ppoly        *ppoly;
                                /* Compute xdata and fdata */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                                /* Compute cubic spline interpolant */
   ppoly = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);

                                /* Print results */
   printf("    x         F(x)      Interpolant   Error\n");
   for (i = 0; i < 2*NDATA-1; i++){
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_cub_spline_value(x,ppoly,0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                           fabs(F(x)-y));
   }
}
```

### Output

```
   x         F(x)     Interpolant   Error
0.000       0.000       0.000       0.0000
0.050       0.682       0.809       0.1270
0.100       0.997       0.997       0.0000
0.150       0.778       0.723       0.0552
0.200       0.141       0.141       0.0000
0.250      -0.572      -0.549       0.0228
0.300      -0.978      -0.978       0.0000
0.350      -0.859      -0.843       0.0162
0.400      -0.279      -0.279       0.0000
0.450       0.450       0.441       0.0093
0.500       0.938       0.938       0.0000
0.550       0.923       0.903       0.0199
0.600       0.412       0.412       0.0000
0.650      -0.320      -0.315       0.0049
0.700      -0.880      -0.880       0.0000
0.750      -0.968      -0.938       0.0295
```

```
0.800     -0.537      -0.537      0.0000
0.850      0.183       0.148      0.0347
0.900      0.804       0.804      0.0000
0.950      0.994       1.086      0.0926
1.000      0.650       0.650      0.0000
```

## Example 2

In this example, a cubic spline interpolant to a function *f* is computed. The value of the derivative at the left endpoint and the value of the second derivative at the right endpoint are specified. The values of this spline are then compared with the exact function values.

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

#define NDATA 11
                              /* Define function */
#define F(x)    (float)(sin(15.0*x))

int main()
{
   int             i, ileft, iright;
   float           left, right, x, y, fdata[NDATA], xdata[NDATA];
   Imsl_f_ppoly    *pp;
                             /* Compute xdata and fdata */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)(i)/(NDATA-1);
       fdata[i] = F(xdata[i]);
   }
                              /* Specify end conditions */
   ileft  = 1;
   left  = 0.0;
   iright = 2;
   right =-225.0*sin(15.0);
                              /* Compute cubic spline interpolant */
   pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                     IMSL_LEFT, ileft, left,
                                     IMSL_RIGHT, iright, right,
                                     0);
                              /* Print results for first half */
                              /* of interval */
   printf("    x        F(x)      Interpolant   Error\n\n");
   for (i=0; i<NDATA; i++){
       x = (float)(i)/(float)(2*NDATA-2);
       y = imsl_f_cub_spline_value(x,pp,0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                              fabs(F(x)-y));
   }
}
```

### Output

```
   x         F(x)     Interpolant    Error
0.000       0.000        0.000       0.0000
0.050       0.682        0.438       0.2441
0.100       0.997        0.997       0.0000
0.150       0.778        0.822       0.0442
0.200       0.141        0.141       0.0000
0.250      -0.572       -0.575       0.0038
0.300      -0.978       -0.978       0.0000
0.350      -0.859       -0.836       0.0233
0.400      -0.279       -0.279       0.0000
0.450       0.450        0.439       0.0111
0.500       0.938        0.938       0.0000
```

## Example 3

This example computes the *natural* cubic spline interpolant to a function *f* by forcing the second derivative of the interpolant to be zero at both endpoints. As in the previous example, the exact function values are computed with the values of the spline.

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

#define NDATA 11
                               /* Define function */
#define F(x)    (float)(sin(15.0*x))

int main()
{
   int              i, ileft, iright;
   float            left, right, x, y, fdata[NDATA],
                     xdata[NDATA];
   Imsl_f_ppoly     *pp;
                               /* Compute xdata and fdata */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)(i)/(NDATA-1);
       fdata[i] = F(xdata[i]);
   }
                               /* Specify end conditions */
   ileft  = 2;
   left   = 0.0;
   iright = 2;
   right  = 0.0;
                               /* Compute cubic spline interpolant */
   pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                       IMSL_LEFT, ileft, left,
                                       IMSL_RIGHT, iright, right,
                                       0);
```

```
                                     /* Print results for first half */
                                     /* of interval */
    printf("    x         F(x)       Interpolant    Error\n\n");
    for (i = 0; i < NDATA; i++){
        x = (float)(i)/(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x,pp,0);
        printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                                    fabs(F(x)-y));
    }
}
```

### Output

```
   x         F(x)      Interpolant    Error
0.000        0.000        0.000       0.0000
0.050        0.682        0.667       0.0150
0.100        0.997        0.997       0.0000
0.150        0.778        0.761       0.0172
0.200        0.141        0.141       0.0000
0.250       -0.572       -0.559       0.0126
0.300       -0.978       -0.978       0.0000
0.350       -0.859       -0.840       0.0189
0.400       -0.279       -0.279       0.0000
0.450        0.450        0.440       0.0098
0.500        0.938        0.938       0.0000
```

## Example 4

This example computes the cubic spline interpolant to a functions, and imposes the periodic end conditions $s(a) = s(b)$, $s'(a) = s'(b)$, and $s''(a) = s''(b)$, where $a$ is the leftmost abscissa and $b$ is the rightmost abscissa.

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

#define NDATA 11
                              /* Define function*/
#define F(x)    (float)(sin(x))

int main()
{
   int                 i;
   float               x, y, twopi, fdata[NDATA], xdata[NDATA];
   Imsl_f_ppoly        *pp;
                              /* Compute xdata and fdata */
   twopi = 2.0*imsl_f_constant("pi", 0);
   for (i = 0; i < NDATA; i++) {
       xdata[i] = twopi*(float)(i)/(NDATA-1);
       fdata[i] = F(xdata[i]);
   }
   fdata[NDATA-1] = fdata[0];
                              /* Compute periodic cubic spline */
                              /* interpolant */
   pp = imsl_f_cub_spline_interp_e_cnd(NDATA, xdata, fdata,
                                       IMSL_PERIODIC,
                                       0);
                              /* Print results for first half */
                              /* of interval */
   printf("    x         F(x)      Interpolant   Error\n\n");
   for (i = 0; i < NDATA; i++){
       x = (twopi/20.)*i;
       y = imsl_f_cub_spline_value(x, pp, 0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n",x,F(x), y,
                                            fabs(F(x)-y));
   }
}
```

## Output

```
   x         F(x)     Interpolant   Error
0.000       0.000       0.000       0.0000
0.314       0.309       0.309       0.0001
0.628       0.588       0.588       0.0000
0.942       0.809       0.809       0.0004
1.257       0.951       0.951       0.0000
1.571       1.000       1.000       0.0004
1.885       0.951       0.951       0.0000
2.199       0.809       0.809       0.0004
2.513       0.588       0.588       0.0000
2.827       0.309       0.309       0.0001
3.142      -0.000      -0.000       0.0000
```

## Warning Errors

| | |
|---|---|
| `IMSL_NOT_PERIODIC` | The data is not periodic. The rightmost fdata value is set to the leftmost fdata value. |

## Fatal Errors

| | |
|---|---|
| `IMSL_DUPLICATE_XDATA_VALUES` | The xdata values must be distinct. |

# cub_spline_interp_shape

Computes a shape-preserving cubic spline.

## Synopsis

*#include* <imsl.h>

*Imsl_f_ppoly* *imsl_f_cub_spline_interp_shape (*int* ndata, *float* xdata[], *float* fdata[], …, 0)

The type *Imsl_d_ppoly* function is imsl_d_cub_spline_interp_shape.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the interpolation problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates for the interpolation problem.

## Return Value

A pointer to the structure that represents the cubic spline interpolant. If an interpolant cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_ppoly* *imsl_f_cub_spline_interp_shape (*int* ndata, *float* xdata[], *float* fdata[],

IMSL_CONCAVE,

IMSL_CONCAVE_ITMAX, *int* itmax,

0)

## Optional Arguments

IMSL_CONCAVE
This option produces a cubic interpolant that will preserve the concavity of the data.

`IMSL_CONCAVE_ITMAX`, *int* `itmax` (Input)

This option allows the user to set the maximum number of iterations of Newton's Method.

Default: `itmax` = 25.

## Description

The function `imsl_f_cub_spline_interp_shape` computes a $C^1$ cubic spline interpolant to a set of data points$(x_i, f_i)$ for $i = 0, \ldots,$ `ndata` $- 1 = n$. The breakpoints of the spline are the abscissas. This computation is based on a method by Akima (1970) to combat wiggles in the interpolant. Endpoint conditions are automatically determined by the program; see Akima (1970) or de Boor (1978).

If the optional argument `IMSL_CONCAVE` is chosen, then this function computes a cubic spline interpolant to the data. For ease of explanation, we will assume that $x_i < x_{i+1}$, although it is not necessary for the user to sort these data values. If the data are strictly convex, then the computed spline is convex, $C^2$, and minimizes the expression

$$\int_{x_1}^{x_n} (g'')^2$$

over all convex $C^1$ functions that interpolate the data. In the general case, when the data have both convex and concave regions, the convexity of the spline is consistent with the data, and the above integral is minimized under the appropriate constraints. For more information on this interpolation scheme, refer to Michelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this function is that it is not possible, a priori, to predict the number of breakpoints of the resulting interpolant. In most cases, there will be breakpoints at places other than data locations. This function should be used when it is important to preserve the convex and concave regions implied by the data.

Both methods are nonlinear, and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.) This explains the theoretical error estimate below.

If the data points arise from the values of a smooth (say $C^4$) function $f$, i.*e.* $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for either of the above spline interpolants. Then, the maximum absolute error satisfies

$$\left\| f - s \right\|_{[\xi_0, \xi_m]} \le C \left\| f^{(2)} \right\|_{[\xi_0, \xi_m]} |\xi|^2$$

where

$$|\xi| := \max_{i=0,\ldots,m-1} |\xi_{i+1} - \xi_i|$$

and $\xi_m$ is the last breakpoint.

The return value for this function is a pointer of the type *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*ppoly`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y*.

```
y = imsl_f_cub_spline_value (x, ppoly, 0)
```

The difference between the convexity-preserving spline and Akima's spline is illustrated in the following figure. Note that the convexity-preserving interpolant exhibits linear segments where the convexity constraints are binding.

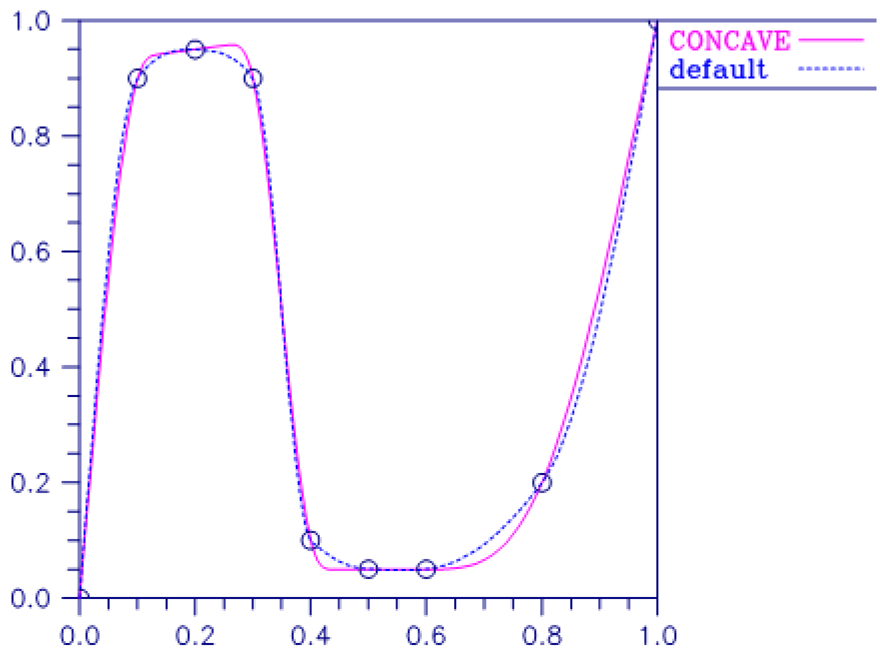



Figure 3.2 — Two Shape-Preserving Splines

## Examples

### Example 1

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values.

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

#define NDATA 11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))
```

```
int main()
{
    int                 i;
    float               fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly        *pp;
                                   /* Compute xdata and fdata */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = (float)(i)/(NDATA-1);
        fdata[i] = F(xdata[i]);
    }
                                   /* Compute cubic spline interpolant */
    pp = imsl_f_cub_spline_interp_shape(NDATA, xdata, fdata, 0);
                                   /* Print results */
    printf("    x         F(x)      Interpolant    Error\n\n");
    for (i = 0; i < 2*NDATA-1; i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                                  fabs(F(x)-y));
    }
}
```

### Output

```
   x         F(x)     Interpolant    Error
0.000       0.000        0.000      0.0000
0.050       0.682        0.818      0.1360
0.100       0.997        0.997      0.0000
0.150       0.778        0.615      0.1635
0.200       0.141        0.141      0.0000
0.250      -0.572       -0.478      0.0934
0.300      -0.978       -0.978      0.0000
0.350      -0.859       -0.812      0.0464
0.400      -0.279       -0.279      0.0000
0.450       0.450        0.386      0.0645
0.500       0.938        0.938      0.0000
0.550       0.923        0.854      0.0683
0.600       0.412        0.412      0.0000
0.650      -0.320       -0.276      0.0433
0.700      -0.880       -0.880      0.0000
0.750      -0.968       -0.889      0.0789
0.800      -0.537       -0.537      0.0000
0.850       0.183        0.149      0.0338
0.900       0.804        0.804      0.0000
0.950       0.994        0.932      0.0613
1.000       0.650        0.650      0.0000
```

## Example 2

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values.

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

#define NDATA  11
                                /* Define function */
#define F(x)   (float)(sin(15.0*x))

int main()
{
   int               i;
   float             fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_ppoly      *pp;
                                /* Compute xdata and fdata */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)(i)/(NDATA-1);
       fdata[i] = F(xdata[i]);
   }
                                /* Compute cubic spline interpolant */
   pp = imsl_f_cub_spline_interp_shape(NDATA, xdata, fdata,
                                       IMSL_CONCAVE,
                                       0);
                               /* Print results */
   printf("    x         F(x)      Interpolant   Error\n\n");
   for (i = 0; i < 2*NDATA-1; i++){
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_cub_spline_value(x, pp, 0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                                  fabs(F(x)-y));
   }
}
```

### Output

```
   x        F(x)     Interpolant   Error
0.000       0.000       0.000      0.0000
0.050       0.682       0.667      0.0150
0.100       0.997       0.997      0.0000
0.150       0.778       0.761      0.0172
0.200       0.141       0.141      0.0000
0.250      -0.572      -0.559      0.0126
0.300      -0.978      -0.978      0.0000
0.350      -0.859      -0.840      0.0189
0.400      -0.279      -0.279      0.0000
0.450       0.450       0.440      0.0098
0.500       0.938       0.938      0.0000
0.550       0.923       0.902      0.0208
0.600       0.412       0.412      0.0000
0.650      -0.320      -0.311      0.0086
0.700      -0.880      -0.880      0.0000
```

```
0.750      -0.968      -0.952      0.0156
0.800      -0.537      -0.537      0.0000
0.850       0.183       0.200      0.0174
0.900       0.804       0.804      0.0000
0.950       0.994       0.892      0.1020
1.000       0.650       0.650      0.0000
```

## Warning Errors

| | |
|---|---|
| `IMSL_MAX_ITERATIONS_REACHED` | The maximum number of iterations has been reached. The best approximation is returned. |

## Fatal Errors

| | |
|---|---|
| `IMSL_DUPLICATE_XDATA_VALUES` | The xdata values must be distinct. |

# cub_spline_tcb

Computes a tension-continuity-bias (TCB) cubic spline interpolant. This is also called a Kochanek-Bartels spline and is a generalization of the Catmull–Rom spline.

## Synopsis

*#include* <imsl.h>

*Imsl_f_ppoly* *imsl_f_cub_spline_tcb (*int* ndata, *float* xdata[], *float* fdata[], …, 0)

The type *Imsl_d_ppoly* function is imsl_d_cub_spline_tcb.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the interpolation problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates for the interpolation problem.

## Return Value

A pointer to the structure that represents the interpolant. If an interpolant cannot be computed, NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_ppoly* *imsl_f_cub_spline_tcb (*int* ndata, *float* xdata[], *float* fdata[],

IMSL_TENSION, *float* tension[],

IMSL_CONTINUITY, *float* continuity[],

IMSL_BIAS, *float* bias[],

IMSL_LEFT, *double* left,

IMSL_RIGHT, *double* right,

0)

## Optional Arguments

`IMSL_TENSION`, *float* `tension[]` (Input)

Sets the tension values at the data points. The array `tension` is of length `ndata` and contains tension values in the interval [-1,1]. For each point, if the tension value is near +1 the curve is tightened at that point. If it is near -1, the curve is slack.
Default: All values of `tension` are zero.

`IMSL_CONTINUITY`, *float* `continuity[]` (Input)

Sets the continuity values at the data points. The array `continuity` is of length `ndata` and contains continuity values in the interval [-1,1]. For each point, if the continuity value is zero the curve is $C^1$ at that point. Otherwise the curve has a corner at that point, but is still continuous ($C^0$).
Default: All values of `continuity` are zero.

`IMSL_BIAS`, *float* `bias[]` (Input)

Sets the bias values at the data points. The array `bias` is of length `ndata` and contains bias values in the interval [-1,1]. For each point, if the bias value is zero the left and right side tangents are equally weighted. If the value is near +1 the left-side tangent dominates. If the value is near -1 the right-side tangent dominates.
Default: All values of `bias` are zero.

`IMSL_LEFT`, *double* `left` (Input)

Sets the value of the tangent at the leftmost endpoint.
Default: `left` = 0.

`IMSL_RIGHT`, *double* `right` (Input)

Sets the value of the tangent at the rightmost endpoint.
Default: `right` = 0.

## Description

The function `imsl_f_cub_spline_tcb` computes the Kochanek-Bartels spline, a piecewise cubic Hermite spline interpolant to a set of data points $\{x_i, f_i\}$ for I = 0, ..., `ndata`-1. The breakpoints of the spline are the abscissas. As with all of the univariate interpolation functions, the abscissas need not be sorted.

The $\{x_i\}$ values are the knots, so the *i*-th interval is $[x_i, x_{i+1}]$. (To simplify the explanation, it is assumed that the data points are given in increasing order.) The cubic Hermite in the *i*-th segment has a starting value of $f_i$ and an ending value of $f_{i+1}$. Its incoming tangent is

$$DS_i = \frac{1}{2}\left(1-t_i\right)\left(1-c_i\right)\left(1+b_i\right)\frac{y_i-y_{i-1}}{x_{i+1}-x_i} + \frac{1}{2}\left(1-t_i\right)\left(1+c_i\right)\left(1-b_i\right)\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$$

where $t_i$ is the *i*-th tension value, $c_i$ is the *i*-th continuity value, and $b_i$ is the *i*-th bias value. Its outgoing tangent is

$$DD_i = \frac{1}{2}\left(1-t_i\right)\left(1+c_i\right)\left(1+b_i\right)\frac{y_i - y_{i-1}}{x_{i+1}-x_i} + \frac{1}{2}\left(1-t_i\right)\left(1-c_i\right)\left(1-b_i\right)\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$$

The optional arguments `left` and `right` are used at the endpoints:

$$\frac{y_0 - y_{-1}}{x_1 - x_o} = left \text{ and } \frac{y_n - y_{n-1}}{x_n - x_{n-1}} = \text{right}$$

Both ***left*** and ***right*** default to zero.

The spline has a continuous first derivative (is $C^1$) if at each data point the left and right tangents are equal. This is true if the continuity parameters, $c_i$, are all zero. For any values of the parameters the spline is continuous ($C^0$).

If $t_i = c_i = b_i = 0$ for all $i$, then the curve is the ***Catmull-Rom*** spline.

The following chart shows the same data points interpolated with different parameter values. All of the tension, continuity and bias parameters are zero except for the labeled parameter, which has the indicated value at all data points.

Tension controls how sharply the spline bends at the data points. If `tension` is near +1, the curve tightens. If `tension` is near -1, the curve slackens.

The continuity parameter controls the continuity of the first derivative. If continuity is zero, the spline's first derivative is continuous, so the spline is $C^1$.

The bias parameter controls the weighting of the left and right tangents. If zero, the tangents are equally weighted. If the bias parameter is near +1, the left tangent dominates. If the bias parameter is near -1, the right tangent dominates.

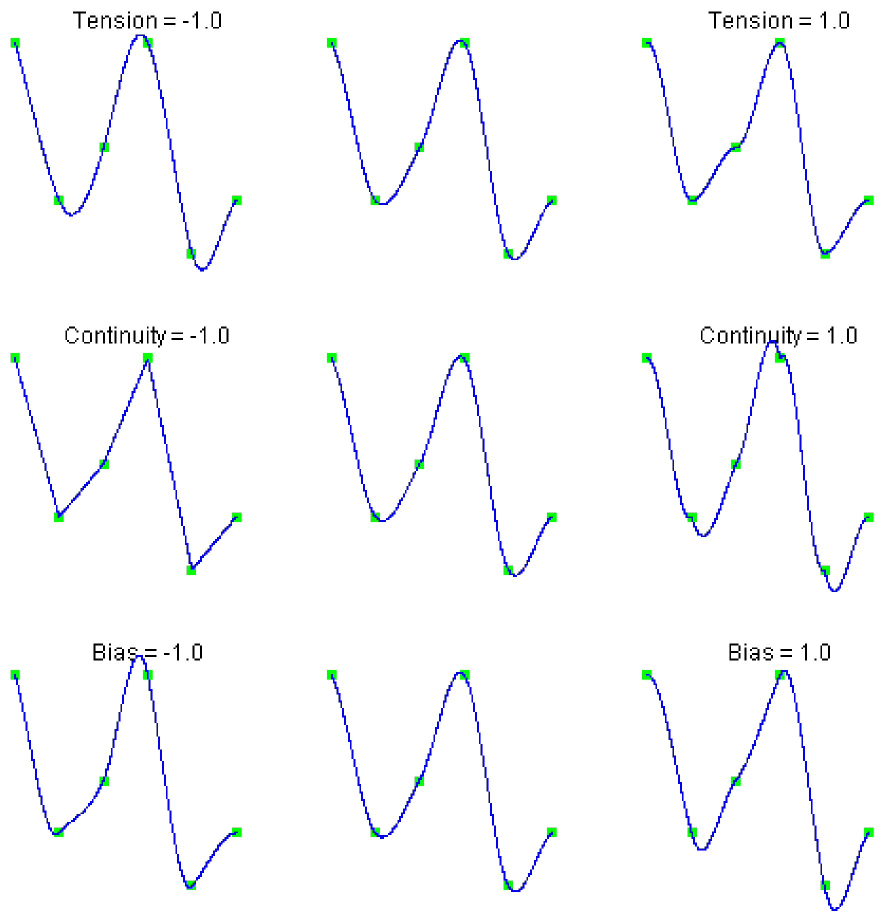



Figure 3.3 — Data Points Interpolated with Different Parameter Values

# Examples

## Example 1

This example interpolates to a set of points. At *x* = 3 the continuity and tension parameters are -1. At all other points, they are zero. Interpolated values are then printed.

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

int main()
{
   int    ndata = 6;
   float    xdata[] = {0, 1, 2, 3, 4, 5};
   float    fdata[] = {5, 2, 3, 5, 1, 2};
   float    continuity[] = {0, 0, 0, -1, 0, 0};
   float    tension[] = {0, 0, 0, -1, 0, 0};
   Imsl_f_ppoly  *ppoly;
```

```
    int      m = 2;
    int     i;

    ppoly = imsl_f_cub_spline_tcb(ndata, xdata, fdata,
        IMSL_CONTINUITY, continuity,
        IMSL_TENSION, tension,
        0);
    for (i = 0; i < m*(ndata-1)+1; i++) {
        float x = i / (float)m;
        float y = imsl_f_cub_spline_value(x, ppoly, 0);
        printf(" %6.3f %10.4f\n", x, y);
    }
}
```

### Output

```
0.000    5.0000
0.500    3.4375
1.000    2.0000
1.500    2.1875
2.000    3.0000
2.500    3.6875
3.000    5.0000
3.500    2.1875
4.000    1.0000
4.500    1.2500
5.000    2.0000
```

## Example 2

It is possible to use an interpolating spline for approximation by using an optimization function to compute its parameters. In this example a series of $n$ interest rates, $r_i$, for different maturities, $x_i$, is given, $\{x_i, r_i\}$ for $i = 0, \ldots, n-1$. Since the dates are given on a widely varying time scale, the base 10 logarithm of the dates is used for interpolation.

A TCB spline is constructed using a subset of the given data points for knot locations, $\{p_i, q_i\}$, for $i = 0, \ldots, m-1$. The $p$ values are a subset of the $\log_{10} x_i$ values. The $q$ values are to be determined by the optimizer. The spline has non-zero values of the continuity parameter, $c_i$, for $i = 0, \ldots, m-1$.

The optimization problem finds the spline, $s\left(r_i; p, q, c, left, right\right)$, which interpolates the points $\{p_i, q_i\}$ and has continuity parameters, $c$, and specified $left$ and $right$ parameters.

The optimization problem is

$$\min_{q,c,left,right} \sum_{i=0}^{i=n-1} \left| s\left(r_i; q, c, left, right\right) - r_i \right|^2$$

subject to the bounds, for all $i$.

$$0.1 \le q_i \le 10$$

$$-0.95 \le c_i \le 0.95$$

The function constrained_nlp is used as the optimizer. The unknowns ***q***, ***c***, ***left*** and ***right*** are packed into the array `x`, respectively.

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

void fcn(int n, double x[], int iact, double *result, int *ierr);
double objective(double *yknots, double *continuity,
   double left, double right);

#define N_DATA          15

static int days[] = {3, 31, 62, 94, 185, 367, 731, 1096, 1461, 1826, 2194,
   2558, 2922, 3287, 3653};
static double log_days[N_DATA];
static double rate[] = {5.01772, 4.98284, 4.97234, 4.96157, 4.99058,
   5.09389, 5.79733, 6.30595, 6.73464, 6.94816, 7.08807, 7.27527,
   7.30852, 7.3979, 7.49015};

/* Knots are set on a subset of the data points */
#define N_KNOTS         4
static double           xknots[N_KNOTS];
static int              subset[] = {0, 5, 10, 14};

#define Y_KNOTS         0
#define CONTINUITY      (Y_KNOTS + N_KNOTS)
#define LEFT            (CONTINUITY + N_KNOTS)
#define RIGHT           (LEFT + 1)
#define N_VARIABLES     (RIGHT + 1)

int main()
{
   Imsl_d_ppoly *ppoly;
   double         *x, xlb[N_VARIABLES], xub[N_VARIABLES];
   double         xguess[N_VARIABLES];
   int            n_constraints, ibtype, i;

   n_constraints = 0;
   ibtype = 0;

   for (i = 0; i < N_KNOTS; i++) {
       xlb[Y_KNOTS+i] = 0.1;       /* lower bound on rate */
       xub[Y_KNOTS+i] = 10.0;      /* upper bound on rate */
       xlb[CONTINUITY+i] = -0.95; /* lower bound on continuity */
       xub[CONTINUITY+i] = 0.95;  /* upper bound on continuity */
   }
```

```
    /* Set bounds wide enough on LEFT and RIGHT so they are not binding */
    xlb[LEFT] = -100.0;   xub[LEFT] = 100.0;
    xlb[RIGHT] = -100.0;  xub[RIGHT] = 100.0;

    for (i = 0; i < N_DATA; i++) {
        log_days[i] = log10(days[i]);
    }

    for (i = 0; i < N_KNOTS; i++) {
        xknots[i] = log_days[subset[i]];
        xguess[Y_KNOTS+i] = rate[subset[i]];
        xguess[CONTINUITY+i] = 0.0;
    }

    xguess[LEFT] = xguess[RIGHT] = 0.0;
    /* Find the optimial curve */
    x = imsl_d_constrained_nlp(fcn, n_constraints, 0, N_VARIABLES,
        ibtype, xlb, xub,
        IMSL_XGUESS, xguess,
        IMSL_DIFFTYPE, 3,
        0);

    /* Report results */
    ppoly = imsl_d_cub_spline_tcb(N_KNOTS, xknots, x,
        IMSL_CONTINUITY, x+CONTINUITY,
        0);

    printf("Days  Rate Curve Error \n");
    for (i = 0; i < N_DATA; i++) {
        double y = imsl_d_cub_spline_value(log_days[i], ppoly, 0);
        printf("%4d %6.3f %6.3f %6.3f\n",
            days[i], rate[i], y, y-rate[i]);
    }

    printf("\n");
    for (i = 0; i < N_KNOTS; i++) {
        printf(" continuity[%2d] = %6.3f\n", days[i], x[CONTINUITY+i]);
    }

    printf("\n left = %6.3f\n right = %6.3f\n", x[LEFT], x[RIGHT]);
}

/* Function passed to imsl_d_constrained_nlp */
void fcn(int n, double x[], int iact, double *result, int *ierr)
{
    if (iact == 0) {
        *result = objective(x+Y_KNOTS, x+CONTINUITY, x[LEFT], x[RIGHT]);
    }
}
```

```
/* * Compute the objective function, the sum of squares error */
double objective(double *yknots, double *continuity,
   double left, double right)
{
   Imsl_d_ppoly *ppoly;
   double          error;
   int            i;

   ppoly = imsl_d_cub_spline_tcb(N_KNOTS, xknots, yknots,
       IMSL_CONTINUITY, continuity,
       IMSL_LEFT, left,
       IMSL_RIGHT, right,
       0);

   error = 0.0;
   for (i = 0; i < N_DATA; i++) {
       double y = imsl_d_cub_spline_value(log_days[i], ppoly, 0);
       double diff = (y - rate[i]);
       error += diff * diff;
   }

   imsl_free(ppoly);
   return error/N_DATA;
}
```

### Output

```
Days Rate Curve Error
3 5.018 5.019 0.002
31 4.983 4.926 -0.057
62 4.972 4.911 -0.061
94 4.962 4.919 -0.043
185 4.991 4.970 -0.021
367 5.094 5.084 -0.010
731 5.797 5.842 0.045
1096 6.306 6.340 0.034
1461 6.735 6.683 -0.052
1826 6.948 6.931 -0.017
2194 7.088 7.118 0.030
2558 7.275 7.240 -0.035
2922 7.309 7.332 0.023
3287 7.398 7.409 0.011
3653 7.490 7.479 -0.011

continuity[ 3] = 0.009
continuity[31] = -0.630
continuity[62] = -0.184
continuity[94] = -0.950

left = 0.534
right = 0.266
```

# cub_spline_value

Computes the value of a cubic spline or the value of one of its derivatives.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_cub_spline_value` (*float* `x`, *Imsl_f_ppoly* `*ppoly`, …, 0)

The type *double* function is `imsl_d_cub_spline_value`.

## Required Arguments

*float* `x` (Input)
Evaluation point for the cubic spline.

*Imsl_f_ppoly* `*ppoly` (Input)
Pointer to the piecewise polynomial structure that represents the cubic spline.

## Return Value

The value of a cubic spline or one of its derivatives at the point *x*. If no value can be computed, then NaN is returned.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `imsl_f_cub_spline_value` (*float* `x`, *Imsl_f_ppoly* `*ppoly`,

`IMSL_DERIV`, *int* `deriv`,

`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value`,

`IMSL_GRID_USER`, *int* `n`, *float* `*xvec`, *float* `value_user[]`,

0)

## Optional Arguments

`IMSL_DERIV`, *int* `deriv` (Input)
Let $d$ = `deriv` and let $s$ be the cubic spline that is represented by the structure `*ppoly`, then this option produces the $d$-th derivative of $s$ at $x$, $s^{(d)}(x)$.

`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value` (Input/Output)

The array `xvec` of length `n` contains the points at which the cubic spline is to be evaluated. The *d*-th derivative of the spline at the points in `xvec` is returned in `value`. The values in array `xvec` must appear sorted and non-decreasing. Arranging for this requirement may benefit by use of the function `imsl_f_sort`, Chapter 12.

`IMSL_GRID_USER`, *int* `n`, *float* `*xvec`, *float* `value_user[]` (Input/Output)

The array `xvec` of length `n` contains the points at which the cubic spline is to be evaluated. The *d*-th derivative of the spline at the points in `xvec` is returned in the user-supplied space `value_user`. The values in array `xvec` must appear sorted and non-decreasing..

## Description

The function `imsl_f_cub_spline_value` computes the value of a cubic spline or one of its derivatives. The first and last pieces of the cubic spline are extrapolated. As a result, the cubic spline structures returned by the cubic spline routines are defined and can be evaluated on the entire real line. This routine is based on the routine PPVALU by de Boor (1978, p. 89).

## Examples

### Example 1

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

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

#define NDATA   11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

int main()
{
   int                  i;
   float                fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_ppoly         *pp;
                                /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                               /* Compute cubic spline interpolant */
   pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);
```

```
                                    /* Print results */
    printf("     x          F(x)       Interpolant    Error\n");
    for (i = NDATA/2; i < 3*NDATA/2; i++) {
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp, 0);
        printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                                   fabs(F(x)-y));
    }
}
```

## Output

```
   x        F(x)      Interpolant    Error
0.250     -0.572        -0.549       0.0228
0.300     -0.978        -0.978       0.0000
0.350     -0.859        -0.843       0.0162
0.400     -0.279        -0.279       0.0000
0.450      0.450         0.441       0.0093
0.500      0.938         0.938       0.0000
0.550      0.923         0.903       0.0199
0.600      0.412         0.412       0.0000
0.650     -0.320        -0.315       0.0049
0.700     -0.880        -0.880       0.0000
0.750     -0.968        -0.938       0.0295
```

## Example 2

Recall that in the first example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. This example compares the values of the first derivatives.

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

#define NDATA   11
                                    /* Define functions */
#define F(x)     (float)(sin(15.0*x))
#define FP(x)    (float)(15.*cos(15.0*x))

int main()
{
    int                  i;
    float                fdata[NDATA], xdata[NDATA], x, y;
    Imsl_f_ppoly         *pp;
                                    /* Set up a grid */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = (float)i /((float)(NDATA-1));
        fdata[i] = F(xdata[i]);
    }
                                    /* Compute cubic spline interpolant */
```

```
    pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata,fdata, 0);
                                    /* Print results */
    printf("    x         FP(x)     Interpolant  Deriv Error\n");
    for (i = NDATA/2; i < 3*NDATA/2; i++){
        x = (float) i /(float)(2*NDATA-2);
        y = imsl_f_cub_spline_value(x, pp,
                                     IMSL_DERIV, 1,
                                     0);
        printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, FP(x), y,
                                                  fabs(FP(x)-y));
        }
}
```

## Output

```
  x        FP(x)  Interpolant  Deriv Error
0.250    -12.308      -12.559      0.2510
0.300     -3.162       -3.218      0.0560
0.350      7.681        7.796      0.1151
0.400     14.403       13.919      0.4833
0.450     13.395       13.530      0.1346
0.500      5.200        5.007      0.1926
0.550     -5.786       -5.840      0.0535
0.600    -13.667      -13.201      0.4660
0.650    -14.214      -14.393      0.1798
0.700     -7.133       -6.734      0.3990
0.750      3.775        3.911      0.1359
```

# cub_spline_integral

Computes the integral of a cubic spline.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_cub_spline_integral (*float* a, *float* b, *Imsl_f_ppoly* *ppoly)

The type ***double*** function is imsl_d_cub_spline_integral.

## Required Arguments

*float* a (Input)

*float* b (Input)
Endpoints for integration.

*Imsl_f_ppoly* *ppoly (Input)
Pointer to the piecewise polynomial structure that represents the cubic spline.

## Return Value

The integral from ***a*** to ***b*** of the cubic spline. If no value can be computed, then NaN is returned.

## Description

The function imsl_f_cub_spline_integral computes the integral of a cubic spline from ***a*** to ***b***.

$$\int_a^b s(x)\,dx$$

## Example

In this example, a cubic spline interpolant to a function *f* is computed. The values of the integral of this spline are then compared with the exact integral values. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

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

#define NDATA  21
                              /* Define function */
```

```
#define F(x)    (float)(sin(15.0*x))
                                /* Integral from 0 to x */
#define FI(x)   (float)((1.-cos(15.0*x))/15.)

int main()
{
   int                  i;
   float                fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_ppoly         *pp;
                                /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                                /* Compute cubic spline interpolant */
   pp = imsl_f_cub_spline_interp_e_cnd (NDATA, xdata, fdata, 0);
                                /* Print results */
   printf("    x        FI(x)     Interpolant  Integral Error\n");
   for (i = NDATA/2; i < 3*NDATA/2; i++){
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_cub_spline_integral(0.0, x, pp);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, FI(x), y,
                                                   fabs(FI(x)-y));
       }
}
```

## Output

```
x           FI(x)   Interpolant  Integral Error
0.250       0.121       0.121      0.0001
0.275       0.104       0.104      0.0001
0.300       0.081       0.081      0.0001
0.325       0.056       0.056      0.0001
0.350       0.033       0.033      0.0001
0.375       0.014       0.014      0.0002
0.400       0.003       0.003      0.0002
0.425       0.000       0.000      0.0002
0.450       0.007       0.007      0.0002
0.475       0.022       0.022      0.0001
0.500       0.044       0.044      0.0001
0.525       0.068       0.068      0.0001
0.550       0.092       0.092      0.0001
0.575       0.113       0.113      0.0001
0.600       0.127       0.128      0.0001
0.625       0.133       0.133      0.0001
0.650       0.130       0.130      0.0001
0.675       0.118       0.118      0.0001
0.700       0.098       0.098      0.0001
0.725       0.075       0.075      0.0001
0.750       0.050       0.050      0.0001
```

# spline_interp

Compute a spline interpolant.

## Synopsis

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_interp (*int* ndata, *float* xdata[], *float* fdata[], …, 0)

The type *Imsl_d_spline* function is imsl_d_spline_interp.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the interpolation problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates of the interpolation problem.

## Return Value

A pointer to the structure that represents the spline interpolant. If an interpolant cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_interp (*int* ndata, *float* xdata[], *float* fdata[],

IMSL_ORDER, *int* order,

IMSL_KNOTS, *float* knots[],

0)

## Optional Arguments

`IMSL_ORDER`, *int* `order` (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: `order` = 4, i.e., cubic splines

`IMSL_KNOTS`, *float* `knots[]` (Input)
This option requires the user to provide the knots.
Default: knots are selected by the function `imsl_f_spline_knots` using its defaults.

## Description

Given the data points $x$ = `xdata`, $f$ = `fdata`, and the number $n$ = `ndata` of elements in `xdata` and `fdata`, the default action of `imsl_f_spline_interp` computes a cubic ($k$ = 4) spline interpolant $s$ to the data using the default knot sequence generated by `imsl_f_spline_knots`.

The optional argument `IMSL_ORDER` allows the user to choose the order of the spline interpolant. The optional argument `IMSL_KNOTS` allows user specification of knots.

The function `imsl_f_spline_interp` is based on the routine SPLINT by de Boor (1978, p. 204).

First, `imsl_f_spline_interp` sorts the `xdata` vector and stores the result in $x$. The elements of the `fdata` vector are permuted appropriately and stored in $f$, yielding the equivalent data $(x_i, f_i)$ for $i = 0$ to $n - 1$.

The following preliminary checks are performed on the data. We verify that

$$\begin{array}{ll} x_i < x_{i+1} & i = 0, \ldots, n-2 \\ t_i < t_{i+k} & i = 0, \ldots, n-1 \\ t_i < t_{i+1} & i = 0, \ldots, n+k-2 \end{array}$$

The first test checks to see that the abscissas are distinct. The second and third inequalities verify that a valid knot sequence has been specified.

In order for the interpolation matrix to be nonsingular, we also check $\mathbf{t}_{k-1} \le x_i \le \mathbf{t}_n$ for $i = 0$ to $n - 1$. This first inequality in the last check is necessary since the method used to generate the entries of the interpolation matrix requires that the $k$ possibly nonzero B-splines at $x_i$,

$$B_{j-k+1}, \ldots, B_j \qquad \text{where } j \text{ satisfies } \mathbf{t}_j \le x_i < \mathbf{t}_{j+1}$$

be well-defined (that is, $j - k + 1 \ge 0$).

General conditions are not known for the exact behavior of the error in spline interpolation; however, if $\mathbf{t}$ and $x$ are selected properly and the data points arise from the values of a smooth (say $C^k$) function $f$, i.e. $f_j = f(x_j)$, then the error will behave in a predictable fashion. The maximum absolute error satisfies

$$\left\| f - s \right\|_{[t_{k-1}, t_n]} \leq C \left\| f^{(k)} \right\|_{[t_{k-1}, t_n]} |t|^k$$

where

$$|t| := \max_{i=k-1,\ldots,n-1} |t_{i+1} - t_i|$$

For more information on this problem, see de Boor (1978, Chapter 13) and his reference. This function can be used in place of the IMSL function `imsl_f_cub_spline_interp`.

The return value for this function is a pointer of type *Imsl_f_spline*. The calling program must receive this in a pointer *Imsl_f_spline* `*sp`. This structure contains all the information to determine the spline (stored as a linear combination of B-splines) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y.*

```
y = imsl_f_spline_value (x, sp, 0)
```

Three spline interpolants of order 2, 3, and 5 are plotted. These splines use the default knots.

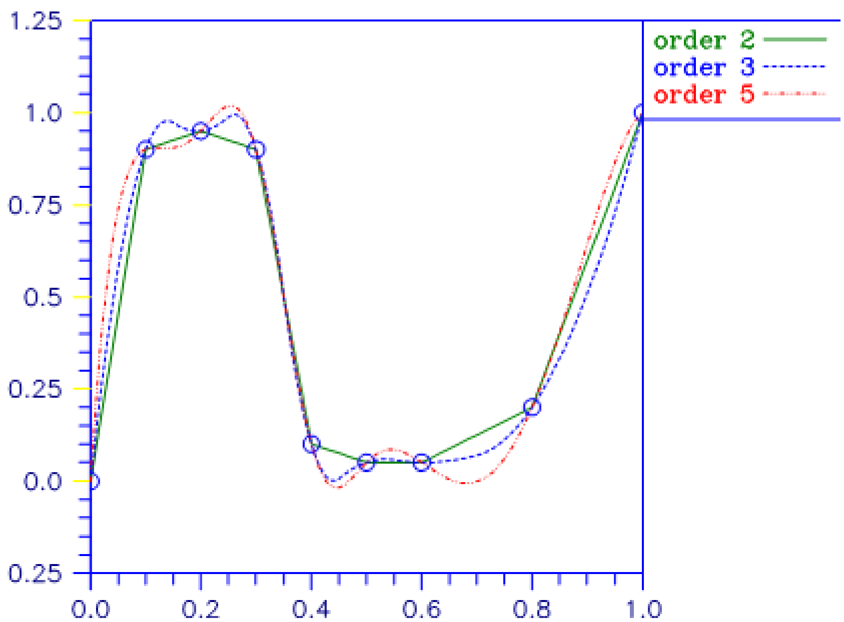



Figure 3.4 — Three Spline Interpolants

# Examples

## Example 1

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

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

#define NDATA  11
                              /* Define function */
#define F(x)   (float)(sin(15.0*x))

int main()
{
   int                 i;
   float               xdata[NDATA], fdata[NDATA], x, y;
   Imsl_f_spline       *sp;
                              /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                              /* Compute cubic spline interpolant */
   sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                              /* Print results */
   printf("    x        F(x)      Interpolant   Error\n");
   for (i = 0; i < 2*NDATA-1; i++){
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_spline_value(x, sp, 0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                          fabs(F(x)-y));
   }
}
```

### Output

```
  x        F(x)      Interpolant   Error
0.000      0.000        0.000      0.0000
0.050      0.682        0.809      0.1270
0.100      0.997        0.997      0.0000
0.150      0.778        0.723      0.0552
0.200      0.141        0.141      0.0000
0.250     -0.572       -0.549      0.0228
0.300     -0.978       -0.978      0.0000
0.350     -0.859       -0.843      0.0162
0.400     -0.279       -0.279      0.0000
0.450      0.450        0.441      0.0093
0.500      0.938        0.938      0.0000
0.550      0.923        0.903      0.0199
0.600      0.412        0.412      0.0000
0.650     -0.320       -0.315      0.0049
0.700     -0.880       -0.880      0.0000
0.750     -0.968       -0.938      0.0295
0.800     -0.537       -0.537      0.0000
```

```
0.850        0.183        0.148        0.0347
0.900        0.804        0.804        0.0000
0.950        0.994        1.086        0.0926
1.000        0.650        0.650        0.0000
```

## Example 2

Recall that in the first example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. This example chooses to use a quadratic (*k* = 3) and a quintic *k* = 6 spline interpolant to the data instead of the default values.

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

#define NDATA 11
                                /* Define function */
#define F(x)     (float)(sin(15.0*x))

int main()
{
   int                  i, order;
   float                fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_spline        *sp;
                                /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
   for (order =3; order<7; order += 3) {
                                /* Compute cubic spline interpolant */
       sp = imsl_f_spline_interp (NDATA, xdata, fdata,
                                  IMSL_ORDER, order,
                                  0);
                                /* Print results */
       printf("\nThe order of the spline is %d\n", order);
       printf("    x        F(x)      Interpolant   Error\n");
       for (i = NDATA/2; i < 3*NDATA/2; i++){
           x = (float) i /(float)(2*NDATA-2);
           y = imsl_f_spline_value(x,sp,0);
           printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                              fabs(F(x)-y));
       }
   }
}
```

## Output

```
The order of the spline is 3
    x        F(x)      Interpolant   Error
```

```
  0.250     -0.572      -0.542      0.0299
  0.300     -0.978      -0.978      0.0000
  0.350     -0.859      -0.819      0.0397
  0.400     -0.279      -0.279      0.0000
  0.450      0.450       0.429      0.0210
  0.500      0.938       0.938      0.0000
  0.550      0.923       0.879      0.0433
  0.600      0.412       0.412      0.0000
  0.650     -0.320      -0.305      0.0149
  0.700     -0.880      -0.880      0.0000
  0.750     -0.968      -0.922      0.0459

The order of the spline is 6
    x         F(x)      Interpolant   Error
  0.250     -0.572      -0.573      0.0016
  0.300     -0.978      -0.978      0.0000
  0.350     -0.859      -0.856      0.0031
  0.400     -0.279      -0.279      0.0000
  0.450      0.450       0.448      0.0020
  0.500      0.938       0.938      0.0000
  0.550      0.923       0.922      0.0003
  0.600      0.412       0.412      0.0000
  0.650     -0.320      -0.322      0.0025
  0.700     -0.880      -0.880      0.0000
  0.750     -0.968      -0.959      0.0090
```

## Warning Errors

| | |
|---|---|
| IMSL_ILL_COND_INTERP_PROB | The interpolation matrix is ill-conditioned. The solution might not be accurate. |

## Fatal Errors

| | |
|---|---|
| IMSL_DUPLICATE_XDATA_VALUES | The xdata values must be distinct. |
| IMSL_KNOT_MULTIPLICITY | Multiplicity of the knots cannot exceed the order of the spline. |
| IMSL_KNOT_NOT_INCREASING | The knots must be nondecreasing. |
| IMSL_KNOT_XDATA_INTERLACING | The *i*-th smallest element of xdata ($x_j$) must satisfy $\mathbf{t}_j \le x_j < \mathbf{t}_{j+\mathrm{order}}$ where **t** is the knot sequence. |
| IMSL_XDATA_TOO_LARGE | The array xdata must satisfy $\mathrm{xdata}_j \le \mathbf{t}_{\mathrm{ndata}}$, for $i = 1, \ldots, \mathrm{ndata}$. |
| IMSL_XDATA_TOO_SMALL | The array xdata must satisfy $\mathrm{xdata}_j \ge \mathbf{t}_{\mathrm{order}-1}$, for $i = 1, \ldots, \mathrm{ndata}$. |

# spline_knots

Computes the knots for a spline interpolant

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_spline_knots (*int* ndata, *float* xdata[], …, 0)

The type *double* function is imsl_d_spline_knots.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the interpolation problem.

## Return Value

A pointer to the knots. If the knots cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_spline_knots (*int* ndata, *float* xdata[],

IMSL_ORDER, *int* order,

IMSL_OPT,

IMSL_OPT_ITMAX, *int* itmax,

IMSL_RETURN_USER, *float* knots[],

0)

## Optional Arguments

`IMSL_ORDER`, *int* `order` (Input)

The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.

Default: `order` = 4, i.e., cubic splines

`IMSL_OPT`

This option produces knots that satisfy an optimality criterion.

`IMSL_OPT_ITMAX`, *int* `itmax` (Input)

This option allows the user to set the maximum number of iterations of Newton's method.

Default: `itmax` = 10

`IMSL_RETURN_USER`, *float* `knots[]` (Output)

This option requires the user to provide the space for the return knots. For example, the user could declare `float knots`[100]; and pass in `knots`. The return value is then also set to `knots`.

## Description

Given the data points $x$ = `xdata`, the order of the spline $k$ = `order`, and the number $n$ = `ndata` of elements in `xdata`, the default action of `imsl_f_spline_knots` returns a pointer to a knot sequence that is appropriate for interpolation of data on $x$ by splines of order $k$ (the default order is $k = 4$). The knot sequence is contained in its first $n + k$ positions. If $k$ is even, and we assume that the entries in the input vector $x$ are increasing, then the resulting knot sequence **t** is returned as

$$t_i = x_0 \quad \text{for} \quad i = 0, \dots, k-1$$
$$t_i = x_{j-k/2-1} \quad \text{for} \quad i = k, \dots, n-1$$
$$t_i = x_{n-1} \quad \text{for} \quad i = n, \dots, n+k-1$$

There is some discussion concerning this selection of knots in de Boor (1978, p. 211). If $k$ is odd, then **t** is returned as

$$t_\mathrm{i} = x_0 \quad \text{for} \quad i = 0, \dots, k-1$$
$$t_\mathrm{i} = \left( x_{\mathrm{i} - \frac{\mathrm{k}-1}{2} - 1} + x_{\mathrm{i} - 1 - \frac{\mathrm{k}-2}{2}} \right) / 2 \quad \text{for} \quad i = k, \dots, n-1$$
$$t_\mathrm{i} = x_{\mathrm{n}-1} \quad \text{for} \quad i = n, \dots, n+k-1$$

It is not necessary to sort the values in `xdata`.

If the option `IMSL_OPT` is selected, then the knot sequence returned minimizes the constant $c$ in the error estimate

$$\|f - s\| \le c \|f^{(\mathrm{k})}\|$$

In the above formula, $f$ is any function in $C^k$, and $s$ is the spline interpolant to $f$ at the abscissas $x$ with knot sequence **t**.

The algorithm is based on a routine described in de Boor (1978, p. 204), which in turn is based on a theorem of Micchelli et al. (1976).

# Examples

## Example 1

In this example, knots for a cubic spline are generated and printed. Notice that the knots are stacked at the endpoints and that the second and next to last data points are not knots.

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

#define NDATA 6

int main()
{
   int        i;
   float      *knots, xdata[NDATA];

   for(i = 0; i < NDATA; i++)
       xdata[i] = i;
   knots = imsl_f_spline_knots(NDATA, xdata, 0);
   imsl_f_write_matrix("The knots for the cubic spline are:\n",
                       1, NDATA+4, knots,
                       IMSL_COL_NUMBER_ZERO,
                       0);
}
```

### Output

```
                The knots for the cubic spline are:

        0          1          2          3          4          5
        0          0          0          0          2          3

        6          7          8          9
        5          5          5          5
```

## Example 2

This is a continuation of the examples for `imsl_f_spline_interp`. Recall that in these examples, a cubic spline interpolant to a function *f* is computed first. The values of this spline are then compared with the exact function values. The second example uses a quadratic ($k$ = 3) and a quintic ($k$ = 6) spline interpolant to the data. Now, instead of using the default knots, select the "optimal" knots as described above. Notice that the error is actually worse in this case.

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

#define NDATA  11
                              /* Define function */
#define F(x)   (float)(sin(15.0*x))

int main()
{
   int                  i, order;
   float                fdata[NDATA], xdata[NDATA], *knots, x, y;
   Imsl_f_spline        *sp;
                              /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
   for(order = 3; order < 7; order += 3) {
       knots = imsl_f_spline_knots(NDATA, xdata, IMSL_ORDER, order,
                                    IMSL_OPT,
                                    0);
                              /* Compute spline interpolant */
       sp = imsl_f_spline_interp (NDATA, xdata,fdata,
                                   IMSL_ORDER, order,
                                   IMSL_KNOTS, knots,
                                   0);
                              /* Print results */
       printf("\nThe order of the spline is %d\n", order);
       printf("    x         F(x)      Interpolant   Error\n");
       for (i = NDATA/2; i < 3*NDATA/2; i++) {
           x = (float) i /(float)(2*NDATA-2);
           y = imsl_f_spline_value(x, sp, 0);
           printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
                                                   fabs(F(x)-y));
       }
   }
}
```

### Output

```
The order of the spline is 3
    x         F(x)      Interpolant   Error
  0.250     -0.572      -0.543      0.0290
  0.300     -0.978      -0.978      0.0000
  0.350     -0.859      -0.819      0.0401
  0.400     -0.279      -0.279      0.0000
  0.450      0.450       0.429      0.0210
  0.500      0.938       0.938      0.0000
  0.550      0.923       0.879      0.0433
```

```
  0.600      0.412      0.412      0.0000
  0.650     -0.320     -0.305      0.0150
  0.700     -0.880     -0.880      0.0000
  0.750     -0.968     -0.920      0.0478

The order of the spline is 6
    x        F(x)      Interpolant   Error
  0.250     -0.572     -0.578      0.0061
  0.300     -0.978     -0.978      0.0000
  0.350     -0.859     -0.854      0.0054
  0.400     -0.279     -0.279      0.0000
  0.450      0.450      0.448      0.0019
  0.500      0.938      0.938      0.0000
  0.550      0.923      0.920      0.0022
  0.600      0.412      0.412      0.0000
  0.650     -0.320     -0.317      0.0020
  0.700     -0.880     -0.880      0.0000
  0.750     -0.968     -0.966      0.0023
```

# Warning Errors

| | |
|---|---|
| IMSL_NO_CONV_NEWTON | Newton's method iteration did not converge. |

# Fatal Errors

| | |
|---|---|
| IMSL_DUPLICATE_XDATA_VALUES | The xdata values must be distinct. |
| IMSL_ILL_COND_LIN_SYS | Interpolation matrix is singular. The xdata values may be too close together. |

# spline_2d_interp

Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data.

## Synopsis

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_2d_interp (*int* num_xdata, *float* xdata [], *int* num_ydata, *float* ydata [], *float* fdata [], …, 0)

The type *Imsl_d_spline* function is imsl_d_spline_2d_interp.

## Required Arguments

*int* num_xdata (Input)
Number of data points in the *X* direction.

*float* xdata [] (Input)
Array with num_xdata components containing the data points in the *X* direction.

*int* num_ydata (Input)
Number of data points in the *Y* direction.

*float* ydata [] (Input)
Array with num_ydata components containing the data points in the *Y* direction.

*float* fdata [] (Input)
Array of size num_xdata × num_ydata containing the values to be interpolated. fdata[i][j] is the value at (xdata[i], ydata[j]).

## Return Value

A pointer to the structure that represents the tensor-product spline interpolant. If an interpolant cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_2d_interp (*int* num_xdata, *float* xdata [], *int* num_ydata, *float* ydata [], *float* fdata [],

IMSL_ORDER, *int* xorder, *int* yorder,

IMSL_KNOTS, *float* xknots[], *float* yknots[],

IMSL_FDATA_COL_DIM, *int* fdata_col_dim,

0)

## Optional Arguments

IMSL_ORDER, *int* xorder, *int* yorder (Input)
This option is used to communicate the order of the spline subspace.
Default: xorder, yorder = 4, (i.e., tensor-product cubic splines)

IMSL_KNOTS, *float* xknots[], *float* yknots[] (Input)
This option requires the user to provide the knots. The default knots are selected by the function imsl_f_spline_knots using its defaults.

IMSL_FDATA_COL_DIM, *int* fdata_col_dim (Input)
The column dimension of the matrix fdata.
Default: fdata_col_dim = num_ydata

## Description

The function imsl_f_spline_2d_interp computes a tensor-product spline interpolant. The tensor-product spline interpolant to data $\{(x_j, y_j, f_{jj})\}$, where $0 \le i \le n_x - 1$ and $0 \le j \le n_y - 1$ has the form

$$\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_x-1} c_{nm} B_{n,\, k_x,\, t_x}(x) B_{m,\, k_y,\, t_y}(y)$$

where $k_x$ and $k_y$ are the orders of the splines. These numbers are defaulted to be 4, but can be set to any positive integer using the keyword, IMSL_ORDER. Likewise, $\mathbf{t}_x$ and $\mathbf{t}_y$ are the corresponding knot sequences (xknots and yknots). These values are defaulted to the knots returned by imsl_f_spline_knots. The algorithm requires that

$$t_x\left(k_x - 1\right) \le x_i \le t_x\left(n_x\right) \quad 0 \le i \le n_x - 1$$
$$t_y\left(k_y - 1\right) \le y_i \le t_y\left(n_y - 1\right) \quad 0 \le j \le n_y - 1$$

Tensor-product spline interpolants in two dimensions can be computed quite efficiently by solving (repeatedly) two univariate interpolation problems.

The computation is motivated by the following observations. It is necessary to solve the system of equations

$$\sum_{m=0}^{n_y-1}\sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}\left(x_i\right) B_{m,k_y,t_y}\left(y_j\right) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x-1} c_{nm} B_{n,\, k_x,\, t_x}(x_i)$$

note that for each fixed *i* from 1 to $n_x - 1$, we have $n_y - 1$ linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=0}^{n_y-1} h_{mi} B_{m,k_y,t_y}(y_i) = f_{ij}$$

The same matrix appears in all of the equations above:

$$\left[ B_{m,k_y,t_y}(y_j) \right] \quad 1 \le m,j \le n_y - 1$$

Thus, only factor this matrix once and then apply this factorization to the $n_x$ right-hand sides. Once this is done and $h_{mi}$ is computed, then solve for the coefficients $c_{nm}$ using the relation

$$\sum_{n=0}^{n_x-1} c_{nm} B_{n,\, k_x,\, t_x}(x_i) = h_{mi}$$

for *m* from 0 to $n_y - 1$, which again involves one factorization and $n_y$ solutions to the different right-hand sides. The function `imsl_f_spline_2d_interp` is based on the routine SPLI2D by de Boor (1978, p. 347).

The return value for this function is a pointer to the structure `imsl_f_spline`. The calling program must receive this in a pointer `imsl_f_spline *sp`. This structure contains all the information to determine the spline (stored in B-spline format) that is computed by this procedure. For example, the following code sequence evaluates this spline at (*x,y*) and returns the value in *z*.

```
z = imsl_f_spline_2d_value (x, y, sp, 0);
```

# Examples

## Example 1

In this example, a tensor-product spline interpolant to a function *f* is computed. The values of the interpolant and the error on a 4 × 4 grid are displayed.

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

#define NDATA            11
#define OUTDATA           2
                                /* Define function */
#define F(x, y)           (float)(x*x*x+y*y)
```

```
int main()
{
    int                  i, j, num_xdata, num_ydata;
    float                fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float                x, y, z;
    Imsl_f_spline         *sp;
                                  /* Set up grid */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = ydata[i] = (float)i / ((float)(NDATA-1));
    }
    for (i = 0; i < NDATA; i++) {
        for (j = 0; j < NDATA; j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                  /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                            ydata, (float*) fdata, 0);
                                  /* Print results */
    printf("    x      y       F(x, y)   Interpolant   Error \n");
    for (i = 0; i < OUTDATA; i++) {
        x = (float) i / (float) (OUTDATA);
        for (j = 0; j < OUTDATA; j++) {
            y = (float) j / (float) (OUTDATA);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                   x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

### Output

```
  x       y       F(x, y)   Interpolant    Error
0.000   0.000      0.000       0.000       0.0000
0.000   0.500      0.250       0.250       0.0000
0.500   0.000      0.125       0.125       0.0000
0.500   0.500      0.375       0.375       0.0000
```

## Example 2

Recall that in the first example, a tensor-product spline interpolant to a function *f* is computed. The values of the interpolant and the error on a 4 × 4 grid are displayed. Notice that the first interpolant with `order` = 3 does not reproduce the cubic data, while the second interpolant with `order` = 6 does reproduce the data.

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

```
#define NDATA           7
#define OUTDATA         4
                                    /* Define function */
#define F(x,y)          (float)(x*x*x+y*y)

int main()
{
   int                  i, j, num_xdata, num_ydata, order;
   float                fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
   float                x, y, z;
   Imsl_f_spline         *sp;
                                    /* Set up grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = ydata[i] = (float) i / ((float) (NDATA - 1));
   }
   for (i = 0; i < NDATA; i++) {
       for (j = 0; j < NDATA; j++) {
           fdata[i][j] = F(xdata[i], ydata[j]);
       }
   }
   num_xdata = num_ydata = NDATA;

   for(order = 3; order < 7; order += 3) {
                                    /* Compute tensor-product interpolant */
       sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                ydata, (float *)fdata,
                                         IMSL_ORDER, order, order,
                                         0);
                                    /* Print results */
      printf("\nThe order of the spline is %d \n", order);
      printf("    x       y       F(x, y)    Interpolant    Error\n");
      for (i = 0; i < OUTDATA; i++) {
           x = (float) i / (float) (OUTDATA);
           for (j = 0; j < OUTDATA; j++) {
               y = (float) j / (float) (OUTDATA);
               z = imsl_f_spline_2d_value(x, y, sp, 0);
               printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f  \n",
                      x, y, F(x,y), z, fabs(F(x,y)-z));
           }
       }
   }
}
```

### Output

```
The order of the spline is 3
    x       y       F(x, y)    Interpolant    Error
  0.000   0.000      0.000        0.000      0.0000
  0.000   0.250      0.062        0.063      0.0000
  0.000   0.500      0.250        0.250      0.0000
  0.000   0.750      0.562        0.562      0.0000
```

```
  0.250   0.000      0.016       0.016      0.0002
  0.250   0.250      0.078       0.078      0.0002
  0.250   0.500      0.266       0.266      0.0002
  0.250   0.750      0.578       0.578      0.0002
  0.500   0.000      0.125       0.125      0.0000
  0.500   0.250      0.188       0.188      0.0000
  0.500   0.500      0.375       0.375      0.0000
  0.500   0.750      0.688       0.687      0.0000
  0.750   0.000      0.422       0.422      0.0002
  0.750   0.250      0.484       0.484      0.0002
  0.750   0.500      0.672       0.672      0.0002
  0.750   0.750      0.984       0.984      0.0002

The order of the spline is 6
    x       y        F(x, y)   Interpolant   Error
  0.000   0.000      0.000       0.000      0.0000
  0.000   0.250      0.062       0.063      0.0000
  0.000   0.500      0.250       0.250      0.0000
  0.000   0.750      0.562       0.562      0.0000
  0.250   0.000      0.016       0.016      0.0000
  0.250   0.250      0.078       0.078      0.0000
  0.250   0.500      0.266       0.266      0.0000
  0.250   0.750      0.578       0.578      0.0000
  0.500   0.000      0.125       0.125      0.0000
  0.500   0.250      0.188       0.188      0.0000
  0.500   0.500      0.375       0.375      0.0000
  0.500   0.750      0.688       0.688      0.0000
  0.750   0.000      0.422       0.422      0.0000
  0.750   0.250      0.484       0.484      0.0000
  0.750   0.500      0.672       0.672      0.0000
  0.750   0.750      0.984       0.984      0.0000
```

## Warning Errors

| | |
|---|---|
| `IMSL_ILL_COND_INTERP_PROB` | The interpolation matrix is ill-conditioned. The solution might not be accurate. |

## Fatal Errors

| | |
|---|---|
| `IMSL_XDATA_NOT_INCREASING` | The `xdata` values must be strictly increasing. |
| `IMSL_YDATA_NOT_INCREASING` | The `ydata` values must be strictly increasing. |
| `IMSL_KNOT_MULTIPLICITY` | Multiplicity of the knots cannot exceed the order of the spline. |
| `IMSL_KNOT_NOT_INCREASING` | The knots must be nondecreasing. |
| `IMSL_KNOT_DATA_INTERLACING` | The *i*-th smallest element of the data arrays `xdata` and `ydata` must satisfy $\mathbf{t}_i \le \mathtt{data}_i < \mathbf{t}_{i+\mathrm{order}}$, where $\mathbf{t}$ is the knot sequence. |

| | |
|---|---|
| IMSL_DATA_TOO_LARGE | The data arrays xdata and ydata must satisfy $\text{data}_i \leq \mathbf{t}_{\text{num\_data}}$, for $i$ = 1, …, num_data. |
| IMSL_DATA_TOO_SMALL | The data arrays xdata and ydata must satisfy $\text{data}_i \geq \mathbf{t}_{\text{order-1}}$, for $i$ = 1, …, num_data. |

# spline_value

Computes the value of a spline or the value of one of its derivatives.

## Synopsis

*#include* <imsl.h>

*float* `imsl_f_spline_value` (*float* `x`, *Imsl_f_spline* `*sp`, …, 0)

The type ***double*** function is `imsl_d_spline_value`.

## Required Arguments

*float* `x` (Input)
Evaluation point for the spline.

*Imsl_f_spline* `*sp` (Input)
Pointer to the structure that represents the spline.

## Return Value

The value of a spline or one of its derivatives at the point ***x***. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* `imsl_f_spline_value` (*float* `x`, *Imsl_f_spline* `*sp`,

`IMSL_DERIV`, *int* `deriv`,

`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value`,

`IMSL_GRID_USER`, *int* `n`, *float* `*xvec`, *float* `value_user[]`,

0)

## Optional Arguments

`IMSL_DERIV`, *int* `deriv` (Input)
Let ***d*** = `deriv` and let ***s*** be the spline that is represented by the structure `*sp`. Then, this option produces the ***d***-th derivative of ***s*** at ***x***, $s^{(d)}(x)$.
Default: `deriv` = 0

`IMSL_GRID`, *int* `n`, *float* `*xvec`, *float* `**value` (Input/Output)
The argument `xvec` is the array of length `n` containing the points at which the spline is to be evaluated. The ***d***-th derivative of the spline at the points in `xvec` is returned in `value`.

`IMSL_GRID_USER` *int* `n`, *float* `*xvec`, *float* `value_user[]` (Input/Output)
The argument `xvec` is the array of length `n` containing the points at which the spline is to be evaluated. The ***d***-th derivative of the spline at the points in `xvec` is returned in `value_user`.

# Description

The function `imsl_f_spline_value` computes the value of a spline or one of its derivatives. This function is based on the routine BVALUE by de Boor (1978, p. 144).

# Examples

## Example 1

In this example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. Since the default settings are used, the interpolant is determined by the “not-a-knot” condition (see de Boor 1978).

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

#define NDATA  11
                                /* Define function */
#define F(x)    (float)(sin(15.0*x))

int main()
{
   int                 i;
   float               fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_spline       *sp;
                                /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                                /* Compute cubic spline interpolant */
   sp = imsl_f_spline_interp (NDATA, xdata,fdata, 0);
                                /* Print results */
   printf("    x        F(x)      Interpolant   Error\n");
   for (i = NDATA/2; i < 3*NDATA/2; i++){
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_spline_value(x, sp, 0);
       printf(" %6.3f %10.3f  %10.3f  %10.4f\n", x, F(x), y,
```

```
                                                  fabs(F(x)-y));
    }
}
```

### Output

```
  x        F(x)     Interpolant   Error
0.250     -0.572      -0.549      0.0228
0.300     -0.978      -0.978      0.0000
0.350     -0.859      -0.843      0.0162
0.400     -0.279      -0.279      0.0000
0.450      0.450       0.441      0.0093
0.500      0.938       0.938      0.0000
0.550      0.923       0.903      0.0199
0.600      0.412       0.412      0.0000
0.650     -0.320      -0.315      0.0049
0.700     -0.880      -0.880      0.0000
0.750     -0.968      -0.938      0.0295
```

## Example 2

Recall that in the first example, a cubic spline interpolant to a function *f* is computed. The values of this spline are then compared with the exact function values. This example compares the values of the first derivatives.

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

#define NDATA  11
                               /* Define function */
#define F(x)    (float)(sin(15.0*x))
#define FP(x)   (float)(15.*cos(15.0*x))

int main()
{
   int                  i;
   float                fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_spline         *sp;
                               /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                               /* Compute cubic spline interpolant */
   sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                               /* Print results */
   printf("    x        FP(x)     Interpolant  Deriv Error\n");
   for (i = NDATA/2; i < 3*NDATA/2; i++) {
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_spline_value(x, sp, IMSL_DERIV, 1, 0);
```

```
        printf(" %6.3f %10.3f  %10.3f  %10.4f  \n", x, FP(x), y,
                                                   fabs(FP(x)-y));
    }
}
```

### Output

```
  x        FP(x)  Interpolant  Deriv Error
0.250    -12.308     -12.559      0.2510
0.300     -3.162      -3.218      0.0560
0.350      7.681       7.796      0.1151
0.400     14.403      13.919      0.4833
0.450     13.395      13.530      0.1346
0.500      5.200       5.007      0.1926
0.550     -5.786      -5.840      0.0535
0.600    -13.667     -13.201      0.4660
0.650    -14.214     -14.393      0.1798
0.700     -7.133      -6.734      0.3990
0.750      3.775       3.911      0.1359
```

## Fatal Errors

| | |
|---|---|
| IMSL_KNOT_MULTIPLICITY | Multiplicity of the knots cannot exceed the order of the spline. |
| IMSL_KNOT_NOT_INCREASING | The knots must be nondecreasing. |

# spline_integral

Computes the integral of a spline.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_spline_integral (*float* a, *float* b, *Imsl_f_spline* *sp)

The type *double* function is imsl_d_spline_integral.

## Required Arguments

*float* a (Input)
The lower limit of integration.

*float* b (Input)
Endpoints for integration.

*Imsl_f_spline* *sp (Input)
Pointer to the structure that represents the spline.

## Return Value

The integral of a spline. If no value can be computed, then NaN is returned.

## Description

The function imsl_f_spline_integral computes the integral of a spline from ***a*** to ***b***

$$\int_a^b s(x)\,dx$$

This routine uses the identity (22) on page 151 of de Boor (1978).

## Example

In this example, a cubic spline interpolant to a function *f* is computed. The values of the integral of this spline are then compared with the exact integral values. Since the default settings are used, the interpolant is determined by the "not-a-knot" condition (see de Boor 1978).

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

```
#include <math.h>

#define NDATA   21
                              /* Define function */
#define F(x)    (float)(sin(15.0*x))
                              /* Integral from 0 to x */
#define FI(x)   (float)((1.-cos(15.0*x))/15.)

int main()
{
   int              i;
   float            fdata[NDATA], xdata[NDATA], x, y;
   Imsl_f_spline    *sp;
                              /* Set up a grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = (float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]);
   }
                              /* Compute cubic spline interpolant */
   sp = imsl_f_spline_interp (NDATA, xdata, fdata, 0);
                              /* Print results */
   printf("    x        FI(x)     Interpolant  Integral Error\n");
   for (i = NDATA/2; i < 3*NDATA/2; i++) {
       x = (float) i /(float)(2*NDATA-2);
       y = imsl_f_spline_integral(0.0, x, sp);
       printf(" %6.3f %10.3f  %10.3f  %10.4f  \n", x, FI(x), y,
                                                    fabs(FI(x)-y));
   }
}
```

## Output

```
  x       FI(x)   Interpolant Integral Error
0.250      0.121       0.121       0.0001
0.275      0.104       0.104       0.0001
0.300      0.081       0.081       0.0001
0.325      0.056       0.056       0.0001
0.350      0.033       0.033       0.0001
0.375      0.014       0.014       0.0002
0.400      0.003       0.003       0.0002
0.425      0.000       0.000       0.0002
0.450      0.007       0.007       0.0002
0.475      0.022       0.022       0.0001
0.500      0.044       0.044       0.0001
0.525      0.068       0.068       0.0001
0.550      0.092       0.092       0.0001
0.575      0.113       0.113       0.0001
0.600      0.127       0.128       0.0001
0.625      0.133       0.133       0.0001
0.650      0.130       0.130       0.0001
```

```
0.675       0.118       0.118       0.0001
0.700       0.098       0.098       0.0001
0.725       0.075       0.075       0.0001
0.750       0.050       0.050       0.0001
```

## Warning Errors

| | |
|---|---|
| `IMSL_SPLINE_SMLST_ELEMNT` | The data arrays xdata and ydata must satisfy $data_i \le \mathbf{t}_{order-1}$, for $i = 1, \ldots,$ num_data. |
| `IMSL_SPLINE_EQUAL_LIMITS` | The upper and lower endpoints of integration are equal. The indefinite integral is zero. |
| `IMSL_LIMITS_LOWER_TOO_SMALL` | The left endpoint is less than $\mathbf{t}_{order-1}$. Integration occurs only from $\mathbf{t}_{order-1}$ to b. |
| `IMSL_LIMITS_UPPER_TOO_SMALL` | The right endpoint is less than $\mathbf{t}_{order-1}$. Integration occurs only from $\mathbf{t}_{order-1}$ to a. |
| `IMSL_LIMITS_UPPER_TOO_BIG` | The right endpoint is greater than $\mathbf{t}_{spline\_space\_dim-1}$. Integration occurs only from a to $\mathbf{t}_{spline\_space\_dim-1}$. |
| `IMSL_LIMITS_LOWER_TOO_BIG` | The left endpoint is greater than $\mathbf{t}_{spline\_space\_dim-1}$. Integration occurs only from b to $\mathbf{t}_{spline\_space\_dim-1}$. |

## Fatal Errors

| | |
|---|---|
| `IMSL_KNOT_MULTIPLICITY` | Multiplicity of the knots cannot exceed the order of the spline. |
| `IMSL_KNOT_NOT_INCREASING` | The knots must be nondecreasing. |

# spline_2d_value

Computes the value of a tensor-product spline or the value of one of its partial derivatives.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_spline_2d_value (*float* x, *float* y, *Imsl_f_spline* *sp, …, 0)

The type *double* function is imsl_d_spline_2d_value.

## Required Arguments

*float* x (Input)

*float* y (Input)
The (*x*, *y*) coordinates of the evaluation point for the tensor-product spline.

*Imsl_f_spline* *sp (Input)
Pointer to the structure that represents the spline.

## Return Value

The value of a tensor-product spline or one of its derivatives at the point (*x*, *y*).

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_spline_2d_value (*float* x, *float* y, *Imsl_f_spline* *sp,

IMSL_DERIV, *int* x_partial, *int* y_partial,

IMSL_GRID, *int* nx, *float* *xvec, *int* ny, *float* *yvec, *float* **value,

IMSL_GRID_USER, *int* nx, *float* *xvec, *int* ny, *float* *yvec, *float* value_user[],

0)

## Optional Arguments

IMSL_DERIV, *int* x_partial, *int* y_partial (Input)

Let $p$ = x_partial and $q$ = y_partial, and let $s$ be the spline that is represented by the structure *sp, then this option produces the $(p, q)$-th derivative of $s$ at $(x, y)$, $s^{(p,q)}(x, y)$.

Default: x_partial = y_partial = 0

IMSL_GRID, *int* nx, *float* *xvec, *int* ny, *float* *yvec, *float* **value (Input/Output)

The argument xvec is the array of length nx containing the *X* coordinates at which the spline is to be evaluated. The argument yvec is the array of length ny containing the *Y* coordinates at which the spline is to be evaluated. The value of the spline on the nx by ny grid is returned in value.

IMSL_GRID_USER, *int* nx, *float* *xvec, *int* ny, *float* *yvec, *float* value_user[] (Input/Output)

The argument xvec is the array of length nx containing the *X* coordinates at which the spline is to be evaluated. The argument yvec is the array of length ny containing the *Y* coordinates at which the spline is to be evaluated. The value of the spline on the nx by ny grid is returned in the user-supplied space value_user.

## Description

The function imsl_f_spline_2d_value computes the value of a tensor-product spline or one of its derivatives. This function is based on the discussion in de Boor (1978, pp. 351−353).

## Examples

### Example 1

In this example, a spline interpolant $s$ to a function $f$ is constructed. Using the procedure imsl_f_spline_2d_interp to compute the interpolant, imsl_f_spline_2d_value is employed to compute $s(x, y)$. The values of this partial derivative and the error are computed on a 4 × 4 grid and then displayed.

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

#define NDATA           11
#define OUTDATA          2
                               /* Define function */
#define F(x,y)          (float)(x*x*x+y*y)

int main()
{
   int                 i, j, num_xdata, num_ydata;
```

```
    float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
    float               x, y, z;
    Imsl_f_spline       *sp;
                                 /* Set up grid */
    for (i = 0; i < NDATA; i++) {
        xdata[i] = ydata[i] = (float) i / ((float) (NDATA - 1));
    }
    for (i = 0; i < NDATA; i++) {
        for (j = 0; j < NDATA; j++) {
            fdata[i][j] = F(xdata[i], ydata[j]);
        }
    }
    num_xdata = num_ydata = NDATA;
                                 /* Compute tensor-product interpolant */
    sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                      ydata, fdata, 0);
                                 /* Print results */
    printf("    x       y       F(x, y)       Value      Error\n");
    for (i = 0; i < OUTDATA; i++) {
        x = (float) (1+i) / (float) (OUTDATA+1);
        for (j = 0; j < OUTDATA; j++) {
            y = (float) (1+j) / (float) (OUTDATA+1);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                   x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

### Output

```
  x      y        F(x, y)      Value       Error
0.333  0.333       0.148       0.148       0.0000
0.333  0.667       0.481       0.481       0.0000
0.667  0.333       0.407       0.407       0.0000
0.667  0.667       0.741       0.741       0.0000
```

## Example 2

In this example, a spline interpolant ***s*** to a function ***f*** is constructed. Using function `imsl_f_spline_2d_interp` to compute the interpolant, then `imsl_f_spline_2d_value` is employed to compute $s^{(2,1)}(x, y)$. The values of this partial derivative and the error are computed on a 4 × 4 grid and then displayed.

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

#define NDATA           11
#define OUTDATA          2
```

```
                                /* Define function */
#define F(x, y)         (float)(x*x*x*y*y)
#define F21(x,y)        (float)(6.*x*2.*y)

int main()
{
   int                  i, j, num_xdata, num_ydata;
   float                fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
   float                x, y, z;
   Imsl_f_spline        *sp;
                                /* Set up grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = ydata[i] = (float)i / ((float)(NDATA-1));
   }
   for (i = 0; i < NDATA; i++) {
       for (j = 0; j < NDATA; j++) {
           fdata[i][j] = F(xdata[i], ydata[j]);
       }
   }
   num_xdata = num_ydata = NDATA;
                                /* Compute tensor-product interpolant */
   sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                       ydata, fdata, 0);
                                /* Print results */
   printf("    x      y      F21(x, y)  21InterpDeriv  Error\n");
   for (i = 0; i < OUTDATA; i++) {
       x = (float) (1+i) / (float) (OUTDATA+1);
       for (j = 0; j < OUTDATA; j++) {
           y = (float) (1+j) / (float) (OUTDATA+1);
           z = imsl_f_spline_2d_value(x, y, sp,
                                  IMSL_DERIV, 2, 1,
                                  0);
           printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                  x, y, F21(x, y), z, fabs(F21(x,y)-z));
       }
   }
}
```

## Output

```
   x      y    F21(x, y)  21InterpDeriv  Error
0.333  0.333     1.333       1.333      0.0000
0.333  0.667     2.667       2.667      0.0000
0.667  0.333     2.667       2.667      0.0000
0.667  0.667     5.333       5.333      0.0001
```

## Warning Errors

| | |
|---|---|
| IMSL_X_NOT_WITHIN_KNOTS | The value of x does not lie within the knot sequence. |
| IMSL_Y_NOT_WITHIN_KNOTS | The value of y does not lie within the knot sequence. |

## Fatal Errors

| | |
|---|---|
| IMSL_KNOT_MULTIPLICITY | Multiplicity of the knots cannot exceed the order of the spline. |
| IMSL_KNOT_NOT_INCREASING | The knots must be nondecreasing. |

# spline_2d_integral

Evaluates the integral of a tensor-product spline on a rectangular domain.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_spline_2d_integral (*float* a, *float* b, *float* c, *float* d, *Imsl_f_spline* *sp)

The type ***double*** function is imsl_d_spline_2d_integral.

## Required Arguments

*float* a (Input)
blah

*float* b (Input)
The integration limits for the first variable of the tensor-product spline.

*float* c (Input)
blah

*float* d (Input)
The integration limits for the second variable of the tensor-product spline.

*Imsl_f_spline* *sp (Input)
Pointer to the structure that represents the spline.

## Return Value

The value of the integral of the tensor-product spline over the rectangle $[a, b] \times [c, d]$. If no value can be computed, NaN is returned.

## Description

The function imsl_f_spline_2d_integral computes the integral of a tensor-product spline. If $s$ is the spline, then this function returns

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

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

$$\int_{t_0}^{x}\sum_{i=0}^{n-1}\alpha_i B_{i,k}(\tau)d\tau = \sum_{i=0}^{r-1}\left[\sum_{j=0}^{i}\alpha_j\frac{t_{j+k}-t_j}{k}\right]B_{i,k+1}(x)$$

where $\mathbf{t}_0 \le x \le \mathbf{t}_r$.

It assumes (for all knot sequences) that the first and last $k$ knots are stacked, that is, $\mathbf{t}_0 = \ldots = \mathbf{t}_{k-1}$ and $\mathbf{t}_n = \ldots = \mathbf{t}_{n+k-1}$ , where $k$ is the order of the spline in the $x$ or $y$ direction.

## Example

This example integrates a two-dimensional, tensor-product spline over the rectangle $[0, x] \times [0, y]$.

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

#define NDATA            11
#define OUTDATA           2
                               /* Define function */
#define F(x,y)           (float)(x*x*x+y*y)
                               /* The integral of F from 0 to x */
                               /* and 0 to y */
#define FI(x,y)          (float)(y*x*x*x*x/4. + x*y*y*y/3.)

int main()
{
   int                 i, j, num_xdata, num_ydata;
   float               fdata[NDATA][NDATA], xdata[NDATA], ydata[NDATA];
   float               x, y, z;
   Imsl_f_spline       *sp;
                               /* Set up grid */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = ydata[i] = (float) i / ((float)(NDATA-1));
   }
   for (i = 0; i < NDATA; i++) {
       for (j = 0; j < NDATA; j++) {
           fdata[i][j] = F(xdata[i],ydata[j]);
       }
   }
   num_xdata = num_ydata = NDATA;
                               /* Compute tensor-product interpolant */
   sp = imsl_f_spline_2d_interp(num_xdata, xdata, num_ydata,
                                                   ydata, fdata, 0);
                               /* Print results */
   printf("    x       y       FI(x, y)      Integral    Error\n");
   for (i = 0; i < OUTDATA; i++) {
       x = (float) (1+i) / (float) (OUTDATA+1);
       for (j = 0; j < OUTDATA; j++) {
```

```
            y = (float) (1+j) / (float) (OUTDATA+1);
            z = imsl_f_spline_2d_integral(0.0, x, 0.0, y, sp);
            printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                   x, y, FI(x, y), z, fabs(FI(x,y)-z));
        }
    }
}
```

## Output

```
   x        y     FI(x, y)     Integral       Error
0.333   0.333        0.005        0.005      0.0000
0.333   0.667        0.035        0.035      0.0000
0.667   0.333        0.025        0.025      0.0000
0.667   0.667        0.099        0.099      0.0000
```

## Warning Errors

| | |
|---|---|
| `IMSL_SPLINE_LEFT_ENDPT` | The left endpoint of *X* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to `b`. |
| `IMSL_SPLINE_RIGHT_ENDPT` | The right endpoint of *X* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to `a`. |
| `IMSL_SPLINE_LEFT_ENDPT_1` | The left endpoint of *X* integration is not within the knot sequence. Integration occurs only from `b` to $\mathbf{t}_{spline\_space\_dim-1}$. |
| `IMSL_SPLINE_RIGHT_ENDPT_1` | The right endpoint of *X* integration is not within the knot sequence. Integration occurs only from `a` to $\mathbf{t}_{spline\_space\_dim-1}$. |
| `IMSL_SPLINE_LEFT_ENDPT_2` | The left endpoint of *Y* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to `d`. |
| `IMSL_SPLINE_RIGHT_ENDPT_2` | The right endpoint of *Y* integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{order-1}$ to `c`. |
| `IMSL_SPLINE_LEFT_ENDPT_3` | The left endpoint of *Y* integration is not within the knot sequence. Integration occurs only from `d` to $\mathbf{t}_{spline\_space\_dim-1}$. |
| `IMSL_SPLINE_RIGHT_ENDPT_3` | The right endpoint of *Y* integration is not within the knot sequence. Integration occurs only from `c` to $\mathbf{t}_{spline\_space\_dim-1}$. |

## Fatal Errors

| | |
|---|---|
| `IMSL_KNOT_MULTIPLICITY` | Multiplicity of the knots cannot exceed the order of the spline. |
| `IMSL_KNOT_NOT_INCREASING` | The knots must be nondecreasing. |

# spline_nd_interp

Performs multidimensional interpolation and differentiation for up to 7 dimensions.

## Synopsis

*#include* <imsl.h>

*float* `imsl_f_spline_nd_interp` (*int* `n`, *int* `d[]`, *float* `x[]`, *float* `xdata[]`, *float* `fdata[]`, ..., 0)

The type *double* function is `imsl_d_spline_nd_interp`.

## Required Arguments

*int* `n` (Input)
The dimension of the problem. `n` cannot be greater than seven.

*int* `d[]` (Input)
Array of length `n`. `d[`*i*`]` contains the number of gridpoints in the *i-th* direction.

*float* `x[]` (Input)
Array of length `n` containing the point at which interpolation is to be done. An interpolant is to be calculated at the point:

$$\left(X_1, X_2, \ldots, X_n\right)$$

where

$$X_1 = x\left[0\right], X_2 = x\left[1\right], \ldots$$

*float* `xdata[]` (Input)
Array of size `n` * *max*(`d[0]`, ..., `d[n-1]`) containing the gridpoint values for the grid.

*float* `fdata[]` (Input)
Array of length `d[0]` * `d[1]` * ...* `d[n-1]` containing the values of the function to be interpolated at the gridpoints.
`fdata`(*i, j, k*, ...) is the value of the function at

$$\left(Z_{1,\,i}, Z_{2,\,j}, Z_{3,\,k}, \ldots\right)$$

where

$$Z_{1,\,i} = xdata[0][i-1]$$

$$Z_{2,\,j} = xdata[1][j-1]$$

$$Z_{3,\,k} = xdata[2][k-1]$$

$$\text{for } i = 1, \ldots, d[0],\ j = 1, \ldots, d[1],\ k = 1, \ldots, d[2], \ldots$$

## Return Value

Interpolated value of the function. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_spline_nd_interp (*int* n, *int* d[], *float* x[], *float* xdata[], *float* fdata[],

IMSL_NDEGREE, *int* ndeg[],

IMSL_ORDER, *int* nders,

IMSL_DERIV, *float* **deriv,

IMSL_DERIV_USER, *float* deriv[],

IMSL_ERR_EST, *float* *error,

0)

## Optional Arguments

IMSL_NDEGREE, *int* ndeg[] (Input)

Array of length ***n*** containing the degree of polynomial interpolation to be used in each dimension. ndeg[*i*] must be less than or equal to 15.

Default: ndeg[*i*] = 5, for *i* = 0, ..., ***n−1***.

IMSL_ORDER, *int* nders (Input)

Maximum order of derivatives to be computed with respect to each variable. nders cannot be larger than ***max*** (7− ***n***, 2). All partial derivatives up to and including order nders are returned in each of the n dimensions. See deriv for more details.

Default: nders = 0.

IMSL_DERIV, *float* **deriv (Output)

Address of a pointer to an internally allocated ***n*** dimensional array, dimensioned (nders +1) × (nders +1) × ..., containing derivative estimates at the interpolation point.

`deriv [i] [j]` ... will hold an estimate of the derivative with respect to $x_1$ *i* times, and with respect to $x_2$ *j* times, etc. where *i* = 0, ..., `nders`, *j* = 0, ..., `nders`, .... The 0–*th* derivative means the function value, thus, `deriv[0][0] ... = imsl_f_spline_nd_interp`.

`IMSL_DERIV_USER`, *float* `deriv[]` (Output)
Storage for `deriv` is provided by the user. See `IMSL_DERIV`.

`IMSL_ERR_EST`, *float* `*error` (Output)
Estimate of the error.

## Description

The function `imsl_f_spline_nd_interp` interpolates a function of up to 7 variables, defined on a (possibly nonuniform) grid. It fits a polynomial of up to degree 15 in each variable through the grid points nearest the interpolation point. Approximations of partial derivatives are calculated, if requested. If derivatives are desired, high precision is strongly recommended. For more details, see Krogh (1970).

## Example

The 3D function $f(x, y, z) = exp(x + 2y + 3z)$, defined on a 20 by 30 by 40 uniform grid, is interpolated together with several partial derivatives.

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

#define  N  3
#define  ND1  20
#define  ND2  30
#define  ND3  40
#define  NDERS  1

int main() {
   char order[3];
   int i, j, k, ndeg[N], d[N], nders=NDERS;
   float xdata[N][ND3], fdata[ND1][ND2][ND3], x[N], xx, yout, yy,
       zz, derout[NDERS+1][NDERS+1][NDERS+1], error, relerr, tr;

   d[0] = ND1;
   d[1] = ND2;
   d[2] = ND3;

   /*
    * 20 by 30 by 40 uniform grid used for
    * interpolation of F(x,y,z) = exp(x+2*y+3*z)
    */
   ndeg[0] = 8;
```

```
ndeg[1] = 7;
ndeg[2] = 9;
for (i=0; i < ND1; i++)
    xdata[0][i] = 0.05*(i);

for (j=0; j < ND2; j++)
    xdata[1][j] = 0.03*(j);

for (k=0; k < ND3; k++)
    xdata[2][k] = 0.025*(k);

for (i=0; i < ND1; i++) {
    for (j=0; j < ND2; j++) {
        for (k=0; k < ND3; k++) {
            xx = xdata[0][i];
            yy = xdata[1][j];
            zz = xdata[2][k];
            fdata[i][j][k] = exp(xx+2*yy+3*zz);
        }
    }
}

/* Interpolate at (0.18,0.43,0.35)*/
x[0] = 0.18;
x[1] = 0.43;
x[2] = 0.35;

yout = imsl_f_spline_nd_interp(N, d, x, &xdata[0][0],
    &fdata[0][0][0],
    IMSL_NDEGREE, ndeg,
    IMSL_ORDER, nders,
    IMSL_DERIV_USER, &derout[0][0][0],
    IMSL_ERR_EST, &error, 0);

/*
 * Output F,Fx,Fy,Fz,Fxy,Fxz,Fyz,Fxyz at
 * interpolation point
 */
xx = x[0];
yy = x[1];
zz = x[2];

printf("EST. VALUE = %g, EST. ERROR = %g\n\n", yout, error);
printf("       Computed Der.  True Der.     Rel. Err\n");
for (k=0; k <= NDERS; k++) {
    for (j=0; j <= NDERS; j++) {
        for (i=0; i <= NDERS; i++) {
            order[0] = ' ';
            order[1] = ' ';
            order[2] = ' ';
            if (i == 1) order[0] = 'x';
```

```
                if (j == 1) order[1] = 'y';
                if (k == 1) order[2] = 'z';
                tr = pow(2,j) * pow(3,k) * exp(xx+2*yy+3*zz);
                relerr = (derout[i][j][k] - tr)/tr;
                printf("F%s", order);
                printf("%14.6f %14.6f %14.3e\n", derout[i][j][k],
                    tr, relerr);
            }
        }
    }
}
```

## Output

```
Est. Value = 8.08491, Est. Error = 4.18959e-006

       Computed Der.  True Der.     Rel. Err
F        8.084914      8.084915   -1.180e-007
Fx       8.084922      8.084915    8.257e-007
F y     16.169794     16.169830   -2.241e-006
Fxy     16.170071     16.169830    1.486e-005
F z    24.254747     24.254745    7.864e-008
Fx z    24.253994     24.254745   -3.098e-005
F yz    48.510410     48.509491    1.895e-005
Fxyz    48.533176     48.509491    4.883e-004
```

## Warning Errors

| | |
|---|---|
| IMSL_ARG_TOO_BIG | "nders" is too large, it has been reset to *max*(7−n,2). |
| IMSL_INTERP_OUTSIDE_DOMAIN | The interpolation point is outside the domain of the table, so extrapolation is used. |

## Fatal Errors

| | |
|---|---|
| IMSL_TOO_MANY_DERIVATIVES | Too many derivatives requested for the polynomial degree used. |
| IMSL_POLY_DEGREE_TOO_LARGE | One of the polynomial degrees requested is too large for the number of gridlines in that direction. |

# user_fcn_least_squares

Computes a least-squares fit using user-supplied functions.

## Synopsis

*#include* <imsl.h>

*float* `*imsl_f_user_fcn_least_squares` (*float* `fcn` (*int* `k`, *float* `x`), *int* `nbasis`, *int* `ndata`, *float* `xdata[]`, *float* `ydata[]`, …, 0)

The type ***double*** function is `imsl_d_user_fcn_least_squares`.

## Required Arguments

*float* `fcn` (*int* `k`, *float* `x`) (Input)
User-supplied function that defines the subspace from which the least-squares fit is to be performed. The ***k***-th basis function evaluated at `x` is `f(k, x)` where ***k*** = 1, 2, …, `nbasis`.

*int* `nbasis` (Input)
Number of basis functions.

*int* `ndata` (Input)
Number of data points.

*float* `xdata[]` (Input)
Array with `ndata` components containing the abscissas of the least-squares problem.

*float* `ydata[]` (Input)
Array with `ndata` components containing the ordinates of the least-squares problem.

## Return Value

A pointer to the vector containing the coefficients of the basis functions. If a fit cannot be computed, then `NULL` is returned. To release this space, use `imsl_free`.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* `*imsl_f_user_fcn_least_squares` (*float* `fcn()`, *int* `nbasis`, *int* `ndata`, *float* `xdata[]`, *float* `ydata[]`,

`IMSL_RETURN_USER`, *float* `coef[]`,

IMSL_INTERCEPT, *float* *intercept,

IMSL_SSE, *float* *ssq_err,

IMSL_WEIGHTS, *float* weights[],

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* coef[] (Output)
The coefficients are stored in the user-supplied array.

IMSL_INTERCEPT, *float* *intercept (Output)
This option adds an intercept to the model. Thus, the least-squares fit is computed using the user-supplied basis functions augmented by the constant function. The coefficient of the constant function is stored in intercept.

IMSL_SSE, *float* *ssq_err (Output)
This option returns the error sum of squares.

IMSL_WEIGHTS, *float* weights[] (Input)
This option requires the user to provide the weights.
Default: all weights equal one

IMSL_FCN_W_DATA, *float* fcn (*int* k, *float* x, *float* *data), *void* *data, (Input)
User supplied function that defines the subspace from which the least-squares fit is to be performed, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

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

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

by *M* basis functions

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

(where *M* = nbasis). In particular, the default for this function returns the coefficients *a* which minimize

$$\sum_{i=0}^{n-1} w_i \left[ f_i - \sum_{j=1}^{M} a_{j-1} F_j(x_i) \right]^2$$

where $w$ = weights, $n$ = ndata, $x$ = xdata, and $f$ = ydata.

If the optional argument IMSL_INTCERCEPT is chosen, then an intercept is placed in the model, and the coefficients $a$, returned by imsl_f_user_fcn_least_squares, minimize the error sum of squares as indicated below.

$$\sum_{i=0}^{n-1} w_i \left[ f_i - intercept - \sum_{j=1}^{M} a_{j-1} F_j(x_i) \right]^2$$

# Remarks

Note that although the system is linear, for very large problems the Chapter 8 function, imsl_f_nonlin_least_squares, might be better suited. This is because imsl_f_user_fcn_least_squares will gather the matrix entries one at a time by calls to the user-supplied function. By using the nonlinear solver and supplying the Jacobian the user can sidestep this issue and likely achieve accurate results since the nonlinear solver utilizes regularization and iterative refinement. Example 3 below demonstrates the use of the nonlinear solver imsl_f_nonlin_least_squares as an alternative to imsl_f_user_fcn_least_squares.

# Examples

## Example 1

This example fits the following two functions (indexed by δ):

1 + sin$x$ + 7 sin3$x$ + δϵ

where ε is a random uniform deviate over the range [−1, 1] and δ is 0 for the first function and 1 for the second. These functions are evaluated at 90 equally spaced points on the interval [0, 6]. Four basis functions are used: 1, sin$x$, sin2$x$, sin3$x$.

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

#define NDATA   90
                               /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

float           fcn(int n, float x);
```

```
int main()
{
   int        nbasis = 4, i, delta;
   float      ydata[NDATA], xdata[NDATA], *random, *coef;
                                /* Generate random numbers */
   imsl_random_seed_set(1234567);
   random = imsl_f_random_uniform(NDATA, 0);
                                /* Set up data */
   for(delta = 0; delta < 2; delta++) {
       for (i = 0; i < NDATA; i++) {
           xdata[i] = 6.*(float)i /((float)(NDATA-1));
           ydata[i] = F(xdata[i]) + (delta)*2.*(random[i]-.5);
       }
       coef = imsl_f_user_fcn_least_squares(fcn, nbasis, NDATA, xdata,
                                                              ydata, 0);
       printf("\nFor delta = %1d", delta);
       imsl_f_write_matrix("the computed coefficients are\n",
                            1, nbasis, coef, 0);
   }
}


float fcn(int n, float x)
{
   return (n == 1) ? 1.0 : sin((n-1)*x);
}
```

### Output

```
For delta = 0
       the computed coefficients are

        1          2          3          4
        1          1         -0          7

For delta = 1
       the computed coefficients are

        1          2          3          4
    0.979      0.998      0.096      6.839
```

## Example 2

Recall that the first example fitted the following two functions (indexed by $\delta$):

$1 + \sin x + 7 \sin 3x + \delta\epsilon$

where $\epsilon$ is a random uniform deviate over the range$[-1, 1]$, and $\delta$ is 0 for the first function and 1 for the second. These functions are evaluated at 90 equally spaced points on the interval [0, 6]. Previously, the four basis functions were used: 1, sin*x*, sin2*x*, sin3*x*. This example uses the four basis functions: sin*x*, sin2*x*, sin3*x*, sin4*x*, combined with the intercept option.

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

#define NDATA  90
                              /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

float           fcn(int n, float x);

int main()
{
   int        nbasis = 4, i, delta;
   float      ydata[NDATA], xdata[NDATA], *random, *coef, intercept;
                              /* Generate random numbers */
   imsl_random_seed_set(1234567);
   random = imsl_f_random_uniform(NDATA, 0);
                             /* Set up data */
   for(delta = 0; delta < 2; delta++){
       for (i = 0; i < NDATA; i++) {
           xdata[i] = 6.*(float)i /((float)(NDATA-1));
           ydata[i] = F(xdata[i]) + (delta)*2.*(random[i]-.5);
       }
       coef = imsl_f_user_fcn_least_squares(fcn, nbasis, NDATA, xdata,
                                                              ydata,
                                                    IMSL_INTERCEPT, &intercept,
                                                    0);
       printf("\nFor delta = %1d\n", delta);
       printf("The predicted intercept value is %10.3f\n" ,
               intercept);
       imsl_f_write_matrix("the computed coefficients are\n",
                           1, nbasis, coef, 0);
   }
}


float fcn(int n, float x)
{
   return sin(n*x);
}
```

### Output

```
For delta = 0
The predicted intercept value is      1.000
        the computed coefficients are

        1          2          3          4
        1          0          7         -0
```

```
For delta = 1
The predicted intercept value is       0.978

       the computed coefficients are

         1           2           3           4
     0.998       0.097       6.841       0.075
```

## Example 3

This example solves the same problem as Example 1, demonstrating the use of `imsl_f_nonlin_least_squares` as an alternative to `imsl_f_user_fcn_least_squares`. See the Remarks section above for a discussion of why, for some problems, `imsl_f_nonlinear_least_squares` might be better suited than `imsl_f_user_fcn_least_squares`.

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

float fcn(int n, float x);
void fcn2(int m, int n, float x[], float f[], void* data);
void fcn2jac(int m, int n, float x[], float fjac[], int fjac_col_dim,
            void *data);
/*
* The following structure type will be used to pass
* additional data to imsl_f_nonlin_least_squares() using
* the optional argument IMSL_FCN_W_DATA
*/
typedef struct {
    float *xdata;
    float *ydata;
} Problem_data;

int main()
{
#define NDATA    90
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))
    int          nbasis = 4, i, delta;
    float       *random, *coef, *coef2;
    float        ydata[NDATA], xdata[NDATA];
    Problem_data data;
    data.xdata = xdata;
    data.ydata = ydata;

    imsl_random_seed_set(1234567);
    random = imsl_f_random_uniform(NDATA, 0);
    for(delta = 0;  delta < 2;  delta++) {
        printf("\n\nFor delta = %1d", delta);
        for (i = 0; i < NDATA; i++) {
            xdata[i] = 6.*(float)i /((float)(NDATA-1));
            ydata[i] = F(xdata[i]) + (delta)*2.*(random[i]-.5);
        }

        coef = imsl_f_user_fcn_least_squares(fcn, nbasis, NDATA, xdata,
                                             ydata, 0);
        imsl_f_write_matrix(
            "   Coefficients from user_fcn_least_squares\n",
            1, nbasis,  coef, IMSL_NO_COL_LABELS, IMSL_NO_ROW_LABELS,
            IMSL_WRITE_FORMAT, "%6.3f", 0);

        coef2 = imsl_f_nonlin_least_squares(NULL, NDATA, nbasis,
            IMSL_FCN_W_DATA, fcn2, &data,
            IMSL_JACOBIAN_W_DATA, fcn2jac, &data, 0);
        imsl_f_write_matrix(
            "   Coefficients from nonlin_least_squares  \n",
```

```
            1, nbasis, coef2, IMSL_NO_COL_LABELS, IMSL_NO_ROW_LABELS,
            IMSL_WRITE_FORMAT, "%6.3f", 0);
    }
}

float fcn(int n, float x)
{
    return  (n == 1) ? 1.0 : sin((n-1)*x);
}

void fcn2(int m, int n, float x[], float f[], void *data)
{
    int i;
    float *xdata, *ydata;
    xdata = ((Problem_data*)data)->xdata;
    ydata = ((Problem_data*)data)->ydata;

    for (i=0;i<m;i++)
        f[i] = x[0] + x[1]*sin(xdata[i]) + x[2]*sin(2.0*xdata[i]) +
        x[3]*sin(3.0*xdata[i]) - ydata[i];
}

void fcn2jac(int m, int n, float x[], float fjac[], int fjac_col_dim,
            void *data)
{
    int i, j;
    float *xdata;
    xdata = ((Problem_data*)data)->xdata;

    for (i=0;i<m;i++)
        for (j=0;j<n;j++)
            fjac[i*fjac_col_dim+j] = (j==0) ? 1.0 : sin(j*xdata[i]);
}
```

### Output

```
For delta = 0
  Coefficients from user_fcn_least_squares

       1.000  1.000  0.000  7.000

  Coefficients from nonlin_least_squares

       1.000  1.000  0.000  7.000


For delta = 1
  Coefficients from user_fcn_least_squares

       0.979  0.998  0.096  6.839
```

```
Coefficients from nonlin_least_squares

     0.979   0.998   0.096   6.839
```

## Warning Errors

| | |
|---|---|
| IMSL_LINEAR_DEPENDENCE | Linear dependence of the basis functions exists. One or more components of coef are set to zero. |
| IMSL_LINEAR_DEPENDENCE_CONST | Linear dependence of the constant function and basis functions exists. One or more components of coef are set to zero. |

## Fatal Errors

| | |
|---|---|
| IMSL_NEGATIVE_WEIGHTS_2 | All weights must be greater than or equal to zero. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# spline_least_squares

Computes a least-squares spline approximation.

## Synopsis

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_least_squares (*int* ndata, *float* xdata[], *float* fdata[], *int* spline_space_dim, …, 0)

The type *Imsl_d_spline* function is imsl_d_spline_least_squares.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the least-squares problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates of the least-squares problem.

*int* spline_space_dim (Input)
The linear dimension of the spline subspace. It should be smaller than ndata and greater than or equal to order (whose default value is 4).

## Return Value

A pointer to the structure that represents the spline fit. If a fit cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_least_squares (*int* ndata, *float* xdata[], *float* fdata[], *int* spline_space_dim,

IMSL_SSE, *float* *sse_err,

IMSL_WEIGHTS, *float* weights[],

IMSL_ORDER, *int* order,

IMSL_KNOTS, *float* knots[],

IMSL_OPTIMIZE,

0)

# Optional Arguments

IMSL_SSE, *float* *sse (Output)
This option places the weighted error sum of squares in the place pointed to by sse.

IMSL_WEIGHTS, *float* weights[] (Input)
This option requires the user to provide the weights.
Default: all weights equal one.

IMSL_ORDER, *int* order (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: order = 4, (i.e., cubic splines).

IMSL_KNOTS, *float* knots[] (Input)
This option requires the user to provide the knots. The user must provide a knot sequence of length spline_space_dimension + order.
Default: an appropriate knot sequence is selected. See below for more details.

IMSL_OPTIMIZE
This option optimizes the knot locations, by attempting to minimize the least-squares error as a function of the knots. The optimal knots are available in the returned spline structure.

# Description

Let's make the identifications

$n$ = ndata

$x$ = xdata

$f$ = fdata

$m$ = spline_space_dim

$k$ = order

For convenience, we assume that the sequence $x$ is increasing, although the function does not require this.

By default, $k = 4$, and the knot sequence we select equally distributes the knots through the distinct $x_i$'s. In particular, the $m + k$ knots will be generated in $[x_1, x_n]$ with $k$ knots stacked at each of the extreme values. The interior knots will be equally spaced in the interval.

Once knots **t** and weights *w* are determined (and assuming that the option `IMSL_OPTIMIZE` is not chosen), then the function computes the spline least-squares fit to the data by minimizing over the linear coefficients $a_j$

$$\sum_{i=0}^{n-1} w_i \left[ f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right]^2$$

where the $B_j$, $j = 1, \ldots, m$ are a (B-spline) basis for the spline subspace.

The optional argument `IMSL_ORDER` allows the user to choose the order of the spline fit. The optional argument `IMSL_KNOTS` allows user specification of knots. The function `imsl_f_spline_least_squares` is based on the routine L2APPR by de Boor (1978, p. 255).

If the option `IMSL_OPTIMIZE` is chosen, then the procedure attempts to find the best placement of knots that will minimize the least-squares error to the given data by a spline of order *k* with *m* coefficients. For this problem to make sense, it is necessary that $m > k$. We then attempt to find the minimum of the functional

$$F(a,t) = \sum_{i=0}^{n-1} w_i \left[ f_i - \sum_{j=0}^{m-1} a_j B_{j,k,t}(x_i) \right]$$

The technique employed here uses the fact that for a fixed knot sequence **t** the minimization in *a* is a linear least-squares problem that can be easily solved. Thus, we can think of our objective function *F* as a function of just **t** by setting

$$G(t) = \min_{\mathrm{a}} F(a,t)$$

A Gauss-Seidel (cyclic coordinate) method is then used to reduce the value of the new objective function *G*. In addition to this local method, there is a global heuristic built into the algorithm that will be useful if the data arise from a smooth function. This heuristic is based on the routine NEWNOT of de Boor (1978, pp. 184 and 258−261).

The initial guess, $\mathbf{t}^g$, for the knot sequence is either provided by the user or is the default. This guess must be a ***valid*** knot sequence for splines of order *k* with

$$t_0^g \le \ \ldots \ \le t_{k-1}^g \le x_i \le t_m^g \le \ \ldots \ \le t_{m+k-1}^g i = 1, \ \ldots \ , M$$

with $\mathbf{t}^g$ nondecreasing, and

$$t_i^g < t_{i+k}^g \text{ for } i = 0, \ldots , m-1$$

In regard to execution speed, this function can be several orders of magnitude slower than a simple least-squares fit.

The return value for this function is a pointer of type `Imsl_f_spline`. The calling program must receive this in a pointer `Imsl_f_spline *sp`. This structure contains all the information to determine the spline (stored in B-spline form) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y*.

```
y = imsl_f_spline_value (x, sp, 0);
```

In the figure below, two cubic splines are fit to

$$\sqrt{|x|}$$

Both splines are cubics with the same `spline_space_dim` = 8. The first spline is computed with the default settings, while the second spline is computed by optimizing the knot locations using the keyword `IMSL_OPTIMIZE`.

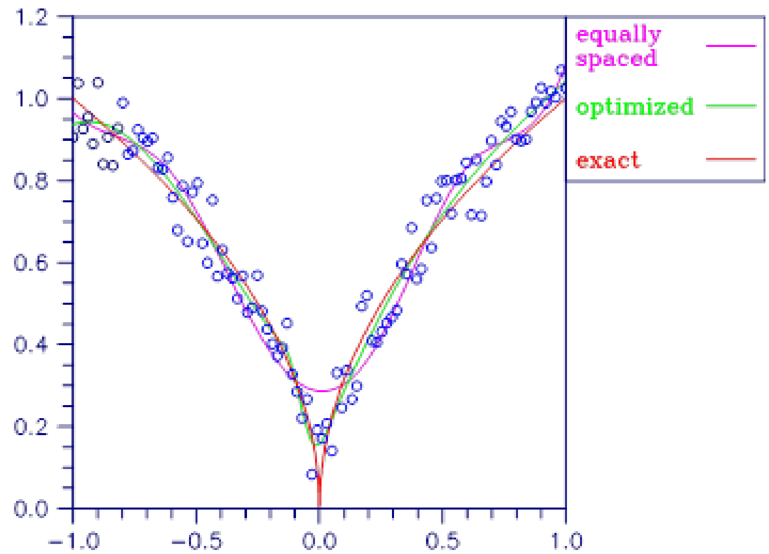



Figure 3.5 — Two Cubic Splines

# Examples

## Example 1

This example fits data generated from a trigonometric polynomial

$$1 + \sin x + 7\,\sin 3x + \epsilon$$

where **ε** is a random uniform deviate over the range [−1, 1]. The data are obtained by evaluating this function at 90 equally spaced points on the interval [0, 6]. This data is fitted with a cubic spline with 12 degrees of freedom (eight equally spaced interior knots). The error at 10 equally spaced points is printed out.

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

#define NDATA   90
                                /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))
```

```
int main()
{
   int                  i, spline_space_dim = 12;
   float                fdata[NDATA], xdata[NDATA], *random;
   Imsl_f_spline        *sp;
                                 /* Generate random numbers */
   imsl_random_seed_set(123457);
   random = imsl_f_random_uniform(NDATA, 0);
                                 /* Set up data */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = 6.*(float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]) + 2.*(random[i]-.5);
   }
   sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                          spline_space_dim, 0);
   printf("      x        error \n");
   for(i = 0; i < 10; i++) {
       float x, error;
       x = 6.*i/9.;
       error = F(x) - imsl_f_spline_value(x, sp, 0);
       printf("%10.3f %10.3f\n", x, error);
   }
}
```

### Output

```
  x       Error
0.000     -0.356
0.667     -0.004
1.333      0.434
2.000     -0.069
2.667     -0.494
3.333      0.362
4.000     -0.273
4.667     -0.247
5.333      0.303
6.000      0.578
```

## Example 2

This example continues with the first example in which we fit data generated from the trigonometric polynomial

$$1 + \sin x + 7\sin 3x + \epsilon$$

where $\varepsilon$ is random uniform deviate over the range [−1, 1]. The data is obtained by evaluating this function at 90 equally spaced points on the interval [0, 6]. This data was fitted with a cubic spline with 12 degrees of freedom (in this case, the default gives us eight equally spaced interior knots) and the error sum of squares was printed. In this example, the knot locations are optimized and the error sum of squares is printed. Then, the error at 10 equally spaced points is printed.

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

#define NDATA  90
                              /* Define function */
#define F(x)       (float)(1.+ sin(x)+7.*sin(3.0*x))

int main()
{
   int                i, spline_space_dim = 12;
   float              fdata[NDATA], xdata[NDATA], *random, sse1, sse2;
   Imsl_f_spline       *sp;
                              /* Generate random numbers */
   imsl_random_seed_set(123457);
   random = imsl_f_random_uniform(NDATA, 0);
                              /* Set up data */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = 6.*(float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]) + 2.*(random[i]-.5);
   }
   sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                                   spline_space_dim,
                                   IMSL_SSE, &sse1,
                                   0);
   sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
                                                  spline_space_dim,
                                   IMSL_OPTIMIZE,
                                   IMSL_SSE, &sse2,
                                   0);
   printf("The error sum of squares before optimizing is %10.1f\n",
           sse1);
   printf("The error sum of squares after optimizing is %10.1f\n\n", sse2);
   printf("      x        error\n");
   for(i = 0; i < 10; i++){
       float  x, error;
       x = 6.*i/9.;
       error = F(x) - imsl_f_spline_value(x, sp, 0);
       printf("%10.3f %10.3f\n", x, error);
   }
}
```

### Output

```
The error sum of squares before optimizing is      32.6
The error sum of squares after optimizing is      27.0

     x        Error
   0.000     -0.656
   0.667      0.107
```

```
1.333      0.055
2.000     -0.243
2.667     -0.063
3.333     -0.015
4.000     -0.424
4.667     -0.138
5.333      0.133
6.000      0.494
```

# Warning Errors

| | |
|---|---|
| IMSL_OPT_KNOTS_STACKED_1 | The knots found to be optimal are stacked more than order. This indicates fewer knots will produce the same error sum of squares. The knots have been separated slightly. |

# Fatal Errors

| | |
|---|---|
| IMSL_XDATA_TOO_LARGE | The array xdata must satisfy $\mathrm{xdata}_i \le \mathbf{t}_{\mathrm{ndata}}$, for $i = 1, \ldots,$ ndata. |
| IMSL_XDATA_TOO_SMALL | The array xdata must satisfy $\mathrm{xdata}_i \ge \mathbf{t}_{\mathrm{order}-1}$, for $i = 1, \ldots,$ ndata. |
| IMSL_NEGATIVE_WEIGHTS | All weights must be greater than or equal to zero. |
| IMSL_KNOT_MULTIPLICITY | Multiplicity of the knots cannot exceed the order of the spline. |
| IMSL_KNOT_NOT_INCREASING | The knots must be nondecreasing. |
| IMSL_OPT_KNOTS_STACKED_2 | The knots found to be optimal are stacked more than order. This indicates fewer knots will produce the same error sum of squares. |

# spline_2d_least_squares

Computes a two-dimensional, tensor-product spline approximant using least-squares.

## Synopsis

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_2d_least_squares (*int* num_xdata, *float* xdata[],
*int* num_ydata, *float* ydata[], *float* fdata[], *int* x_spline_space_dim,
*int* y_spline_space_dim, …, 0)

The type *Imsl_d_spline* function is imsl_d_spline_2d_least_squares.

## Required Arguments

*int* num_xdata (Input)
Number of data points in the *x* direction.

*float* xdata[] (Input)
Array with num_xdata components containing the data points in the *x* direction.

*int* num_ydata (Input)
Number of data points in the *y* direction.

*float* ydata[] (Input)
Array with num_ydata components containing the data points in the *y* direction.

*float* fdata[] (Input)
Array of size num_xdata × num_ydata containing the values to be approximated. fdata[i][j] is the value at (xdata[i], ydata[j]).

*int* x_spline_space_dim (Input)
The linear dimension of the spline subspace for the *x* variable. It should be smaller than num_xdata and greater than or equal to xorder whose default value is 4.

*int* y_spline_space_dim (Input)
The linear dimension of the spline subspace for the *y* variable. It should be smaller than num_ydata and greater than or equal to yorder whose default value is 4.

## Return Value

This is a pointer to the structure that represents the tensor-product spline interpolant. If an interpolant cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*Imsl_f_spline* `*imsl_f_spline_2d_least_squares` (*int* `num_xdata`, *float* `xdata[]`,
*int* `num_ydata`, *float* `ydata[]`, *float* `fdata[]`, *int* `x_spline_space_dim`,
*int* `y_spline_space_dim`,

`IMSL_SSE`, *float* `*sse`,

`IMSL_ORDER`, *int* `xorder`, *int* `yorder`,

`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]`,

`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim`,

`IMSL_WEIGHTS`, *float* `xweights[]`, *float* `yweights[]`,

0)

## Optional Arguments

`IMSL_SSE`, *float* `*sse` (Output)
This option places the weighted error sum of squares in the place pointed to by `sse`.

`IMSL_ORDER`, *int* `xorder`, *int* `yorder` (Input)
This option is used to communicate the order of the spline subspace.
Default: `xorder`, `yorder` = 4 i.e., tensor-product cubic splines

`IMSL_KNOTS`, *float* `xknots[]`, *float* `yknots[]` (Input)
This option requires the user to provide the knots.
Default: The default knots are equally spaced in the *x* and *y* dimensions.

`IMSL_FDATA_COL_DIM`, *int* `fdata_col_dim` (Input)
The column dimension of `fdata`.
Default: `fdata_col_dim = num_ydata`

`IMSL_WEIGHTS`, *float* `xweights[]`, *float* `yweights[]` (Input)
This option requires the user to provide the weights for the least-squares fit.
Default: all weights are equal to 1.

## Description

The `imsl_f_spline_2d_least_squares` procedure computes a tensor-product spline least-squares approximation to weighted tensor-product data. The input for this function consists of data vectors to specify the tensor-product grid for the data, two vectors with the weights (optional, the default is 1), the values of the surface on the grid, and the specification for the tensor-product spline (optional, a default is chosen). The grid is specified by the two arrays $x$ = `xdata` and $y$ = `ydata` of length $n$ = `num_xdata` and $m$ = `num_ydata`, respectively. A two-dimensional array $f$ = `fdata` contains the data values which are to be fit. The two vectors $w_x$ = `xweights` and $w_y$ = `yweights` contain the weights for the weighted least-squares problem. The information for the approximating tensor-product spline can be provided using the keywords `IMSL_ORDER` and `IMSL_KNOTS`. This information is contained in $k_x$ = `xorder`, $\mathbf{t}_x$ = `xknots`, and $N$ = `xspline_space_dim` for the spline in the first variable, and in $k_y$ = `yorder`, $\mathbf{t}_y$ = `yknots` and $M$ = `y_spline_space_dim` for the spline in the second variable.

This function computes coefficients for the tensor-product spline by solving the normal equations in tensor-product form as discussed in de Boor (1978, Chapter 17). Also see the paper by Grosse (1980).

As the computation proceeds, we obtain coefficients $c$ minimizing

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} w_x(i) w_y(j) \left[\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl} B_{kl}(x_i, y_i) - f_{ij}\right]^2$$

where the function $B_{kl}$ is the tensor-product of two B-splines of order $k_x$ and $k_y$. Specifically, we have

$$B_{kl}(x, y) = B_{k,\, k_x,\, t_x}(x) B_{l,\, k_y,\, t_y}(y)$$

The spline

$$\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl} B_{kl}(x, y)$$

and its partial derivatives can be evaluated at any point ($x, y$) using `imsl_f_spline_2d_value`.

The return value for this function is a pointer to the structure *Imsl_f_spline*. The calling program must receive this in a pointer of type *Imsl_f_spline*. This structure contains all the information to determine the spline that is computed by this procedure. For example, the following code sequence evaluates this spline (stored in the structure `sp` at ($x$, $y$) and returns the value in `v`.

```
v = imsl_f_spline_2d_value (x, y, sp, 0)
```

# Examples

## Example 1

The data for this example comes from the function $e^x \sin(x + y)$ on the rectangle $[0, 3] \times [0, 5]$. This function is sampled on a $50 \times 25$ grid. Then recover or smooth the data by using tensor-product cubic splines and least-squares fitting. The values of the function $e^x \sin(x + y)$ are printed on a $2 \times 2$ grid and compared with the values of the tensor-product spline least-squares fit.

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

```
#define NXDATA          50
#define NYDATA          25
#define OUTDATA          2
                                /* Define function */
#define F(x,y)          (float)(exp(x)*sin(x+y))

int main()
{
   int                 i, j, num_xdata, num_ydata;
   float               fdata[NXDATA][NYDATA];
   float               xdata[NXDATA], ydata[NYDATA];
   Imsl_f_spline       *sp;
                                /* Set up grid */
   for (i = 0; i < NXDATA; i++) {
       xdata[i] = 3.*(float) i / ((float)(NXDATA-1));
   }
   for (i = 0; i < NYDATA; i++) {
       ydata[i] = 5.*(float) i / ((float)(NYDATA-1));
   }
                                /* Compute function values on grid */
   for (i = 0; i < NXDATA; i++) {
       for (j = 0; j < NYDATA; j++) {
           fdata[i][j] = F(xdata[i], ydata[j]);
       }
   }
   num_xdata = NXDATA;
   num_ydata = NYDATA;
                                /* Compute tensor-product interpolant */
   sp = imsl_f_spline_2d_least_squares(num_xdata, &xdata[0], num_ydata,
                                        &ydata[0], &fdata[0][0], 5, 7, 0);
                                /* Print results */
   printf("    x      y       F(x, y)   Spline Fit  Abs. Error\n");
   for (i = 0; i < OUTDATA; i++) {
       x = (float)i / (float)(OUTDATA);
       for (j = 0; j < OUTDATA; j++) {
           y = (float)j / (float)(OUTDATA);
           z = imsl_f_spline_2d_value(x, y, sp, 0);
           printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                                  x, y, F(x, y), z, fabs(F(x,y)-z));
       }
     }
   }
```

## Output

```
    x       y        F(x, y)   Spline Fit  Abs. Error
  0.000   0.000       0.000      -0.020      0.0204
  0.000   0.500       0.479       0.500      0.0208
```

```
  0.500   0.000       0.790       0.816       0.0253
  0.500   0.500       1.387       1.384       0.0031
```

### Example 2

The same data is used as in the previous example. Optional argument `IMSL_SSE` is used to return the error sum of squares.

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

#define NXDATA          50
#define NYDATA          25
#define OUTDATA          2
                                /* Define function */
#define F(x,y)          (float)(exp(x)*sin(x+y))

int main()
{
   int                  i, j, num_xdata, num_ydata;
   float                fdata[NXDATA][NYDATA];
   float                xdata[NXDATA], ydata[NYDATA], sse, x, y, z;
   Imsl_f_spline        *sp;
                                /* Set up grid */
   for (i = 0; i < NXDATA; i++) {
       xdata[i] = 3.*(float) i / ((float) (NXDATA - 1));
   }
   for (i = 0; i < NYDATA; i++) {
       ydata[i] = 5.*(float) i / ((float) (NYDATA - 1));
   }
                                /* Compute function values on grid */
   for (i = 0; i < NXDATA; i++) {
       for (j = 0; j < NYDATA; j++) {
           fdata[i][j] = F(xdata[i], ydata[j]);
       }
   }
   num_xdata = NXDATA;
   num_ydata = NYDATA;
                                /* Compute tensor-product interpolant */
   sp = imsl_f_spline_2d_least_squares(num_xdata, &xdata[0], num_ydata,
                                    &ydata[0], &fdata[0][0], 5, 7,
                                    IMSL_SSE, &sse, 0);

                                /* Print results */
   printf("The error sum of squares is %10.3f\n\n", sse);
   printf("    x       y       F(x, y)    Spline Fit   Abs. Error\n");
   for (i = 0; i < OUTDATA; i++) {
       x = (float) i / (float) (OUTDATA);
       for (j = 0; j < OUTDATA; j++) {
```

```
            y = (float) j / (float) (OUTDATA);
            z = imsl_f_spline_2d_value(x, y, sp, 0);
            printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                                  x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

### Output

```
The error sum of squares is     3.753

    x       y        F(x, y)    Spline Fit  Abs. Error
  0.000   0.000      0.000        -0.020      0.0204
  0.000   0.500      0.479         0.500      0.0208
  0.500   0.000      0.790         0.816      0.0253
  0.500   0.500      1.387         1.384      0.0031
```

## Warning Errors

| | |
|---|---|
| `IMSL_ILL_COND_LSQ_PROB` | The least-squares matrix is ill-conditioned. The solution might not be accurate. |
| `IMSL_SPLINE_LOW_ACCURACY` | There may be less than one digit of accuracy in the least-squares fit. Try using a higher precision if possible. |

## Fatal Errors

| | |
|---|---|
| `IMSL_KNOT_MULTIPLICITY` | Multiplicity of the knots cannot exceed the order of the spline. |
| `IMSL_KNOT_NOT_INCREASING` | The knots must be nondecreasing. |
| `IMSL_SPLINE_LRGST_ELEMNT` | The data arrays `xdata` and `ydata` must satisfy $\mathrm{data}_i \le \mathbf{t}_{\mathrm{spline\_space\_dim}}$, for $i = 1, \ldots,$ `num_data`. |
| `IMSL_SPLINE_SMLST_ELEMNT` | The data arrays `xdata` and `ydata` must satisfy $\mathrm{data}_i \ge \mathbf{t}_{\mathrm{order}-1}$, for $i = 1, \ldots,$ `num_data`. |
| `IMSL_NEGATIVE_WEIGHTS` | All weights must be greater than or equal to zero. |
| `IMSL_DATA_DECREASING` | The `xdata` values must be nondecreasing. |

# cub_spline_smooth

Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.

## Synopsis

*#include* <imsl.h>

*Imsl_f_ppoly* *imsl_f_cub_spline_smooth (*int* ndata, *float* xdata[], *float* fdata[], …, 0)

The type *Imsl_d_ppoly* function is imsl_d_cub_spline_smooth.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates of the problem.

## Return Value

A pointer to the structure that represents the cubic spline. If a smoothed cubic spline cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>s

*Imsl_f_ppoly* *imsl_f_cub_spline_smooth (*int* ndata, *float* xdata[], *float* fdata[],

IMSL_WEIGHTS, *float* weights[],

IMSL_SMOOTHING_PAR, *float* sigma,

0)

## Optional Arguments

`IMSL_WEIGHTS`, *float* `weights[]` (Input)
This option requires the user to provide the weights.
Default: all weights are equal to 1.

`IMSL_SMOOTHING_PAR`, *float* `sigma` (Input)
This option sets the smoothing parameter **σ** = `sigma` explicitly.

## Description

The function `imsl_f_cub_spline_smooth` is designed to produce a $C^2$ cubic spline approximation to a data set in which the function values are noisy. This spline is called a ***smoothing spline***.

Consider first the situation when the optional argument `IMSL_SMOOTHING_PAR` is selected. Then, a natural cubic spline with knots at all the data abscissas ***x*** = `xdata` is computed, but it does ***not*** interpolate the data ($x_i$, $f_i$). The smoothing spline ***s*** is the unique $C^2$ function which minimizes

$$\int_a^b s''(x)^2 dx$$

subject to the constraint

$$\sum_{i=0}^{n-1} |\left(s\left(x_i\right) - f_i\right) w_i|^2 \le \sigma$$

where ***w*** = `weights`, **σ** = `sigma` is the smoothing parameter, and ***n*** = `ndata`.

Recommended values for **σ** depend on the weights ***w***. If an estimate for the standard deviation of the error in the value $f_i$ is available, then $w_i$ should be set to the inverse of this value; and the smoothing parameter **σ** should be chosen in the confidence interval corresponding to the left side of the above inequality. That is,

$$n - \sqrt{2n} \le \sigma \le n + \sqrt{2n}$$

The function `imsl_f_cub_spline_smooth` is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pp. 235−243).

The default for this function chooses the smoothing parameter **σ** by a statistical technique called ***cross-validation***. For more information on this topic, refer to Craven and Wahba (1979).

The return value for this function is a pointer to the structure *Imsl_f_ppoly*. The calling program must receive this in a pointer *Imsl_f_ppoly* `*pp`. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this procedure. For example, the following code sequence evaluates this spline at ***x*** and returns the value in `y`.

```
y = imsl_f_cub_spline_value (x, pp, 0);
```

# Examples

## Example 1

In this example, function values are contaminated by adding a small "random" amount to the correct values. The function `imsl_f_cub_spline_smooth` is used to approximate the original, uncontaminated data.

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

#define NDATA   90
                              /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

int main()
{
   int                  i;
   float                fdata[NDATA], xdata[NDATA], *random;
   Imsl_f_ppoly         *pp;
                              /* Generate random numbers */
   imsl_random_seed_set(123457);
   random = imsl_f_random_uniform(NDATA, 0);
                              /* Set up data */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = 6.*(float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]) + .5*(random[i]-.5);
   }
   pp = imsl_f_cub_spline_smooth(NDATA, xdata, fdata, 0);
   printf("     x        error \n");
   for(i = 0; i < 10; i++){
       float x, error;
       x = 6.*i/9.;
       error = F(x) - imsl_f_cub_spline_value(x, pp, 0);
       printf("%10.3f %10.3f\n", x, error);
   }
}
```

### Output

```
  x       Error
0.000     -0.201
0.667      0.070
1.333     -0.008
2.000     -0.058
2.667     -0.025
3.333      0.076
4.000     -0.002
4.667     -0.008
5.333      0.045
```

```
6.000      0.276
```

## Example 2

Recall that in the first example, function values are contaminated by adding a small “random” amount to the correct values. Then, `imsl_f_cub_spline_smooth` is used to approximate the original, uncontaminated data. This example explicitly inputs the value of the smoothing parameter to be 5.

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

#define NDATA   90
                              /* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

int main()
{
   int                  i;
   float                fdata[NDATA], xdata[NDATA], *random;
   Imsl_f_ppoly         *pp;
                              /* Generate random numbers */
   imsl_random_seed_set(123457);
   random = imsl_f_random_uniform(NDATA, 0);
                              /* Set up data */
   for (i = 0; i < NDATA; i++) {
       xdata[i] = 6.*(float)i /((float)(NDATA-1));
       fdata[i] = F(xdata[i]) + .5*(random[i]-.5);
   }
   pp = imsl_f_cub_spline_smooth(NDATA, xdata, fdata,
                                IMSL_SMOOTHING_PAR, 5.0,
                                0);
   printf("      x        error \n");
   for(i = 0; i < 10; i++){
       float x, error;
       x = 6.*i/9.;
       error = F(x) - imsl_f_cub_spline_value(x, pp, 0);
       printf("%10.3f %10.3f\n", x, error);
   }
}
```

### Output

```
x        Error
0.000     -0.593
0.667      0.230
1.333     -0.116
2.000     -0.106
2.667      0.176
3.333     -0.071
```

```
4.000      -0.171
4.667       0.196
5.333      -0.036
6.000       0.971
```

# Warning Errors

| | |
|---|---|
| `IMSL_MAX_ITERATIONS_REACHED` | The maximum number of iterations has been reached. The best approximation is returned. |

# Fatal Errors

| | |
|---|---|
| `IMSL_DUPLICATE_XDATA_VALUES` | The xdata values must be distinct. |
| `IMSL_NEGATIVE_WEIGHTS` | All weights must be greater than or equal to zero. |

# spline_lsq_constrained

Computes a least-squares constrained spline approximation.

## Synopsis

*#include* <imsl.h>

*Imsl_f_spline* *imsl_f_spline_lsq_constrained (*int* ndata, *float* xdata[], *float* fdata[], *int* spline_space_dim, *int* num_con_pts, *f_constraint_struct* constraints[], …, 0)

The type *Imsl_d_spline* function is imsl_d_spline_lsq_constrained.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the least-squares problem.

*float* fdata[] (Input)
Array with ndata components containing the ordinates of the least-squares problem.

*int* spline_space_dim (Input)
The linear dimension of the spline subspace. It should be smaller than ndata and greater than or equal to order (whose default value is 4).

*int* num_con_pts (Input)
The number of points in the vector constraints.

*f_constraint_struct* constraints[] (Input)
A structure containing the abscissas at which the fit is to be constrained, the derivative of the spline that is to be constrained, the type of constraints, and any lower or upper limits. A description of the structure fields follows:

| Field | Description |
|---|---|
| xval | point at which fit is constrained |
| der | derivative value of the spline to be constrained |
| type | types of the general constraints |
| bl | lower limit of the general constraints |
| bu | upper limit of the general constraints |

**Notes:** If you want to constrain the integral of the spline over the closed interval (*c*, *d*), then set `constraints[i].der` = `constraints[i+1].der` = $-1$ and `constraints[i].xval` = *c* and `constraints[i+1].xval` = *d*. For consistency, insist that `constraints[i].type` = `constraints[i+1].type` $\geq 0$ and $c \leq d$. Note that every `der` must be at least $-1$.

| constraints [i].type | *i*-th constraint |
|---|---|
| 1 | $bl_i = f^{(d_i)}(x_i)$ |
| 2 | $f^{(d_i)}(x_i) \leq bu_i$ |
| 3 | $f^{(d_i)}(x_i) \geq bl_i$ |
| 4 | $bl_i \leq f^{(d_i)}(x_i) \leq bu_i$ |
| 5 | $bl_i = \int_c^d f(t)\,dt$ |
| 6 | $\int_c^d f(t)\,dt \leq bu_i$ |
| 7 | $\int_c^d f(t)\,dt \geq bl_i$ |
| 8 | $bl_i \leq \int_c^d f(t)\,dt \leq bu_i$ |
| 20 | periodic end conditions |
| 99 | disregard this constraint |

In order to have two point constraints, must have

`constraints[i].type` = `constraints[i+1].type`

| constraints [i]. type | *i*-th constraint |
|---|---|
| 9 | $bl_i = f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1})$ |
| 10 | $f^{(d_i)}(x_i) \leq bu_i$ |
| 11 | $f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \geq bl_i$ |
| 12 | $bl_i \leq f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \leq bu_i$ |

## Return Value

A pointer to the structure that represents the spline fit. If a fit cannot be computed, then `NULL` is returned. To release this space, use `imsl_free`.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*Imsl_f_spline* `*imsl_f_spline_lsq_constrained` (*int* `ndata`, *float* `xdata[]`, *float* `fdata[]`, *int* `spline_space_dim`, *int* `num_con_pts`, *f_constraint_struct* `constraints[]`,

`IMSL_NHARD`, *int* `nhard`,

`IMSL_WEIGHTS`, *float* `weights[]`,

`IMSL_ORDER`, *int* `order`,

`IMSL_KNOTS`, *float* `knots[]`,

0)

## Optional Arguments

`IMSL_NHARD`, *int* `nhard` (Input)
The argument `nhard` is the number of entries of constraints involved in the "hard" constraints. Note that 0 ≤ `nhard` ≤ `num_con_pts`. The default, `nhard` = 0, always results in a fit, while setting `nhard` = `num_con_pts` forces all constraints to be met. The "hard" constraints must be met, or else the function signals failure. The "soft" constraints need not be satisfied, but there will be an attempt to satisfy the "soft" constraints. The constraints must be listed in terms of priority with the most important constraints first. Thus, all of the "hard" constraints must precede the "soft" constraints. If infeasibility is detected among the "soft" constraints, we satisfy, in order, as many of the "soft" constraints as possible.
Default: `nhard` = 0

`IMSL_WEIGHTS`, *float* `weights[]` (Input)
This option requires the user to provide the weights.
Default: all weights equal one

`IMSL_ORDER`, *int* `order` (Input)
The order of the spline subspace for which the knots are desired. This option is used to communicate the order of the spline subspace.
Default: `order` = 4 (i.e., cubic splines)

`IMSL_KNOTS`, *float* `knots[]` (Input)

This option requires the user to provide the knots. The user must provide a knot sequence of length `spline_space_dimension + order`.

Default: an appropriate knot sequence is selected. See below for more details.

## Description

The function `imsl_f_spline_lsq_constrained` produces a constrained, weighted least-squares fit to data from a spline subspace. Constraints involving one point, two points, or integrals over an interval are allowed. The types of constraints supported by the functions are of four types:

| | |
|---|---|
| $E_p[f]$ | $= f^{(j_p)}(y_p)$ |
| *Or* | $= f^{(j_p)}(y_p) - f^{(j_{p+1})}(y_{p+1})$ |
| *Or* | $= \int_{y_p}^{y_{p+1}} f(t)\,dt$ |
| *Or* | = periodic end conditions |

An interval, $I_p$ (which may be a point, a finite interval, or a semi-infinite interval), is associated with each of these constraints.

The input for this function consists of several items; first, the data set ($x_i$, $f_i$) for $i = 1, \ldots N$ (where $N$ = `NDATA`), that is the data which is to be fit. Second, we have the weights to be used in the least-squares fit ($w$ = `WEIGHT`, defaulting to 1). The vector `constraints` contains the abscissas of the points involved in specifying the constraints, as well as information relating the type of constraints and the constraint interval.

Let $n_f$ denote the number of feasible constraints as described above. Then, the function solved the problem

$$\sum_{i=1}^{n} \left| f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right|^2 w_i$$

subject to

$$E_p\left[\sum_{j=1}^{m} a_j B_j\right] \in I_p \quad p = 1, \ldots, n_f$$

This linearly constrained least-squares problem is treated as a quadratic program and is solved by invoking the function `imsl_f_quadratic_prog` (See Chapter 8, “Optimization”)

The choice of weights depends on the data uncertainty in the problem. In some cases, there is a natural choice for the weights based on the estimates of errors in the data points.

Determining feasibility of linear constraints is a numerically sensitive task. If you encounter difficulties, a quick fix would be to widen the constraint intervals $I_p$.

# Examples

## Example 1

This is a simple application of `imsl_f_lsq_constrained`. Data is generated from the function

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

and contaminated with random noise and fit with cubic splines. The function is increasing, so least-squares fit should also be increasing. This is not the case for the unconstrained least-squares fit generated by `imsl_f_spline_least_squares`. Then, the derivative is forced to be greater than 0 at `num_con_pts` = 15 equally spaced points and `imsl_f_lsq_constrained` is called. The resulting curve is monotone. The error is printed for the two fits averaged over 100 equally spaced points.

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

#define MXKORD  4
#define MXNCOF 20
#define MXNDAT 51
#define MXNXVL 15

int main()
{
   f_constraint_struct constraint[MXNXVL];
   int  i, korder, ncoef, ndata, nxval;
   float *noise, errlsq, errnft, grdsiz, x;
   float fdata[MXNDAT], xdata[MXNDAT];
   Imsl_f_spline *sp, *spls;


#define F1(x)   (float)(.5*(x) + sin( .5*(x) ))

   korder = 4;
   ndata = 15;
   nxval = 15;
   ncoef = 8;
   /*
    * Compute original xdata and fdata with random noise.
    */
   imsl_random_seed_set (234579);
   noise = imsl_f_random_uniform (ndata, 0);
   grdsiz = 10.0;
   for (i = 0; i < ndata; i++) {
       xdata[i] = grdsiz * ((float) (i) / (float) (ndata - 1));
       fdata[i] = F1 (xdata[i]) + (noise[i] - .5);
   }
```

```
    /* Compute least-squares fit. */

    spls = imsl_f_spline_least_squares (ndata, xdata, fdata, ncoef, 0);
    /*
     * Construct the constraints.
     */
    for (i = 0; i < nxval; i++) {
        constraint[i].xval = grdsiz * (float)(i) / (float)(nxval - 1);
        constraint[i].type = 3;
        constraint[i].der = 1;
        constraint[i].bl = 0.0;
    }
    /* Compute constrained least-squares fit. */
    sp = imsl_f_spline_lsq_constrained (ndata, xdata, fdata, ncoef,
               nxval, constraint, 0);
    /*
     * Compute the average error of 100 points in the interval.
     */
    errlsq = 0.0;
    errnft = 0.0;
    for (i = 0; i < 100; i++) {
        x = grdsiz * (float) (i) / 99.0;
        errnft += fabs (F1 (x) - imsl_f_spline_value(x,sp,0));
        errlsq += fabs (F1 (x) - imsl_f_spline_value(x,spls,0));
    }
    /* Print results */
    printf (" Average error with spline_least_squares fit:   %8.5f\n",
              errlsq / 100.0);
    printf (" Average error with spline_lsq_constrained fit: %8.5f\n",
              errnft / 100.0);
}
```

### Output

```
Average error with spline_least_squares fit:    0.20250
Average error with spline_lsq_constrained fit:  0.14334
```

## Example 2

Now, try to recover the function

$$\frac{1}{1+x^4}$$

from noisy data. First, try the unconstrained least-squares fit using `imsl_f_spline_least_squares`. Finding that fit somewhat unsatisfactory, several constraints are applied using `imsl_f_spline_lsq_constrained`. First, notice that the unconstrained fit oscillates through the true function at both ends of the interval. This is common for flat data. To remove this oscillation, the cubic spline is constrained to have zero second derivative at the first and last four knots. This forces the cubic spline to reduce to a linear polynomial on the first and last three knot intervals. In addition, the fit is constrained (called *s*) as follows:

$$s(-7) \geq 0$$

$$\int_{-7}^{7} s(x)\,dx \leq 2.3$$

$$s(-7) = s(7)$$

Notice that the last constraint was generated using the periodic option (requiring only the ***zero***-th derivative to be periodic). The error is printed for the two fits averaged over 100 equally spaced points.

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

#define KORDER  4
#define NDATA  51
#define NXVAL  12
#define NCOEF  13

int main()
{
   f_constraint_struct constraint[NXVAL];
   int  i;
   float *noise, errlsq, errnft, grdsiz, x;
   float fdata[NDATA], xdata[NDATA], xknot[NDATA+KORDER];
   Imsl_f_spline *sp, *spls;


#define F1(x)   (float)(1.0/(1.0+x*x*x*x))

    /* Compute original xdata and fdata with random noise */

   imsl_random_seed_set (234579);
   noise = imsl_f_random_uniform (NDATA, 0);
   grdsiz = 14.0;
   for (i = 0; i < NDATA; i++) {
       xdata[i] = grdsiz * ((float)(i)/(float)(NDATA - 1))
                  - grdsiz/2.0;
       fdata[i] = F1 (xdata[i]) + 0.125*(noise[i] - .5);
   }
/* Generate knots. */
   for (i = 0; i < NCOEF-KORDER+2; i++) {
       xknot[i+KORDER-1] = grdsiz * ((float)(i)/
                           (float)(NCOEF-KORDER+1)) - grdsiz/2.0;
   }
   for (i = 0; i < KORDER - 1; i++) {
       xknot[i] = xknot[KORDER-1];
       xknot[i+NCOEF+1] = xknot[NCOEF];
   }

   /* Compute spline_least_squares fit */

   spls = imsl_f_spline_least_squares (NDATA, xdata, fdata, NCOEF,
            IMSL_KNOTS, xknot, 0);
```

```
     /* Construct the constraints for CONFT */

    for (i = 0; i < 4; i++) {
        constraint[i].xval = xknot[KORDER+i-1];
        constraint[i+4].xval = xknot[NCOEF-3+i];
        constraint[i].type = 1;
        constraint[i+4].type = 1;
        constraint[i].der = 2;
        constraint[i+4].der = 2;
        constraint[i].bl = 0.0;
        constraint[i+4].bl = 0.0;
    }
    constraint[8].xval = -7.0;
    constraint[8].type = 3;
    constraint[8].der = 0;
    constraint[8].bl = 0.0;

    constraint[9].xval = -7.0;
    constraint[9].type = 6;
    constraint[9].bu = 2.3;

    constraint[10].xval = 7.0;
    constraint[10].type = 6;
    constraint[10].bu = 2.3;

    constraint[11].xval = -7.0;
    constraint[11].type = 20;
    constraint[11].der = 0;

    sp = imsl_f_spline_lsq_constrained (NDATA, xdata, fdata, NCOEF,
                NXVAL, constraint, IMSL_KNOTS, xknot, 0);

    /* Compute the average error of 100 points in the interval */

    errlsq = 0.0;
    errnft = 0.0;
    for (i = 0; i < 100; i++) {
        x = grdsiz * (float) (i) / 99.0 - grdsiz/2.0;
        errnft += fabs (F1 (x) - imsl_f_spline_value(x,sp,0));
        errlsq += fabs (F1 (x) - imsl_f_spline_value(x,spls,0));
    }
    /* Print results */
    printf (" Average error with BSLSQ fit: %8.5f\n",
              errlsq / 100.0);
    printf (" Average error with CONFT fit: %8.5f\n",
              errnft / 100.0);
}
```

## Output

```
Average error with BSLSQ fit:  0.01783
Average error with CONFT fit:  0.01339
```

# smooth_1d_data

Smooth one-dimensional data by error detection.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_smooth_1d_data (*int* ndata, *float* xdata[], *float* fdata[], …, 0)

The type *double* function is imsl_d_smooth_1d_data.

## Required Arguments

*int* ndata (Input)
Number of data points.

*float* xdata[] (Input)
Array with ndata components containing the abscissas of the data points.

*float* ydata[] (Input)
Array with ndata components containing the ordinates of the data points.

## Return Value

A pointer to the vector of length ndata containing the smoothed data.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_smooth_1d_data (*int* ndata, *float* xdata[], *float* fdata[],

IMSL_RETURN_USER, *float* sdata[],

IMSL_ITMAX, *int* itmax,

IMSL_DISTANCE, *float* dis,

IMSL_STOPPING_CRITERION, *float* sc,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* sdata[] (Output)
The smoothed data is stored in the user-supplied array.

IMSL_ITMAX, *int* itmax (Input)
The maximum number of iterations allowed.
Default: itmax = 500

IMSL_DISTANCE, *float* dis (Input)
Proportion of the distance the ordinate in error is moved to its interpolating curve. It must be in the range 0.0 to 1.0.
Default: dis = 1.0

IMSL_STOPPING_CRITERION, *float* sc (Input)
The stopping criterion. sc should be greater than or equal to zero.
Default: sc = 0.0

## Description

The function imsl_f_smooth_1d_data is designed to smooth a data set that is mildly contaminated with isolated errors. In general, the routine will not work well if more than 25% of the data points are in error. The routine imsl_f_smooth_1d_data is based on an algorithm of Guerra and Tapia (1974).

Setting ndata = $n$, ydata = $f$, sdata = $s$ and xdata = $x$, the algorithm proceeds as follows. Although the user need not input an ordered xdata sequence, we will assume that $x$ is increasing for simplicity. The algorithm first sorts the xdata values into an increasing sequence and then continues. A cubic spline interpolant is computed for each of the 6-point data sets (initially setting $s = f$)

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

where $i = 4, \ldots, n - 3$. For each $i$ the interpolant, which we will call $S_i$, is compared with the current value of $s_i$, and a 'point energy' is computed as

$$pe_i = S_i(x_i) - s_i$$

Setting sc = $sc$, the algorithm terminates either if itmax iterations have taken place or if

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

If the above inequality is violated for any $i$, then we update the $i$-th element of $s$ by setting $s_i = s_i + d(pe_i)$, where $d$ = dis. Note that neither the first three nor the last three data points are changed. Thus, if these points are inaccurate, care must be taken to interpret the results.

The choice of the parameters $d$, $sc$ and itmax are crucial to the successful usage of this subroutine. If the user has specific information about the extent of the contamination, then he should choose the parameters as follows: $d = 1$, $sc = 0$ and itmax to be the number of data points in error. On the other hand, if no such specific information is available, then choose $d = .5$, itmax $\leq 2n$, and

$$sc = .5\frac{\max s - \min s}{\left(x_n - x_1\right)}$$

In any case, we would encourage the user to experiment with these values.

## Example

We take 91 uniform samples from the function $5 + (5 + t^2 \sin t)/t$ on the interval [1, 10]. Then, we contaminate 10 of the samples and try to recover the original function values.

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

#define NDATA 91
#define F(X) (X*X*sin((double)(X))+5.0)/X + 5.0

int main()
{
 int i, maxit;
 int isub[10] = {5, 16, 25, 33, 41, 48, 55, 61, 74, 82};
 float dis, fdata[NDATA], sc, *sdata=NULL;
 float xdata[NDATA], s_user[NDATA];
 float rnoise[10] = {2.5, -3., -2., 2.5, 3.,
                     -2., -2.5, 2., -2., 3.};

 /* Example 1: No specific information available. */
 dis = .5;
 sc = .56;
 maxit = 182;

 /* Set values for xdata and fdata. */
 xdata[0] = 1.;
 fdata[0] = F(xdata[0]);
 for (i=1;i<NDATA;i++) {
   xdata[i] = xdata[i-1]+.1;
   fdata[i] = F(xdata[i]);
 }

 /* Contaminate the data. */
 for (i=0;i<10;i++) fdata[isub[i]] += rnoise[i];

 /* Smooth the data. */
 sdata = imsl_f_smooth_1d_data(NDATA, xdata, fdata,
                               IMSL_DISTANCE, dis,
                               IMSL_STOPPING_CRITERION, sc,
                               IMSL_ITMAX, maxit,
                               0);
```

```
    /* Output the result. */
    printf("Case A - No specific information available. \n");
    printf("  F(X)       F(X)+noise        sdata\n");

    for (i=0;i<10;i++) printf("%7.3f\t%15.3f\t%15.3f\n",
                              F(xdata[isub[i]]),
                              fdata[isub[i]],
                              sdata[isub[i]]);

    /* Example 2: No specific information is available. */
    dis = 1.0;
    sc = 0.0;
    maxit = 10;

    /*
     * A warning message is produced because the maximum
     * number of iterations is reached.
     */

    /* Smooth the data. */
   sdata = imsl_f_smooth_1d_data(NDATA, xdata, fdata,
                                IMSL_DISTANCE, dis,
                                IMSL_STOPPING_CRITERION, sc,
                                IMSL_ITMAX, maxit,
                                IMSL_RETURN_USER, s_user,
                                0);

    /* Output the result. */
    printf("Case B - Specific information available. \n");
    printf("  F(X)       F(X)+noise        sdata\n");

    for (i=0;i<10;i++) printf("%7.3f\t%15.3f\t%15.3f\n",
                              F(xdata[isub[i]]),
                              fdata[isub[i]],
                              s_user[isub[i]]);
}
```

## Output

```
Case A - No specific information available.
  F(X)        F(X)+noise         sdata
 9.830          12.330           9.870
 8.263           5.263           8.215
 5.201           3.201           5.168
 2.223           4.723           2.264
 1.259           4.259           1.308
 3.167           1.167           3.138
 7.167           4.667           7.131
10.880          12.880          10.909
12.774          10.774          12.708
```

```
 7.594         10.594          7.639

*** WARNING Error IMSL_ITMAX_EXCEEDED from imsl_f_smooth_1d_data.
*** Maximum number of iterations limit "itmax" = 10 exceeded.
*** The best answer found is returned.

Case B - Specific information available.
  F(X)       F(X)+noise        sdata
 9.830         12.330          9.831
 8.263          5.263          8.262
 5.201          3.201          5.199
 2.223          4.723          2.225
 1.259          4.259          1.261
 3.167          1.167          3.170
 7.167          4.667          7.170
10.880         12.880         10.878
12.774         10.774         12.770
 7.594         10.594          7.592
```

# scattered_2d_interp

Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

## Synopsis

*#include* <imsl.h>

*float* `*imsl_f_scattered_2d_interp` (*int* `ndata`, *float* `xydata[]`, *float* `fdata[]`, *int* `nx_out`, *int* `ny_out`, *float* `x_out[]`, *float* `y_out[]`, …, 0)

The type ***double*** function is `imsl_d_scattered_2d_interp`.

## Required Arguments

*int* `ndata` (Input)
Number of data points.

*float* `xydata[]` (Input)
Array with `ndata*2` components containing the data points for the interpolation problem. The *i*-th data point ($x_i$, $y_i$) is stored consecutively in the 2*i* and 2*i* + 1 positions of `xydata`.

*float* `fdata[]` (Input)
Array of size `ndata` containing the values to be interpolated.

*int* `nx_out` (Input)
Number of data points in the *x* direction for the output grid.

*int* `ny_out` (Input)
Number of data points in the *y* direction for the output grid.

*float* `x_out[]` (Input)
Array of length `nx_out` specifying the *x* values for the output grid. It must be strictly increasing.

*float* `y_out[]` (Input)
Array of length `ny_out` specifying the *y* values for the output grid. It must be strictly increasing.s

## Return Value

A pointer to the `nx_out` × `ny_out` grid of values of the interpolant. If no answer can be computed, then `NULL` is returned. To release this space, use `imsl_free`.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_scattered_2d_interp (*int* ndata, *float* xydata[], *float* fdata[], *int* nx_out, *int* ny_out, *float* x_out[], *float* y_out[],

IMSL_RETURN_USER, *float* surface[],

IMSL_SUR_COL_DIM, *int* surface_col_dim,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* surface[] (Output)
This option allows the user to provide his own space for the result. In this case, the answer will be returned in surface.

IMSL_SUR_COL_DIM, *int* surface_col_dim (Input)
This option requires the user to provide the column dimension of the two-dimensional array surface.
Default: surface_col_dim = ny_out

## Description

The function imsl_f_scattered_2d_interp computes a $C^1$ interpolant to scattered data in the plane. Given the data points

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

in $\mathbf{R}^3$ where $n$ = ndata, imsl_f_scattered_2d_interp returns the values of the interpolant $s$ on the user-specified grid. The computation of $s$ is as follows: First the Delaunay triangulation of the points

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

is computed. On each triangle $T$ in this triangulation, $s$ has the form

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

Thus, $s$ is a bivariate quintic polynomial on each triangle of the triangulation. In addition, we have

$$s\left(x_i,y_i\right) = f_i \quad \text{for} \quad i = 0,\, \ldots\, ,n-1$$

and $s$ is continuously differentiable across the boundaries of neighboring triangles. These conditions do not exhaust the freedom implied by the above representation. This additional freedom is exploited in an attempt to produce an interpolant that is faithful to the global shape properties implied by the data. For more information on this procedure, refer to the article by Akima (1978). The output grid is specified by the two integer variables `nx_out` and `ny_out` that represent the number of grid points in the first (second) variable and by two real vectors that represent the first (second) coordinates of the grid.

# Examples

## Example 1

In this example, the interpolant to the linear function (3 + 7*x* + 2*y*) is computed from 20 data points equally spaced on the circle of radius 3. Then the values are printed on a 3 ×3 grid.

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

#define NDATA          20
#define OUTDATA         3
                              /* Define function */
#define F(x,y)         (float)(3.+7.*x+2.*y)

#define SURF(I,J)      surf[(J) +(I)*OUTDATA]

int main()
{
   int                i, j;
   float              fdata[NDATA], xydata[2*NDATA], *surf;
   float              x, y, z, x_out[OUTDATA], y_out[OUTDATA], pi;

   pi = imsl_f_constant("pi", 0);
                              /* Set up output grid */
   for (i = 0; i < OUTDATA; i++) {
       x_out[i] = y_out[i] = (float) i / ((float) (OUTDATA - 1));
   }
   for (i = 0; i < 2*NDATA; i += 2) {
       xydata[i]   = 3.*cos(pi*i/NDATA);
       xydata[i+1] = 3.*sin(pi*i/NDATA);
       fdata[i/2] = F(xydata[i], xydata[i+1]);
   }
                              /* Compute scattered data interpolant */
   surf = imsl_f_scattered_2d_interp (NDATA, xydata, fdata, OUTDATA,
                                      OUTDATA, x_out, y_out, 0);
                              /* Print results */
   printf("    x      y       F(x, y)   Interpolant   Error\n");
   for (i = 0; i < OUTDATA; i++) {
       for (j = 0; j < OUTDATA; j++) {
           x = x_out[i];
           y = y_out[j];
           z = SURF(i,j);
           printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
```

```
                                     x, y, F(x,y), z, fabs(F(x,y)-z));
        }

    }
}
```

### Output

```
   x       y      F(x, y)   Interpolant   Error
0.000   0.000      3.000       3.000       0.0000
0.000   0.500      4.000       4.000       0.0000
0.000   1.000      5.000       5.000       0.0000
0.500   0.000      6.500       6.500       0.0000
0.500   0.500      7.500       7.500       0.0000
0.500   1.000      8.500       8.500       0.0000
1.000   0.000     10.000      10.000       0.0000
1.000   0.500     11.000      11.000       0.0000
1.000   1.000     12.000      12.000       0.0000
```

## Example 2

Recall that in the first example, the interpolant to the linear function $3 + 7x + 2y$ is computed from 20 data points equally spaced on the circle of radius 3. We then print the values on a $3 \times 3$ grid. This example used the optional arguments to indicate that the answer is stored noncontiguously in a two-dimensional array `surf` with column dimension equal to 11.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>
#define NDATA          20
#define OUTDATA         3
#define COLDIM         11
                               /* Define function */
#define F(x,y)         (float)(3.+7.*x+2.*y)

int main()
{
   int                 i, j;
   float               fdata[NDATA], xydata[2*NDATA];
   float               surf[OUTDATA][COLDIM];
   float               x, y, z, x_out[OUTDATA], y_out[OUTDATA], pi;

   pi = imsl_f_constant("pi", 0);
                               /* Set up output grid */
   for (i = 0; i < OUTDATA; i++) {
       x_out[i] = y_out[i] = (float) i / ((float) (OUTDATA - 1));
   }
   for (i = 0; i < 2*NDATA; i += 2) {
       xydata[i]   = 3.*cos(pi*i/NDATA);
       xydata[i+1] = 3.*sin(pi*i/NDATA);
```

```
        fdata[i/2] = F(xydata[i], xydata[i+1]);
    }
                                /* Compute scattered data interpolant */
    imsl_f_scattered_2d_interp (NDATA, xydata, fdata, OUTDATA,
                                          OUTDATA, x_out, y_out,
                                IMSL_RETURN_USER, surf,
                                IMSL_SUR_COL_DIM, COLDIM,
                                0);
                                /* Print results */
    printf("    x      y       F(x, y)   Interpolant   Error\n");
    for (i = 0; i < OUTDATA; i++) {
        for (j = 0; j < OUTDATA; j++) {
            x = x_out[i];
            y = y_out[j];
            z = surf[i][j];
            printf(" %6.3f %6.3f %10.3f  %10.3f  %10.4f\n",
                                x, y, F(x,y), z, fabs(F(x,y)-z));
        }
    }
}
```

### Output

```
   x      y      F(x, y)   Interpolant   Error
0.000  0.000     3.000       3.000       0.0000
0.000  0.500     4.000       4.000       0.0000
0.000  1.000     5.000       5.000       0.0000
0.500  0.000     6.500       6.500       0.0000
0.500  0.500     7.500       7.500       0.0000
0.500  1.000     8.500       8.500       0.0000
1.000  0.000    10.000      10.000       0.0000
1.000  0.500    11.000      11.000       0.0000
1.000  1.000    12.000      12.000       0.0000
```

### Fatal Errors

| | |
|---|---|
| `IMSL_DUPLICATE_XYDATA_VALUES` | The two-dimensional data values must be distinct. |
| `IMSL_XOUT_NOT_STRICTLY_INCRSING` | The vector `x_out` must be strictly increasing. |
| `IMSL_YOUT_NOT_STRICTLY_INCRSING` | The vector `y_out` must be strictly increasing. |

# radial_scattered_fit

Computes an approximation to scattered data in $\Re^n$ for $n \geq 1$ using radial-basis functions.

## Synopsis

*#include* <imsl.h>

*Imsl_f_radial_basis_fit* *imsl_f_radial_scattered_fit (*int* dimension, *int* num_points, *float* abscissae[], *float* fdata[], *int* num_centers, …, 0)

The type *Imsl_d_radial_basis_fit* function is imsl_d_radial_scattered_fit.

## Required Arguments

*int* dimension (Input)
Number of dimensions.

*int* num_points (Input)
The number of data points.

*float* abscissae[] (Input)
Array of size dimension × num_points containing the abscissae of the data points. The argument abscissae[i][j] is the abscissa value of the (i+1)-th data point in the (j+1)-th dimension.

*float* fdata[] (Input)
Array with num_points components containing the ordinates for the problem.

*int* num_centers (Input)
The number of centers to be used when computing the radial-basis fit. The argument num_centers should be less than or equal to num_points.

## Return Value

A pointer to the structure that represents the radial-basis fit. If a fit cannot be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*Imsl_f_radial_basis_fit* *imsl_f_radial_scattered_fit (*int* dimension, *int* num_points, *float* abscissae[], *float* fdata[], *int* num_centers,

IMSL_CENTERS, *float* centers[],

IMSL_CENTERS_RATIO, *float* ratio,

IMSL_RANDOM_SEED, *int* seed,

IMSL_SUPPLY_BASIS, *float* radial_function(),

IMSL_SUPPLY_BASIS_W_DATA, *float* radial_function(), *void* *data,

IMSL_SUPPLY_DELTA, *float* delta,

IMSL_WEIGHTS, *float* weights[],

IMSL_NO_SVD,

0)

# Optional Arguments

IMSL_CENTERS (Input)
User-supplied centers. See the Description section of this function for details.

IMSL_CENTERS_RATIO, *float* ratio (Input)
The desired ratio of centers placed on an evenly spaced grid to the total number of centers. The condition that the same number of centers placed on a grid for each dimension must be equal. Thus, the actual number of centers placed on a grid is usually less than ratio × num_centers, but will never be more than ratio × num_centers. The remaining centers are randomly chosen from the set of abscissae given in abscissae.
Default: ratio = 0.5

IMSL_RANDOM_SEED, *int* seed
The value of the random seed used when determining the random subset of abscissae to use as centers. By changing the value of seed on different calls to imsl_f_radial_scattered_fit, with the same data set, a different set of random centers will be chosen. Setting seed to zero forces the random number seed to be based on the system clock, so a possibly different set of centers will be chosen each time the program is executed.
Default: seed = 234579

IMSL_SUPPLY_BASIS, *float* radial_function (*float* distance) (Input)
User-supplied function to compute the values of the radial functions.
Default: Hardy multiquadric

IMSL_SUPPLY_BASIS_W_DATA, *float* radial_function (*float* distance, *void* *data), *void* *data (Input)
User-supplied function to compute the values of the radial functions, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.
Default: Hardy multiquadric

IMSL_SUPPLY_DELTA, *float* delta (Input)

The delta used in the default basis function

$$\phi(r) = \sqrt{r^2 + \delta^2}$$

Default: delta = 1

IMSL_WEIGHTS, *float* weights[]

This option requires the user to provide the weights.
Default: all weights equal one

IMSL_NO_SVD

This option forces the use of a *QR* decomposition instead of a singular value decomposition. This may result in space savings for large problems.

# Description

The function imsl_f_radial_scattered_fit computed a least-squares fit to scattered data in $\Re^d$ where *d* = dimension. More precisely, let *n* = ndata, *x* = abscissae, *f* = fdata, and *d* = dimension. Then we have

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

This function computes a function *F* which approximates the above data in the sense that it minimizes the sum-of-squares error

$$\sum_{i=0}^{n-1} w_i \left( F\left(x^i\right) - f_i \right)^2$$

where *w* = weights. Of course, we must restrict the functional form of *F*. This is done as follows:

$$F(x) := \sum_{j=0}^{k-1} \alpha_j \phi\left( \|x - c\|^2 + \delta^2 \right)^{1/2}$$

The function $\phi$ is called the radial function. It maps $\Re^1$ into $\Re^1$, only defined for the nonnegative reals. For the purpose of this routine, the user-supplied function

$$\phi(r) = \left(r^2 + \delta^2\right)^{1/2}$$

Note that the value of delta is defaulted to 1. It can be set by the user by using the keyword IMSL_DELTA. The parameter $\delta$ is used to scale the problem. Generally choose $\delta$ to be near the minimum spacing of the centers.

The default basis function is called the Hardy multiquadric, and it is defined as

$$\phi(r) = \left(r^2 + \delta^2\right)^{1/2}$$

A key feature of this routine is the user's control over the selection of the basis function.

To obtain the default selection of centers, we first compute the number of centers that will be on a grid and how many are on a random subset of the abscissae. Next, we compute those centers on a grid. Finally, a random subset of abscissa are obtained determining where the centers are placed. Let us examine the selection of centers in more detail.

First, we restrict the computed grid to have the same number of grid values in each of the `dimension` directions. Then, the number of centers placed on a grid, `num_gridded`, is computed as follows:

$$\alpha = (\texttt{centers\_ratio})\,(\texttt{num\_centers})$$

$$\beta = \lfloor \alpha^{1/\text{dimension}} \rfloor$$

$$\texttt{num\_gridded} = \beta^{\text{dimension}}$$

Note that there are $\beta$ grid values in each of the `dimension` directions. Then we have

$$\texttt{num\_random} = (\texttt{num\_centers}) - (\texttt{num\_gridded})$$

Now we know how many centers will be placed on a grid and how many will be placed on a random subset of the abscissae. The gridded centers are computed such that they are equally spaced in each of the `dimension` directions. The last problem is to compute a random subset, without replacement, of the abscissa. The selection is based on a random seed. The default seed is 234579. The user can change this using the optional argument `IMSL_RANDOM_SEED`. Once the subset is computed, we use the abscissae as centers.

Since the selection of good centers for a specific problem is an unsolved problem at this time, we have given the ultimate flexibility to the user. That is, you can select your own centers using the keyword `IMSL_CENTERS`. As a rule of thumb, the centers should be interspersed with the abscissae.

The return value for this function is a pointer to the structure, which contains all the information necessary to evaluate the fit. This pointer is then passed to the function `imsl_f_radial_evaluate` to produce values of the fitted function.

# Examples

## Example 1

This example, generates data from a function and contaminates it with noise on a grid of 10 equally spaced points.The fit is evaluated on a finer grid and compared with the actual function values.

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

#define NDATA          10
#define NUM_CENTERS    5
#define NOISE_SIZE     0.25
#define F(x)           ((float)(sin(2*pi*x)))
```

```
int main ()
{
   int        i;
   int        dim = 1;
   float      fdata[NDATA];
   float      *fdata2;
   float      xdata[NDATA];
   float      xdata2[2*NDATA];
   float      pi;
   float      *noise;
   Imsl_f_radial_basis_fit  *radial_fit;

   pi = imsl_f_constant ("pi", 0);

   imsl_random_seed_set (234579);
   noise = imsl_f_random_uniform(NDATA, 0);

/* Set up the sampled data points with noise. */

   for (i = 0; i < NDATA; ++i) {
      xdata[i] = (float)(i)/(float)(NDATA-1);
      fdata[i] = F(xdata[i]) + NOISE_SIZE*(1.0 - 2.0*noise[i]);
   }
/* Compute the radial fit. */

   radial_fit = imsl_f_radial_scattered_fit (dim, NDATA, xdata,
                fdata, NUM_CENTERS, 0);

/* Compare result to the original function at twice as many values as
  there were original data points. */

   for (i = 0; i < 2*NDATA; ++i)
       xdata2[i] = (float)(i/(float)(2*(NDATA-1)));
/* Evaluate the fit at these new points. */

   fdata2 = imsl_f_radial_evaluate(2*NDATA, xdata2, radial_fit, 0);

   printf("   I    TRUE      APPROX    ERROR\n");
   for (i = 0; i < 2*NDATA; ++i)
   printf("%5d %10.5f %10.5f %10.5f\n",i+1,F(xdata2[i]), fdata2[i],
           F(xdata2[i])-fdata2[i]);
 }
```

### Output

```
I    TRUE      APPROX    ERROR
1   0.00000  -0.08980   0.08980
2   0.34202   0.38795  -0.04593
3   0.64279   0.75470  -0.11191
4   0.86603   0.99915  -0.13312
```

```
5   0.98481   1.11597  -0.13116
6   0.98481   1.10692  -0.12211
7   0.86603   0.98183  -0.11580
8   0.64279   0.75826  -0.11547
9   0.34202   0.46078  -0.11876
10  -0.00000   0.11996  -0.11996
11  -0.34202  -0.23007  -0.11195
12  -0.64279  -0.55348  -0.08931
13  -0.86603  -0.81624  -0.04979
14  -0.98481  -0.98752   0.00271
15  -0.98481  -1.04276   0.05795
16  -0.86603  -0.96471   0.09868
17  -0.64279  -0.74472   0.10193
18  -0.34202  -0.38203   0.04001
19   0.00000   0.11600  -0.11600
20   0.34202   0.73553  -0.39351
```

## Example 2

This example generates data from a function and contaminates it with noise.We fit this data successively on grids of size 10, 20, …, 100. Now interpolate and print the 2-norm of the difference between the interpolated result and actual function values. Note that double precision is used for higher accuracy.

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

#define NDATA           100
#define NUM_CENTERS     100
#define NRANDOM         200
#define NOISE_SIZE      1.0
#define G(x,y)          (exp((y)/2.0)*sin(x) - cos((y)/2.0))

double radial_function (double r);

int main()
{
   int       i;
   int       ndata;
   double    *fit;
   double    ratio;
   double    fdata[NDATA+1];
   double    xydata[2 * NDATA+1];
   double    pi;
   double    *noise;
   int       num_centers;
   Imsl_d_radial_basis_fit *radial_struct;

   pi = imsl_d_constant ("pi", 0);
```

```
   /* Get the random numbers used for the noise. */

   imsl_random_seed_set (234579);
   noise = imsl_d_random_uniform (NRANDOM+1, 0);
   for (i = 0; i < NRANDOM; ++i) noise[i] = 1.0 - 2.0 * noise[i];
   printf("   NDATA         || Error ||_2 \n");

   for (ndata = 10; ndata <= 100 ; ndata += 10) {
       num_centers = ndata;

   /* Set up the sampled data points with noise. */
       for (i = 0; i < 2 * ndata; i += 2) {
           xydata[i] = 3. * (noise[i]);
           xydata[i + 1] = 3. * (noise[i + 1]);
           fdata[i / 2] = G(xydata[i], xydata[i + 1])
                        + NOISE_SIZE * noise[i];
       }

   /* Compute the radial fit. */
       ratio = 0.5;
       radial_struct= imsl_d_radial_scattered_fit (2, ndata, xydata,
                    fdata, num_centers,
                    IMSL_CENTERS_RATIO, ratio,
                    IMSL_SUPPLY_BASIS, radial_function,
                    0);
       fit = imsl_d_radial_evaluate (ndata, xydata, radial_struct, 0);

       for (i = 0; i < ndata; ++i) fit[i] -= fdata[i];

       printf("%8d %17.8f \n", ndata,
               imsl_d_vector_norm(ndata, fit, 0));
   }

}

double radial_function (double r)
{
   return log(1.0+r);
}
```

### Output

```
NDATA    || Error ||_2
10        0.00000000
20        0.00000000
30        0.00000000
40        0.00000000
50        0.00000000
60        0.00000000
70        0.00000000
80        0.00000000
90        0.00000000
100       0.00000000
```

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm. User flag = "#". |

# radial_evaluate

Evaluates a radial-basis fit.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_radial_evaluate (*int* n, *float* x[], *Imsl_d_radial_basis_fit* *radial_fit, …, 0)

The type *double* function is imsl_d_evaluate.

## Required Arguments

*int* n (Input)
The number of points at which the fit will be evaluated.

*float* x[] (Input)
Array of size (radial_fit -> dimension) × *n* containing the abscissae of the data points at which the fit will be evaluated. The argument x[i][j] is the abscissa value of the (i+1)-th data point in the (j+1)-th dimension.

*Imsl_f_radial_basis_fit* *radial_fit (Input)
A pointer to radial-basis structure to be used for the evaluation. (Input).

## Return Value

A pointer to an array of length n containing the values of the radial-basis fit at the desired values. If no value can be computed, then NULL is returned. To release this space, use imsl_free.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_radial_evaluate (*int* n, *float* x[], *Imsl_f_radial_basis_fit* *radial_fit

IMSL_RETURN_USER, *float* value[],

0)

## Optional Arguments

IMSL_RETURN_USER, *float* value[] (Input)
A user-allocated array of length n containing the returned values.

## Description

The function `imsl_f_radial_evaluate` evaluates a radial-basis fit from data generated by `imsl_f_radial_scattered_fit`.

## Example

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

#define NDATA          10
#define NUM_CENTERS    5
#define NOISE_SIZE     0.25
#define F(x)           ((float)(sin(2*pi*x)))

int main ()
{
   int       i;
   int       dim = 1;
   float     fdata[NDATA];
   float     *fdata2;
   float     xdata[NDATA];
   float     xdata2[2*NDATA];
   float     pi;
   float     *noise;
   Imsl_f_radial_basis_fit  *radial_fit;

   pi = imsl_f_constant ("pi", 0);

   imsl_random_seed_set (234579);
   noise = imsl_f_random_uniform(NDATA, 0);

/* Set up the sampled data points with noise */

   for (i = 0; i < NDATA; ++i) {
      xdata[i] = (float)(i)/(float)(NDATA-1);
      fdata[i] = F(xdata[i]) + NOISE_SIZE*(1.0 - 2.0*noise[i]);
   }
/* Compute the radial fit */

   radial_fit = imsl_f_radial_scattered_fit (dim, NDATA, xdata,
                fdata, NUM_CENTERS, 0);

/* Compare result to the original function at twice as many values as there
  were original data points */

   for (i = 0; i < 2*NDATA; ++i)
       xdata2[i] = (float)(i/(float)(2*(NDATA-1)));

/* Evaluate the fit at these new points */

   fdata2 = imsl_f_radial_evaluate(2*NDATA, xdata2, radial_fit, 0);

   printf("    I     TRUE       APPROX     ERROR\n");
   for (i = 0; i < 2*NDATA; ++i)
```

```
   printf("%5d %10.5f %10.5f %10.5f\n",i+1,F(xdata2[i]), fdata2[i],
          F(xdata2[i])-fdata2[i]);
}
```

## Output

```
I    TRUE      APPROX    ERROR
1   0.00000  -0.08980   0.08980
2   0.34202   0.38795  -0.04593
3   0.64279   0.75470  -0.11191
4   0.86603   0.99915  -0.13312
5   0.98481   1.11597  -0.13116
6   0.98481   1.10692  -0.12211
7   0.86603   0.98183  -0.11580
8   0.64279   0.75826  -0.11547
9   0.34202   0.46078  -0.11876
10  -0.00000   0.11996  -0.11996
11  -0.34202  -0.23007  -0.11195
12  -0.64279  -0.55348  -0.08931
13  -0.86603  -0.81624  -0.04979
14  -0.98481  -0.98752   0.00271
15  -0.98481  -1.04276   0.05795
16  -0.86603  -0.96471   0.09868
17  -0.64279  -0.74472   0.10193
18  -0.34202  -0.38203   0.04001
19   0.00000   0.11600  -0.11600
20   0.34202   0.73553  -0.39351
```

# Chapter 4 Quadrature

## Functions

### Univariate Quadrature



### Multivariate Quadrature



### Gauss Quadrature



### Differentiation

# Usage Notes

## Univariate Quadrature

The first nine functions in this chapter section are designed to compute approximations to integrals of the form

$$\int_c^b f(x)w(x)dx$$

The weight function *w* is used to incorporate known singularities (either algebraic or logarithmic) or to incorporate oscillations. For general-purpose integration, we recommend the use of `int_fcn_sing` (even if no endpoint singularities are present). If more efficiency is desired, then the use of one of the more specialized functions should be considered. These functions are organized as follows:

- $w = 1$

  `imsl_f_int_fcn_sing`

  `int_fcn_sing_1d`

  `imsl_f_int_fcn`

  `imsl_f_int_fcn_sing_pts`

  `imsl_f_int_fcn_inf`

  `imsl_f_int_fcn_smooth`

- $w(x) = \sin \omega x$ or $w(x) = \cos \omega x$

  `imsl_f_int_fcn_trig` (for a finite interval)

  `imsl_f_int_fcn_fourier` (for an infinite interval)

- $w(x) = (x - a)^a(b - x)^b \ln(x - a)\ln(b - x)$ where the ln factors are optional

  `imsl_f_int_fcn_alg_log`

- $w(x) = 1/(x - c)$

  `imsl_f_int_fcn_cauchy`

The calling sequences for these functions are very similar. The function to be integrated is always `fcn`, and the lower and upper limits are `a` and `b`, respectively. The requested absolute error $\varepsilon$ is `err_abs`, while the requested relative error $\rho$ is `err_rel`. These quadrature functions return the estimated answer *R*. An optional value `err_est` = *E* estimates the error. These numbers are related as follows:

$$\left| \int_a^b f(x)w(x)dx - R \right| \le E \le \max\left\{ \varepsilon, \rho \left| \int_a^b f(x)w(x)dx \right| \right\}$$

Several of the univariate quadrature functions have arguments of type `imsl_quad`, which is defined in `imsl.h`.

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The functions described above do not directly address this question. However, the standard method for handling this problem is first to interpolate the data, and then to integrate the interpolant. This can be accomplished by using the IMSL spline interpolation functions with one of the spline integration functions, which can be found in Interpolation and Approximation

## Multivariate Quadrature

Four functions have been included in this chapter that are of use in approximating certain multivariate integrals. In particular, the functions `imsl_f_int_fcn_2d` and `imsl_f_int_fcn_sing_2d` return an approximation to an iterated two-dimensional integral of the form

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

while `imsl_f_int_fcn_sing_3d` returns an approximation to an iterated three-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} \int_{p(x, y)}^{q(x, y)} f(x, y, z)\, dz dy dx$$

The fourth function, `imsl_f_int_fcn_hyper_rect`, returns an approximation to the integral of a function of $n$ variables over a hyper-rectangle

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} f(x_1, \dots, x_n)\, dx_n \dots dx_1$$

When working with two-dimensional tensor-product tabular data, use the IMSL spline interpolation function `imsl_f_spline_2d_interp`, followed by the IMSL spline integration function `imsl_f_spline_2d_integral` described in Chapter 3, "Interpolation and Approximation".

## Gauss Quadrature

Before computing Gauss quadratures, you must compute so-called Gauss quadrature rules that integrate polynomials of as high degree as possible. These quadrature rules can be easily computed using the function `imsl_f_gauss_quad_rule`, which produces the points $\{w_i\}$ for $i = 1, \dots, N$ that satisfy

$$\int_a^b f(x)w(x)\,dx = \sum_{i=1}^{N} f(x_i)w_i$$

for all functions *f* that are polynomials of degree less than 2*N*. The weight functions ***w*** may be selected from the following table.

| *w*(*x*) | **Interval** | **Name** |
|---|---|---|
| 1 | (−1, 1) | Legendre |
| $1/\left(\sqrt{1-x^2}\right)$ | (−1, 1) | Chebyshev 1st kind |
| $\sqrt{1-x^2}$ | (−1, 1) | Chebyshev 2nd kind |
| $e^{-x^2}$ | (−∞, ∞) | Hermite |
| $(1 + x)^a (1 - x)^b$ | (−1, 1) | Jacobi |
| $e^{-x}x^a$ | (0, ∞) | Generalized Laguerre |
| 1/cosh (*x*) | (−∞, ∞) | Hyperbolic cosine |

Where permissible, `imsl_f_gauss_quad_rule` also computes Gauss-Radau and Gauss-Lobatto quadrature rules.

# int_fcn_sing

more...

Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_sing (*float* fcn (), *float* a, *float* b, …, 0)

The type ***double*** function is imsl_d_int_fcn_sing.

## Required Arguments

*float* fcn (*float* x) (input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration.

## Return Value

An estimate of

$$\int_a^b fcn(x)\,dx$$

If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_sing (*float* fcn (), *float* a, *float* b,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

# Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

This function is designed to handle functions with endpoint singularities. However, the performance on functions that are well-behaved at the endpoints is also quite good.

The function imsl_f_int_fcn_sing is a general-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error. It subdivides the interval [*a*, *b*] and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-

point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This function uses an extrapolation procedure known as the ε-algorithm.

On some platforms, `imsl_f_int_fcn_sing` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_sing` is based on the subroutine QAGS by Piessens et al. (1983).

# Examples

## Example 1

The value of

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

is estimated.

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

float fcn(float x);

int main()
{
   float q, exact;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_sing (fcn, 0.0, 1.0,
       0);

   /* Print the result and */
   /*the exact answer */
   exact = -4.0;
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
```

```
   return log(x)/sqrt(x);
}
```

### Output

```
integral =    -4.000
exact    =    -4.000
```

## Example 2

The value of

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

is again estimated. The values of the actual and estimated errors are printed as well. Note that these numbers are machine dependent. Furthermore, usually the error estimate is pessimistic. That is, the actual error is usually smaller than the error estimate as is in this example.

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

float fcn(float x);

int main()
{
   float q, exact, err_est, exact_err;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_sing (fcn, 0.0, 1.0,
       IMSL_ERR_EST, &err_est,
       0);

   /* Print the result and */
   /* the exact answer */
   exact = -4.0;
   exact_err = fabs(exact - q);
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
   printf("error estimate  = %e\nexact error     = %e\n", err_est,
       exact_err);
}

float fcn(float x)
{
   return log(x)/sqrt(x);
```

```
}
```

### Output

```
integral   =      -4.000
exact      =      -4.000
error estimate    = 2.708435e-004
exact error       = 2.241135e-005
```

## Warning Errors

| | |
|---|---|
| IMSL_ROUNDOFF_CONTAMINATION | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| IMSL_PRECISION_DEGRADATION | A degradation in precision has been detected. |
| IMSL_EXTRAPOLATION_ROUNDOFF | Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected. |

## Fatal Errors

| | |
|---|---|
| IMSL_DIVERGENT | Integral is probably divergent or slowly convergent. |
| IMSL_PRECISION_DEGRADATION | Integral is probably divergent or slowly convergent. |
| IMSL_MAX_SUBINTERVALS | The maximum number of subintervals allowed has been reached. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_sing_1d

Integrates a function with a possible internal or endpoint singularity.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_sing_1d (*float* fcn (), *float* a, *float* b, …, 0)

The type *double* function is imsl_d_int_fcn_sing_1d.

## Required Arguments

*float* fcn (*float* x) (Input/Output)
User-supplied function to be integrated.

**Arguments**

*float* x (Input)
Independent variable.

**Return Value**

The computed function value at the point x.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration. The relative values of a and b are interpreted properly. Thus if one exchanges a and b, the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of b − a.

## Return Value

An estimate of

$$\int_a^b fcn(x)\,dx$$

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_sing_1d (*float* fcn (), *float* a, *float* b, IMSL_FCN_W_DATA, *float* fcn(), *float* *err_post, *void* *data,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_FRAC, *float* err_frac,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_PRIOR, *float* err_prior,

IMSL_MAX_EVALS, *int* maxfn,

IMSL_SINGULARITY, *float* singularity, *int* singularity_type,

IMSL_N_EVALS, *int* *n_evals,

IMSL_ERR_EST, *float* *err_est,

IMSL_ISTATUS, *int* *istatus,

0)

## Optional Arguments

IMSL_FCN_W_DATA, *float* fcn (*float* x, *float* *err_post, *void* *data), *float* *err_post, *void* *data (Input)

*float* fcn (*float* x, *float* *err_post, *void* *data) (Input)

User supplied function to be integrated, which also accepts a pointer to an a posteriori estimate of the absolute value of the error committed while evaluating the integrand, and a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* x (Input)

The point at which the function is evaluated.

*float* *err_post (Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. This argument provides a means for the user to have fcn compute this value as output. Although this argument must appear in the argument list of fcn, it need not be referenced in the function. See Example 2 for an example of this.

*void* *data (Input)

A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point x.

*float* *err_post (Input/Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. On input, the user may supply this estimate and that value will be used as the estimate thereafter provided fcn does not calculate a new value. If an a posteriori estimate of the value of the error is not known, set err_post to 0.0 on input. On output, err_post will contain either the input value set by the user or the value calculated by fcn.

*void* *data (Input)

A pointer to the data to be passed to the user-supplied function.

IMSL_ERR_ABS, *float* err_abs (Input)

Absolute error tolerance. See Remark 1 for a discussion on the error tolerances.

Default: err_abs = 0.0

IMSL_ERR_FRAC, *float* err_frac (Input)

A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See Remark 1 for a discussion on accuracy.

Default: err_frac = 0.75

IMSL_ERR_REL, *float* err_rel (Input)

The error tolerance relative to the value of the integral. See Remark 1 for a discussion on the error tolerances.

Default: err_rel = 0.0

IMSL_ERR_PRIOR, *float* err_prior (Input)

An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral.

Default: err_prior = imsl_f_machine(4)

IMSL_MAX_EVALS, *int* maxfn (Input)

The maximum number of function evaluations to use to compute the integral.

Default: The number of function values is not bounded.

IMSL_SINGULARITY, *float* singularity, *int* singularity_type (Input)

singularity is the real part of the abscissa of a singularity or discontinuity in the integrand. singularity_type is a signed integer specifying the type of singularity which occurs in the integrand. If the singularity has a leading term of the form $x^{\alpha}$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n-1)/2$ where $n$ is a nonnegative integer, or the singularity is well outside the interval, set singularity_type to a positive integer. Otherwise, set singularity_type to a negative integer. Also see Remark 2.

Default: It is assumed that there is no singularity in the integrand so singularity and singularity_type are not set.

IMSL_N_EVALS, *int* *n_evals (Output)

Number of function evaluations used to calculate the integral.

IMSL_ERR_EST, *float* *err_est (Output)

An estimate of the upper bound of the magnitude of the difference between the value returned by imsl_f_int_fcn_sing_1d and the true value of the integral.

IMSL_ISTATUS, *int* *istatus (Output)

A status flag indicating the error criteria which was satisfied on exit.

| istatus | Description |
|---|---|
| -1 | Indicates normal termination with either the absolute or relative error tolerance criteria satisfied. |
| -2 | Indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied. |
| -3 | Indicates normal termination with none of the error tolerance criteria satisfied. |
| Other | Any value other than the above indicates abnormal termination due to an error condition. |

## Description

The function imsl_f_int_fcn_sing_1d is based on the JPL Library routine SINT1. The integral is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the $k^{th}$ formula used all the integrand values used in the $k\text{-}1^{st}$ formula, and added $2^{k-1}$ new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to $k = 8$.

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

## Remarks

### Remark 1

The user provides the absolute error tolerance through optional argument `IMSL_ERR_ABS`. Optional argument `IMSL_ERR_FRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). Optional argument `IMSL_ERR_REL` represents the error tolerance relative to the value of the integral. The internal value for `err_frac` is bounded between .5 and 1. By default, `err_abs` and `err_rel` are set to 0.0 and `err_frac` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.

The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.

### Remark 2

Optional argument `IMSL_SINGULARITY` provides the user with a means to give the routine information about the location and type of any known singularity of the integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside `[a,b]`.

If the singularity has a leading term of the form $x^\alpha$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n - 1)/2$ where n is a nonnegative integer, or the singularity is well outside the interval, set `singularity_type` to a positive value. Otherwise, set `singularity_type` to a negative value. The meaning of "large" depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered "large". For a singularity of the form $x^\alpha \log x$ use the above rule, even if $\alpha$ is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed, some experimentation may be useful.

When `singularity_type` is positive, a transformation of the form $T = TA + (X - TA)^2 / (TB - TA)$ is applied, where *TA* is the abscissa of the singularity and *TB* is the end of the interval. If *TA* is outside the interval, *TB* will be the end of the interval farthest from *TA*. If *TA* is inside the interval, the interval will immediately be subdivided at *TA*, and both parts will be separately integrated with *TB* equal to each end of the original interval, respectively. When `singularity_type` is negative, a transformation of the form $T = TA + (X - TA)^4 / (TB - TA)^3$ is applied, with *TA* and *TB* as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

# Examples

## Example 1

The value of

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

is estimated. Note that the optional argument `IMSL_SINGULARITY` is used.

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

float fcn (float x);

int main(){
    int n_evals, singularity_type=-1;
    float a=0.0, b=1.0, singularity=0.0, errabs=0.0, errest, result;

    result = imsl_f_int_fcn_sing_1d(fcn,a,b,
        IMSL_ERR_ABS, errabs,
        IMSL_SINGULARITY, singularity, singularity_type,
        IMSL_ERR_EST, &errest,
        IMSL_N_EVALS, &n_evals, 0);
    printf("The approximation to the integral is %f\n", result);
    printf("The estimated absolute error is %f\n", errest);
    printf("The number of evaluations taken is %d\n", n_evals);
}


float fcn (float x) {
    return  log(x)/sqrt(x);
}
```

### Output

```
The approximation to the integral is -4.000000
The estimated absolute error is 6.0e-007
```

```
The number of evaluations taken is 32
```

### Example 2

The value of

$$\int_1^2 (2x + kx)\, dx = 6$$

is estimated. Note that the optional argument `IMSL_FCN_W_DATA` is used to set the value of $k = 2$ in the user-supplied function, `fcn2`. We do not attempt to calculate the an a posteriori error in the function evaluation so `err_post` is set to 0.0.

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

float fcn (float x, float *err_post, void *fcn_data);

int main(){
    float a=1.0, b=2.0, err_post=0.0,rdata[1]={2.0}, errest, result;

    result = imsl_f_int_fcn_sing_1d(NULL, a, b,
        IMSL_FCN_W_DATA, fcn, &err_post, (void *)rdata,
        IMSL_ERR_EST, &errest, 0);
    printf("The approximation to the integral is %f\n", result);
    printf("The estimated absolute error is %6.1e\n", errest);
}


float fcn (float x, float *err_post, void *fcn_data) {
    float k = ((float*)fcn_data)[0];
    return  2.0*x + k*x;
}
```

### Output

```
The approximation to the integral is 6.000000
The estimated absolute error is 1.2e-006
```

## Fatal Errors

| | |
|---|---|
| `IMSL_NONINTEGRABLE` | The integrand apparently contains a noninte-grable singularity. The abscissa of the singularity is near #. The result has been set to NaN. |
| `IMSL_MAX_FCN_EVAL_EXCEEDED_NAN` | The maximum number of function evalua-tions allowed, "`maxfn`", has been exceeded. "`maxfn`" is currently set to a value of #. The result has been set to NaN. |

| | |
|---|---|
| IMSL_CRITERIA_NOT_SATISFIED | The algorithm has terminated without satisfying any of the error tolerance criteria. The error estimate is #. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn

Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn (*float* fcn(), *float* a, *float* b, …, 0)

The type *double* function is imsl_d_int_fcn.

## Required Arguments

*float* fcn (*float x*) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration.

## Return Value

The value of

$$\int_a^b fcn(x)\,dx$$

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn (*float* fcn (*float* x), *float* a, *float* b,

IMSL_RULE, *int* rule,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_RULE, *int* rule (Input)

Choice of quadrature rule.

| rule | Gauss-Kronrod Rule |
|---|---|
| 1 | 7-15 points |
| 2 | 10-21 points |
| 3 | 15-31 points |
| 4 | 20-41 points |
| 5 | 25-51 points |
| 6 | 30-61 points |

Default: rule = 1

IMSL_ERR_ABS, *float* err_abs (Input)

Absolute accuracy desired.

Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)

Relative accuracy desired.

Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)

Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)

Number of subintervals allowed.

Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)

Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)

Address to store the number of evaluations of fcn.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

The function `imsl_f_int_fcn` is a general-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It subdivides the interval [$a$, $b$] and uses a ($2k$ + 1)-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the $k$-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. The function `imsl_f_int_fcn` is based on the subroutine QAG by Piessens et al. (1983).

On some platforms, `imsl_f_int_fcn` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

Should `imsl_f_int_fcn` fail to produce acceptable results, consider one of the more specialized functions documented in this chaptersection.

# Examples

## Example 1

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is computed. Since the integrand is not oscillatory, all of the default values are used. The values of the actual and estimated error are machine dependent.

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

float fcn(float x);
float q;
float exact;

int main()
{
   imsl_omp_options(
```

```
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);
    /* evaluate the integral */
    q = imsl_f_int_fcn (fcn, 0.0, 2.0, 0);

    /* print the result and the exact answer */
    exact = exp(2.0) + 1.0;
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
    float y;

    y = x * (exp(x));
    return y;
}
```

### Output

```
integral =     8.389
exact    =     8.389
```

## Example 2

The value of

$$\int_0^1 \sin\left(1/x\right)dx$$

is computed. Since the integrand is oscillatory, `rule` = 6 is used. The exact value is 0.50406706. The values of the actual and estimated error are machine dependent.

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

float fcn(float x);

int main()
{
    float q, err_est, err_abs= 0.0001, exact = 0.50406706, error;

    imsl_omp_options(
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    /* intergrate fcn(x) from 0 to 1 */
    q = imsl_f_int_fcn (fcn, 0.0, 1.0,
        IMSL_ERR_ABS,   err_abs,/* set abs error value*/
```

```
        IMSL_RULE,      6,
        IMSL_ERR_EST,   &err_est, /* pass in address */
        0);

    error = q - exact;
    /* print the result and the exact answer */
    printf(" integral = %10.3f\n   exact = %10.3f\n   error = %10.3f\n ",
        q, exact , error);
    printf("  err_est = %g\n", err_est);
}

float fcn(float x)
{
    /* compute sin(1/x), avoiding division by zero */
    return ((x)>1.0e-5) ? sin(1.0/(x)) : 0.0;
}
```

### Output

```
integral =     0.504
  exact =     0.504
  error =     0.000
  err_est = 0.000170593
```

## Warning Errors

| | |
|---|---|
| IMSL_ROUNDOFF_CONTAMINATION | Roundoff error has been detected. The requested tolerances, "err_abs" = # and "err_rel" cannot be reached. |
| IMSL_PRECISION_DEGRADATION | Precision is degraded due to too fine a subdivision relative to the requested tolerance. This may be due to bad integrand behavior in the interval (#,#). Higher precision may alleviate this problem. |

## Fatal Errors

| | |
|---|---|
| IMSL_MAX_SUBINTERVALS | The maximum number of subintervals allowed "max_sub" has been reached. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_sing_pts

more...

Integrates a function with singularity points given.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_sing_pts` (*float* `fcn ()`, *float* `a`, *float* `b`, *int* `npoints`, *float* `points []`, …, 0)

The type ***double*** function is `imsl_d_int_fcn_sing_pts`.

## Required Arguments

*float* `fcn` (*float* `x`) (Input)
User-supplied function to be integrated.

*float* `a` (Input)
Lower limit of integration.

*float* `b` (Input)
Upper limit of integration.

*int* `npoints` (Input)
The number of singularities of the integrand.

*float* `points []` (Input)
The abscissas of the singularities. These values should be interior to the interval [*a, b*].

## Return Value

The value of

$$\int_a^b fcn(x)\,dx$$

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* imsl_f_int_fcn_sing_pts (*float* fcn (), *float* a, *float* b, *int* npoints, *float* points [],

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functionsin the introduction to this manual for more details.

## Description

The function `imsl_f_int_fcn_sing_pts` is a special-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error. It subdivides the interval [*a*, *b*] into `npoints` + 1 user-supplied subintervals and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This function uses an extrapolation procedure known as the ε-algorithm.

On some platforms,`imsl_f_int_fcn_sing_pts` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_sing_pts` is based on the subroutine QAGP by Piessens et al. (1983).

## Examples

### Example 1

The value of

$$\int_0^3 x^3 \ln\left|\left(x^2-1\right)\left(x^2-2\right)\right| dx = 61\ln 2 + \frac{77}{4}\ln 7 - 27$$

is computed. The values of the actual and estimated error are machine dependent. Note that this function never evaluates the user-supplied function at the user-supplied breakpoints.

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

float fcn(float x);

int main()
{
   int        npoints = 2;
   float      q, exact, points[2];

   /* Set singular points */
   points[0] = 1.0;
   points[1] = sqrt(2.);

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
```

```
   /* Evaluate the integral */
   q = imsl_f_int_fcn_sing_pts (fcn, 0.0, 3.0, npoints, points,
       0);

   /* print the result and */
   /* the exact answer */
   exact = 61.*log(2.) + (77./4)*log(7.) - 27.;
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
   return x*x*x*(log(fabs((x*x-1.)*(x*x-2.))));
}
```

### Output

```
integral =    52.741
exact    =    52.741
```

## Example 2

The value of

$$\int_0^3 x^3 \ln\left|\left(x^2-1\right)\left(x^2-2\right)\right| dx = 61\ln 2 + \frac{77}{4}\ln 7 - 27$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

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

float fcn(float x);

int main()
{
   int   n_evals, npoints = 2;
   float q, exact, err_est, exact_err, points[2];

   /* Set singular points */
   points[0] = 1.0;
   points[1] = sqrt(2.);

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
```

```
   /* Evaluate the integral and get the */
   /* error estimate and the number of */
   /* evaluations */
   q = imsl_f_int_fcn_sing_pts (fcn, 0.0, 3.0, npoints, points,
       IMSL_ERR_EST, &err_est,
       IMSL_N_EVALS, &n_evals,
       0);

   /* Print the result and the */
   /* exact answer */
   exact = 61.*log(2.) + (77./4)*log(7.) - 27.;
   exact_err = fabs(exact - q);
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
   printf("error estimate  = %e\nexact error     = %e\n", err_est,
       exact_err);
   printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
   return x*x*x*(log(fabs((x*x-1.)*(x*x-2.))));
}
```

### Output

```
integral =    52.741
exact    =    52.741
error estimate  = 1.258850e-04
exact error     = 3.051758e-05
The number of function evaluations = 819
```

## Warning Errors

| | |
|---|---|
| `IMSL_ROUNDOFF_CONTAMINATION` | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| `IMSL_PRECISION_DEGRADATION` | A degradation in precision has been detected. |
| `IMSL_EXTRAPOLATION_ROUNDOFF` | Roundoff error has been detected in the extrapolation table. The tolerances, "`err_abs`" = # and "`err_rel`" = # cannot be reached. |

## Fatal Errors

| | |
|---|---|
| `IMSL_DIVERGENT` | Integral is probably divergent or slowly convergent. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_alg_log

more...

Integrates a function with algebraic-logarithmic singularities.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_alg_log (*float* fcn (), *float* a, *float* b, *Imsl_quad* weight, *float* alpha, *float* beta, …, 0)

The type *double* function is imsl_d_int_fcn_alg_log.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration.

*Imsl_quad* weight, *float* alpha, *float* beta (Input)
These three parameters are used to describe the weight function that may have algebraic or logarithmic singularities at the endpoints. The parameter weight can take on four values as described below. The parameters alpha = α and beta = β specify the strength of the singularities at *a* or *b* and hence, must be greater than −1.

| **Weight** | **Integration Weight** |
|---|---|
| IMSL_ALG | $(x - a)^{a} (b - x)^{b}$ |
| IMSL_ALG_LEFT_LOG | $(x - a)^{a} (b - x)^{b} \log (x - a)$ |
| IMSL_ALG_RIGHT_LOG | $(x - a)^{a} (b - x)^{b} \log (b - x)$ |
| IMSL_ALG_LOG | $(x - a)^{a} (b - x)^{b} \log (x - a) \log (b - x)$ |

## Return Value

The value of

$$\int_a^b fcn(x)w(x)dx$$

is returned where *w*(*x*) is one of the four weights above. If no value can be computed, then NaN is returned.

# Synopsis with Optional Arguments

*#include* <imsl.h>

*float* `imsl_f_int_fcn_alg_log` (*float* `fcn` (*float* `x`), *float* `a`, *float* `b`, *Imsl_quad* `weight`, *float* `alpha`, *float* `beta`,

`IMSL_ERR_ABS`, *float* `err_abs`,

`IMSL_ERR_REL`, *float* `err_rel`,

`IMSL_ERR_EST`, *float* `*err_est`,

`IMSL_MAX_SUBINTER`, *int* `max_subinter`,

`IMSL_N_SUBINTER`, *int* `*n_subinter`,

`IMSL_N_EVALS`, *int* `*n_evals`,

`IMSL_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,

0)

# Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)

Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)

User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function imsl_f_int_fcn_alg_log is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form $w(x)f(x)$ where $w(x)$ is a weight function described above. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas is employed. This function is based on the subroutine QAWS, which is fully documented by Piessens et al. (1983).

On some platforms,imsl_f_int_fcn_alg_log can evaluate the user-supplied function fcn in parallel. This is done only if the function imsl_omp_options is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

## Examples

### Example 1

The value of

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

is computed.

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

float fcn(float x);

int main()
{
   float q, exact;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
   /* Evaluate the integral */
   q = imsl_f_int_fcn_alg_log (fcn, 0.0, 1.0,
```

```
        IMSL_ALG_LEFT_LOG, 1.0, 0.5,
        0);

    /* Print the result and the */
    /* exact answer */
    exact = (3.*log(2.)-4.)/9.;
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return sqrt(1+x);
}
```

### Output

```
integral =    -0.213
exact    =    -0.213
```

## Example 2

The value of

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

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

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

float fcn(float x);

int main()
{
    int   n_evals;
    float q, exact, err_est, exact_err;

    imsl_omp_options(
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);
    /* Evaluate the integral */
    q = imsl_f_int_fcn_alg_log (fcn, 0.0, 1.0,
        IMSL_ALG_LEFT_LOG, 1.0, 0.5,
        IMSL_ERR_EST, &err_est,
        IMSL_N_EVALS, &n_evals,
```

```
        0);

    /* Print the result and the */
    /* exact answer */
    exact = (3.*log(2.)-4.)/9.;
    exact_err = fabs(exact - q);
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
    printf("error estimate  = %e\nexact error     = %e\n", err_est,
        exact_err);
    printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
    return sqrt(1+x);
}
```

### Output

```
integral =    -0.213
exact    =    -0.213
error estimate  = 3.725290e-09
exact error     = 1.490116e-08
The number of function evaluations = 50
```

## Warning Errors

| | |
|---|---|
| `IMSL_ROUNDOFF_CONTAMINATION` | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| `IMSL_PRECISION_DEGRADATION` | A degradation in precision has been detected. |

## Fatal Errors

| | |
|---|---|
| `IMSL_MAX_SUBINTERVALS` | The maximum number of subintervals allowed has been reached. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_inf

more...

Integrates a function over an infinite or semi-infinite interval.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_inf (*float* fcn (), *float* bound, *Imsl_quad* interval, …, 0)

The type *double* procedure is imsl_d_int_fcn_inf.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function to be integrated.

*float* bound (Input)
Finite limit of integration. This argument is ignored if interval has the value IMSL_INF_INF.

*Imsl_quad* interval (Input)
Flag indicating integration limits. The following settings are allowed:

| interval | Integration Limits |
|---|---|
| IMSL_INF_BOUND | $(-\infty$, bound) |
| IMSL_BOUND_INF | (bound, $\infty$) |
| IMSL_INF_INF | $(-\infty, \infty)$ |

## Return Value

The value of

$$\int_a^b fcn(x)\,dx$$

is returned where ***a*** and ***b*** are appropriate integration limits. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_inf (*float* fcn, *float* bound, *Imsl_quad* interval,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)

Absolute accuracy desired.

Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)

Relative accuracy desired.

Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)

Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)

Number of subintervals allowed.

Default: max_subinter = 500.

IMSL_N_SUBINTER, *int* *n_subinter (Output)

Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)

Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)

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

## Description

The function `imsl_f_int_fcn_inf` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It initially transforms an infinite or semi-infinite interval into the finite interval [0, 1]. It then uses the same strategy as the function `imsl_f_int_fcn_sing`.

On some platforms, `imsl_f_int_fcn_inf` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_inf` is based on the subroutine QAGI by Piessens et al. (1983).

## Examples

### Example 1

The value of

$$\int_0^{\infty} \frac{\ln(x)}{1+(10x)^2} dx = \frac{-\pi \ln(10)}{20}$$

is computed.

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

float fcn(float x);

int main()
{
   float q, exact, pi;

   pi = imsl_f_constant("pi", 0);

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_inf (fcn, 0.0,
       IMSL_BOUND_INF,
       0);

   /* Print the result and the */
   /* exact answer */
   exact = -pi*log(10.)/20.;
```

```
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
   float      z;

   z = 10.*x;
   return log(x)/(1+ z*z);
}
```

### Output

```
integral =    -0.362
exact    =    -0.362
```

## Example 2

The value of

$$\int_0^\infty \frac{\ln x}{1+\left(10x\right)^2}dx = \frac{-\pi \ln\left(10\right)}{20}$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as in this example. The number of function evaluations also are printed.

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

float fcn(float x);

int main()
{
   int   n_evals;
   float q, exact, err_est, exact_err, pi;

   pi = imsl_f_constant("pi",
       0);

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_inf (fcn, 0.0,
       IMSL_BOUND_INF,
       IMSL_ERR_EST, &err_est,
       IMSL_N_EVALS, &n_evals,
```

```
        0);

    /* Print the result and the */
    /* exact answer */
    exact = -pi*log(10.)/20.;
    exact_err = fabs(exact - q);
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
    printf("error estimate  = %e\nexact error     = %e\n", err_est,
        exact_err);
    printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
    float      z;

    z = 10.*x;
    return log(x)/(1+ z*z);
}
```

### Output

```
integral =    -0.362
exact    =    -0.362
error estimate  = 2.801418e-06
exact error     = 2.980232e-08
The number of function evaluations = 285
```

## Warning Errors

| | |
|---|---|
| IMSL_ROUNDOFF_CONTAMINATION | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| IMSL_PRECISION_DEGRADATION | A degradation in precision has been detected. |
| IMSL_EXTRAPOLATION_ROUNDOFF | Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected. |

## Fatal Errors

| | |
|---|---|
| IMSL_DIVERGENT | Integral is probably divergent or slowly convergent. |
| IMSL_MAX_SUBINTERVALS | The maximum number of subintervals allowed has been reached. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_trig

more...

Integrates a function containing a sine or a cosine factor.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_trig (*float* fcn (), *float* a, *float* b, *Imsl_quad* weight, *float* omega, …, 0)

The type *double* function is imsl_d_int_fcn_trig.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration.

*Imsl_quad* weight and *float* omega (Input)
These two parameters are used to describe the trigonometric weight. The parameter weight can take on the two values described below, and the parameter omega = ω specifies the frequency of the trigonometric weighting function.

| weight | Integration Weight |
|---|---|
| IMSL_COS | cos ($\omega x$) |
| IMSL_SIN | sin ($\omega x$) |

## Return Value

The value of

$$\int_a^b fcn(x)\cos(\omega x)\,dx$$

is returned if weight = IMSL_COS. If weight = IMSL_SIN, then the cosine factor is replaced with a sine factor. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_trig (*float* fcn(), *float* a, *float* b, *Imsl_quad* weight, *float* omega,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_MAX_MOMENTS, *int* max_moments,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)

Absolute accuracy desired.

Default: $err\_abs = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)

Relative accuracy desired.

Default: $err\_rel = \sqrt{\varepsilon}$ where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)

Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)

Number of subintervals allowed.

Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)

Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)

Address to store the number of evaluations of fcn.

`IMSL_MAX_MOMENTS`, *int* `max_moments` (Input)

This is an upper bound on the number of Chebyshev moments that can be stored. Increasing (decreasing) this number may increase (decrease) execution speed and space used.

Default: `max_moments` = 21

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function `imsl_f_int_fcn_trig` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form $w(x)f(x)$ where $w(x)$ is either $\cos(\omega x)$ or $\sin(\omega x)$. Depending on the length of the subinterval in relation to the size of $\omega$, either a modified Clenshaw-Curtis procedure or a Gauss-Kronrod 7/15 rule is employed to approximate the integral on a subinterval. This function uses the general strategy of the function `imsl_f_int_fcn_sing`.

On some platforms,`imsl_f_int_fcn_trig` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_trig` is based on the subroutine QAWO by Piessens et al. (1983).

## Examples

### Example 1

The value of

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

is computed. Notice that we have coded around the singularity at zero. This is necessary since this procedure evaluates the integrand at the two endpoints.

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

float fcn(float x);

int main()
{
   float q, exact, omega;
```

```
    omega = 10*imsl_f_constant("pi",
        0);

    imsl_omp_options(
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    /* Evaluate the integral */
    q = imsl_f_int_fcn_trig (fcn, 0.0, 1.0,
        IMSL_SIN, omega,
        0);

    /* Print the result and the */
    /* exact answer */
    exact = -.1281316;
    printf("integral = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return (x==0.0) ? 0.0 : log(x);
}
```

### Output

```
integral =     -0.128
exact    =     -0.128
```

## Example 2

The value of

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

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, it is usually the case that the error estimate is pessimistic. That is, the actual error is usually smaller than the error estimate as is the case in this example. The number of function evaluations are also printed.

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

float fcn(float x);

int main()
{
```

```
    int   n_evals;

    float q, exact, omega, err_est, exact_err;

    omega = 10*imsl_f_constant("pi",
        0);

    imsl_omp_options(
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    /* Evaluate the integral */
    q = imsl_f_int_fcn_trig (fcn, 0.0, 1.0,
        IMSL_SIN, omega,
        IMSL_ERR_EST, &err_est,
        IMSL_N_EVALS, &n_evals,
        0);

    /* Print the result and the */
    /* exact answer */
    exact = -.1281316;
    exact_err = fabs(exact - q);
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
    printf("error estimate  = %e\nexact error     = %e\n", err_est,
        exact_err);
    printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
    return (x==0.0) ? 0.0 : log(x);
}
```

### Output

```
integral =    -0.128
exact    =    -0.128
error estimate  = 7.504603e-05
exact error     = 5.245209e-06
The number of function evaluations = 215
```

## Warning Errors

| | |
|---|---|
| `IMSL_ROUNDOFF_CONTAMINATION` | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| `IMSL_PRECISION_DEGRADATION` | A degradation in precision has been detected. |
| `IMSL_EXTRAPOLATION_ROUNDOFF` | Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected. |

## Fatal Errors

| | |
|---|---|
| `IMSL_DIVERGENT` | Integral is probably divergent or slowly convergent. |
| `IMSL_MAX_SUBINTERVALS` | The maximum number of subintervals allowed has been reached. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_fourier

OpenMP
more...

Computes a Fourier sine or cosine transform.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_fourier (*float* fcn (), *float* a, *Imsl_quad* weight, *float* omega, …, 0)

The type ***double*** function is imsl_d_int_fcn_fourier.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration. The upper limit of integration is ∞.

*Imsl_quad* weight and *float* omega (Input)
These two parameters are used to describe the trigonometric weight. The parameter weight can take on the two values described below, and the parameter omega = ω specifies the frequency of the trigonometric weighting function.

| weight | Integration Weight |
|---|---|
| IMSL_COS | cos ($\omega x$) |
| IMSL_SIN | sin ($\omega x$) |

## Return Value

The return value is

$$\int_a^\infty fcn(x)\cos(\omega x)\,dx$$

if weight = IMSL_COS. If weight = IMSL_SIN, then the cosine factor is replaced with a sine factor. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_fourier (*float* fcn (), *float* a, *Imsl_quad* weight, *float* omega,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_MAX_CYCLES, *int* max_cycles,

IMSL_MAX_MOMENTS, *int* max_moments,

IMSL_N_CYCLES, *int* *n_cycles,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)

Absolute accuracy desired.

Default: $err\_abs = \sqrt{\varepsilon}$ where ε is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)

Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)

Number of subintervals allowed.

Default: max_subinter = 500

IMSL_MAX_CYCLES, *int* max_cycles (Input)

Number of cycles allowed.

Default: max_subinter = 50

IMSL_MAX_MOMENTS, *int* max_moments (Input)

Number of subintervals allowed in the partition of each cycle.

Default: max_moments = 21

IMSL_N_CYCLES, *int* *n_cycles (Output)

Address to store the number of cycles generated.

IMSL_N_EVALS, *int* *n_evals (Output)

Address to store the number of evaluations of fcn.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

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

## Description

The function `imsl_f_int_fcn_fourier` is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form $w(x)f(x)$ where $w(x)$ is either $\cos\omega x$ or $\sin\omega x$. The integration interval is always semi-infinite of the form $[a, \infty]$. These Fourier integrals are approximated by repeated calls to the function `imsl_f_int_fcn_trig` followed by extrapolation.

On some platforms,`imsl_f_int_fcn_fourier` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_fourier` is based on the subroutine QAWF by Piessens et al. (1983).

## Examples

### Example 1

The value of

$$\int_0^\infty x^{-1/2}\cos(\pi x/2)\,dx = 1$$

is computed. Notice that the integrand is coded to protect for the singularity at zero.

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

float fcn(float x);

int main()
{
   float q, exact, omega;

   omega = imsl_f_constant("pi",0) / 2.;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
```

```
    /* Evaluate the integral */
    q = imsl_f_int_fcn_fourier (fcn, 0.0,
        IMSL_COS, omega,
        0);

    /* Print the result and the */
    /* exact answer */
    exact = 1.0;
    printf("integral = %10.3f\nexact     = %10.3f\n", q, exact);
}

float fcn(float x)
{
    return (x==0.) ? 0. : 1./sqrt(x);
}
```

### Output

```
integral =     1.000
exact    =     1.000
```

## Example 2

The value of

$$\int_0^{\infty} x^{-1/2} \cos\left(\pi x/2\right) dx = 1$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate,

as is the case in this example.The number of function evaluations also are printed. Notice that the integrand is coded to protect for the singularity at zero.

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

float fcn(float x);

int main()
{
    int   n_evals;
    float q, exact, omega, err_est, exact_err;

    omega = imsl_f_constant("pi",0) / 2.0;

    imsl_omp_options(
        IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);
```

```
    /* Evaluate the integral */
    q = imsl_f_int_fcn_fourier (fcn, 0.0,
        IMSL_COS, omega,
        IMSL_ERR_EST, &err_est,
        IMSL_N_EVALS, &n_evals,
        0);

    /* Print the result and the */
    /* exact answer */
    exact = 1.;
    exact_err = fabs(exact - q);
    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
    printf("error estimate  = %e\nexact error     = %e\n", err_est,
        exact_err);
    printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
    return (x==0.) ? 0. : 1./sqrt(x);
}
```

### Output

```
integral =     1.000
exact    =     1.000
error estimate  = 1.803637e-04
exact error     = 1.013279e-06
The number of function evaluations = 405
```

## Warning Errors

| | |
|---|---|
| IMSL_BAD_INTEGRAND_BEHAVIOR | Bad integrand behavior occurred in one or more cycles. |
| IMSL_EXTRAPOLATION_PROBLEMS | Extrapolation table constructed for convergence acceleration of the series formed by the integral contributions of the cycles does not converge to the requested accuracy. |

## Fatal Errors

| | |
|---|---|
| IMSL_MAX_CYCLES | Maximum number of cycles allowed has been reached. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_cauchy

more...

Computes integrals of the form

$$\int_a^b \frac{f(x)}{x-c}dx$$

in the Cauchy principal value sense.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_cauchy` (*float* `fcn()`, *float* `a`, *float* `b`, *float* `c`, …, 0)

The type ***double*** function is `imsl_d_int_fcn_cauchy`.

## Required Arguments

*float* `fcn` (*float* `x`) (Input)
User-supplied function to be integrated.

*float* `a` (Input)
Lower limit of integration.

*float* `b` (Input)
Upper limit of integration.

*float* `c` (Input)
Singular point, ***c*** must not equal ***a*** or ***b***.

## Return Value

The value of

$$\int_a^b \frac{fcn(x)}{x-c}dx$$

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_cauchy` (*float* `fcn ()`, *float* `a`, *float* `b`, *float* `c`,

`IMSL_ERR_ABS`, *float* `err_abs`,

`IMSL_ERR_REL`, *float* `err_rel`,

`IMSL_ERR_EST`, *float* `*err_est`,

`IMSL_MAX_SUBINTER`, *int* `max_subinter`,

`IMSL_N_SUBINTER`, *int* `*n_subinter`,

`IMSL_N_EVALS`, *int* `*n_evals`,

`IMSL_FCN_W_DATA`, *float* `fcn ()`, *void* `*data`,

0)

## Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$ , where **ε** is the machine precision

`IMSL_ERR_REL`, *float* `err_rel` (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$ , where **ε** is the machine precision

`IMSL_ERR_EST`, *float* `*err_est` (Output)
Address to store an estimate of the absolute value of the error.

`IMSL_MAX_SUBINTER`, *int* `max_subinter` (Input)
Number of subintervals allowed.
Default: `max_subinter` = 500

`IMSL_N_SUBINTER`, *int* `*n_subinter` (Output)
Address to store the number of subintervals generated.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)
Address to store the number of evaluations of `fcn`.

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

## Description

The function `imsl_f_int_fcn_cauchy` uses a globally adaptive scheme in an attempt to reduce the absolute error. It computes integrals whose integrands have the special form $w(x)f(x)$ where $w(x) = 1/(x - c)$. If $c$ lies in the interval of integration, then the integral is interpreted as a Cauchy principal value. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas are employed.

On some platforms, `imsl_f_int_fcn_cauchy` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function `imsl_f_int_fcn_cauchy` is an implementation of the subroutine QAWC by Piessens et al. (1983).

## Examples

### Example 1

The Cauchy principal value of

$$\int_{-1}^{5} \frac{1}{x(5x^3+6)}dx = \frac{\ln(125/631)}{18}$$

is computed.

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

float fcn(float x);

int main()
{
   float q, exact;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_cauchy (fcn, -1.0, 5.0, 0.0, 0);

   /* Print the result and the */
   /* exact answer */
   exact = log(125./631.)/18.;
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}
```

```
float fcn(float x)
{
   return 1.0/(5.0*x*x*x+6.0);
}
```

### Output

```
integral =    -0.090
exact    =    -0.090
```

## Example 2

The Cauchy principal value of

$$\int_{-1}^{5} \frac{1}{x(5x^3+6)} dx = \frac{\ln(125/631)}{18}$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate,
as is the case in this example. The number of function evaluations also are printed.

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

float          fcn(float x);

int main()
{
   int        n_evals;
   float q, exact, err_est, exact_err;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
   /* Evaluate the integral */
   q = imsl_f_int_fcn_cauchy (fcn, -1.0, 5.0, 0.0,
       IMSL_ERR_EST, &err_est,
       IMSL_N_EVALS, &n_evals,
       0);

   /* Print the result and the */
   /* exact answer */
   exact = log(125./631.)/18.;
   exact_err = fabs(exact - q);
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
   printf("error estimate  = %e\nexact error     = %e\n", err_est,
       exact_err);
```

```
   printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x)
{
   return 1.0/(5.0*x*x*x+6.0);
}
```

### Output

```
integral =    -0.090
exact    =    -0.090
error estimate  = 2.160174e-06
exact error     = 0.000000e+00
The number of function evaluations = 215
```

## Warning Errors

| | |
|---|---|
| `IMSL_ROUNDOFF_CONTAMINATION` | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| `IMSL_PRECISION_DEGRADATION` | A degradation in precision has been detected. |

## Fatal Errors

| | |
|---|---|
| `IMSL_MAX_SUBINTERVALS` | The maximum number of subintervals allowed has been reached. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_smooth

more...

Integrates a smooth function using a nonadaptive rule.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_smooth (*float* fcn (), *float* a, *float* b, …, 0)

The type *double* function is imsl_d_int_fcn_smooth.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of integration.

*float* b (Input)
Upper limit of integration.

## Return Value

The value of

$$\int_a^b fcn(x)\,dx$$

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_smooth (*float* fcn (), *float* a, *float* b,

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$, where **ε** is the machine precision

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$, where **ε** is the machine precision

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function imsl_f_int_fcn_smooth is designed to integrate smooth functions. It implements a nonadaptive quadrature procedure based on nested Paterson rules of order 10, 21, 43, and 87. These rules are positive quadrature rules with degree of accuracy 19, 31, 64, and 130, respectively. The function imsl_f_int_fcn_smooth applies these rules successively, estimating the error, until either the error estimate satisfies the user-supplied constraints or the last rule is applied.

This function is not very robust, but for certain smooth functions it can be efficient. If imsl_f_int_fcn_smooth should not perform well, we recommend the use of the function imsl_f_int_fcn_sing.

On some platforms, imsl_f_int_fcn_smooth can evaluate the user-supplied function fcn in parallel. This is done only if the function imsl_omp_options is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

The function imsl_f_int_fcn_smooth is based on the subroutine QNG by Piessens et al. (1983).

# Examples

## Example 1

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is computed.

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

float fcn(float x);

int main()
{
   float q, exact;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_smooth (fcn, 0., 2.,
       0);

   /* Print the result and the */
   /* exact answer */
   exact = exp(2.0) + 1.0;
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
   return x * exp(x);
}
```

### Output

```
integral =     8.389
exact    =     8.389
```

## Example 2

The value of

$$\int_0^2 xe^x dx = e^2 + 1$$

is again computed. The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example.

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

float fcn(float x);

int main()
{
   float q, exact, err_est, exact_err;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Evaluate the integral */
   q = imsl_f_int_fcn_smooth (fcn, 0.0, 2.0,
       IMSL_ERR_EST, &err_est,
       0);

   /* Print the result and the */
   /* exact answer */
   exact = exp(2.0) + 1.0;
   exact_err = fabs(exact - q);
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
   printf("error estimate  = %e\nexact error     = %e\n", err_est,
       exact_err);
}

float fcn(float x)
{
   return x * exp(x);
}
```

### Output

```
integral =     8.389
exact    =     8.389
error estimate  = 5.000267e-05
exact error     = 9.536743e-07
```

## Fatal Errors

| | |
|---|---|
| `IMSL_MAX_STEPS` | The maximum number of steps allowed have been taken. The integrand is too difficult for this routine. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_2d

Computes a two-dimensional iterated integral.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_2d (*float* fcn (), *float* a, *float* b, *float* gcn (*float* x), *float* hcn (*float* x), …, 0)

The type *double* function is imsl_d_int_fcn_2d.

## Required Arguments

*float* fcn (*float* x, *float* y) (Input)
User-supplied function to be integrated.

*float* a (Input)
Lower limit of outer integral.

*float* b (Input)
Upper limit of outer integral.

*float* gcn (*float* x) (Input)
User-supplied function to evaluate the lower limit of the inner integral.

*float* hcn (*float* x) (Input)
User-supplied function to evaluate the upper limit of the inner integral.

## Return Value

The value of

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

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_int_fcn_2d (*float* fcn (), *float* a, *float* b, *float* gcn (), *float* hcn (),

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_SUBINTER, *int* max_subinter,

IMSL_N_SUBINTER, *int* *n_subinter,

IMSL_N_EVALS, *int* *n_evals,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

IMSL_GCN_W_DATA, *float* gcn(), *void* *data,

IMSL_HCN_W_DATA, *float* hcn(), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: $err\_abs = \sqrt{\varepsilon}$, where **ε** is the machine precision.

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: $err\_rel = \sqrt{\varepsilon}$, where **ε** is the machine precision.

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_SUBINTER, *int* max_subinter (Input)
Number of subintervals allowed.
Default: max_subinter = 500

IMSL_N_SUBINTER, *int* *n_subinter (Output)
Address to store the number of subintervals generated.

IMSL_N_EVALS, *int* *n_evals (Output)
Address to store the number of evaluations of fcn.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *float* y, *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_GCN_W_DATA`, *float* `gcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

User supplied function to evaluate the lower limit of the inner integral, which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_HCN_W_DATA`, *float* `hcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

User supplied function to evaluate the upper limit of the inner integral, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

The function `imsl_f_int_fcn_2d` approximates the two-dimensional iterated integral

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

An estimate of the error is returned in `err_est`. The lower-numbered rules are used for less smooth integrands while the higher-order rules are more efficient for smooth (oscillatory) integrands.

# Examples

## Example 1

In this example, compute the value of the integral

$$\int_0^1 \int_1^3 y \cos\left(x + y^2\right) dy dx$$

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

float           fcn(float x, float y), gcn(float x), hcn(float x);

int main()
{
   float       q, exact;

                                /* Evaluate the integral */
   q = imsl_f_int_fcn_2d (fcn, 0.0, 1.0, gcn, hcn, 0);
                        /* print the result and the exact answer */
   exact = 0.5*(cos(9.0)+cos(2.0)-cos(10.0)-cos(1.0));
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}
```

```
float fcn(float x, float y)
{
   return y * cos(x+y*y);
}

float gcn(float x)
{
   return 1.0;
}

float hcn(float x)
{
   return 3.0;
}
```

### Output

```
integral =    -0.514
exact    =    -0.514
```

## Example 2

In this example, compute the value of the integral

$$\int_0^1 \int_1^3 y\cos\left(x + y^2\right) dydx$$

The values of the actual and estimated error are printed as well. Note that these numbers are machine dependent. Furthermore, the error estimate is usually pessimistic. That is, the actual error is usually smaller than the error estimate, as is the case in this example. The number of function evaluations also is printed.

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

float fcn(float x, float y), gcn(float x), hcn(float x);

int main()
{
   int   n_evals;
   float q, exact, err_est, exact_err;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
   /* Evaluate the integral */
   q = imsl_f_int_fcn_2d (fcn, 0., 1., gcn, hcn,
       IMSL_ERR_EST, &err_est,
```

```
        IMSL_N_EVALS, &n_evals,
        0);
    /* Print the result and the */
    /* exact answer */
    exact = 0.5*(cos(9.0)+cos(2.0)-cos(10.0)-cos(1.0));
    exact_err = fabs(exact - q);

    printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
    printf("error estimate  = %e\nexact error     = %e\n", err_est,
        exact_err);
    printf("The number of function evaluations = %d\n", n_evals);
}

float fcn(float x, float y)
{
    return y * cos(x+y*y);
}

float gcn(float x)
{
    return 1.0;
}

float hcn(float x)
{
    return 3.0;
}
```

### Output

```
integral =     -0.514
exact    =     -0.514
error estimate  = 3.065193e-06
exact error     = 1.192093e-07
The number of function evaluations = 441
```

# Warning Errors

| | |
|---|---|
| `IMSL_ROUNDOFF_CONTAMINATION` | Roundoff error, preventing the requested tolerance from being achieved, has been detected. |
| `IMSL_PRECISION_DEGRADATION` | A degradation in precision has been detected. |

# Fatal Errors

| | |
|---|---|
| `IMSL_MAX_SUBINTERVALS` | The maximum number of subintervals allowed has been reached. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_sing_2d

Integrates a function of two variables with a possible internal or endpoint singularity.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_sing_2d (*float* fcn (), *float* a, float b, *float* gcn (), *float* hcn (), …, 0)

The type *double* function is imsl_d_int_fcn_sing_2d.

## Required Arguments

*float* fcn (*float* x, *float* y) (Input/Output)
User-supplied function to be integrated.

**Arguments**

*float* x (Input)
Independent variable.

*float* y (Input)
Independent variable.

**Return Value**

The computed function value.

*float* a (Input)
Lower limit of integration for outer dimension.

*float* b (Input)
Upper limit of integration for outer dimension. The relative values of a and b are interpreted properly. Thus if one exchanges a and b, the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of b − a.

*float* gcn (*float* x) (Input/Output)
User-supplied function to compute the lower limit of integration for the inner dimension.

**Arguments**

*float* x (Input)
Independent variable.

**Return Value**

The computed function value at the point x.

*float* `hcn` (*float* `x`) (Input/Output)

User-supplied function to compute the upper limit of integration for the inner dimension.

**Arguments**

*float* x (Input)

Independent variable.

**Return Value**

The computed function value at the point `x`.

# Return Value

An estimate of

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

# Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_sing_2d` (*float* `fcn ()`, *float* `a`, *float* `b`, *float* `gcn ()`, *float* `hcn ()`,

`IMSL_FCN_W_DATA`, *float* `fcn ()`, *float* *err_post, *void* `*data`,

`IMSL_GCN_W_DATA`, *float* `gcn ()`, *void* `*data`,

`IMSL_HCN_W_DATA`, *float* `hcn ()`, *void* `*data`,

`IMSL_ERR_ABS`, *float* `err_abs`,

`IMSL_ERR_FRAC`, *float* `err_frac`,

`IMSL_ERR_REL`, *float* `err_rel`,

`IMSL_ERR_PRIOR`, *float* `err_prior`,

`IMSL_MAX_EVALS`, *int* `maxfn`,

`IMSL_SINGULARITY`, *float* `singularity`, *int* `singularity_type`,

`IMSL_N_EVALS`, *int* `*n_evals`,

`IMSL_ERR_EST`, *float* `*err_est`,

`IMSL_ISTATUS`, *int* `*istatus`,

0)

## Optional Arguments

IMSL_FCN_W_DATA, *float* fcn (*float* x, *float* y, *float* *err_post, *void* *data), *float* *err_post, *void* *data (Input)

*float* fcn (*float* x, *float* y, *float* *err_post, *void* *data) (Input)

User supplied function to be integrated, which also accepts a pointer to an a posteriori estimate of the absolute value of the error committed while evaluating the integrand, and a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* x (Input)

Independent variable.

*float* y (Input)

Independent variable.

*float* *err_post (Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. This argument provides a means for the user to have fcn compute this value as output. Although this argument must appear in the argument list of fcn, it need not be referenced in the function. See Example 2 of int_fcn_sing_1d for an example of this.

*void* *data (Input)

A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value.

*float* *err_post (Input/Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. On input, the user may supply this estimate and that value will be used as the estimate thereafter provided fcn does not calculate a new value. If an a posteriori estimate of the value of the error is not known, set err_post to 0.0 on input. On output, err_post will contain either the input value set by the user or the value calculated by fcn.

*void* *data (Input)

A pointer to the data to be passed to the user-supplied function.

IMSL_GCN_W_DATA, *float* gcn (*float* x, *void* *data), *void* *data (Input)

*float* gcn (*float* x, *void* *data) (Input)

User supplied function to compute the lower limit of integration for the inner dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

#### Arguments

*float* `x` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

#### Return Value

The computed function value at the point `x`.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_HCN_W_DATA`, *float* `hcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

*float* `hcn` (*float* `x`, *void* `*data`) (Input)
User supplied function to compute the upper limit of integration for the inner dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

#### Arguments

*float* `x` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

#### Return Value

The computed function value at the point `x`.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute error tolerance. See Remark 1 for a discussion on the error tolerances.
Default: `err_abs` = 0.0

`IMSL_ERR_FRAC`, *float* `err_frac` (Input)
A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See Remark 1 for a discussion on accuracy.
Default: `err_frac` = 0.75

`IMSL_ERR_REL`, *float* `err_rel` (Input)
The error tolerance relative to the value of the integral. See Remark 1 for a discussion on the error tolerances.
Default: `err_rel` = 0.0

IMSL_ERR_PRIOR, *float* err_prior (Input)

An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral.

Default: Default: err_prior = imsl_f_machine(4)

IMSL_MAX_EVALS, *int* maxfn (Input)

The maximum number of function evaluations to use to compute the integral.

Default: The number of function values is not bounded.

IMSL_SINGULARITY, *float* singularity, *int* singularity_type (Input)

singularity is the real part of the abscissa of a singularity or discontinuity in the innermost integrand. singularity_type is a signed integer specifying the type of singularity which occurs in the integrand. If the singularity has a leading term of the form $x^{\alpha}$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n-1)/2$ where $n$ is a nonnegative integer, or the singularity is well outside the interval, set singularity_type to a positive integer. Otherwise, set singularity_type to a negative integer. Also see Remark 2.

Default: It is assumed that there is no singularity in the innermost integrand so singularity and singularity_type are not set.

IMSL_N_EVALS, *int* *n_evals (Output)

Number of function evaluations used to calculate the integral.

IMSL_ERR_EST, *float* *err_est (Output)

An estimate of the upper bound of the magnitude of the difference between the value returned by imsl_f_int_fcn_sing_2d and the true value of the integral.

IMSL_ISTATUS, *int* *istatus (Output)

A status flag indicating the error criteria which was satisfied on exit.

| istatus | Description |
|---|---|
| -2 | Indicates normal termination with either the absolute or relative error tolerance criteria satisfied. |
| -3 | Indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied. |
| -4 | Indicates normal termination with none of the error tolerance criteria satisfied. |
| Other | Any value other than the above indicates abnormal termination due to an error condition. |

## Description

The function imsl_f_int_fcn_sing_2d, based on the JPL Library routine SINTM, approximates an iterated two-dimensional integral of the form

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

The integral over two dimensions is computed by repeated integration over one dimension. The integration over one dimension is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the $k^{th}$ formula used all the integrand values used in the $k\text{-}1^{st}$ formula, and added $2^{k-1}$ new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to $k = 8$.

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

# Remarks

**Remark 1**

The user provides the absolute error tolerance through optional argument `IMSL_ERR_ABS`. Optional argument `IMSL_ERR_FRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). Optional argument `IMSL_ERR_REL` represents the error tolerance relative to the value of the integral. The internal value for `err_frac` is bounded between .5 and 1. By default, `err_abs` and `err_rel` are set to 0.0 and `err_frac` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.

The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.

**Remark 2**

Optional argument `IMSL_SINGULARITY` provides the user with a means to give the routine information about the location and type of any known singularity of the innermost integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside `[a,b]`. If the singularity has a leading term of the form $x^{\alpha}$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n - 1)/2$ where n is a nonnegative integer, or the singularity is well outside the interval, set `singularity_type` to a positive value. Otherwise, set `singularity_type` to a negative value. The meaning of "large" depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered "large". For a singularity of the form $x^{\alpha} \log x$ use the above rule, even if $\alpha$ is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed, some experimentation may be useful.

When `singularity_type` is positive, a transformation of the form $T = TA + (X - TA)^2 / (TB - TA)$ is applied, where *TA* is the abscissa of the singularity and *TB* is the end of the interval. If *TA* is outside the interval, *TB* will be the end of the interval farthest from *TA*. If *TA* is inside the interval, the interval will immediately be subdivided at *TA*, and both parts will be separately integrated with *TB* equal to each end of the original interval, respectively. When `singularity_type` is negative, a transformation of the form $T = TA + (X - TA)^4 / (TB - TA)^3$ is applied, with *TA* and *TB* as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

# Example

The value of

$$\int_0^1 \int_1^3 y \cos\left(x + y^2\right) dy\, dx$$

is estimated.

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

float fcn (float x, float y);
float gcn (float x);
float hcn (float x);


int main(){

   float a=0.0, b=1.0, errest, result;

   result = imsl_f_int_fcn_sing_2d(fcn, a, b, gcn, hcn,
            IMSL_ERR_EST, &errest, 0);
   printf("The approximation to the integral is %f\n", result);
   printf("The estimated error is %6.1e\n", errest);
}


float fcn (float x, float y) {
   return y*cos(x+y*y);
}

float gcn (float x) {
   return 1.0;
}

float hcn (float x) {
   return 3.0;
}
```

# Output

```
The approximation to the integral is -0.514254
The estimated error is 5.3e-006
```

## Fatal Errors

| | |
|---|---|
| `IMSL_NONINTEGRABLE` | The integrand apparently contains a nonintegrable singularity. The abscissa of the singularity is near #. The result has been set to NaN. |
| `IMSL_MAX_FCN_EVAL_EXCEEDED_NAN` | The maximum number of function evaluations allowed, "`maxfn`", has been exceeded. "`maxfn`" is currently set to a value of #. The result has been set to NaN. |
| `IMSL_CRITERIA_NOT_SATISFIED` | The algorithm has terminated without satisfying any of the error tolerance criteria. The error estimate is #. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_sing_3d

Integrates a function of three variables with a possible internal or endpoint singularity.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_sing_3d` (*float* `fcn ()`, *float* `a`, *float* `b`, *float* `gcn ()`, *float* `hcn ()`, *float* `pcn ()`, *float* `qcn ()`, …, 0)

The type ***double*** function is `imsl_d_int_fcn_sing_3d`.

## Required Arguments

*float* `fcn` (*float* `x`, *float* `y`, *float* `z`) (Input/Output)
User-supplied function to be integrated.

**Arguments**

*float* `x` (Input)
Independent variable..

*float* `y` (Input)
Independent variable.

*float* `z` (Input)
Independent variable.

**Return Value**

The computed function value.

*float* `a` (Input)
Lower limit of integration for outer dimension.

*float* `b` (Input)
Upper limit of integration for outer dimension. The relative values of `a` and `b` are interpreted properly. Thus if one exchanges `a` and `b`, the sign of the answer is changed. When the integrand is positive, the sign of the result is the same as the sign of `b` − `a`.

*float* `gcn` (*float* `x`) (Input/Output)
User-supplied function to compute the lower limit of integration for the middle dimension.

**Arguments**

*float* `x` (Input)
Independent variable.

**Return Value**

The computed function value at the point `x`.

*float* `hcn` (*float* `x`) (Input/Output)

User-supplied function to compute the upper limit of integration for the middle dimension.

**Arguments**

*float* `x` (Input)

Independent variable..

**Return Value**

The computed function value.

*float* `pcn` (*float* `x`, *float* `y`) (Input/Output)

User-supplied function to compute the lower limit of integration for the inner dimension.

**Arguments**

*float* `x` (Input)

Independent variable.

*float* `y` (Input)

Independent variable.

**Return Value**

The computed function value.

*float* `qcn` (*float* `x`, *float* `y`) (Input/Output)

User-supplied function to compute the upper limit of integration for the inner dimension.

**Arguments**

*float* `x` (Input)

Independent variable.

*float* `y` (Input)

Independent variable.

**Return Value**

The computed function value.

## Return Value

An estimate of

$$\int_a^b \int_{gcn(x)}^{hcn(x)} \int_{pcn(x,\,y)}^{qcn(x,\,y)} f\left(x,\, y,\, z\right) dzdydx$$

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_sing_3d` (*float* `fcn ()`, *float* `a`, *float* `b`, *float* `gcn ()`, *float* `hcn ()`, *float* `pcn ()`, *float* `qcn ()`,

`IMSL_FCN_W_DATA`, *float* `fcn ()`, *float* *err_post, *void* `*data`,

`IMSL_GCN_W_DATA`, *float* `gcn ()`, *void* `*data`,

`IMSL_HCN_W_DATA`, *float* `hcn ()`, *void* `*data`,

`IMSL_PCN_W_DATA`, *float* `pcn ()`, *void* `*data`,

`IMSL_QCN_W_DATA`, *float* `qcn ()`, *void* `*data`,

`IMSL_ERR_ABS`, *float* `err_abs`,

`IMSL_ERR_FRAC`, *float* `err_frac`,

`IMSL_ERR_REL`, *float* `err_rel`,

`IMSL_ERR_PRIOR`, *float* `err_prior`,

`IMSL_MAX_EVALS`, *int* `maxfn`,

`IMSL_SINGULARITY`, *float* `singularity`, *int* `singularity_type`,

`IMSL_N_EVALS`, *int* `*n_evals`,

`IMSL_ERR_EST`, *float* `*err_est`,

`IMSL_ISTATUS`, *int* `*istatus`,

0)

## Optional Arguments

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *float* `y`, *float* `z`, *float* `*err_post`, *void* `*data`),
*float* `*err_post`, *void* `*data` (Input)

*float* `fcn` (*float* `x`, *float* `y`, *float* `z`, *float* `*err_post`, *void* `*data`) (Input)

User supplied function to be integrated, which also accepts a pointer to an a posteriori estimate of the absolute value of the error committed while evaluating the integrand, and a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* `x` (Input)

Independent variable.

*float* `y` (Input)

Independent variable.

*float* `z` (Input)

Independent variable.

*float* `*err_post` (Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand This argument provides a means for the user to have `fcn` compute this value as output. Although this argument must appear in the argument list of `fcn`, it need not be referenced in the function. See Example 2 of `int_fcn_sing_1d` for an example of this.

*void* `*data` (Input)

A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value.

*float* `*err_post` (Input/Output)

An a posteriori estimate of the absolute value of the error committed while evaluating the integrand. On input, the user may supply this estimate and that value will be used as the estimate thereafter provided `fcn` does not calculate a new value. If an a posteriori estimate of the value of the error is not known, set `err_post` to 0.0 on input. On output, `err_post` will contain either the input value set by the user or the value calculated by `fcn`.

*void* `*data` (Input)

A pointer to the data to be passed to the user-supplied function.

`IMSL_GCN_W_DATA`, *float* `gcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

*float* `gcn` (*float* `x`, *void* `*data`) (Input)

User supplied function to compute the lower limit of integration for the middle dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* `x` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point `x`.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_HCN_W_DATA`, *float* `hcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

*float* `hcn` (*float* `x`, *void* `*data`) (Input)
User supplied function to compute the upper limit of integration for the middle dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* `x` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point `x`.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_PCN_W_DATA`, *float* `pcn` (*float* `x`, *float* `y`, *void* `*data`), *void* `*data` (Input)

*float* `pcn` (*float* `x`, *float* `y`, *void* `*data`) (Input)
User supplied function to compute the lower limit of integration for the inner dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* `x` (Input)
Independent variable.

*float* `y` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_QCN_W_DATA`, *float* `qcn` (*float* `x`, *float* `y`, *void* `*data`), *void* `*data` (Input)

*float* `qcn` (*float* `x`, *float* `y`, *void* `*data`) (Input)
User supplied function to compute the lower limit of integration for the inner dimension which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* `x` (Input)
Independent variable.

*float* `y` (Input)
Independent variable.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value.

*void* `*data` (Input)
A pointer to the data to be passed to the user-supplied function.

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
Absolute error tolerance. See Remark 1 for a discussion on the error tolerances.
Default: `err_abs` = 0.0

`IMSL_ERR_FRAC`, *float* `err_frac` (Input)
A fraction expressing the (number of correct digits of accuracy desired)/(number of digits of achievable precision). See Remark 1 for a discussion on accuracy.
Default: `err_frac` = 0.75

`IMSL_ERR_REL`, *float* `err_rel` (Input)
The error tolerance relative to the value of the integral. See Remark 1 for a discussion on the error tolerances.
Default: `err_rel` = 0.0

`IMSL_ERR_PRIOR`, *float* `err_prior` (Input)
An a priori estimate of the absolute value of the relative error expected to be committed while evaluating the integrand. Changes to this value are not detected during evaluation of the integral.
Default: `err_prior = imsl_f_machine(4)`

`IMSL_MAX_EVALS`, *float* `maxfn` (Input)

The maximum number of function evaluations to use to compute the integral.

Default: The number of function values is not bounded.

`IMSL_SINGULARITY`, *float* `singularity`, *int* `singularity_type` (Input)

`singularity` is the real part of the abscissa of a singularity or discontinuity in the innermost integrand. By default it is assumed that there is no singularity in the integrand. `singularity_type` is a signed integer specifying the type of singularity which occurs in the integrand. If the singularity has a leading term of the form $x^\alpha$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n-1)/2$ where $n$ is a nonnegative integer, or the singularity is well outside the interval, set `singularity_type` to a positive integer. Otherwise, set `singularity_type` to a negative integer. Also see Remark 2.

Default: It is assumed that there is no singularity in the innermost integrand so `singularity` and `singularity_type` are not set.

`IMSL_N_EVALS`, *int* `*n_evals` (Output)

Number of function evaluations used to calculate the integral.

`IMSL_ERR_EST`, *float* `*err_est` (Output)

An estimate of the upper bound of the magnitude of the difference between the value returned by `imsl_f_int_fcn_sing_3d` and the true value of the integral.

`IMSL_ISTATUS`, *int* `*istatus` (Output)

A status flag indicating the error criteria which was satisfied on exit.

| istatus | Description |
|---|---|
| -3 | Indicates normal termination with either the absolute or relative error tolerance criteria satisfied. |
| -4 | Indicates normal termination with neither the absolute nor the relative error tolerance criteria satisfied, but the error tolerance based on the locally achievable precision is satisfied. |
| -5 | Indicates normal termination with none of the error tolerance criteria satisfied. |
| Other | Any value other than the above indicates abnormal termination due to an error condition. |

## Description

The function `imsl_f_int_fcn_sing_3d`, based on the JPL Library routine `SINTM`, approximates an iterated three-dimensional integral of the form

$$\int_a^b \int_{g(x)}^{h(x)} \int_{p(x,y)}^{q(x,y)} f(x, y, z)\, dz dy dx$$

The integral over three dimensions is computed by repeated integration over one dimension. The integration over one dimension is estimated using quadrature formulae due to T. N. L. Patterson (1968). Patterson described a family of formulae in which the $k^{th}$ formula used all the integrand values used in the $k-1^{st}$ formula, and added $2^{k-1}$ new integrand values in an optimal way. The first formula is the midpoint rule, the second is the three point Gauss formula, and the third is the seven point Kronrod formula. Formulae of this family of higher degree had not previously been described. This program uses formulae up to $k = 8$.

An error estimate is obtained by comparing the values of the integral estimated by two adjacent formulae, examining differences up to the fifteenth order, integrating round-off error, integrating error declared to have been committed during computation of the integrand, integrating a first order estimate of the effect round-off error in the abscissa has on integrand values, and including errors in the limits. The latter four methods are also used to derive a bound on the achievable precision.

If the integral over an interval cannot be estimated with sufficient accuracy, the interval is subdivided. The difference table is used to discover whether the integral is difficult to compute because the integrand is too complex or has singular behavior. In the former case, the estimated error, requested error tolerance, and difference table are used to choose a step size.

In the latter case, the difference table is used in a search algorithm to find the abscissa of the singular behavior. If the singular behavior is discovered on the end of an interval, a change of independent variable is applied to reduce the strength of the singularity.

The program also uses the difference table to detect nonintegrable singularities, jump discontinuities, and computational noise.

# Remarks

### Remark 1

The user provides the absolute error tolerance through optional argument `IMSL_ERR_ABS`. Optional argument `IMSL_ERR_FRAC` represents the ratio of the (number of correct digits of accuracy desired) to (number of digits of achievable precision). Optional argument `IMSL_ERR_REL` represents the error tolerance relative to the value of the integral. The internal value for `err_frac` is bounded between .5 and 1. By default, `err_abs` and `err_rel` are set to 0.0 and `err_frac` is set to .75. These default values usually provide all the accuracy that can be obtained efficiently.

The error tolerance relative to the value of the integral is applied globally (over the entire region of integration) rather than locally (one step at a time). This policy provides true control of error relative to the value of the integral when the integrand is not sign definite, as well as when the integrand is sign definite. To apply the criterion of error tolerance relative to the value of the integral, the value of the integral over the entire region, estimated without refinement of the region, is used to derive an absolute error tolerance that may be applied locally. If the preliminary estimate of the value of the integral is significantly in error, and the least restrictive error tolerance is relative to the value of the integral, the cost of computing the integral will be larger than the cost of computing the integral to the same degree of accuracy using appropriate values of either of the other tolerance criteria. The preliminary estimate of the integral may be significantly in error if the integrand is not sign definite or has large variation.

**Remark 2**

Optional argument `IMSL_SINGULARITY` provides the user with a means to give the routine information about the location and type of any known singularity of the innermost integrand. When an integrand appears to have singular behavior at the end of the interval, a transformation of the variable of integration is applied to reduce the strength of the singularity. When an integrand appears to have singular behavior inside the interval, the abscissa of the singularity is determined as precisely as necessary, depending on the error tolerance, and the interval is subdivided. The discovery of singular behavior and determination of the abscissa of singular behavior are expensive. If the user knows of the existence of a singularity, the efficiency of computation of the integral may be improved by requesting an immediate transformation of the independent variable or subdivision of the interval. It is recommended that the user select these optional arguments for all singularities, even those outside `[a, b]`. If the singularity has a leading term of the form $x^{\alpha}$ where $\alpha$ is not an integer, if $\alpha$ is "large" or has the form $\alpha = (2n-1)/2$ where n is a nonnegative integer, or the singularity is well outside the interval, set `singularity_type` to a positive value. Otherwise, set `singularity_type` to a negative value. The meaning of "large" depends on the rest of the integrand and the length of the interval. For the typical case, a value of about 2 is considered "large". For a singularity of the form $x^{\alpha} \log x$ use the above rule, even if $\alpha$ is an integer. For other types of singularities make a reasonable guess based on the above. If several similar integrals are to be computed, some experimentation may be useful.

When `singularity_type` is positive, a transformation of the form $T = TA + (X - TA)^2 / (TB - TA)$ is applied, where *TA* is the abscissa of the singularity and *TB* is the end of the interval. If *TA* is outside the interval, *TB* will be the end of the interval farthest from *TA*. If *TA* is inside the interval, the interval will immediately be subdivided at *TA*, and both parts will be separately integrated with *TB* equal to each end of the original interval, respectively. When `singularity_type` is negative, a transformation of the form $T = TA + (X - TA)^4 / (TB - TA)^3$ is applied, with *TA* and *TB* as above.

If the integrand has singularities at more than one abscissa within the region, or more than one pole near the real axis such that the real parts are within the region of integration, then the interval should be subdivided at the abscissa of the singularities or the real parts of the poles, and the integrals should be computed as separate problems, with the results summed.

## Example

The value of

$$\int_0^1 \int_0^{1-x} \int_0^{1-x-y} \left(1.0 + x + y + 2z\right) dz\,dy\,dx$$

is estimated.

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

```
float fcn (float x, float y, float z);
float gcn (float x);
float hcn (float x);
float pcn (float x, float y);
float qcn (float x, float y);


int main(){
   float a=0.0, b=1.0, errest, result;

   result = imsl_f_int_fcn_sing_3d(fcn, a, b, gcn, hcn, pcn, qcn,
            IMSL_ERR_EST, &errest, 0);
   printf("The approximation to the integral is %f\n", result);
   printf("The estimated error is %6.1e\n", errest);
}


float fcn (float x, float y, float z) {
   return 1.0 + x + y + 2.0*z;
}

float gcn (float x) {
   return 0.0;
}

float hcn (float x) {
   return 1.0 - x;
}

float pcn (float x, float y) {
   return 0.0;
}

float qcn (float x, float y) {
   return 1.0 - x - y;
}
```

## Output

```
The approximation to the integral is 0.333333
The estimated error is 1.9e-007
```

## Fatal Errors

| | |
|---|---|
| `IMSL_NONINTEGRABLE` | The integrand apparently contains a nonintegrable singularity. The abscissa of the singularity is near #. The result has been set to NaN. |
| `IMSL_MAX_FCN_EVAL_EXCEEDED_NAN` | The maximum number of function evaluations allowed, “`maxfn`”, has been exceeded. “`maxfn`” is currently set to a value of #. The result has been set to NaN. |
| `IMSL_CRITERIA_NOT_SATISFIED` | The algorithm has terminated without satisfying any of the error tolerance criteria. The error estimate is #. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# int_fcn_hyper_rect

more...

Integrate a function on a hyper-rectangle,

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

## Synopsis

*#include* <imsl.h>

*float* `imsl_f_int_fcn_hyper_rect` (*float* `fcn ()`, *int* `ndim`, *float* `a []`, *float* `b []`, …, 0)

The type ***double*** function is `imsl_d_int_fcn_hyper_rect`.

## Required Arguments

*float* `fcn` (*int* `ndim`, *float* `x []`) (Input)
User-supplied function to be integrated.

*int* `ndim` (Input)
The dimension of the hyper-rectangle.

*float* `a []` (Input)
Lower limits of integration.

*float* `b []` (Input)
Upper limits of integration.

## Return Value

The value of

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

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

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `imsl_f_int_fcn_hyper_rect` (*float* `fcn ()`, *int* `ndim`, *float* `a []`, *float* `b []`,

`IMSL_ERR_ABS`, *float* `err_abs`,

`IMSL_ERR_REL`, *float* `err_rel`,

`IMSL_ERR_EST`, *float* `*err_est`,

`IMSL_MAX_EVALS`, *int* `max_evals`,

`IMSL_FCN_W_DATA`, *float* `fcn ()`, *void* `*data`,

0)

# Optional Arguments

`IMSL_ERR_ABS`, *float* `err_abs` (Input)

Absolute accuracy desired.

Default: $err\_abs = \sqrt{\varepsilon}$, where **ε** is the machine precision.

`IMSL_ERR_REL`, *float* `err_rel` (Input)

Relative accuracy desired.

Default: $err\_rel = \sqrt{\varepsilon}$, where **ε** is the machine precision.

`IMSL_ERR_EST`, *float* `*err_est` (Output)

Address to store an estimate of the absolute value of the error.

`IMSL_MAX_EVALS`, *int* `max_evals` (Input)

Number of evaluations allowed.

Default: `max_evals` = $32^n$.

`IMSL_FCN_W_DATA`, *float* `fcn` (*int* `ndim`, *float* `x []`, *void* `*data`), *void* `*data` (Input)

User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

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

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

An estimate of the error is returned in the optional argument `err_est`. The approximation is achieved by iterated applications of product Gauss formulas. The integral is first estimated by a two-point tensor product formula in each direction. Then for ***i*** = 1, …, ***n***, the function calculates a new estimate by doubling the number of points in the ***i***-th direction, then halving the number immediately afterwards if the new estimate does not change appreciably. This process is repeated until either one complete sweep results in no increase in the number of sample points in any dimension; the number of Gauss points in one direction exceeds 256; or the number of function evaluations needed to complete a sweep exceeds `max_evals`.

On some platforms, `imsl_f_int_fcn_hyper_rect` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

## Example

In this example, we compute the integral of

$$e^{-\left(x_1^2+x_2^2+x_3^2\right)}$$

on an expanding cube. The values of the error estimates are machine dependent. The exact integral over $\mathbf{R}^3$ is $\pi^{3/2}$.

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

float fcn(int n, float x[]);

int main()
{
   int   i, j, ndim = 3;
   float q, limit, a[3], b[3];

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   printf("      integral      limit \n");
   limit = pow(imsl_f_constant("pi",0), 1.5);

   /* Evaluate the integral */
   for (i = 0; i < 6; i++) {
       for (j = 0; j < 3; j++) {
           a[j] = -(i+1)/2.;
           b[j] = (i+1)/2.;
       }
       q = imsl_f_int_fcn_hyper_rect (fcn, ndim, a, b,
           0);

       /* Print the result and the */
       /* limiting answer */
       printf("  %10.3f   %10.3f\n", q, limit);
   }
}

float fcn(int n, float x[])
{
   float    s;

   s = x[0]*x[0] + x[1]*x[1] + x[2]*x[2];
   return exp(-s);
}
```

## Output

```
integral      limit
 0.785        5.568
 3.332        5.568
 5.021        5.568
 5.491        5.568
```

```
5.561        5.568
5.568        5.568
```

## Warning Errors

| | |
|---|---|
| IMSL_MAX_EVALS_TOO_LARGE | The argument max_evals was set greater than $2^{8n}$. |

## Fatal Errors

| | |
|---|---|
| IMSL_NOT_CONVERGENT | The maximum number of function evaluations has been reached, and convergence has not been attained |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# int_fcn_qmc

more...

Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_int_fcn_qmc (*float* fcn (), *int* ndim, *float* a [], *float* b [], …, 0)

The type *double* function is imsl_d_int_fcn_qmc.

## Required Arguments

*float* fcn (*int* ndim, *float* x []) (Input)
User-supplied function to be integrated.

*int* ndim (Input)
The dimension of the hyper-rectangle.

*float* a [] (Input)
Lower limits of integration.

*float* b [] (Input)
Upper limits of integration.

## Return Value

The value of

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

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

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_int_fcn_qmc (*float* fcn (), *int* ndim, *float* a [], *float* b [],

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_REL, *float* err_rel,

IMSL_ERR_EST, *float* *err_est,

IMSL_MAX_EVALS, *int* max_evals,

IMSL_BASE, *int* base,

IMSL_SKIP, *int* skip,

IMSL_FCN_W_DATA, *float* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_ERR_ABS, *float* err_abs (Input)
Absolute accuracy desired.
Default: err_abs = 1.0e-4.

IMSL_ERR_REL, *float* err_rel (Input)
Relative accuracy desired.
Default: err_abs = 1.0e-4.

IMSL_ERR_EST, *float* *err_est (Output)
Address to store an estimate of the absolute value of the error.

IMSL_MAX_EVALS, *int* max_evals (Input)
Number of evaluations allowed.
Default: No limit.

IMSL_BASE, *int* base (Input)
The value of IMSL_BASE used to compute the Faure sequence.

IMSL_SKIP, *int* skip (Input)
The value of IMSL_SKIP used to compute the Faure sequence.

IMSL_FCN_W_DATA, *float* fcn (*int* ndim, *float* x[], *void* *data), *void* *data (Input)
User supplied function to be integrated, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

Integration of functions over hypercubes by direct methods, such as imsl_f_fcn_hyper_rect, is practical only for fairly low dimensional hypercubes. This is because the amount of work required increases exponential as the dimension increases.

An alternative to direct methods is Monte Carlo, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like $1/n^{1/2}$, where $n$ is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to at sequence of random points is a ***low-discrepancy*** sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence as computed by `imsl_f_faure_next_point` (see Chapter 10, "Statistics and Random Number Generation").

On some platforms, `imsl_f_int_fcn_qmc` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

## Example

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

float fcn(int ndim, float x[]);

int main()
{
   int   k, ndim = 10;
   float q, a[10], b[10];

   for (k = 0; k < ndim; k++) {
       a[k] = 0.0;
       b[k] = 1.0;
   }

   q = imsl_f_int_fcn_qmc (fcn, ndim, a, b,
       0);
   printf ("integral=%10.3f\n", q);
}

float fcn (int ndim, float x[])
{
   int   i, j;
   float prod, sum = 0.0, sign = -1.0;

   for (i = 0; i < ndim; i++) {
       prod = 1.0;
       for (j = 0; j <= i; j++) {
           prod *= x[j];
       }
       sum += sign * prod;
       sign = -sign;
   }
   return sum;
}
```

## Output

```
q = -0.333
```

## Fatal Errors

| | |
|---|---|
| `IMSL_NOT_CONVERGENT` | The maximum number of function evaluations has been reached, and convergence has not been attained |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# gauss_quad_rule

Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_gauss_quad_rule (*int* n, *float* weights[], *float* points[], …, 0)

The type *double* procedure is imsl_d_gauss_quad_rule.

## Required Arguments

*int* n (Input)
Number of quadrature points.

*float* weights[] (Output)
Array of length *n* containing the quadrature weights.

*float* points[] (Output)
Array of length *n* containing quadrature points. The default action of this routine is to produce the Gauss Legendre points and weights.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_f_gauss_quad_rule (*int* n, *float* weights[], *float* points[],

IMSL_CHEBYSHEV_FIRST,

IMSL_CHEBYSHEV_SECOND,

IMSL_HERMITE,

IMSL_COSH,

IMSL_JACOBI, *float* alpha, *float* beta,

IMSL_GEN_LAGUERRE, *float* alpha,

IMSL_FIXED_POINT, *float* a,

IMSL_TWO_FIXED_POINTS, *float* a, *float* b,

0)

## Optional Arguments

`IMSL_CHEBYSHEV_FIRST`

Compute the Gauss points and weights using the weight function

$$1/\sqrt{1-x^2}$$

on the interval $(-1, 1)$.

`IMSL_CHEBYSHEV_SECOND`

Compute the Gauss points and weights using the weight function

$$\sqrt{1-x^2}$$

on the interval $(-1, 1)$.

`IMSL_HERMITE`

Compute the Gauss points and weights using the weight function $\exp(-x^2)$ on the interval $(-\infty, \infty)$.

`IMSL_COSH`

Compute the Gauss points and weights using the weight function $1 / (\cosh(x))$ on the interval $(-\infty, \infty)$.

`IMSL_JACOBI`, *float* `alpha`, *float* `beta` (Input)

Compute the Gauss points and weights using the weight function $(1 - x)^a (1 + x)^b$ on the interval $(-1, 1)$.

`IMSL_GEN_LAGUERRE`, *float* `alpha` (Input)

Compute the Gauss points and weights using the weight function $\exp(-x)x^a$ on the interval $(0, \infty)$.

`IMSL_FIXED_POINT`, *float* `a` (Input)

Compute the Gauss-Radau points and weights using the specified weight function and the fixed point *a*. This formula will integrate polynomials of degree less than $2n - 1$ exactly.

`IMSL_TWO_FIXED_POINTS`, *float* `a`, *float* `b` (Input)

Compute the Gauss-Lobatto points and weights using the specified weight function and the fixed points *a* and *b*. This formula will integrate polynomials of degree less than $2n - 2$ exactly.

## Description

The function `imsl_f_gauss_quad_rule` produces the points and weights for the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature formulas for some of the most popular weights. The default weight is the weight function identically equal to 1 on the interval $(-1, 1)$. In fact, it is slightly more general than this suggests, because the extra one or two points that may be specified do not have to lie at the endpoints of the interval. This function is a modification of the subroutine GAUSSQUADRULE (Golub and Welsch 1969).

In the default case, the function returns points in $x$ = `points` and weights in $w$ = weights so that

$$\int_a^b f(x)w(x)dx = \sum_{i=1}^{N} f(x_i)w_i$$

for all functions *f* that are polynomials of degree less than 2*n*.

If the keyword `IMSL_FIXED_POINT` is specified, then one of the above $x_i$ is equal to *a*. Similarly, if the keyword `IMSL_TWO_FIXED_POINTS` is specified, then two of the components of *x* are equal to *a* and *b*. In general, the accuracy of the above quadrature formula degrades when *n* increases. The quadrature rule will integrate all functions *f* that are polynomials of degree less than $2n - F$, where *F* is the number of fixed points.

# Examples

## Example 1

The three-point Gauss Legendre quadrature points and weights are computed and used to approximate the integrals

$$\int_{-1}^{1} x^i dx \quad i = 0, \ldots, 6$$

Notice that the integrals are exact for the first six monomials, but that the last approximation is in error. In general, the Gauss rules with *k* points integrate polynomials with degree less than 2*k* exactly.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define QUADPTS 3
#define POWERS  7

int main()
{
   int        i, j;
   float      weights[QUADPTS], points[QUADPTS], s[POWERS];

   /* Produce the Gauss Legendre */
   /* quadrature points */
   imsl_f_gauss_quad_rule (QUADPTS, weights, points,
       0);

   /* integrate the functions */
   /* 1, x, ..., pow(x,POWERS-1) */
   for(i = 0; i < POWERS; i++) {
       s[i] = 0.0;
       for(j = 0; j < QUADPTS; j++) {
           s[i] += weights[j]*imsl_fi_power(points[j], i);
       }
```

```
    }
    printf("The integral from -1 to 1 of pow(x, i) is\n");
    printf("Function           Quadrature   Exact\n\n");

    for(i = 0; i < POWERS; i++){
        float  z;

        z = (1-i%2)*2./(i+1.);
        printf("pow(x, %d)       %10.3f %10.3f\n", i, s[i], z);
    }
}
```

### Output

```
The integral from -1 to 1 of pow(x, i) is
Function           Quadrature   Exact

pow(x, 0)            2.000      2.000
pow(x, 1)            0.000      0.000
pow(x, 2)            0.667      0.667
pow(x, 3)            0.000      0.000
pow(x, 4)            0.400      0.400
pow(x, 5)            0.000      0.000
pow(x, 6)            0.240      0.286
```

## Example 2

The three-point Gauss Laguerre quadrature points and weights are computed and used to approximate the integrals

$$\int_0^\infty x^i x e^{-x} dx = i! \quad i = 0, \ldots ,6$$

Notice that the integrals are exact for the first six monomials, but that the last approximation is in error. In general, the Gauss rules with *k* points integrate polynomials with degree less than 2*k* exactly.

```
#include <imsl.h>
#include <stdio.h>

#define QUADPTS 3
#define POWERS  7

int main()
{
    int        i, j;
    float      weights[QUADPTS], points[QUADPTS], s[POWERS], z;

    /* Produce the Gauss Legendre */
    /* quadrature points */
```

```
    imsl_f_gauss_quad_rule (QUADPTS, weights, points,
        IMSL_GEN_LAGUERRE, 1.0,
        0);

    /* Integrate the functions */
    /* 1, x, ..., pow(x,POWERS-1) */
    for(i = 0; i < POWERS; i++) {
        s[i] = 0.0;
        for(j = 0; j < QUADPTS; j++){
            s[i] += weights[j]*imsl_fi_power(points[j], i);
        }
    }
    printf("The integral from 0 to infinity of pow(x, i)*x*exp(x) is\n");
    printf("Function           Quadrature   Exact\n\n");

    for(z = 1.0, i = 0; i < POWERS; i++){
        z *= (i+1);
        printf("pow(x, %d)       %10.3f %10.3f \n", i, s[i], z);
    }
}
```

### Output

```
The integral from 0 to infinity of pow(x, i)*x*exp(x) is
Function           Quadrature   Exact

pow(x, 0)             1.000      1.000
pow(x, 1)             2.000      2.000
pow(x, 2)             6.000      6.000
pow(x, 3)            24.000     24.000
pow(x, 4)           120.000    120.000
pow(x, 5)           720.000    720.000
pow(x, 6)          4896.000   5040.000
```

# fcn_derivative

Computes the first, second, or third derivative of a user-supplied function.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_fcn_derivative (*float* fcn (), *float* x, …, 0)

The type *double* procedure is imsl_d_fcn_derivative.

## Required Arguments

*float* fcn(*float* x) (Input)
User-supplied function whose derivative at x will be computed.

*float* x (Input)
Point at which the derivative will be evaluated.

## Return Value

An estimate of the first, second or third derivative of fcn at x. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_fcn_derivative (*float* fcn (), *float* x,

IMSL_ORDER, *int* order,

IMSL_INITIAL_STEPSIZE, *float* stepize,

IMSL_RELATIVE_ERROR, *float* tolerance,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

0)

## Optional Arguments

IMSL_ORDER, *int* order (Input)
The order of the desired derivative (1, 2 or 3).
Default: order = 1.

`IMSL_INITIAL_STEPSIZE`, *float* `stepsize` (Input)

Beginning value used to compute the size of the interval for approximating the derivative. Stepsize must be chosen small enough that `fcn` is defined and reasonably smooth in the interval (`x` – 4.0 × `stepsize`, `x` + 4.0 × `stepsize`), yet large enough to avoid roundoff problems.

Default: `stepsize` = 0.01

`IMSL_RELATIVE_ERROR`, *float* `tolerance` (Input)

The relative error desired in the derivative estimate. Convergence is assumed when $(2/3)\ |d_2 - d_1| <$ `tolerance`, for two successive derivative estimates, $d_1$ and $d_2$.

Default: `tolerance` = $\sqrt[4]{\varepsilon}$ where **ε** is the machine precision.

`IMSL_FCN_W_DATA`, *float* `fcn` (*float* `x`, *void* `*data`), *void* `*data` (Input)

User supplied function whose derivative at `x` will be computed, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

The function `imsl_f_fcn_derivative` produces an estimate to the first, second, or third derivative of a function. The estimate originates from first computing a spline interpolant to the input function using value within the interval (`x` – 4.0 × `stepsize`, `x` + 4.0 × `stepsize`), then differentiating the spline at `x`.

# Examples

## Example 1

This example obtains the approximate first derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
   float fcn(float);
   float x;
   float deriv;

   x = 2.0;

   deriv = imsl_f_fcn_derivative(fcn, x, 0);
   printf ("f'(x) = %7.4f\n", deriv);
}

float fcn(float x)
{
```

```
    return -2.0*sin(1.5*x);
}
```

### Output

```
f’(x) = 2.9701
```

## Example 2

This example obtains the approximate first, second, and third derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
    double fcn(double);
    double x, tolerance, deriv;

    x = 2.0;

    deriv = imsl_d_fcn_derivative(fcn, x, 0);
    printf ("f'(x)   = %7.3f, error = %5.2e\n", deriv,
        fabs(deriv+3.0*cos(1.5*x)));

    deriv = imsl_d_fcn_derivative(fcn, x, IMSL_ORDER, 2, 0);
    printf ("f''(x) = %7.4f, error = %5.2e\n", deriv,
        fabs(deriv-4.5*sin(1.5*x)));

    deriv = imsl_d_fcn_derivative(fcn, x, IMSL_ORDER, 3, 0);
    printf ("f'''(x) = %7.4f, error = %5.2e\n", deriv,
        fabs(deriv-6.75*cos(1.5*x)));
}



double fcn(double x)
{
    return -2.0*sin(1.5*x);
}
```

### Output

```
f’(x)   =  2.970, error = 1.11e-07
f’’(x) = 0.6350, error = 8.52e-09
f’’’(x) = -6.6824, error = 1.12e-08
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# Chapter 5 Differential Equations

## Functions

# Usage Notes

## Ordinary Differential Equations

An ***ordinary differential equation*** is an equation involving one or more dependent variables $y_i$, an independent variable $t$, and derivatives of the $y_i$ with respect to $t$.

In the ***initial-value problem*** (IVP), the initial or starting values of the dependent variables $y_i$ at a known value $t = t_0$ are given. Values of $y_i(t)$ for $t > t_0$ or $t < t_0$ are required.

The functions `imsl_f_ode_runge_kutta` and `imsl_f_ode_adams_krogh` solve the IVP for ODEs of the form

$$\frac{dy_i}{dt} = y'_i = f_i\left(t, y_1, \ldots y_N\right) \quad i = 1, \ldots N$$

with $y_i(t = t_0)$ specified. Here $f_i$ is a user-supplied function that must be evaluated at any set of values ($t$, $y_1, \ldots, y_N$), $i = 1, \ldots, N$.

This problem statement is abbreviated by writing it as a ***system*** of first-order ODEs, $y(t) = [y_1(t), \ldots, y_N(t)]^T$, $f(t, y) = [f_1(t, y), \ldots, f_N(t, y)]^T$, so that the problem becomes $y' = f(t, y)$ with initial values $y(t_0)$.

The system $\frac{dy}{dt} = y' = f\left(t, y\right)$ is said to be ***stiff*** if some of the eigenvalues of the Jacobian matrix $\left\{\partial y'_i / \partial y_j\right\}$ are large and negative. An alternate definition is based on the disparate integration times using a non-stiff solver compared to an implicit integration solver. Frequently differential equations modeling the behavior of physical systems are stiff, such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reactions in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or ***time constants***).

Users typically identify stiff systems by the fact that numerical differential equation solvers such as `imsl_f_ode_runge_kutta` are inefficient, or else completely fail. Special methods are often required. The most common inefficiency is that a large number of evaluations of $f(t, y)$ (and hence an excessive amount of computer time) are required to satisfy the accuracy and stability requirements of the software. In such cases, use the IMSL function `imsl_f_ode_adams_krogh`. For more discussion about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

The function `imsl_f_pde_method_of_lines` solves the boundary value problem (BVP) for first order systems of the form $y' = f\left(t, y, p\right)$ subject to the boundary conditions $h\left(y_{left}, y_{right}\right) = 0$. Both functions $f, h$ are user-supplied. The function assumes that the user has embedded the problem into a one-parameter family of problems. In this formulation, $p$ is an optional continuation parameter. It can be useful in solving nonlinear problems. When used, $p = 0$ corresponds to an easy-to-solve problem and $p = 1$ corresponds to the actual problem.

The function`imsl_ode_adams_krogh` solves systems of ordinary differential equations of order one, order two, or mixed order one and two.

## Differential-algebraic Equations

Frequently, it is not possible or not convenient to express the model of a dynamical system as a set of ODEs. Rather, an implicit equation is available in the form

$$g_{\mathrm{i}}\left(t, y, \ldots, y_{\mathrm{N}}, y'_1, \ldots, y'_{\mathrm{N}}\right) = 0 \quad i = 1, \ldots, N$$

The $g_i$ are user-supplied functions. The system is abbreviated as

$$g(t, y, y') = \left[g_1(t, y, y'), \ \ldots, g_N(t, y, y')\right]^T = 0$$

With initial value $y(t_0)$. Any system of ODEs can be trivially written as a differential-algebraic system by defining

$$g(t, y, y') = f(t, y) - y'$$

The function `imsl_f_differential_algebraic_eqs` solves differential-algebraic systems of index 1 or index 0. For a definition of ***index*** of a differential-algebraic system, see (Brenan et al. 1989). Also, see Gear and Petzold (1984) for an outline of the computing methods used.

## Partial Differential Equations

There is a section Introduction to pde_1d_mg in this chapter for `imsl_pde_1d_mg` with greater details. This software is a variable grid-variable order integrator. It solves a problem

$$\sum_{k=1}^{NPDE} C_{j,k}\left(x, t, u, u_x\right)\frac{\partial u^k}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m R_j\left(x, t, u, u_x\right)\right) - Q_j\left(x, t, u, u_x\right),$$
$$j = 1, \ \ldots, NPDE, \quad x_L < x < x_{R,} \quad t > t_0, \quad m \in \{0, 1, 2\}$$

with boundary conditions

$$\beta_j(x,t)R_j\left(x,t,u,u_x\right) = \gamma_j\left(x,t,u,u_x\right),$$
$$at x = x_L \text{ and } x = x_R, j = 1,\ldots NPDE$$

The function `imsl_f_modified_method_of_lines` solves the IVP problem for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ \ldots, u_N, \frac{\partial u_1}{\partial x}, \ \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

subject to the boundary conditions

$$\begin{aligned} \alpha_1^{(i)} u_i(a) + \beta_1^{(i)}\frac{\partial u_i}{\partial x}(a) &= \gamma_1(t) \\ \alpha_2^{(i)} u_i(b) + \beta_2^{(i)}\frac{\partial u_i}{\partial x}(b) &= \gamma_2(t) \end{aligned}$$

and subject to the initial conditions $u_i(x, t = t_0) = g_i(x)$, for $i$ = 1, …, $N$. Here, $f_i$, $g_i$, $\alpha_j^{(i)}$, and $\beta_j^{(i)}$ are user-supplied, $j$ = 1, 2.

The function imsl_f_feynman_kac solves a single equation on a finite interval $[x_{\min}, x_{\max}]$. This equation often arises in applications from financial engineering and that is the primary focus of the document examples. The equation, initial conditions and Feynman-Kac boundary values are given by

$$f_t + \mu(x,t) f_x + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t) f = \phi(f,x,t),$$

$$f(x,T) = p(x), \{f_t = \frac{\partial f}{\partial t}, etc.\}$$

$$a(x,t) f + b(x,t) f_x + c(x,t) f_{xx} = d(x,t), \quad x = x_{\min} x_{\max}$$

The solution is approximated by a piece-wise series of Hermite quintic polynomials on a grid of the interval $[x_{\min}, x_{\max}]$ that yields a twice differentiable solution. To assist in the evaluation of the approximate solution and its derivatives there is the function imsl_f_feynman_kac_evaluate.

The function imsl_f_fast_poisson_2d solves Laplace's, Poisson's, or Helmholtz's equation in two dimensions. This function uses a fast Poisson method to solve a PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = f(x,y)$$

over a rectangle, subject to boundary conditions on each of the four sides. The scalar constant ***c*** and the function ***f*** are user specified.

# ode_runge_kutta

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_ode_runge_kutta_mgr (*int* task, *void* **state, …, 0)

*void* imsl_f_ode_runge_kutta (*int* neq, *float* *t, *float* tend, *float* y[], *void* *state, *void* fcn ())

The type ***double*** functions are imsl_d_ode_runge_kutta_mgr and imsl_d_ode_runge_kutta.

## Required Arguments for imsl_ f_ode_runge_kutta_mgr

*int* task (Input)
This function must be called with task set to IMSL_ODE_INITIALIZE to set up for solving an ODE system and with task equal to IMSL_ODE_RESET to clean up after it has been solved. These values for task are defined in the include file, imsl.h.

*void* **state (Input/Output)
The current state of the ODE solution is held in a structure pointed to by state. It cannot be directly manipulated.

## Required Arguments for imsl_f_ode_runge_kutta

*int* neq (Input)
Number of differential equations.

*float* *t (Input/Output)
Independent variable. On input, t is the initial independent variable value. On output, t is replaced by tend, unless error conditions arise.

*float* tend (Input)
Value of t at which the solution is desired. The value tend may be less than the initial value of t.

*float* y[] (Input/Output)
Array with neq components containing a vector of dependent variables. On input, y contains the initial values. On output, y contains the approximate solution.

*void* *state (Input/Output)
The current state of the ODE solution is held in a structure pointed to by state. It must be initialized by a call to imsl_f_ode_runge_kutta_mgr. It cannot be directly manipulated.

*void* fcn (*int* neq, *float* t, *float* *y, *float* *yprime)

User-supplied function to evaluate the right-hand side where *float* *yprime (Output) Array with neq components containing the vector y′. This function computes

$$\texttt{yprime} = \frac{dy}{dt} = y' = f\left(t, y\right)$$

and neq, t, and *y are defined immediately preceding this function.

# Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_ode_runge_kutta_mgr (*int* task, *void* **state,

IMSL_TOL, *float* tol,

IMSL_HINIT, *float* hinit,

IMSL_HMIN, *float* hmin,

IMSL_HMAX, *float* hmax,

IMSL_MAX_NUMBER_STEPS, *int* max_steps,

IMSL_MAX_NUMBER_FCN_EVALS, *int* max_fcn_evals,

IMSL_SCALE, *float* scale,

IMSL_NORM, *int* norm,

IMSL_FLOOR, *float* floor,

IMSL_NSTEP, *int* *nstep,

IMSL_NFCN, *int* *nfcn,

IMSL_HTRIAL, *float* *htrial,

IMSL_FCN_W_DATA, *void* fcn (), *void* *data,

0)

# Optional Arguments

IMSL_TOL, *float* tol (Input)

Tolerance for error control. An attempt is made to control the norm of the local error such that the global error is proportional to tol.

Default: tol = 100.0*imsl_f_machine(4)

IMSL_HINIT, *float* hinit (Input)

Initial value for the step size *h*. Steps are applied in the direction of integration.

Default: hinit = 0.001|tend – t|

IMSL_HMIN, *float* hmin (Input)

Minimum value for the step size *h*.

Default: hmin – 0.0.

IMSL_HMAX, *float* hmax (Input)

Maximum value for the step size *h*.

Default: hmax = 2.0.

IMSL_MAX_NUMBER_STEPS, *int* max_steps (Input)

Maximum number of steps allowed.

Default: max_steps = 500.

IMSL_MAX_NUMBER_FCN_EVALS, *int* max_fcn_evals (Input)

Maximum number of function evaluations allowed.

Default: max_fcn_evals = No enforced limit.

IMSL_SCALE, *float* scale (Input)

A measure of the scale of the problem, such as an approximation to the Jacobian along the trajectory.

Default: scale = 1.

IMSL_NORM, *int* norm (Input)

Switch determining the error norm. In the following, $e_i$ is the absolute value of the error estimate for $y_i$.

| norm | Error norm used |
|---|---|
| 0 | minimum of the absolute error and the relative error, equals the maximum of $e_i$ / max ($\lvert y_i \rvert$, 1) for $i$ = 1, …, neq. |
| 1 | absolute error, equals $\max_i e_i$. |
| 2 | $\max_i(e_i / w_i)$ where $w_i$ = max ($\lvert y_i \rvert$, floor). The value of floor is reset using IMSL_FLOOR. |

Default: norm = 0.

IMSL_FLOOR, *float* floor (Input)

This is used with IMSL_NORM. It provides a positive lower bound for the error norm option with value 2.

Default: floor = 1.0.

IMSL_NSTEP, *int* *nstep (Output)

Returns the number of steps taken.

IMSL_NFCN, *int* *nfcn (Output)

Returns the number of function evaluations used.

IMSL_HTRIAL, *float* *htrial (Output)

Returns the current trial step size.

IMSL_FCN_W_DATA, *void* fcn(*int* neq, *float* t, *float* *y, *float* *yprime, *void* *data), *void* *data, (Input)

User-supplied function to evaluate the right-hand side, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function imsl_f_ode_runge_kutta finds an approximation to the solution of a system of first-order differential equations of the form

$$\mathtt{yprime} = \frac{dy}{dt} = y' = f(t, y)$$

with given initial conditions for *y* at the starting value for *t*. The function attempts to keep the global error proportional to a user-specified tolerance. The proportionality depends on the differential equation and the range of integration.

The function imsl_f_ode_runge_kutta is efficient for nonstiff systems where the evaluations of *f*(*t*, *y*) are not expensive. The code is based on an algorithm designed by Hull et al. (1976, 1978). It uses Runge-Kutta formulas of order five and six developed by J.H. Verner.

## Examples

### Example 1

This example solves

$$\frac{dy}{dt} = -y$$

over the interval [0, 1] with the initial condition *y*(0) = 1. The solution is $y(t) = e^{-t}$.

The ODE solver is initialized by a call to imsl_f_ode_runge_kutta_mgr with IMSL_ODE_INITIALIZE. This is the simplest use of the solver, so none of the default values are changed. The function imsl_f_ode_runge_kutta is then called to integrate from *t* = 0 to *t* = 1.

```
#include <imsl.h>
#include <math.h>

void       fcn (int neq, float t, float y[], float yprime[]);

int main()
{
   int         neq = 1;        /* Number of ode's */
```

```
   float      t = 0.0;       /* Initial time */
   float      tend = 1.0;    /* Final time */
   float      y[1] = {1.0};  /* Initial condition */
   void       *state;
                              /* Initialize the ODE solver */
   imsl_f_ode_runge_kutta_mgr(IMSL_ODE_INITIALIZE, &state, 0);
                              /* Integrate from t=0 to tend=1 */
   imsl_f_ode_runge_kutta (neq, &t, tend, y, state, fcn);
                              /* Print the solution and error */
   printf("y[%f] = %f\n", t, y[0]);
   printf("Error is: %e\n", exp( (double)(-tend) )-y[0]);
}

void fcn (int neq, float t, float y[], float yprime[])
{
   yprime[0] = -y[0];
}
```

### Output

```
y[1.000000] = 0.367879
Error is: -9.149755e-09
```

## Example 2

Consider a predator-prey problem with rabbits and foxes. Let *r* be the density of rabbits, and let *f* be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. Mathematically, the model without species interaction is approximated by the equation

$$r' = 2r$$
$$f' = f$$

With species interaction, the rate at which the rabbits are consumed by the foxes is assumed to equal the value *2rf*. The rate at which the foxes increase, because they are consuming the rabbits, is equal to *rf*. Thus, the model differential equations to be solved are

$$r' = 2r - 2rf$$
$$f' = -f + rf$$

For illustration, the initial conditions are taken to be $r(0) = 1$ and $f(0) = 3$. The interval of integration is $0 \le t \le 10$. In the program, $y[0] = r$ and $y[1] = f$. The ODE solver is initialized by a call to `imsl_f_ode_runge_kutta_mgr`. The error tolerance is set to 0.0005. Absolute error control is selected by setting `IMSL_NORM` to the value one. We also request that `nstep` be set to the current number of steps in the integration. The function `imsl_f_ode_runge_kutta` is then called in a loop to integrate from $t = 0$ to $t = 10$ in steps of $\delta t = 1$. At each step, the solution is printed. Note that `nstep` is updated even though it is not an argument to this function. Its address has been stored within `imsl_f_ode_runge_kutta_mgr` into the area pointed to by `state`. The last call to `imsl_f_ode_runge_kutta_mgr` with `IMSL_ODE_RESET` releases workspace.

```
#include <imsl.h>

void            fcn(int neq, float t, float y[], float yprime[]);

int main()
{
   int        neq = 2;
   float      t = 0.0;           /* Initial time */
   float      tend;              /* Final time */
   float      y[2] = {1.0, 3.0}; /* Initial conditions */
   int        k;
   int        nstep;
   void       *state;
                              /* Initialize the ODE solver */
   imsl_f_ode_runge_kutta_mgr(IMSL_ODE_INITIALIZE, &state,
                             IMSL_TOL,    0.0005,
                             IMSL_NSTEP, &nstep,
                             IMSL_NORM,   1,
                             0);

   printf("\n Start   End    Density of Density of   Number of" );
   printf("\n Time    Time     Rabbits    Foxes        Steps\n\n");

   for (k = 0;  k < 10;  k++) {
       tend = k + 1;
       imsl_f_ode_runge_kutta (neq, &t, tend, y, state, fcn);
       printf("%3d %12.3f %12.3f %12.3f %12d\n", k, t, y[0], y[1], nstep);
   }
   imsl_f_ode_runge_kutta_mgr(IMSL_ODE_RESET, &state, 0);
}

void fcn (int neq, float t, float y[], float yprime[])
{
                              /* Density change rate for Rabbits: */
   yprime[0] = 2*y[0]*(1 - y[1]);
                              /* Density change rate for Foxes: */
   yprime[1] = -y[1]*(1 - y[0]);
}
```

### Output

```
Start     End      Density of  Density of     Number of
Time      Time      Rabbits      Foxes          Steps

0       1.000       0.078       1.465            4
1       2.000       0.085       0.578            6
2       3.000       0.292       0.250            7
3       4.000       1.449       0.187            8
4       5.000       4.046       1.444           11
5       6.000       0.176       2.256           15
```

```
6        7.000        0.066        0.908            18
7        8.000        0.148        0.367            20
8        9.000        0.655        0.188            21
9       10.000        3.157        0.352            23
```

## Fatal Errors

| | |
|---|---|
| `IMSL_ODE_TOO_MANY_EVALS` | Completion of the next step would make the number of function evaluations #, but only # evaluations are allowed. |
| `IMSL_ODE_TOO_MANY_STEPS` | Maximum number of steps allowed, #, used. The problem may be stiff. |
| `IMSL_ODE_FAIL` | Unable to satisfy the error requirement. “`tol`” = # may be too small. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# ode_adams_gear

***Note***: This function is deprecated and has been replaced by `imsl_f_ode_adams_krogh`. To view the deprecated documentation, see imsl_f_ode_adams_gear.pdf on the Rogue Wave website. You can also access a local copy in your IMSL installation directory at `pdf\deprecated_routines\math\ode_adams_gear.pdf`.

# bvp_finite_difference

Solves a (parameterized) system of differential equations with boundary conditions at two points, using a variable order, variable step size finite difference method with deferred corrections.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_bvp_finite_difference (*void* fcneq(), *void* fcnjac(), *void* fcnbc(), *int* n, *int* nleft, *int* ncupbc, *float* tleft, *float* tright, *int* linear, *float* *nfinal, *float* *xfinal, *float* *yfinal, …, 0)

The type ***double*** function is imsl_d_bvp_finite_difference.

## Required Arguments

*void* fcneq (*int* n, *float* t, *float* y[], *float* p, *float* dydt[]) (Input)
User supplied function to evaluate derivatives.

*int* n (Input)
Number of differential equations.

*float* t (Input)
Independent variable, ***t***.

*float* y[] (Input)
Array of size n containing the dependent variable values, ***y***(t).

*float* p (Input)
Continuation parameter, ***p***. See optional argument IMSL_PROBLEM_EMBEDDED.

*float* dydt[] (Output)
Array of size n containing the derivatives ***y'***(*t*).

*void* fcnjac(*int* n, *float* t, *float* y[], *float* p, *float* dypdy[]) (Input)
User supplied function to evaluate the Jacobian.

*int* n (Input)
Number of differential equations.

*float* t (Input)
Independent variable, ***t***.

*float* y[] (Input)
Array of size n containing the dependent variable values, ***y***(t).

*float* p (Input)

Continuation parameter, ***p***. See optional argument IMSL_PROBLEM_EMBEDDED.

*float* dypdy[] (Output)

n by n array containing the partial derivatives $a_{i,j} = \partial f_i / \partial y_j$ evaluated at (*t*, *y*). The values $a_{i,j}$ are returned in dypdy[(i-1)*n+(j-1)].

*void* fcnbc(*int* n, *float* yleft[], *float* yright[], *float* p, *float* h[]) (Input)

User supplied function to evaluate the boundary conditions.

*int* n (Input)

Number of differential equations.

*float* yleft[] (Input)

Array of size n containing the values of the dependent variable at the left endpoint.

*float* yright[] (Input)

Array of size n containing the values of the dependent variable at the right endpoint.

*float* p (Input)

Continuation parameter, ***p*** See optional argument IMSL_PROBLEM_EMBEDDED.

*float* h[] (Output)

Array of size n containing the boundary condition residuals. The boundary conditions are defined by $h_i = 0$, for *i* = 0, …, n-*1*. The left endpoint conditions must be defined first, then, the conditions involving both endpoints, and finally the right endpoint conditions.

*int* n (Input)

Number of differential equations.

*int* nleft (Input)

Number of initial conditions. The value nleft must be greater than or equal to zero and less than n.

*int* ncupbc (Input)

Number of coupled boundary conditions. The value nleft + ncupbc must be greater than zero and less than or equal to n.

*float* tleft (Input)

The left endpoint.

*float* tright (Input)

The right endpoint.

*int* linear (Input)

Integer flag to indicate if the differential equations and the boundary conditions are linear. Set linear to one if the differential equations and the boundary conditions are linear, otherwise set linear to zero.

*int* *nfinal (Output)
Number of final grid points, including the endpoints.

*float* *tfinal (Output)
Array of size mxgrid containing the final grid points. Only the first nfinal points are significant. See optional argument IMSL_MAX_SUBINTER for definition of mxgrid.

*float* *yfinal (Output)
Array of size mxgrid by n containing the values of Y at the points in tfinal. See optional argument IMSL_MAX_SUBINTER for definition of mxgrid.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_f_bvp_finite_difference (*void* fcneq(), *void* fcnjac(), *void* fcnbc(), *int* n, *int* nleft, *int* ncupbc, *float* tleft, *float* tright, *int* linear, *float* *nfinal, *float* *xfinal[], *float* *yfinal,

IMSL_TOL, *float* tol,

IMSL_HINIT, *int* ninit, *float* tinit[], *float* yinit[][],

IMSL_PRINT, *int* iprint,

IMSL_MAX_SUBINTER, *int* mxgrid,

IMSL_PROBLEM_EMBEDDED, *float* pistep, *void* fcnpeq(), *void* fcnpbc(),

IMSL_ERR_EST, *float* **errest,

IMSL_ERR_EST_USER, *float* errest[],

IMSL_FCN_W_DATA, *void* fcneq(), *void* *data,

IMSL_JACOBIAN_W_DATA, *void* fcnjac (),*void* *data,

IMSL_FCN_BC_W_DATA, *void* fcnbc(), *void* *data,

IMSL_PROBLEM_EMBEDDED_W_DATA, *float* pistep, (), *void* *data, *void* fcnpeq(), *void* fcnpbc(), *void* *data,

0)

## Optional Arguments

IMSL_TOL, *float* tol (Input)
Relative error control parameter. The computations stop when

$$|E_{i,j}| / \max\left(y_{i,j}, 1.0\right) < tol \text{ for all } i = 0, n = 1, \text{ and } j = 0, ngrid - 1$$

Here $E_{i,j}$ is the estimated error on $y_{i,j}$

Default: `tol` = .001.

`IMSL_HINIT`, *int* `ninit`, *float* `tinit[]`, *float* `yinit[][]`, (Input)

Initial gridpoints. Number of initial grid points, including the endpoints, is given by `ninit`. `tinit` is an array of size `ninit` containing the initial grid points. `yinit` is an array size `ninit` by `n` containing an initial guess for the values of `Y` at the points in `tinit`.

Default: `ninit` = 10, `tinit[*]` equally spaced in the interval [`tleft`, `tright`], and `yinit[*][*]` = 0.

`IMSL_PRINT`, *int* `iprint` (Input)

Parameter indicating the desired output level.

| iprint | Action |
|---|---|
| 0 | No output printed. |
| 1 | Intermediate output is printed.<br>Default: `iprint` = 0. |

`IMSL_MAX_SUBINTER`, *int* `mxgrid` (Input)

Maximum number of grid points allowed.

Default: `mxgrid` = 100.

`IMSL_PROBLEM_EMBEDDED`, *float* `pistep`, *void* `fcnpeq()`, *void* `fcnpbc()`

If this optional argument is supplied, then the function `imsl_f_bvp_finite_difference` assumes that the user has embedded the problem into a one-parameter family of problems:

$$y' = y'\left(t, y, p\right)$$

$$h\left(yleft, yright, p\right) = 0$$

such that for ***p*** = 0 the problem is simple. For ***p*** = 1, the original problem is recovered. The function `imsl_f_bvp_finite_difference` automatically attempts to increment from ***p*** = 0 to ***p*** = 1. The value `pistep` is the beginning increment used in this continuation. The increment will usually be changed by function `imsl_f_bvp_finite_difference`, but an arbitrary minimum of 0.01 is imposed.

The argument `p` is the initial increment size for ***p***. The functions `fcnpeq` and `fcnpbc` are user-supplied functions, and are defined:

*void* `fcnpeq`(*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dypdp[]`) (Input)

User supplied function to evaluate the derivative of ***y'*** with respect to the parameter ***p***.

*int* `n` (Input)

Number of differential equations.

*float* `t` (Input)
Independent variable, ***t***.

*float* `y[]` (Input)
Array of size `n` containing the dependent variable values.

*float* `p` (Input)
Continuation parameter, ***p***.

*float* `dypdp[]` (Output)
Array of size `n` containing the derivative ***y'*** with respect to the parameter ***p*** at (***t***, ***y***).

*void* `fcnpbc`(*int* `n`, *float* `yleft[]`, *float* `yright[]`, *float* `p`, *float* `h[]`)(Input)
User supplied function to evaluate the derivative of the boundary conditions with respect to the parameter ***p***.

*int* `n` (Input)
Number of differential equations.

*float* `yleft[]` (Input)
Array of size `n` containing the values of the dependent variable at the left endpoint.

*float* `yright[]` (Input)
Array of size `n` containing the values of the dependent variable at the right endpoint.

*float* `p` (Input)
Continuation parameter, ***p***.

*float* `h[]` (Output)
Array of size `n` containing the derivative of $f_i$ with respect to ***p***.

`IMSL_ERR_EST`, *float* `**errest` (Output)
Address of a pointer to an array of size `n` containing estimated error in y.

`IMSL_ERR_EST_USER`, *float* `errest[]` (Output)
User allocated array of size `n` containing estimated error in y.

`IMSL_FCN_W_DATA`, *void* `fcneq` (*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dydt[]`, *void* `*data`) ,*void* `*data`, (Input)
User-supplied function to evaluate derivatives, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_JACOBIAN_W_DATA`, *void* `fcnjac`(*int* `n`, *float* `t`, *float* `y[]`, *float* `p`, *float* `dypdy[]`, *void* `*data`), *void* `*data`, (Input)
User-supplied function to evaluate the Jacobian, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_FCN_BC_W_DATA`, *void* `fcnbc`(*int* `n`, *float* `yleft[]`, *float* `yright[]`, *float* `p`, *float* `h[]`, *void* `*data`), *void* `*data`, (Input)

User-supplied function to evaluate the boundary conditions , which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_PROBLEM_EMBEDDED_W_DATA`, *float* `pistep`, *void* `fcnpeq(`*void* `*data)`, *void* `fcnpbc()`, *void* `*data`, (Input)

Same as optional argument `IMSL_PROBLEM_EMBEDDED`, except user-supplied functions also accept a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

# Description

The function `imsl_f_bvp_finite_difference` is based on the subprogram `PASVA3` by M. Lentini and V. Pereyra (see Pereyra 1978). The basic discretization is the trapezoidal rule over a nonuniform mesh. This mesh is chosen adaptively, to make the local error approximately the same size everywhere. Higher-order discretizations are obtained by deferred corrections. Global error estimates are produced to control the computation. The resulting nonlinear algebraic system is solved by Newton's method with step control. The linearized system of equations is solved by a special form of Gauss elimination that preserves the sparseness.

# Examples

## Example 1

This example solves the third-order linear equation

$$y''' - 2y'' + y' - y = \sin t$$

subject to the boundary conditions $y(0) = y(2\pi)$ and $y'(0) = y'(2\pi) = 1$. (Its solution is $y = \sin t$.) To use `imsl_f_bvp_finite_difference`, the problem is reduced to a system of first-order equations by defining $y_1 = y$, $y_2 = y'$ and $y_3 = y''$. The resulting system is

$$\begin{aligned} y'_1 &= y_2 & \qquad y_2(0) - 1 &= 0 \\ y'_2 &= y_3 & \qquad y_1(0) - y_1(2\pi) &= 0 \\ y'_3 &= 2y_3 - y_2 + y_1 + \sin t & \qquad y_2(2\pi) - 1 &= 0 \end{aligned}$$

Note that there is one boundary condition at the left endpoint $t = 0$ and one boundary condition coupling the left and right endpoints. The final boundary condition is at the right endpoint. The total number of boundary conditions must be the same as the number of equations (in this case 3).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>
```

```
void fcneqn( int n, float t, float y[], float p, float dydt[]);
void fcnjac( int n, float t, float y[], float p, float dfdy[]);

void fcnbc( int n, float yleft[], float yright[], float p, float h[]);

#define MXGRID 100
#define N 3

int main()
{
   int n = N;
   int nleft = 1;
   int ncupbc = 1;
   float tleft = 0;
   float tright;
   int linear = 1;
   int nfinal;
   float tfinal[MXGRID];
   float yfinal[MXGRID][N];
   float errest[N];
   int i;

   tright = 2.0*imsl_f_constant("pi", 0);

   imsl_f_bvp_finite_difference( fcneqn, fcnjac, fcnbc,
       n, nleft, ncupbc, tleft, tright,
       linear, &nfinal, tfinal,
       (float*)(&yfinal[0][0]),
       IMSL_ERR_EST_USER, errest,
       0);
   printf("          tfinal           y0            y1             y2 \n" );
   for( i=0; i<nfinal; i++ ) {
       printf( "%5d%15.6e%15.6e%15.6e%15.6e\n", i,     tfinal[i],
           yfinal[i][0], yfinal[i][1], yfinal[i][2] );
   }
   printf("Error Estimates    ");
   printf("%15.6e%15.6e%15.6e\n",errest[0],errest[1],errest[2]);
}

void fcneqn( int n, float t, float y[], float p, float dydt[] )
{
   dydt[0] = y[1];
   dydt[1] = y[2];
   dydt[2] = 2*y[2] - y[1] + y[0] + sin(t);
}

void fcnjac( int n, float t, float y[], float p, float dfdy[] )
{
   dfdy[0*n+0] = 0;  /* df1/dy1 */
   dfdy[1*n+0] = 0;  /* df2/dy1 */
```

```
   dfdy[2*n+0] = 1;  /* df3/dy1 */
   dfdy[0*n+1] = 1;  /* df1/dy2 */
   dfdy[1*n+1] = 0;  /* df2/dy2 */
   dfdy[2*n+1] = -1; /* df3/dy2 */
   dfdy[0*n+2] = 0;  /* df1/dy3 */
   dfdy[1*n+2] = 1;  /* df2/dy3 */
   dfdy[2*n+2] = 2;  /* df3/dy3 */
}

void fcnbc( int n, float yleft[], float yright[], float p, float h[] )
{
   h[0] = yleft[1] - 1;
   h[1] = yleft[0] - yright[0];
   h[2] = yright[1] - 1;
}
```

### Output

```
          tfinal            y0            y1            y2
   0  0.000000e+00 -1.123446e-04  1.000000e+00  6.245916e-05
   1  3.490659e-01  3.419106e-01  9.397087e-01 -3.419581e-01
   2  6.981317e-01  6.426907e-01  7.660918e-01 -6.427230e-01
   3  1.396263e+00  9.847531e-01  1.737333e-01 -9.847453e-01
   4  2.094395e+00  8.660527e-01 -4.998748e-01 -8.660057e-01
   5  2.792527e+00  3.421828e-01 -9.395475e-01 -3.420647e-01
   6  3.490659e+00 -3.417236e-01 -9.396111e-01  3.418948e-01
   7  4.188790e+00 -8.656881e-01 -5.000588e-01  8.658734e-01
   8  4.886922e+00 -9.845795e-01  1.734572e-01  9.847519e-01
   9  5.585054e+00 -6.427722e-01  7.658259e-01  6.429526e-01
  10  5.934120e+00 -3.420819e-01  9.395434e-01  3.423984e-01
  11  6.283185e+00 -1.123446e-04  1.000000e+00  6.739637e-04
Error Estimates       2.840487e-04  1.792839e-04  5.587848e-04
```

## Example 2

In this example, the following nonlinear problem is solved:

$$y'' - y^3 + (1 + \sin^2 t) \sin t = 0$$

with $y(0) = y(\pi) = 0$. Its solution is $y = \sin t$. As in Example 1, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$y'_1 = y_2 \quad y_1(0) = 0$$
$$y'_2 = y_1^3 - (1 + \sin^2 t) \sin t \quad y_1(\pi) = 0$$

In this problem, there is one boundary condition at the left endpoint and one at the right endpoint; there are no coupled boundary conditions.

```
#include <imsl.h>
#include <stdio.h>
```

```
#include <math.h>

void fcneqn(int n, float x, float y[], float p, float dydx[]);
void fcnjac(int n, float x, float y[], float p, float dfdy[]);
void fcnbc(int n, float yleft[], float yright[], float p, float h[]);

#define MXGRID 100
#define NINIT 12
#define N 2

int main()
{
   int n = N, nleft = 1, ncupbc = 0, linear = 0;
   int i, nfinal, ninit = NINIT;
   float tleft = 0, tright;
   float tinit[NINIT], yinit[NINIT][N];
   float tfinal[MXGRID], yfinal[MXGRID][N];
   float *errest, step;

   tright = imsl_f_constant("pi", 0);
   step = (tright-tleft) / (ninit-1);

   for( i=0; i<ninit; i++ ) {
       tinit[i] = tleft + i*step;
       yinit[i][0] = 0.4 * (tinit[i]-tleft) * (tright-tinit[i]);
       yinit[i][1] = 0.4 * (tright+tleft-2*tinit[i]);
   }

   imsl_f_bvp_finite_difference(fcneqn, fcnjac, fcnbc, n, nleft,
       ncupbc, tleft, tright, linear, &nfinal, tfinal,
       (float*)(&yfinal[0][0]),
       IMSL_HINIT, ninit, tinit, yinit,
       IMSL_ERR_EST, &errest,
       0);
   printf("             t            y0           y1\n" );
   for( i=0; i<nfinal; i++ ) {
       printf( "%5d%15.6e%15.6e%15.6e\n", i, tfinal[i], yfinal[i][0],
           yfinal[i][1]);
   }
   printf("Error Estimates    ");
   printf("%15.6e%15.6e\n",errest[0],errest[1]);
}

void fcneqn(int n, float t, float y[], float p, float dydt[])
{
   float sx = sin(t);
   dydt[0] = y[1];
   dydt[1] = y[0]*y[0]*y[0] - (sx*sx+1)*sx;
}

void fcnjac(int n, float t, float y[], float p, float dfdy[])
```

```
{
   dfdy[0*n+0] = 0;           /* df1/dy1 */
   dfdy[1*n+0] = 3*y[0]*y[0]; /* df2/dy1 */
   dfdy[0*n+1] = 1;           /* df1/dy2 */
   dfdy[1*n+1] = 0;           /* df2/dy2 */
}

void fcnbc(int n, float yleft[], float yright[], float p, float h[])
{
   h[0] = yleft[0];
   h[1] = yright[0];
}
```

### Output

```
             t             y0            y1
   0  0.000000e+00  0.000000e+00  9.999277e-01
   1  2.855994e-01  2.817682e-01  9.594315e-01
   2  5.711987e-01  5.406458e-01  8.412407e-01
   3  8.567981e-01  7.557380e-01  6.548904e-01
   4  1.142397e+00  9.096186e-01  4.154530e-01
   5  1.427997e+00  9.898143e-01  1.423307e-01
   6  1.713596e+00  9.898143e-01 -1.423308e-01
   7  1.999195e+00  9.096185e-01 -4.154530e-01
   8  2.284795e+00  7.557380e-01 -6.548902e-01
   9  2.570394e+00  5.406460e-01 -8.412405e-01
  10  2.855994e+00  2.817682e-01 -9.594312e-01
  11  3.141593e+00  0.000000e+00 -9.999274e-01
Error Estimates       3.907291e-05  7.124317e-05
```

## Example 3

In this example, the following nonlinear problem is solved:

$$y'' - y^3 = \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3} - \left(t - \frac{1}{2}\right)^8$$

with $y(0) = y(1) = \pi/2$. As in the previous examples, this equation is reduced to a system of first-order differential equations by defining $y_1 = y$ and $y_2 = y'$. The resulting system is

$$\begin{aligned} y'_1 &= y_2 \quad y_1(0) = \pi/2 \\ y'_2 &= y_1^3 - \tfrac{40}{9}\left(t - \tfrac{1}{2}\right)^{2/3} + \left(t - \tfrac{1}{2}\right)^8 y_1(1) = \pi/2 \end{aligned}$$

The problem is embedded in a family of problems by introducing the parameter $p$ and by changing the second differential equation to

$$y'_2 = py_1^3 + \frac{40}{9}\left(t - \frac{1}{2}\right)^{2/3}\left(t - \frac{1}{2}\right)^8$$

At $p$ = 0, the problem is linear; and at $p$ = 1, the original problem is recovered. The derivatives $\partial y'/\partial p$ must now be specified in the subroutine `fcnpeq`. The derivatives $\partial f/\partial p$ are zero in `fcnpbc`.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

void fcneqn(int n, float t, float y[], float p, float dydt[]);
void fcnjac(int n, float t, float y[], float p, float dfdy[]);
void fcnbc(int n, float yleft[], float yright[], float p, float h[]);
void fcnpeq(int n, float t, float y[], float p, float dfdp[]);
void fcnpbc(int n, float yleft[], float yright[], float p, float dhdp[]);

#define MXGRID 45
#define NINIT 12
#define N 2

int main()
{
   int n = 2;
   int nleft = 1;
   int ncupbc = 0;
   float tleft = 0;
   float tright = 1;
   float pistep = 0.1;
   int ninit = 5;
   float tinit[NINIT] = { 0.0, 0.4, 0.5, 0.6, 1.0 };
   float yinit[NINIT][N] =
       { 0.15749,  0.00215,
        0.0,      0.00215,
        0.15749, -0.83995,
       -0.05745,  0.0,
        0.05745,  0.83995 };
   int linear = 0;
   int nfinal;
   float tfinal[MXGRID];
   float yfinal[MXGRID][N];
   float *errest;
   int i;

   imsl_f_bvp_finite_difference( fcneqn, fcnjac, fcnbc, n, nleft,
       ncupbc, tleft, tright,
       linear, &nfinal, tfinal, (float*)(&yfinal[0][0]),
       IMSL_MAX_SUBINTER, MXGRID,
       IMSL_PROBLEM_EMBEDDED, pistep, fcnpeq, fcnpbc,
       IMSL_HINIT, ninit, tinit, yinit,
       IMSL_ERR_EST, &errest,
       0 );
```

```
    printf("              t              y0            y1\n" );
    for( i=0; i<nfinal; i++ ) {
        printf("%5d%15.6e%15.6e%15.6e\n", i, tfinal[i], yfinal[i][0],
            yfinal[i][1]);
    }
    printf("Error Estimates    ");
    printf("%15.6e%15.6e\n",errest[0],errest[1]);
}

void fcneqn(int n, float t, float y[], float p, float dydt[])
{
    float z = t - 0.5;
    dydt[0] = y[1];
    dydt[1] = p*y[0]*y[0]*y[0] + 40./9.*pow(z*z,1./3.) - pow(z,8);
}

void fcnjac(int n, float t, float y[], float p, float dfdy[])
{
    dfdy[0*n+0] = 0;                     /* df0/dy0 */
    dfdy[0*n+1] = 1;                     /* df0/dy1 */
    dfdy[1*n+0] = 3.*(p)*(y[0]*y[0]); /* df1/dy0 */
    dfdy[1*n+1] = 0;                     /* df1/dy1 */
}

void fcnbc(int n, float yleft[], float yright[], float p, float h[])
{
    float pi2 = imsl_f_constant("pi", 0)/2.0;
    h[0] = yleft[0] - pi2;
    h[1] = yright[0] - pi2;
}

void fcnpeq(int n, float t, float y[], float p, float dfdp[])
{
    dfdp[0] = 0;
    dfdp[1] = y[0]*y[0]*y[0];
}

void fcnpbc(int n, float yleft[], float yright[], float p, float dhdp[])
{
    dhdp[0] = 0;
    dhdp[1] = 0;
}
```

### Output

```
             t             y0             y1
    0 0.000000e+000 1.570796e+000 -1.949336e+000
    1 4.444445e-002 1.490495e+000 -1.669567e+000
    2 8.888889e-002 1.421951e+000 -1.419465e+000
    3 1.333333e-001 1.363953e+000 -1.194307e+000
    4 2.000000e-001 1.294526e+000 -8.958461e-001
```

```
    5 2.666667e-001 1.243628e+000 -6.373191e-001
    6 3.333334e-001 1.208785e+000 -4.135206e-001
    7 4.000000e-001 1.187783e+000 -2.219351e-001
    8 4.250000e-001 1.183038e+000 -1.584200e-001
    9 4.500000e-001 1.179822e+000 -9.973147e-002
   10 4.625000e-001 1.178748e+000 -7.233894e-002
   11 4.750000e-001 1.178007e+000 -4.638249e-002
   12 4.812500e-001 1.177756e+000 -3.399764e-002
   13 4.875000e-001 1.177582e+000 -2.205549e-002
   14 4.937500e-001 1.177480e+000 -1.061179e-002
   15 5.000000e-001 1.177447e+000 -1.603742e-007
   16 5.062500e-001 1.177480e+000 1.061152e-002
   17 5.125000e-001 1.177582e+000 2.205516e-002
   18 5.187500e-001 1.177756e+000 3.399726e-002
   19 5.250000e-001 1.178007e+000 4.638217e-002
   20 5.375000e-001 1.178748e+000 7.233874e-002
   21 5.500000e-001 1.179822e+000 9.973122e-002
   22 5.750000e-001 1.183038e+000 1.584198e-001
   23 6.000000e-001 1.187783e+000 2.219350e-001
   24 6.666667e-001 1.208786e+000 4.135205e-001
   25 7.333333e-001 1.243628e+000 6.373190e-001
   26 8.000000e-001 1.294526e+000 8.958460e-001
   27 8.666667e-001 1.363953e+000 1.194307e+000
   28 9.111111e-001 1.421951e+000 1.419465e+000
   29 9.555556e-001 1.490495e+000 1.669566e+000
   30 1.000000e+000 1.570796e+000 1.949336e+000
Error Estimates      3.449248e-006 5.550227e-005
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#".. |

# differential_algebraic_eqs

Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, with optional additional constraints and user-defined linear system solver.

**Note**: imsl_f_differential_algebraic_eqs replaces imsl_f_dea_petzold_gear.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_differential_algebraic_eqs (*int* neq, *float* *t, *float* tend, *int* *ido, *float* y[], *float* yprime[], *void* gcn (), …, 0)

The type *double* function is imsl_d_differential_algebraic_eqs.

## Required Arguments

*int* neq (Input)
Number of dependent variables, and number of differential/algebraic equations, not counting any additional constraints.

*float* *t (Input/Output)
Set t to the starting value $t_0$ at the first step. On output, t is set to the value to which the integration has advanced. Normally, this new value is tend.

*float* tend (Input)
Final value of the independent variable. Update this value when re-entering after output with ido = 2.

*int* *ido (Input/Output)
Flag indicating the state of the computation.

| ido | State |
|---|---|
| 1 | Initial entry |
| 2 | Normal re-entry after obtaining output |
| 3 | Release workspace, last call |

The user sets ido = 1 on the first call at t = $t_0$. The function then sets ido =2, and this value is used for all but the last entry, which is made with ido = 3.

*float* y[] (Input/Output)
Array of length neq containing the dependent variable values, ***y***. On input, y must contain initial values. On output, y contains the computed solution at tend.

*float* `yprime[]` (Input/Output)

Array of length `neq` containing derivative values, ***y′***. This array must contain initial values, but they need not be such that $g(t, y, y') = 0$ at `t` $= t_0$. See the description of parameter `iypr` for more information.

*void* `gcn` (*int* `neq`, *float* `t`, *float* `y[]`, *float* `yprime[]`, *float* `delta[]`, *float* `d[]`, *int* `ldd`, *int* `*ires`) (Input)

User-supplied function to evaluate $g(t, y, y')$, and any constraints. Also partial derivative evaluations and optionally linear solving steps occur here. The equations $g(t, y, y') = 0$ consist of `neq` differential-algebraic equations of the form.

$$F_i\left(t, y_1, \ \ldots , y_{neq}, y'_1, \ \ldots , y'_{neq}\right) \equiv F_i\left(t, y, y'\right) = 0, \quad i = 1, \ \ldots , neq$$

The function `gcn` is also used to evaluate the `ncon` additional algebraic constraints.

$$C_i\left(t, y_1, \ \ldots , y_{neq}\right) \equiv C_i\left(t, y\right) = 0, \quad i = 1, \ \ldots , ncon, \quad ncon \ge 0$$

**Arguments**

*int* `neq` (Input)

Number of dependent variables, and number of differential–algebraic equations, not counting any additional constraints.

*float* `t` (Input)

Integration variable *t*.

*float* `y[]` (Input)

Array of `neq` dependent variables, ***y***.

*float* `yprime[]` (Input)

Array of `neq` derivative values, ***y′***.

*float* `delta[]` (Input/Output)

Array of length *max*(`neq`, `ncon`) containing residuals, $\delta$. See parameter `ires` for a definition.

*float* `d[]` (Input/Output

Array of length `ldd` × `neq` containing partial derivatives, ***d*** See parameter `ires` for a definition.

*int* `ldd` (Input)

Number of rows in `d`.

*int* `*ires` (Input/Output)

Flag indicating what is to be calculated in the user function, `gcn`.

*Note*: `ires` is input only, except when `ires` = 6. It is input/output when `ires` = 6. For a detailed description see the table below.

The code calls `gcn` with `ires` = 0, 1, 2, 3, 4, 5, 6, or 7, defined as follows:

| `ires` | **Description** |
|---|---|
| 0 | Do initializations which may be required in later calls to `gcn`. This is a setup flag that is input to gcn just once per problem.<br>Initializations might be computing parameters to be used internally by `gcn` or taking any other necessary steps for what may follow in terms of evaluating derivatives or linear solves.<br>Return and do nothing if no initializations are needed. |
| 1 | Compute $\delta_i = F_i(t, y, y')$, the *i*–th residual, for *i* =1,...,`neq`. |
| 2 | (Required only if `iujac` =1 and `matstr` = 0 or 1).<br>Compute<br>$d_{i,\,j} = \frac{\partial F_i(t,\,y,\,y')}{\partial y_j}$,<br>the partial derivative matrix. These are derivatives of $F_i$ with respect to $y_j$, for *i* =1,..., `neq` and *j* = 1,...,`neq`. |
| 3 | (Required only if `iujac` =1 and `matstr` = 0 or 1).<br>Compute<br>$d_{i,\,j} = \frac{\partial F_i(t,\,y,\,y')}{\partial y'_j}$,<br>the partial derivative of $F_i$ with respect to $y'_j$, for *i* = 1,...,`neq` and *j* = 1,...,`neq`. |
| 4 | (Required only if `iypr` = 2).<br>Compute<br>$\delta_i = \frac{\partial F_i(t,\,y,\,y')}{\partial t}$,<br>the partial derivative of $F_i$ with respect to *t*, for *i* =1,..., `neq`. |
| 5 | (Required only if `ncon` > 0).<br>Compute $\delta_i = C_i(t, y)$, the *i-th* residual in the additional constraints, for *i* =1,..., `ncon`, and<br>$d_{i,\,j} = \frac{\partial C_i(t,\,y)}{\partial y_j}$,<br>the partial derivative of $C_i$ with respect to $y_j$ for *i* =1,..., `ncon` and *j* =1,..., `neq`. |

| ires | Description |
|---|---|
| 6 | (Required only if `isolve` = 1.) If `matstr` = 2, the user must compute the matrix $A = \frac{\partial F}{\partial y} + cj\frac{\partial F}{\partial y'}$, where *cj* = $\delta_1$, and save this matrix in any user-defined format. This is for later use when `ires` = 7. The matrix may also be factored in this step, if desired. The array `d` is not referenced if `matstr` = 2. If `matstr` = 0 or 1, the *A* matrix will already be defined and passed to `gcn` in the array `d`, which will be in full matrix format if `matstr` = 0, and band matrix format, if `matstr` = 1. The user may factor `d` in this step, if desired. **Note:** For `matstr` = 0, 1, or 2, the user must set `ires` = 0 to signal that *A* is nonsingular. If *A* is nearly singular, leave `ires` = 6. This results in using a smaller step-size internally. |
| 7 | (Required only if `isolve` = 1.) The user must solve $Ax = b$, where $b$ is passed to `gcn` in the vector `delta`, and *x* is returned in `delta`. If `matstr` = 2, *A* is the matrix which was computed and saved at the call with `ires` = 6; if `matstr` = 0 or 1, *A* is passed to `gcn` in the array `d`. In either case, the *A* matrix will remain factored if the user factored it when `ires` = 6. |

# Synopsis with Optional Arguments

*#include* `<imsl.h>`

*void* `imsl_f_differential_algebraic_eqs` (*int* `neq`, *float* `*t`, *float* `tend`, *int* `*ido`, *float* `y[]`, *float* `yprime[]`, *void* `gcn()`,

`IMSL_N_CONSTRAINTS`, *int* `ncon`,

`IMSL_JACOBIAN_OPTION`, *int* `iujac`,

`IMSL_YPRIME_METHOD`, *int* `iypr`,

`IMSL_JACOBIAN_MATRIX_TYPE`, *int* `matstr`,

`IMSL_METHOD`, *int* `isolve`,

`IMSL_N_LOWER_DIAG`, *int* `ml`,

`IMSL_N_UPPER_DIAG`, *int* `mu`,

`IMSL_RELATIVE_TOLERANCE`, *float* `rtol`,

`IMSL_ABSOLUTE_TOLERANCE`, *float* `atol[]`,

`IMSL_INITIAL_STEPSIZE`, *float* `h0`,

`IMSL_MAX_STEPSIZE`, *float* `hmax`,

`IMSL_MAX_ORDER`, *int* `maxord`,

`IMSL_MAX_NUMBER_STEPS`, *int* `maxsteps`,

`IMSL_INTEGRATION_LIMIT`, *float* `tstop`,

`IMSL_ORDER_MAGNITUDE_EST`, *float* `fmag`,

`IMSL_GCN_W_DATA`, *void* `gcn()`, *void* `*data`,

0)

## Optional Arguments

`IMSL_N_CONSTRAINTS`, *int* `ncon` (Input)

Number of additional constraints.

Default: `ncon` = 0.

`IMSL_JACOBIAN_OPTION`, *int* `iujac` (Input)

Jacobian calculation option.

| **iujac** | **Description** |
|---|---|
| 0 | Calculates using finite difference approximations. |
| 1 | User supplies the Jacobian matrices of partial derivatives of $F_i, i = 1, \ldots, neq,$ in the function `gcn`, when `ires` = 2 and 3. |

Default: `iujac` = 0 for `matstr` = 0 or 1. `iujac` = 1 for `matstr` = 2.

`IMSL_YPRIME_METHOD`, *int* `iypr` (Input)

Initial *y′* calculation method.

| **iypr** | **Description** |
|---|---|
| 0 | The initial input values of `yprime` are already consistent with the input values of `Y`. That is $g(t, y, y') = 0$ at $t = t_0$. Any constraints must be satisfied at $t = t_0$. |
| 1 | Consistent values of `yprime` are calculated by Petzold's original `DASSL` algorithm. |
| 2 | Consistent values of `yprime` are calculated using a new algorithm [Hanson and Krogh, 2008], which is generally more robust but requires that `iujac` = 1 and `isolve` = 0, and additional derivatives corresponding to `ires` = 4 are to be calculated in `gcn`. |

Default: `iypr` = 1.

IMSL_JACOBIAN_MATRIX_TYPE, *int* matstr (Input)

Parameter specifying the Jacobian matrix structure.

| matstr | Description |
|---|---|
| 0 | The Jacobian matrices (whether iujac = 0 or 1) are to be stored in full storage mode. |
| 1 | The Jacobian matrices are to be stored in band storage mode. In this case, if iujac = 1, the partial derivative matrices have their entries for row *i and column j*, stored as array elements $d_{(i-j+mu+1,\ j)}$. This occurs when ires = 2 or 3 in gcn. |
| 2 | A user-defined matrix structure is used (see the documentation for 6 for more details). If matstr = 2, isolve and iujac are set to 1 internally. |

Default: matstr = 0.

IMSL_METHOD, *int* isolve (Input)

Solve method.

| isolve | Description |
|---|---|
| 0 | imsl_f_differential_algebraic_eqs solves the linear systems. |
| 1 | The user wishes to solve the linear system in function gcn. See parameter gcn for details. |

Default: isolve = 0 for matstr = 0 or 1. isolve = 1 for matstr = 2.

IMSL_N_LOWER_DIAG, *int* ml (Input)

Number of non-zero diagonals below the main diagonal in the Jacobian matrices when band storage mode is used. ml is ignored if matstr ≠ 1.

Default: ml = neq - 1.

IMSL_N_UPPER_DIAG, *int* mu (Input)

Number of non-zero diagonals above the main diagonal in the Jacobian matrices when band storage mode is used. mu is ignored if matstr ≠ 1.

Default: mu = neq - 1.

IMSL_RELATIVE_TOLERANCE, *float* rtol (Input)

Relative error tolerance for solver. The program attempts to maintain a local error in Y(*i*) less than rtol*|y[*i*]| + atol[*i*].

Default: rtol = $\sqrt{\varepsilon}$, where ε is machine precision.

IMSL_ABSOLUTE_TOLERANCE, *float* atol[] (Input)

Array of size neq containing absolute error tolerances. See description of rtol.

Default: atol[*i*] = 0.0.

IMSL_INITIAL_STEPSIZE, *float* h0 (Input)s

Initial stepsize used by the solver. If h0 = 0.0, the function defines the initial stepsize.

Default: h0 = 0.0.

IMSL_MAX_STEPSIZE, *float* hmax (Input)

Maximum stepsize used by the solver. If hmax = 0.0, the function defines the maximum stepsize.

Default: hmax = 0.0.

IMSL_MAX_ORDER, *int* maxord (Input)

Maximum order of the backward difference formulas used. 1 ≤ maxord ≤ 5.

Default: maxord = 5.

IMSL_MAX_NUMBER_STEPS, *int* maxsteps (Input)

Maximum number of steps allowed from t to tend.

Default: maxsteps = 500.

IMSL_INTEGRATION_LIMIT, *float* tstop (Input)

Integration limit point. For efficiency reasons, the code sometimes integrates past tend and interpolates a solution at tend. If a value for tstop is specified, the code will never integrate past t=tstop.

Default: No tstop value is specified.

IMSL_ORDER_MAGNITUDE_EST, *float* fmag (Input)

Order-of-magnitude estimate. fmag is used as an order-of-magnitude estimate of the magnitude of the functions $F_i$ (see description of gcn), for convergence testing, if iypr=2. fmag is ignored if iypr=0 or 1.

Default: fmag = 1.0.

IMSL_GCN_W_DATA, *void* gcn(*int* neq, *float* t, *float* y[], *float* yprime[], *float* delta[], *float* d[], *int* ldd, *int* *ires, *void* *data), *void* *data (Input)

User-supplied function to evaluate $g(t, y, y')$, and any constraints, 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. Please refer to gcn in the Required Arguments section for more information. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

Function imsl_f_differential_algebraic_eqs finds an approximation to the solution of a system of differential-algebraic equations $g(t, y, y') = 0$ with given initial data for $y$ and $y'$. The function uses BDF formulas, which are appropriate for stiff systems. imsl_f_differential_algebraic_eqs is based on the code DASSL designed by Linda Petzold [1982], and has been modified by Hanson and Krogh [2008]Solving Constrained Differential-Algebraic Systems Using Projections to allow the inclusion of additional constraints, including conservation principles, after each time step. The modified code also provides a more robust algorithm to calculate initial $y'$ values consistent with the given initial $y$ values. This occurs when the initial $y'$ are not known.

A differential-algebraic system of equations is said to have “index 0” if the Jacobian matrix of partial derivatives of the $F_i$ with respect to the $y'_j$ is nonsingular. Thus it is possible to solve for all the initial values of $y'_j$ and put the system in the form of a standard ODE system. If it is possible to reduce the system to a system of index 0 by taking first derivatives of some of the equations, the system has index 1, otherwise the index is greater than 1. See Brenan [1989] for a definition of index. `imsl_f_differential_algebraic_eqs` can generally only solve systems of index 0 or 1; other systems will usually have to be reduced to such a form through differentiation.

# Examples

## Example 1 – Method of Lines PDE Problem

This example solves the partial differential equation $U_t = U_{xx} + U$, with initial condition $U(x, 0) = 1 + x$, and boundary conditions $U(0, t) = e^t$, $U(1, t) = 2e^t$ which has exact solution $U(x, t) = (1 + x)e^t$. If we approximate the $U_{xx}$ term using finite differences, where $x_i = (i - 1)h$, and $h = 1/(n - 1)$, we get:

$$U(x_1, t) = e^t$$

$$U'(x_i, t) = \left[U(x_{i+1}, t) - 2U(x_i, t) + U(x_{i-1}, t)\right]/h^2 + U(x_i, t),\ i = 2, \ldots, n - 1$$

$$U(x_n, t) = 2e^t$$

If $Y_i(t) = U(x_i, t)$, the first and last equations are algebraic and the others are differential equations, so this is a system of differential-algebraic equations. The system has index = 1, since it could be transformed into an ODE system by differentiating the first and last equations. Note that the Jacobian matrices are banded (tridiagonal), with `ml` = `mu` = 1. We use this and specify the option for dealing with banded matrices in `imsl_f_differential_algebraic_eqs`.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

#define  NEQ  101
#define MAX(a,b) ((a) > (b)) ? (a) : (b)

void gcn(int neq, float t, float y[], float yprime[], float delta[],
   float *d, int ldd, int *ires);

int main() {
   int i, ido, iujac=1, iypr=1, matstr=1, ml=1, mu=1, nsteps=10;
   float errmax=0.0, hx, rtol=1.0e-4, t, tend, tr, x,
```

```
        y[NEQ], yprime[NEQ];

    hx = 1.0 / (float) (NEQ - 1);

    /* Initial values */
    for (i = 0; i < NEQ; i++) {
        yprime[i] = 0.0;
        x = ((float) i) * hx;
        y[i] = 1.0 + x;
    }

    /* Always set ido=1 on first call */
    ido = 1;

    for (i = 0; i < nsteps; i++) {
        /* Output solution at t=0.1,0.2,...,1.0 */
        t = 0.1 * (float) i;
        tend = 0.1 * (float) (i + 1);

        /* Set ido = 3 on last call */
        if (i == (nsteps-1))
            ido = 3;

        /* User-supplied jacobian matrix (iujac=1)
           Banded jacobian (matstr=1) */
        imsl_f_differential_algebraic_eqs(NEQ, &t, tend, &ido, y,
            yprime, gcn,
            IMSL_JACOBIAN_OPTION, iujac,
            IMSL_YPRIME_METHOD, iypr,
            IMSL_JACOBIAN_MATRIX_TYPE, matstr,
            IMSL_N_LOWER_DIAG, ml,
            IMSL_N_UPPER_DIAG, mu,
            IMSL_RELATIVE_TOLERANCE, rtol,
            0);
    }

    for (i = 0; i < NEQ; i++) {
        x = ((float) i) * hx;
        tr = (1.0 + x) * exp(t);
        errmax = MAX(errmax,fabs(y[i]-tr));
    }

    printf("Max Error at T=1 is %g.\n", errmax);
}


void gcn(int neq, float t, float y[], float yprime[], float delta[],
    float *d, int ldd, int *ires) {
#define D(I_,J_)  (*(d+(I_)*(neq)+(J_)))
    int i, j, mu;
    float hx;
```

```
    hx = 1.0 / (float) (neq - 1);
    mu = 1;

    if (*ires == 1) {
        /* f_i defined here */
        delta[0] = y[0] - exp(t);

        for (i = 1; i < (neq - 1); i++)
            delta[i] = -yprime[i] + (y[i+1] - 2.0 * y[i] + y[i-1]) /
                pow(hx,2) + y[i];

        delta[neq-1] = y[neq-1] - 2.0 * exp(t);
    } else if (*ires == 2) {
        /* d(i-j+mu+1,j) = d(f_i)/d(y_j)
           in band storage mode */
        D(mu,0) = 1.0;

        for (i = 1; i < (neq - 1); i++) {
            j = i;
            D(i-j+mu+1,j-1) = 1.0 / pow(hx,2);
            j = i + 1;
            D(i-j+mu+1,j-1) = -2.0 / pow(hx,2) + 1.0;
            j = i + 2;
            D(i-j+mu+1,j-1) = 1.0 / pow(hx,2);
        }

        D(mu,neq-1) = 1.0;
    } else if (*ires == 3) {
        /* d(i-j+mu+1,j) = d(f_i)/d(yprime_j ) */
        for (i = 1; i < (neq - 1); i++)
            D(mu,i) = -1.0;
    }
}
```

### Output

```
Max Error at T=1 is 5.53131e-005.
```

## Example 2 – Pendulum Problem

The first-order equations of motion of a point-mass *m* suspended on a massless wire of length $L$ under the influence of gravity, *mg*, and wire tension, $\lambda$, in Cartesian coordinates (*p*,*q*) are

$$p' = u$$
$$q' = v$$
$$mu' = -p\lambda$$
$$mv' = -q\lambda - mg$$
$$p^2 + q^2 - L^2 = 0$$

The problem above has an index number equal to 3, thus it cannot be solved with `imsl_f_differential_algebraic_eqs` directly. Unfortunately, the fact that the index is greater than 1 is not obvious, but an attempt to solve it will generally produce an error message stating the corrector equation did not converge, or if `iypr` = 2 an error message stating that the index appears to be greater than 1 should be issued. The user then differentiates the last equation, which after replacing ***p'*** by ***u*** and ***q'*** by ***v***, gives ***pu***+***qv*** = 0. This system still has index=2 (again not obvious, the user discovers this by unsuccessfully trying to solve the new system) and the last equation must be differentiated again, to finally (after appropriate substitutions) give the equation of total energy balance:

$$m\left(u^2 + v^2\right) - mgq - L^2\lambda = 0$$

With initial conditions and appropriate definitions of the dependent variables, the system becomes:

$$p(0) = L, q(0) = u(0) = v(0) = \lambda(0) = 0$$
$$y_1 = p$$
$$y_2 = q$$
$$y_3 = u$$
$$y_4 = v$$
$$y_5 = \lambda$$
$$F_1 = y_3 - y'_1 = 0$$
$$F_2 = y_4 - y'_2 = 0$$
$$F_3 = -y_1 y_5 - my'_3 = 0$$
$$F_4 = -y_2 y_5 - mg - my'_4 = 0$$
$$F_5 = m\left(y_3^2 + y_4^2\right) - mgy_2 - L^2 y_5 = 0$$

The initial conditions correspond to the pendulum starting in a horizontal position.

Since we have replaced the original constraint, $C_1 = p^2 + q^2 - L^2 = 0$, which requires that the pendulum length be ***L***, by differentiating it twice, this constraint is no longer explicitly enforced, and if we try to solve the above system alone (ie, with `ncon`=0), the pendulum length drifts substantially from ***L*** at larger times. `imsl_f_differential_algebraic_eqs` therefore allows the user to add additional constraints, to be re-enforced after each time step, so we add this original constraint, as well as the intermediate constraint $C_2 = pu + qv = 0$. Using these two supplementary constraints, (`ncon` = 2), the pendulum length is constant.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>
```

```
#define  NEQ  5

void gcn(int neq, float t, float y[], float yprime[], float delta[],
   float *d, int ldd, int *ires);

int main() {
   int i, ido, ncon=2, nsteps=5, iypr=2, iujac=1, maxsteps=50000;
  float atol[NEQ], len, t, tend, tol, y[NEQ], yprime[NEQ],
       mass=1.0, length=1.1, gravity=9.806650;

   /* Initial values */
  tol = 1.0e-5;
  for (i = 0; i < NEQ; i++) {
    y[i] = 0.0;
    yprime[i] = 0.0;
    atol[i] = tol;
  }
   y[0] = length;

   printf("       T                Y(0)           ");
   printf("Y(1)           Length\n");

  /* Always set ido=1 on first call */
  ido = 1;

  for (i = 0; i < nsteps; i++) {
    /* Output solution at t=10,20,30,40,50 */
    t = 10.0 * (float) i;
    tend = 10.0 * (float) (i + 1);

    /* Set ido = 3 on last call*/
    if (i == (nsteps-1))
      ido = 3;

    /* User-supplied jacobian matrix (iujac=1)
      Use new algorithm to get compatible y' */
       imsl_f_differential_algebraic_eqs(NEQ, &t, tend, &ido, y,
           yprime, gcn,
           IMSL_N_CONSTRAINTS, ncon,
           IMSL_JACOBIAN_OPTION, iujac,
           IMSL_YPRIME_METHOD, iypr,
           IMSL_RELATIVE_TOLERANCE, tol,
           IMSL_ABSOLUTE_TOLERANCE, atol,
           IMSL_MAX_NUMBER_STEPS, maxsteps,
           0);

     /* len = pendulum length (should be constant) */
    len = sqrt(pow(y[0],2) + pow(y[1],2));
    printf("%15.7f %15.7f %15.7f %15.7f\n", t, y[0], y[1], len);
   }
}
```

```
void gcn(int neq, float t, float y[], float yprime[], float delta[],
   float *d, int ldd, int *ires) {
#define D(I_,J_)  (*(d+(I_)*(neq)+(J_)))
   float lsq, mg, mass=1.0, length=1.1, gravity=9.806650;

   /* Simple swinging pendulum problem */
   mg = mass * gravity;
   lsq = pow(length,2);

   if (*ires == 1) {
       /* f_i defined here */
       delta[0] = y[2] - yprime[0];
       delta[1] = y[3] - yprime[1];
       delta[2] = -y[0] * y[4] - mass * yprime[2];
       delta[3] = -y[1] * y[4] - mass * yprime[3] - mg;
       delta[4] = mass * (pow(y[2],2) + pow(y[3],2)) -
           mg * y[1] - lsq * y[4];
   } else if (*ires == 2) {
       /* d(i,j) = d(f_i)/d(y_j) */
       D(0,2) = 1.0;
       D(1,3) = 1.0;
       D(2,0) = -y[4];
       D(2,4) = -y[0];
       D(3,1) = -y[4];
       D(3,4) = -y[1];
       D(4,1) = -mg;
       D(4,2) = mass * 2.0 * y[2];
       D(4,3) = mass * 2.0 * y[3];
       D(4,4) = -lsq;
   } else if (*ires == 3) {
       /* d(i,j) = d(f_i)/d(yprime_j) */
       D(0,0) = -1.0;
       D(1,1) = -1.0;
       D(2,2) = -mass;
       D(3,3) = -mass;
   } else if (*ires == 4) {
       /* delta(i) = d(f_i)/dt */
       delta[0] = 0.0;
       delta[1] = 0.0;
       delta[2] = 0.0;
       delta[3] = 0.0;
       delta[4] = 0.0;
   } else if (*ires == 5) {
       /* delta(i) = g_i
          d(i,j) = d(g_i)/d(y_j) */
       delta[0] = pow(y[0],2) + pow(y[1],2) - lsq;
       delta[1] = y[0]*y[2] + y[1]*y[3];
       D(0,0) = 2.0 * y[0];
       D(0,1) = 2.0 * y[1];
```

```
            D(0,2) = 0.0;
            D(0,3) = 0.0;
            D(0,4) = 0.0;
            D(1,0) = y[2];
            D(1,1) = y[3];
            D(1,2) = y[0];
            D(1,3) = y[1];
            D(1,4) = 0.0;
        }
    }
```

### Output

```
       T               Y(0)          Y(1)          Length
   10.0000000       1.0998138     -0.0202353      1.0999999
   20.0000000       1.0970403     -0.0806356      1.0999998
   30.0000000       1.0852183     -0.1797250      1.0999999
   40.0000000       1.0541573     -0.3142486      1.0999999
   50.0000000       0.9912723     -0.4768429      1.1000000
```

## Example 3 – User Solves Linear System

Consider the system of ordinary differential equations, $y' = By$, where $B$ is the bi-diagonal matrix with $(-1, -1/2, -1/3, \ldots, -1/(n-1), 0)$ on the main diagonal and with 1's along the first sub-diagonal. The initial condition is $y(0) = (1,0,0,\ldots,0)^T$, and since $y'(0) = By(0) = (-1,1,0,\ldots,0)^T$, $y'(0)$ is also known for this problem.

Since $B^T v = 0$, where $v_i = 1/(i-1)!$, $v$ is an eigenvector of $B^T$ corresponding to the eigenvalue 0. Thus

$$0 = v^T\left(y' - By\right) = v^T y' - \left(B^T v\right)^T y = v^T y' = \left(v^T y\right)'$$

so $v^T y(t)$ is constant. Since it has the value $v^T y(0) = v_1 = 1$ at $t = 0$, the constraint $v^T y(t) = 1$ is satisfied for all $t$. This constraint is imposed in this example.

This example also illustrates how the user can solve his/her own linear systems (`matstr`=2). Normally, when `ires` = 6, the matrix

$$A = \frac{\partial g}{\partial y} + cj\frac{\partial g}{\partial y'}$$

is computed, saved and possibly factored, using a sparse matrix factorization function of the user's choice. Then when `ires`=7, the system $Ax$ = `delta` is solved, using the matrix $B$ saved and factored earlier, and the solution is returned in `delta`. In this case, $B$ is just a bidiagonal matrix, so there is no need to save or factor $A$ when `ires` = 6, since a bi-diagonal system can be solved directly using forward substitution, when `ires` = 7.

```
#include <imsl.h>
#include <stdio.h>

#define  NEQ   100
```

```
void gcn(int neq, float t, float y[], float yprime[], float delta[],
   float *d, int ldd, int *ires);

int main() {
   int i, ido, nsteps=10, ncon=1, iypr=0, matstr=2;
   float atol[NEQ], con=0.0, t, tend, v[NEQ], y[NEQ], yprime[NEQ];

   /* a^t eigenvector v */
   v[0] = 1.0;
   for (i = 1; i < NEQ; i++)
       v[i] = v[i-1] / (float) i;
   /* initial values */
   for (i = 0; i < NEQ; i++) {
       y[i] = 0.0;
       yprime[i] = 0.0;
       atol[i] = 1.0e-4;
   }

   y[0] = 1.0;
   yprime[0] = -1.0;
   yprime[1] = 1.0;

   /* always set ido=1 on first call */
   ido = 1;
   for (i = 0; i < nsteps; i++) {
       /* output solution at t=1,2,...,10 */
       t = (float) i;
       tend = (float) (i + 1);

       /* set ido = 3 on last call */
       if (i == (nsteps-1))
           ido = 3;

       /* user-defined jacobian matrix structure (matstr=2) */
       imsl_f_differential_algebraic_eqs(NEQ, &t, tend, &ido, y,
           yprime, gcn,
           IMSL_N_CONSTRAINTS, ncon,
           IMSL_YPRIME_METHOD, iypr,
           IMSL_JACOBIAN_MATRIX_TYPE, matstr,
           IMSL_ABSOLUTE_TOLERANCE, atol,
           0);
   }
   /* check if solution satisfies constraint */
   for (i = 0; i < NEQ; i++)
       con += v[i] * y[i];

   printf(" V dot Y = %f\n", con);
}

void gcn(int neq, float t, float y[], float yprime[], float delta[],
   float *d, int ldd, int *ires)
```

```
{
#define D(I_,J_) (*(d+(I_)*(neq)+(J_)))
   int i;
   float con, v[NEQ];
   static float cj;

   v[0] = 1.0;
   for (i = 1; i < NEQ; i++)
       v[i] = v[i-1] / (float) i;

   if (*ires == 1) {
       /* f_i defined here */
       delta[0] = yprime[0] + y[0];

       for (i = 1; i < (NEQ - 1); i++)
           delta[i] = yprime[i] - y[i-1] + y[i] / (float) (i + 1);

       delta[NEQ-1] = yprime[NEQ-1] - y[NEQ-2];
   } else if (*ires == 5) {
       /* constraint is v dot y = 1 */
       con = -1.0;

       for (i = 0; i < NEQ; i++) {
           con += v[i]*y[i];
           D(0,i) = v[i];
       }

       delta[0] = con;
   } else if (*ires == 6) {
       /* normally, compute matrix
       a = df/dy + cj*df/dy' = -b + cj*i
       here. only cj needs to be saved in this case, however,
       since b is bidiagonal, so a*x=delta can be solved (ires=7)
       without saving or factoring b. */
       cj = delta[0];

       /* if cj > 0 not close to zero, a is nonsingular,
       so set ires = 0. */
       if (cj >= 1.0e-4)
           *ires = 0;
   } else if (*ires == 7) {
       /* solve a*x=delta and return x in delta. */
       delta[0] /= 1.0 + cj;

       for (i = 1; i < (NEQ - 1); i++)
           delta[i] = (delta[i] + delta[i-1]) /
           (1.0 / (float) (i + 1) + cj);

       delta[NEQ-1] = (delta[NEQ-1] + delta[NEQ-2]) / cj;
   }
}
```

### Output

```
V dot Y = 1.000000
```

## Fatal Errors

| | |
|---|---|
| IMSL_SYSTEM_CONVERGENCE | The system has index # but convergence of "yprime" values was not achieved. |
| IMSL_SYSTEM_CONVERGENCE | The system appears to have index > 1. |
| IMSL_SYS_INDEX_GT_ONE | For the Pareto distribution, the Hessian cannot be calculated because the parameter estimate is 0. |
| IMSL_SYS_NOT_DIFF_ALG | This is not a differential-algebraic system. |
| IMSL_ACCURACY_EXCEEDED_1 | Accuracy requested exceeds machine precision. |
| IMSL_STEPS_EXCEEDED | More than "maxsteps"=# steps taken between output points. |
| IMSL_ERROR_TEST_FAILURE_2 | The error test has failed repeatedly. |
| IMSL_CORRECTOR_FAILED_3 | The corrector iteration failed repeatedly to converge. |
| IMSL_SINGULAR_MATRIX_1 | The iteration matrix is singular. |
| IMSL_UNABLE_TO_SOLVE_YPR | Unable to solve for initial "yprime". |
| IMSL_TEND_GT_TSTOP | "tend" is greater than "tstop". |
| IMSL_TEND_CLOSE_TO_T | "tend" is too close to "t". |
| IMSL_TSTOP_INCONSIST_T | "tstop" is not consistent with"t". |
| IMSL_CONSTRAINTS_INCONSIST | Constraints appear inconsistent |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# dea_petzold_gear

***Note***: This function is deprecated and has been replaced by `differential_algebraic_eqs`. To view the deprecated documentation, see dea_petzold_gear.pdf on the Rogue Wave website. You can also access a local copy in your IMSL installation directory at `pdf\deprecated_routines\math\dea_petzold_gear.pdf`.

# ode_adams_2nd_order

***Note***: This function is deprecated and has been replaced by `imsl_f_ode_adams_krogh`. To view the deprecated documentation, see imsl_f_ode_adams_2nd_order.pdf on the Rogue Wave website. You can also access a local copy in your IMSL installation directory at `pdf\deprecated_routines\math\ode_adams_2nd_order.pdf`.

# ode_adams_krogh

**Note**: imsl_f_ode_adams_krogh replaces imsl_f_ode_adams_gear and imsl_f_ode_adams_2nd_order.

Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_ode_adams_krogh (*int* neq, *float* *t, *float* tend, *int* *ido, *float* y[], *float* hidrvs[], *void* fcn(), …, 0)

The type ***double*** function is imsl_d_ode_adams_krogh.

## Required Arguments

*int* neq (Input)
Number of differential equations in the system of equations to solve.

*float* *t (Input/Output)
On input, t contains the initial independent variable value. On output, t is replaced by tend unless error conditions arise. See ido for details.

*float* tend (Input)
Value of ***t = tend*** where the solution is required.

*int* *ido (Input/Output)
Flag indicating the state of the computation.

| ido | State |
|---|---|
| 1 | Initial entry input value. |
| 2 | Normal re-entry input value. On output, if ido = 2 then the integration is finished. If the integrator is called with a new value for tend, the integration continues. If the integrator is called with tend unchanged, an error message is issued. |
| 3 | Input value to use on final call to release workspace. |
| >3 | Output value that indicates that a fatal error has occurred. |

The initial call is made with ido = 1. The function then sets ido = 2, and this value is used for all but the last call that is made with ido = 3. This final call is only used to release workspace which was automatically allocated by the initial call with ido = 1.

*float* `y[]` (Input/Output)

An array of length $k$ containing the dependent variables, $y(t)$, and first derivatives, if any. $k$ will be the sum of the orders of the equations in the system of equations to solve, that is, the sum of the elements of `korder`. On input, `y` contains the initial values, $y(t_0)$ and $y'(t_0)$ (if needed). On output, `y` contains the approximate solution, $y(t)$. For example, for a system of first order equations, `y`$[i-1]$ is the $i$–th dependent variable. For a system of second order equations, `y`$[2i-2]$ is the $i$–th dependent variable and `y`$[2i-1]$ is the derivative of the $i$–th dependent variable. For systems of equations in which one or more equations is of order 2, optional argument `IMSL_EQ_ORDER` must be used to denote the order of each equation so that the derivatives in `y` can be identified. By default it is assumed that all equations are of order 1 and `y` contains only dependent variables.

*float* `hidrvs[]` (Output)

An array of length `neq` containing the highest order derivatives at the point `y`.

*void* `fcn` (*int* `neq`, *int* `ido`, *float* `t`, *float* `y[]`, *float* `hidrvs`) (Input)

User-supplied function to evaluate derivatives.

**Arguments**

*int* `neq` (Input)

Number of differential equations in the system of equations to solve.

*int* `ido` (Input)

Flag indicating the state of the computation. This flag corresponds to the `ido` argument described above. If `fcn` has complicated subexpressions, which depend only weakly or not at all on `y` then these subexpressions need only be computed when `ido` = 1 and their values then reused when `ido` = 2.

*float* `t` (Input)

Independent variable, $t$.

*float* `y[]` (Input)

An array of length $k$ containing the dependent variable values, $y$, and first derivatives, if any. $k$ will be the sum of the orders of the equations in the system of equations to solve.

*float* `hidrvs[]` (Output)

An array of length `neq` containing the values of the highest order derivatives evaluated at $(t, y)$.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*void* `imsl_f_ode_adams_krogh` (*int* `neq`, *float* `*t`, *float* `tend`, *int* `*ido`, *float* `y[]`,
*float* `hidrvs[]`, *void* `fcn()`,

`IMSL_EQ_ORDER`, *int* `korder[]`,

IMSL_EQ_ERR, *float* eqnerr[],

IMSL_STEPSIZE_INC, *float* hinc,

IMSL_STEPSIZE_DEC, *float* hdec,

IMSL_MIN_STEPSIZE, *float* hmin,

IMSL_MAX_STEPSIZE, *float* hmax,

IMSL_FCN_W_DATA, *void* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_EQ_ORDER, *int* korder[] (Input)

An array of length neq specifying the orders of the equations in the system of equations to solve. The elements of korder can be 1 or 2. korder must be used with argument y to define systems of mixed or higher order.

Default: korder = [1,1,1,...,1].

IMSL_EQ_ERR, *float* eqnerr[] (Input)

An array of length neq specifying the error tolerance for each equation. Let *e(i)* be the error tolerance for equation *i* for *i* = 0,..., neq −1. Then

| Value | Explanation |
|---|---|
| *e(i)* > 0 | Implies an absolute error tolerance of *e(i)* is to be used for equation *i*. |
| *e(i)* = 0 | Implies no error checking is to be performed for equation *i*. |
| *e(i)* < 0 | Implies a relative error test is to be performed for equation *i*. In this case, the base error tolerance used will be \|*e(i)*\| and the relative error factor used will be (15/16 * \|*e(i)*\|). Thus the actual absolute error tolerance used will be \|*e*(*i*)\|×(15/16×\|*e*(*i*)\|). |

Default: An absolute error tolerance of 1.e−5 is used for single precision and 1.e−10 for double precision for all equations.

IMSL_STEPSIZE_INC, *float* hinc (Input)

Factor used for increasing the stepsize. One should set hinc such that 9/8 <= hinc <= 4.

Default: hinc = 2.0.

IMSL_STEPSIZE_DEC, *float* hdec (Input)

Factor used for decreasing the stepsize. One should set hdec such that 1/4 <= hdec <= 7/8.

Default: hdec = 0.5.

`IMSL_MIN_STEPSIZE`, *float* `hmin` (Input)

Absolute value of the minimum stepsize permitted.

Default: `hmin` = 10.0/`imsl_f_machine`(2).

`IMSL_MAX_STEPSIZE`, *float* `hmax` (Input)

Absolute value of the maximum stepsize permitted.

Default: `hmax` = `imsl_f_machine`(2).

`IMSL_FCN_W_DATA`, *void* `fcn`(*int* `neq`, *int* `ido`, *float* `t`, *float* `y[]`, *float* `hidrvs[]`, *void* `*data`), *void* `*data` (Input)

User-supplied function to evaluate functions, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. Please refer to the `fcn` argument in the Required Arguments section for more information. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

`imsl_f_ode_adams_krogh` is based on the JPL Library routine `SIVA`. `imsl_f_ode_adams_krogh` uses a variable order Adams method to solve the initial value problem

$$\left.\begin{aligned}\frac{dy_i}{dt} &= f_i\left(t, y_1, y_2, \ldots, y_{neq}\right)\\ y_i\left(t_0\right) &= \eta_i\end{aligned}\right\}, i = 1,2, \ldots, neq$$

or more generally

$$z_i^{\left(d_i\right)} = f_i\left(t, y\right), y\left(t_0\right) = \eta_0,\ i = 1,2, \ldots, neq,$$

where ***y*** is the vector

$$\left(z_1, z'_1, \ldots, z_1^{\left(d_1-1\right)}, z_2, \ldots, z_{neq}^{\left(d_{neq}-1\right)}\right),$$

$z_i^{(k)}$ is the ***kth*** derivative of $z_i$ with respect to ***t***, $d_i$ is the order of the ***ith*** differential equation, and η is a vector with the same dimension as ***y***.

Note that the systems of equations solved by `imsl_f_ode_adams_krogh` can be of order one, order two, or mixed order one and two.

See "Changing Stepsize in the Integration of Differential Equations Using Modified Divided Differences," Krogh (1974).

# Examples

## Example 1

In this example a system of two equations of order two is solved.

$$Y''_1 = -Y_1 / \left( \left( Y_1^2 + Y_2^2 \right)^{\frac{3}{2}} \right)$$

$$Y''_2 = -Y_2 / \left( \left( Y_1^2 + Y_2^2 \right)^{\frac{3}{2}} \right)$$

The initial conditions are

$$Y_1\left(0\right) = 1.0,\ Y'_1\left(0\right) = 0.0,\ Y_2\left(0\right) = 0.0,\ Y'_2\left(0\right) = 1.0$$

Since the system is of order two, optional argument `imsl_eq_order` must be used to specify the orders of the equations. Also, because the system is of order two, `y`*[0]* contains the first dependent variable, `y[1]` contains the derivative of the first dependent variable, `y[2]` contains the second dependent variable, and `y[3]` contains the derivative of the second dependent variable.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

#define     NEQ   2

void fcn(int neq, int ido, float t, float y[], float hidrvs[]);

int main() {
   int iend, ido, k, korder[NEQ];
   float delta, t, tend, y[4], hidrvs[NEQ];

   /* Initialize */

   ido = 1;
   t = 0.0;
   y[0] = 1.0;
   y[1] = 0.0;
   y[2] = 0.0;
   y[3] = 1.0;
   korder[0] = 2;
   korder[1] = 2;

   /* Write Title */

   printf("          T            Y1/Y2         Y1P/Y2P       ");
   printf("Y1PP/Y2PP\n");

   /* Integrate ODE */
```

```
   iend = 0;
   delta = 2.0 * imsl_f_constant("PI",0);
   for(k=0;k<5;k++){
       iend += 1;
       tend = t + delta;
       if(tend > 20.0) tend = 20.0;
       imsl_f_ode_adams_krogh (NEQ, &t, tend, &ido, y, hidrvs, fcn,
           IMSL_EQ_ORDER, korder,
           0);
       if(iend < 5){
           printf("%15.4f %15.4f %15.4f %15.4f\n",
               t, y[0], y[1], hidrvs[0]);
           printf("                %15.4f %15.4f %15.4f\n",
           y[2], y[3], hidrvs[1]);
       }

       /*   Finish up */

       if (iend == 4) ido = 3;
   }
}


void fcn(int neq, int ido, float t, float y[], float hidrvs[])
{
   float tp;

   tp = y[0] * y[0] + y[2] * y[2];
   tp = 1.0e0/(tp * sqrt(tp));
   hidrvs[0] = -y[0] * tp;
   hidrvs[1] = -y[2] * tp;
}
```

### Output

```
   T           Y1/Y2       Y1P/Y2P     Y1PP/Y2PP
 6.2832        1.0000      -0.0000      -1.0000
                0.0000       1.0000       0.0000
12.5664        1.0000      -0.0000      -1.0000
                0.0000       1.0000      -0.0000
18.8496        1.0000      -0.0000      -1.0000
                0.0000       1.0000      -0.0000
20.0000        0.4081      -0.9129      -0.4081
                0.9129       0.4081      -0.9129
```

## Example 2

This contrived example illustrates how to use `imsl_f_ode_adams_krogh` to solve a system of equations of mixed order.

The height, ***y(t)***, of an object of mass ***m*** above the surface of the Earth can be modeled using Newton's second law as:

$$my'' = -mg - ky'$$

or

$$y'' = -g - (k/m)y'$$

where ***–mg*** is the downward force of gravity and ***–ky'*** is the force due to air resistance, in a direction opposing the velocity. If the object is a meteor, the mass, ***m,*** and air resistance, ***k***, will decrease as the meteor burns up in the atmosphere. The mass is proportional to $r^3$ ($r$ = radius) and the air resistance, presumably dependent on the surface area, may be assumed to be proportional to $r^2$, so that $k/m = k_0/r$. The rate at which the meteor's radius decreases as it burns up may depend on ***r***, on the velocity ***y'***, and, since the density of the atmosphere depends on ***y***, on ***y*** itself. However, we will construct a very simple model where the rate is just proportional to the square of the velocity,

$$r' = -c_0(y')^2$$

We solve (1) and (2), with $k_0$ = 0.005, $c_0 = 10^{-8}$, $g$ = 9.8 and initial conditions ***y(0)*** = 100,000 meters, ***y'(0)*** = –1000 meters/second, ***r(0)*** = 1 meter.

```
#include <imsl.h>
#include <stdio.h>

#define     NEQ  2

void fcn(int neq, int ido, float t, float y[], float hidrvs[]);

int main() {
   int iend, ido, k, korder[NEQ];
   float delta, t, tend, y[3], eqnerr[NEQ], hidrvs[NEQ];

   /* Initialize */
   ido = 1;
   t = 0.0;
   y[0] = 100000.0;
   y[1] = -1000.0;
   y[2] = 1.0;
   korder[0] = 2;
   korder[1] = 1;
   eqnerr[0] = .003;
   eqnerr[1] = .003;

   /* Write Title */
   printf("          T           Y1/Y2           Y1P          ");
   printf("Y1PP/Y2PP\n");

   /* Integrate ODE */
   iend = 0;
   delta = 10.0;

   for(k=0;k<6;k++){
       iend += 1;
       tend = t + delta;
       if(tend > 50.0) tend = 50.0;
       imsl_f_ode_adams_krogh (NEQ, &t, tend, &ido, y, hidrvs, fcn,
           IMSL_EQ_ORDER, korder,
           IMSL_EQ_ERR, eqnerr,
           0);
       if(iend < 6){
           printf("%15.4f %15.4f %15.4f %15.4f\n",
               t, y[0], y[1], hidrvs[0]);
           printf("               %15.4f                %15.4f\n",
               y[2], hidrvs[1]);
       }

   /*   Finish up */
      if (iend == 5) ido = 3;
  }
}
```

```
void fcn(int neq, int ido, float t, float y[], float hidrvs[])
{
   hidrvs[0] = -9.8 - .005/y[2]*y[1];
   hidrvs[1] = -1.e-8 * y[1] * y[1];
}
```

### Output

```
      T          Y1/Y2          Y1P           Y1PP/Y2PP
   10.0000     89773.0391   -1044.0096         -3.9701
                    0.8954                       -0.0109
   20.0000     79150.9922   -1078.6333         -2.9083
                    0.7826                       -0.0116
   30.0000     68240.9531   -1101.0377         -1.5031
                    0.6635                       -0.0121
   40.0000     57184.9180   -1106.9633          0.4253
                    0.5413                       -0.0123
   50.0000     46178.1445   -1089.8291          3.1699
                    0.4201                       -0.0119
```

## Warning Errors

| | |
|---|---|
| IMSL_ TOLERANCE_TOO_SMALL | The requested error tolerance, # is too small. Using # instead. |
| IMSL_ RESTART | The stepsize has been reduced too rapidly The integrator is going to do a restart. |

## Fatal Errors

| | |
|---|---|
| IMSL_ADJUST_STEPSIZE1 | The current step length = #, is less than the minimum steplength, "hmin" = #, at the conclusion of the starting phase of the integration. Decreasing "hmin" to a value less than or equal to # may help. |
| IMSL_ADJUST_STEPSIZE2 | The integrator needs to take a step smaller than # in order to maintain the requested local error. Decreasing "hmin" to a value less than or equal to # may help. |
| IMSL_INCORRECT_TEND | Either the new output point precedes the last one or it has the same value. "tend" = #. |

| | |
|---|---|
| `IMSL_ADJUST_ERROR_TOLERANCE` | The step length, H = #, is so small that when Tn + H is formed, the result will be the same as `Tn`, where `Tn` is the base value of the independent variable. If this problem is not due to a nonintegrable singularity, it can probably be corrected by translating “`t`” so that it is closer to 0. Reducing the error tolerance for the equations through argument “`eqnerr`” may also help with this problem. |
| `IMSL_ERROR_TOLERANCE` | A local error tolerance of zero has been requested. |
| `IMSL_ERROR_PREVIOUS` | A fatal error has occurred because of the error reported in the previous error message. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# Introduction to pde_1d_mg

The section describes an algorithm and a corresponding integrator function `imsl_f_pde_1d_mg` for solving a system of partial differential equations

**Equation 1**

$$u_t \equiv \frac{\partial u}{\partial t} = f\left(u, x, t\right), x_L < x < x_R, t > t_0$$

This software is a one-dimensional differential equation solver. It requires the user to provide initial and boundary conditions in addition to a function for the evaluation of $u_t$. The integration method is noteworthy due to the maintenance of grid lines in the space variable, $x$. Details for choosing new grid lines are given in Blom and Zegeling, (1994). The class of problems solved with `imsl_f_pde_1d_mg` is expressed by ***Equation 1*** and given in more detail by:

**Equation 2**

$$\sum_{k=1}^{NPDE} C_{j,k}\left(x, t, u, u_x\right)\frac{\partial u^k}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m R_j\left(x, t, u, u_x\right)\right) - Q_j\left(x, t, u, u_x\right),$$
$$j = 1, \ \ldots, NPDE, \quad x_L < x < x_R, \quad t > t_0, \quad m \in \{0, 1, 2\}$$

The vector $u \equiv \left[u^1, \ \ldots, u^{NPDE}\right]^T$ is the solution. The integer value $NPDE \ge 1$ is the number of differential equations. The functions $R_j$ and $Q_j$ can be regarded, in special cases, as flux and source terms. The functions $u, C_{j,k}, R_j, Q_j$ are expected to be continuous. Allowed values for the integer $m$ are any of $m = 0, 1, 2$. These are respectively for problems in Cartesian, cylindrical or polar, and spherical coordinates. In the two cases with $m > 0$, the interval $\left[x_L, x_R\right]$ must not contain $x = 0$ as an interior point.

The boundary conditions have the master equation form

**Equation 3**

$$\beta_j\left(x,t\right)R_j\left(x,t,u,u_x\right) = \gamma_j\left(x,t,u,u_x\right),$$
$$\text{at } x = x_L \text{ and } x = x_R, \ j = 1,\ldots NPDE$$

In the boundary conditions the functions $\beta_j$ and $\gamma_j$ are continuous. In the two cases with $m > 0$, with an end-point of $\left[x_L, x_R\right]$ at 0, the finite value of the solution at $x = 0$ must be ensured. This requires the specification of the solution at $x = 0$, or it implies that $R_j|_{x=x_L} = 0$ or $R_j|_{x=x_R} = 0$. The initial values satisfy $u\left(x, t_0\right) = u_0\left(x\right), \quad x \in \left[x_L, x_R\right]$, where $u_0$ is a piece-wise continuous vector function of $x$ with $NPDE$ components.

The user must pose the problem so that mathematical definitions are known for the functions

$$C_{j,k}, R_j, Q_j, \beta_j, \gamma_j \text{ and } u_0$$

These functions are provided to the function `imsl_f_pde_1d_mg` in the form of two user-supplied functions. This form of the usage interface is explained below and illustrated with several examples. $u_0$ can be supplied as the input argument `u` or by an optional user-supplied function. Users comfortable with the description of this algorithm may skip directly to the Examples section.

## Description Summary

Equation 1 is approximated at $N = ngrids$ time-dependent grid values $x_L = x_0 < x_1 < \ \ldots \ < x_i(t) < \ \ldots \ < x_{N+1} = x_R$. Using the total differential $\frac{du}{dt} = u_t + u_x \frac{dx}{dt}$ transforms the differential equation to the form

$$u_t = \frac{du}{dt} - u_x \frac{dx}{dt} = f(u, x, t), \quad x_L < x < x_R$$

Using central divided differences for the factor $u_x$ leads to the system of ordinary differential equations in implicit form

$$\frac{dU_\mathrm{i}}{dt} - \frac{(U_{\mathrm{i}+1} - U_{\mathrm{i}-1})}{(x_{\mathrm{i}+1} - x_{\mathrm{i}-1})} \frac{dx_\mathrm{i}}{dt} = F_\mathrm{i}, \quad t > t_0, \quad i = 1, \cdots, N$$

The terms $U_i, F_i$ respectively represent the approximate solution to the partial differential equation and the value of $f(u, x, t)$ at the point $(u, x, t) = (U_i, x_i(t), t)$. The truncation error from this approximation is second-order in the space variable $x$. The above ordinary differential equations are underdetermined, so additional equations are added for determining the time-dependent grid points. These additional equations contain parameters that can be adjusted by the user. Often it will be necessary to modify these parameters to solve a difficult problem. For this purpose the following quantities are needed:

$$\begin{aligned} &\Delta x_i = x_{i+1} - x_i, \quad n_i = \Delta x_i^{-1} \\ &\mu_i = n_i - \kappa(\kappa + 1)(n_{i+1} - 2n_i + n_{i-1}), \quad 0 \le i \le N \\ &n_{-1} \equiv n_0, \quad n_{N+1} \equiv n_N \end{aligned}$$

The values $n_i$ are the so-called point concentration of the grid. The parameter $\kappa \ge 0$ denotes a spatial smoothing value. Now the grid points are defined implicitly so that

$$\frac{\mu_{i-1} + \tau \frac{du_{i-1}}{dt}}{M_{i-1}} = \frac{\mu_i + \tau \frac{du_i}{dt}}{M_i}, \quad 1 \le i \le N$$

The parameter $\tau \ge 0$ denotes a time-smoothing value. If the value $\tau$ is chosen to be large, this results in a fixed spatial grid. Increasing $\tau$ from its default value avoids the error condition where grid lines cross. The divisors are defined by

$$M_i^2 = \alpha + NPDE^{-1} \sum_{j=1}^{NPDE} \frac{\left( U_{i+1}^j - U_i^j \right)^2}{\left( \Delta x_i \right)^2}$$

The value $\kappa$ determines the level of clustering or spatial smoothing of the grid points. Decreasing $\kappa$ from its default values also decreases the amount of spatial smoothing. The parameters $M_i$ approximate arc length and help determine the shape of the grid or $x_i$ distribution. The parameter $\tau$ prevents the grid movement from adjusting immediately to new values of the $M_i$, thereby avoiding oscillations in the grid that cause large relative errors in the solution. This is important when applied to solutions with steep gradients.

The discrete form of the differential equation and the smoothing equations are combined to yield the implicit system of differential equations

$$A(Y)\frac{dY}{dt} = L\left( Y \right),$$
$$Y = \left[ U_1^1, \ldots, U_1^{NPDE}, x_1, \ldots \right]^T$$

This is usually a stiff differential-algebraic system. It is solved using the integrator `imsl_f_dea_petzold_gear`. If `imsl_f_dea_petzold_gear` is needed during the evaluations of the differential equations or boundary conditions, it must be done in a separate thread to avoid possible problems with `imsl_f_pde_1d_mg`'s internal use of `imsl_f_dea_petzold_gear`. The only options for `imsl_f_dea_petzold_gear` set by `imsl_f_pde_1d_mg` are the Maximum BDF Order, and the absolute and relative error values, documented as `IMSL_MAX_BDF_ORDER`, and `IMSL_ATOL_RTOL_SCALARS`.

# pde_1d_mg

Solves a system of one-dimensional time-dependent partial differential equations using a moving-grid interface.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_pde_1d_mg_mgr (*int* task, *void* **state, …, 0)

*void* imsl_f_pde_1d_mg (*int* npdes, *int* ngrids, *float* *t, *float* tend *float* u[], *float* xl, *float* xr, *void* *state, *void* pde_systems(), *void* boundary_conditions(), …, 0)

The *void* functions imsl_d_pde_1d_mg_mgr and imsl_d_pde_1d_mg are for *double* type arithmetic accuracy.

The function imsl_f_pde_1d_mg_mgr is used to initialize and reset the problem, and the function imsl_f_pde_1d_mg is the integrator. The descriptions of both functions are provided below.

**NOTE**: The integrator is provided with single or double precision arithmetic. We recommend using the double precision interface imsl_d_pde_1d_mg.

## Required Arguments for imsl_f_pde_1d_mg_mgr

*int* task (Input)

This function must be called with task set to IMSL_PDE_INITIALIZE to set up for solving a system and with task equal to IMSL_PDE_RESET to clean up after it has been solved. These values for task are defined in the include file, imsl.h.

*void* **state (Input/Output)

The current state of the PDE solution is held in a structure pointed to by state. It cannot be directly manipulated.

## Required Arguments for imsl_f_pde_1d_mg

*int* npdes (Input)

The number of differential equations.

*int* ngrids (Input)

The number of spatial grid/mesh points, including the boundary points $x_L$ and $x_R$.

*float* *t (Input/Output)

On input, t is the initial independent variable value. On output, t is replaced by tend, unless error conditions arise. This is first set to the value of the independent variable $t_0$ where the integration of $u_t$ begins. It is set to the value tend on return.

*float* tend (Input)

Mathematical value of $t$ where the integration of $u_t$ ends. Note: Starting values of t < tend imply integration in the forward direction, while values of t > tend imply integration in the backward direction. Either direction is permitted.

*float* u[] (Input/Output)

Array of size npdes+1 by ngrids. On input, the first npdes rows contain initial values for all components of the system at the equally spaced grid of values. It is not required to define the grid values in the last row of u. On output u[] contains the approximate solution value ***$U_i(x_j(tend), tend)$*** at array location u[*i*×ngrids+*j*]. The grid value ***$x_j(tend)$*** is in location u[(npdes*ngrids) +*j*]. Normally the grid values are equally spaced as the integration starts. Variable grid values can be provided by defining them as output from the user function initial_conditions supplied by either imsl_f_pde_1d_mg_mgr's IMSL_INITIAL_CONDITIONS, or IMSL_INITIAL_CONDITIONS_W_DATA optional arguments.

*float* xl (Input)

Lower grid boundary, $x_L$.

*float* xr (Input)

Upper grid boundary, $x_R$.

*void* *state (Input/Output)

The current state of the solution is held in a structure pointed to by state. It must be initialized by a call to imsl_f_pde_1d_mg_mgr. It cannot be directly manipulated.

*void* pde_systems(*float* t, *float* x, *int* npdes, *int* ngrids, *float* *full_u, *float* *grid_u, *float* *dudx, *float* *c, *float* *q, *float* *r, *int* *ires) (Input)

A user-supplied function to evaluate the differential equation, as expressed in Equation 2. Each application requires a function specifically designed for the task, and this function is normally written by the user of the integrator.

Evaluate the terms of the system of ***Equation 2***. A default value of $m = 0$ is assumed, but this can be changed to one of the choices, $m = 1, 2$. Use the optional arguments IMSL_CART_COORDINATES, IMSL_CYL_COORDINATES, IMSL_SPH_COORDIANTES for the respective values $m = 0, 1, 2$. Return the values in the arrays as indicated:

$$u^j = \texttt{grid\_u}[j]$$
$$U = \texttt{full\_u}$$
$$\frac{\partial u^j}{\partial x} = u_x^j = \texttt{dudx}[j]$$
$$\mathrm{c}[l][k] = C_{j,k}\left(x,t,u,u_x\right)$$
$$\mathrm{r}[j] = r_j\left(x,t,u,u_x\right)$$
$$\mathrm{q}[j] = q_j\left(x,t,u,u_x\right)$$
$$j,k = 0,\ldots NPDE - 1$$

If any of the functions cannot be evaluated, set `ires=3`. Otherwise, do not change the value of `ires`.

*void* `boundary_conditions` (*float* `t`, *float* `*beta`, *float* `*gamma`, *float* `*full_u`, *float* `*grid_u`, *float* `*dudx`, *int* `npdes`, *int* `grids`, *int* `left`, *int* `*ires`) (Input)

User-supplied function to supply the boundary conditions, as expressed in *Equation 2*.

$$u^j = \texttt{grid\_u}[j]$$
$$U = \texttt{full\_u}$$
$$\frac{\partial u^j}{\partial x} = u_x^j = \texttt{dudx}[j]$$
$$beta[j] = \beta_j\left(x,t,u,u_x\right)$$
$$\texttt{gamma}[j] = \gamma_j\left(x,t,u,u_x\right)$$
$$j = 0, \ldots NPDE - 1$$

The value $x \in \{x_L, x_R\}$, and the flag `left=1` for $x = x_L$. The flag has the value `left=0` for $x = x_R$.

If any of the functions cannot be evaluated, set `ires=3`. Otherwise, do not change the value of `ires`.

## Synopsis with Optional Arguments for imsl_f_pde_1d_mg_mgr

*#include* `<imsl.h>`

*void* `imsl_f_pde_1d_mg_mgr` (*int* `task`, *void* `**state`,

`IMSL_CART_COORDINATES`, *or*

`IMSL_CYL_COORDINATES`, *or*

`IMSL_SPH_COORDINATES`,

`IMSL_TIME_SMOOTHING`, *float* `tau`,

`IMSL_SPATIAL_SMOOTHING`, *float* `kappa`, IMSL_MONITOR_REGULARIZING, *float* alpha,

IMSL_MAX_BDF_ORDER, *int* max_bdf_order,

IMSL_USER_FACTOR_SOLVE, *int* fac(), *void* sol(),

IMSL_USER_FACTOR_SOLVE_W_DATA, *int* fac(), *void* sol(), *void* data,

IMSL_INITIAL_CONDITIONS, *void* initial_conditions()

IMSL_INITIAL_CONDITIONS_W_DATA, *void* initial_conditions*()*, *void* data,

0)

## Optional Arguments

IMSL_CART_COORDINATES, *or*

IMSL_CYL_COORDINATES, *or*

IMSL_SPH_COORDINATES

IMSL_CART_COORDINATES specifies cartesian coordinates, where $m = 0$ in Equation 2.
IMSL_CYL_COORDINATES specifies cylindrical or polar coordinates, where $m = 1$ in *Equation 2*.
IMSL_SPH_COORDINATES specifies spherical coordinates, where $m = 2$ in *Equation 2*.
Default: IMSL_CART_COORDINATES

IMSL_TIME_SMOOTHING, *float* tau, (Input)

Resets the value of the parameter $\tau \geq 0$, described above.
Default: $\tau = 0$.

IMSL_SPATIAL_SMOOTHING, *float* kappa, (Input)

Resets the value of the parameter $\kappa \geq 0$, described above.
Default: $\kappa = 2$.

IMSL_MONITOR_REGULARIZING, *float* alpha, (Input)

Resets the value of the parameter $\alpha \geq 0$, described above.
Default: $\alpha = 0.01$.

IMSL_MAX_BDF_ORDER, *int* max_bdf_order, (Input)

Resets the maximum order for the *bdf* formulas used in imsl_f_dea_petzold_gear. The new value can be any integer between 1 and 5. Some problems benefit by making this change. The default value of max_bdf_order was chosen because imsl_f_dea_petzold_gear may cycle on its selection of order and step-size with max_bdf_order higher than value 2.
Default: max_bdf_order=2.

IMSL_USER_FACTOR_SOLVE, *int* fac(*int* neq, *int* iband, *float* *a), *void* sol(*int* neq, *int* iband, *float* *g, *float* *y) (Input)

User-supplied functions to factor *A*, and solve the system *A*Δ*y* = Δ*g*. Use of this optional argument allows for handling the factorization and solution steps in a problem-specific manner. If successful `fac` should return 0, if unsuccessful, `fac` should return a non-zero value. See Example 5 - A Flame Propagation Model for sample usage of this optional argument.

`IMSL_USER_FACTOR_SOLVE_W_DATA`, *int* `fac`(*int* `neq`, *int* `iband`, *float* `*a`, *void* `*data`), *void* `sol`(*int* `neq`, *int* `iband`, *float* `*g`, *float* `*y`, *void* `*data`), *void* `*data` (Input)
User-supplied functions to factor *A*, and solve the system $A\Delta y = \Delta g$. The argument `data` is a pointer to the data that is passed to the user-supplied function.

`IMSL_INITIAL_CONDITIONS`, *void* `initial_conditions`(*int* `npdes`, *int* `ngrids`, *float* `*u`) (Input)
User-supplied function to supply the initial values for the system at the starting independent variable value `t`. This function can also provide a non-uniform grid at the initial value. Here `npdes` is the number of differential equations, `ngrids` is the number of grid points, and `u` is an array of size `npdes+1` by `ngrids`, containing the approximate solution value $U_i\left(x_j\left(tend\right), tend\right)$ in location `u[`$i$×`ngrids+`$j$`]`. The grid values are equally spaced on input, but can be updated as desired, provided the values are increasing. Update the grid values in array locations `u[(npdes` × `ngrids) +`$j$`]`, where $0 \le j \le$ `ngrids`.

`IMSL_INITIAL_CONDITIONS_W_DATA`, *void* `initial_conditions`(*int* `npdes`, *int* `ngrids`, *float* `*u`, *float* `*grid`, *void* `*data`), *void* `*data` (Input)
User-supplied function to supply the initial values for the system at the starting independent variable value `t`. This function can also provide a non-uniform grid at the initial value. The argument `data` is a pointer to the data that is passed to the user-supplied function.

## Synopsis with Optional Arguments for imsl_f_pde_1d_mg

*#include* `<imsl.h>`

*void* `imsl_f_pde_1d_mg` (*int* `npdes` ,*int* `ngrids` ,*float* `*t` ,*float* `tend`, *float* `u[]` ,*float* `xl` ,*float* `xr` ,*void* `*state` ,*void* `pde_systems()` ,*void* `boundary_conditions()`,

`IMSL_RELATIVE_TOLERANCE` ,*float* `rtol`,

`IMSL_ABSOLUTE_TOLERANCE` ,*float* `atol`,

`IMSL_PDE_SYS_W_DATA` ,*void* `pde_systems()` ,*void* `*`*data*,

`IMSL_BOUNDARY_COND_W_DATA` ,*void* `boundary_conditions()` ,*void* `*`*data*,

0)

# Optional Arguments

`IMSL_RELATIVE_TOLERANCE`, *float* `rtol`, (Input)
This option resets the value of the relative accuracy parameter used in `imsl_f_dea_petzold_gear`.
Default: `rtol=1.0E-2` for single precision, `rtol=1.0E-4` for double precision.

`IMSL_ABSOLUTE_TOLERANCE`, *float* `atol`, (Input)
This option resets the value of the absolute accuracy parameter used in `imsl_f_dea_petzold_gear`.
Default: `atol=1E-2` for single precision, `atol=1E-4` for double precision.

`IMSL_PDE_SYS_W_DATA`, *void* `pde_systems`(*float* `t`, *float* `x`, *int* `npdes`, *int* `ngrids`, *float* `*full_u`, *float* `*grid_u`, *float* `*dudx`, *float* `*c`, *float* `*q`, *float* `*r`, *int* `*ires`, *void* `*data`), *void* `*data` (Input)
User-supplied function to evaluate the differential equation, as expressed in Equation 2. The argument `data` is a pointer to the data that is passed to the user-supplied function.

`IMSL_BOUNDARY_COND_W_DATA`, *void* `boundary_conditions` (*float* `t`, *float* `*beta`, *float* `*gamma`, *float* `*full_u`, *float* `*grid_u`, *float* `*dudx`, *int* `npdes`, *int* `ngrids`, *int* `left`, *int* `*ires`, *void* `*data`), *void* `*data` (Input)
User-supplied function to supply the boundary conditions, as expressed in *Equation 2*. The argument `data` is a pointer to the data that is passed to the user-supplied function.

# Examples

## Remarks on the Examples

Due to its importance and the complexity of its interface, function `imsl_f_pde_1d_mg` is presented with several examples. Many of the program features are exercised. The problems complete without any change to the optional arguments, except where these changes are required to describe or to solve the problem.

In many applications the solution to a PDE is used as an auxiliary variable, perhaps as part of a larger design or simulation process. The truncation error of the approximate solution is commensurate with piece-wise linear interpolation on the grid of values, at each output point. To show that the solution is reasonable, a graphical display is revealing and helpful. We have not provided graphical output as part of our documentation, but users may already have the Rogue Wave, Inc. product, PV−WAVE, which is not included with IMSL C Numerical Library. Examples 1 through 8 write results in files `pde_ex0#.out` that can be visualized with PV−WAVE. We provide a script of commands, `pde_1d_mg_plot.pro`, for viewing the solutions. This is listed below. The grid of values and each consecutive solution component is displayed in separate plotting windows. The script and data files written by examples 1-8 on a SUN-SPARC system are in the directory for IMSL C Numerical Library examples. When executing PV_WAVE, use the command line

```
pde_1d_mg_plot,filename='pde_ex0#.out'
```

to view the output of a particular example. The symbol '#' will be one of the choices 1,2,...,8. However, it is not necessary to have PV_WAVE installed to execute the examples.

To view the code, see Code for PV-WAVE Plotting.

## Example 1 - Electrodynamics Model

This example is from Blom and Zegeling (1994). The system is

$$
\begin{aligned}
& u_t = \varepsilon p u_{xx} - g(u-v) \\
& v_t = p v_{xx} + g(u-v), \\
& \text{where } g(z) = \exp(\eta z/3) - \exp(-2\eta z/3) \\
& 0 \le x \le 1,\ 0 \le t \le 4 \\
& u_x = 0 \text{ and } v = 0 \text{ at } x = 0 \\
& u = 1 \text{ and } v_x = 0 \text{ at } x = 1 \\
& \varepsilon = 0.143, p = 0.1743, \eta = 17.19
\end{aligned}
$$

We make the connection between the model problem statement and the example:

$$
\begin{aligned}
& C = I_2 \\
& m = 0, R_1 = \varepsilon p u_x, R_2 = p v_x \\
& Q_1 = g(u-v), Q_2 = -Q_1 \\
& u = 1 \text{ and } v = 0 \text{ at } t = 1
\end{aligned}
$$

The boundary conditions are

$$
\begin{aligned}
& \beta_1 = 1, \beta_2 = 0, \gamma_1 = 0, \gamma_2 = v, \text{ at } x = x_L = 0 \\
& \beta_1 = 0, \beta_2 = 1, \gamma_1 = u-1, \gamma_2 = 0, \text{ at } x = x_R = 1
\end{aligned}
$$

This is a non-linear problem with sharply changing conditions near $t = 0$. The default settings of integration parameters allow the problem to be solved. The use of `imsl_f_pde_1d_mg` requires two subroutines provided by the user to describe the differential equations, and boundary conditions.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>

/* prototypes */
static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
                double full_u[], double grid_u[], double dudx[], double *c,
                double q[], double r[], int *ires);
static void boundary_conditions (double t, double beta[], double gamma[],
                double full_u[], double grid_u[], double dudx[], int npdes,
                int ngrids, int left, int *ires);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 2
```

```
#define NFRAMES 5
#define N 51
#define U(I_,J_)     u[I_ * ngrids + J_]
int main ()
{
  char *state = NULL;
  int i, j;
  double u[(NPDE + 1) * N];
  double t0 = 0.0, tout;
  double delta_t = 10.0, tend = 4.0;
  int npdes = NPDE, ngrids = N;
  double xl = 0.0, xr = 1.0;
  FILE *file1;

  file1 = fopen ("pde_ex01.out", "w");
  imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
  fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, NFRAMES);
  fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

  /* initialize u */
  initial_conditions (npdes, ngrids, u);

  imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state, 0);
  tout = 1e-3;
  do
    {
      imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl, xr, state,
                     pde_systems, boundary_conditions, 0);


      fprintf (file1, "%f\n", tout);
      for (i = 0; i < npdes + 1; i++)
       {
         for (j = 0; j < ngrids; j++)
          {
            fprintf (file1, "%16.10f   ", U (i, j));
            if (((j + 1) % 4) == 0)
            fprintf (file1, "\n");
          }
         fprintf (file1, "\n");
       }


      t0 = tout;
      tout = tout * delta_t;
      tout = MIN (tout, tend);
    }
  while (t0 < tend);

  imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);
```

```
#undef MIN
#undef NPDE
#undef NFRAMES
#undef N
#undef U
}


static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)         u[I_ * ngrids + J_]


  int i;
  for (i = 0; i < ngrids; i++)
    {
      U (0, i) = 1.0;
      U (1, i) = 0.0;
    }

#undef U
}
static void
pde_systems (double t, double x, int npdes, int ngrids,
        double full_u[], double grid_u[], double dudx[], double *c,
        double q[], double r[], int *ires)
{
#define C(I_,J_)       c[I_ * npdes + J_]
  double z;
  static double eps = 0.143;
  static double eta = 17.19;
  static double pp = 0.1743;

  C (0, 0) = 1.0;
  C (0, 1) = 0.0;
  C (1, 0) = 0.0;
  C (1, 1) = 1.0;
  r[0] = pp * dudx[0] * eps;
  r[1] = pp * dudx[1];
  z = eta * (grid_u[0] - grid_u[1]) / 3.0;
  q[0] = exp (z) - exp (-2.0 * z);
  q[1] = -q[0];
  return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
        double full_u[], double grid_u[], double dudx[],
        int ngrids, int npdes, int left, int *ires)
```

```
{
  if (left)
    {
      beta[0] = 1.0;
      beta[1] = 0.0;
      gamma[0] = 0.0;
      gamma[1] = grid_u[1];
    }
  else
    {
      beta[0] = 0.0;
      beta[1] = 1.0;
      gamma[0] = grid_u[0] - 1.0;
      gamma[1] = 0.0;
    }
  return;
}
```

## Example 2 - Inviscid Flow on a Plate

This example is a first order system from Pennington and Berzins, (1994). The equations are

$$\begin{aligned}
&u_t = -v_x \\
&uu_t = -vu_x + w_{xx} \\
&w = u_x, \text{ implying that } uu_t = -vu_x + u_{xx} \\
&u(0,t) = v(0,t) = 0, u(\infty, t) = u(x_R, t) = 1, t \ge 0 \\
&u(x,0) = 1, v(x,0) = 0, x \ge 0
\end{aligned}$$

Following elimination of *w*, there remain $NPDE = 2$ differential equations. The variable $t$ is not time, but a second space variable. The integration goes from $t = 0$ to $t = 5$. It is necessary to truncate the variable $x$ at a finite value, say $x_{\max} = x_R = 25$. In terms of the integrator, the system is defined by letting $m = 0$ and

$$C = \{C_{jk}\} = \begin{bmatrix} 1 & 0 \\ u & 0 \end{bmatrix}, R = \begin{bmatrix} -v \\ u_x \end{bmatrix}, Q = \begin{bmatrix} 0 \\ vu_x \end{bmatrix}$$

The boundary conditions are satisfied by

$$\beta = 0, \gamma = \begin{bmatrix} u - \exp(-20t) \\ v \end{bmatrix}, \text{ at } x = x_L$$

$$\beta = 0, \gamma = \begin{bmatrix} u - 1 \\ v_x \end{bmatrix}, \text{ at } x = x_R$$

We use $N = 10 + 51 = 61$ grid points and output the solution at steps of $\Delta t = 0.1$.

This is a non-linear boundary layer problem with sharply changing conditions near $t = 0$. The problem statement was modified so that boundary conditions are continuous near $t = 0$. Without this change the underlying integration software, `imsl_f_dea_petzold_gear`, cannot solve the problem. The continuous blending function $u - \exp(-20t)$ is arbitrary and artfully chosen. This is a mathematical change to the problem, required because of the stated discontinuity at $t = 0$ . Reverse communication is used for the problem data. No additional user-written subroutines are required when using reverse communication. We also have chosen 10 of the initial grid points to be concentrated near $X_L = 0$ , anticipating rapid change in the solution near that point. Optional changes are made to use a pure absolute error tolerance and non-zero time-smoothing.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>

/* prototypes */
static void initial_conditions (int npdes, int ngrids, double u[]);

static void pde_systems (double t, double x, int npdes, int ngrids,
                        double full_u[], double grid_u[],
                        double dudx[], double *c, double q[],
                        double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                                double gamma[], double full_u[],
                                double grid_u[], double dudx[],
                                int npdes, int ngrids, int left,
                                int *ires);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 2
#define N1 10
#define N2 51
#define N (N1+N2)
#define U(I_,J_)           u[I_ * ngrids + J_]
FILE *file1;
int main ()
{
   char *state;
   int i, j;
   int nframes;
   double u[(NPDE + 1) * N];
   double t0 = 0.0, tout;
   double delta_t = 1e-1, tend = 5.0;
   int npdes = NPDE, ngrids = N;
   double xl = 0.0, xr = 25.0;
   double tau = 1.0e-3;
   double atol = 1e-2;
   double rtol = 0.0;


   file1 = fopen ("pde_ex02.out", "w");
   imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
   nframes = (int) ((tend + delta_t) / delta_t);
   fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, nframes);
   fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);
```

```
    imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
        IMSL_TIME_SMOOTHING, tau,
        IMSL_INITIAL_CONDITIONS, initial_conditions, 0);

    t0 = 0.0;
    tout = delta_t;
    do
    {
        imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
            xr, state, pde_systems, boundary_conditions,
            IMSL_RELATIVE_TOLERANCE, rtol,
            IMSL_ABSOLUTE_TOLERANCE, atol, 0);

        t0 = tout;
        fprintf (file1, "%f\n", tout);
        for (i = 0; i < npdes + 1; i++)
        {
            for (j = 0; j < ngrids; j++)
            {
                fprintf (file1, "%16.10f   ", U (i, j));
                if (((j + 1) % 4) == 0)
                    fprintf (file1, "\n");
            }
            fprintf (file1, "\n");
        }
        tout = tout + delta_t;
        tout = MIN (tout, tend);
    }
    while (t0 < tend);

    imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);
    fclose (file1);

#undef MIN
#undef NPDE
#undef NFRAMES
#undef N
#undef U
}

static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)           u[I_* ngrids + J_]


    int i, j, i_, n1 = 10, n2 = 51, n;
    double dx1, dx2;
    double xl = 0.0, xr = 25.0;
```

```
   n = n1 + n2;
   for (i = 0; i < ngrids; i++)
   {
       U (0, i) = 1.0;
       U (1, i) = 0.0;
       U (2, i) = 0.0;
   }

   dx1 = xr / n2;
   dx2 = dx1 / n1;

   /* grid */
   for (i = 1; i <= n1; i++)
   {
       i_ = i - 1;
       U (2, i_) = (i - 1) * dx2;
   }
   for (i = n1 + 1; i <= n; i++)
   {
       i_ = i - 1;
       U (2, i_) = (i - n1) * dx1;
   }
   for (i = 0; i < npdes + 1; i++)
   {
       for (j = 0; j < ngrids; j++)
       {
           fprintf (file1, "%16.10f   ", U (i, j));
           if (((j + 1) % 4) == 0)
               fprintf (file1, "\n");
       }
       fprintf (file1, "\n");
   }

#undef U
}

static void
pde_systems (double t, double x, int npdes, int ngrids,
            double full_u[], double grid_u[], double dudx[],
            double *c, double q[], double r[], int *ires)
{
#define C(I_,J_)       c[I_ * npdes + J_]
   double z;

   C (0, 0) = 1.0;
   C (1, 0) = 0.0;
   C (0, 1) = grid_u[0];
   C (1, 1) = 0.0;

   r[0] = -grid_u[1];
   r[1] = dudx[0];
```

```
   q[0] = 0.0;
   q[1] = grid_u[1] * dudx[0];
   return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
                     double full_u[], double grid_u[], double dudx[],
                     int npdes, int ngrids, int left, int *ires)
{
   double dif;

   beta[0] = 0.0;
   beta[1] = 0.0;
   if (left)
   {
      dif = exp (-20.0 * t);
      gamma[0] = grid_u[0] - dif;
      gamma[1] = grid_u[1];
   }
   else
   {
      gamma[0] = grid_u[0] - 1.0;
      gamma[1] = dudx[1];
   }
   return;
}
```

## Example 3 - Population Dynamics

This example is from Pennington and Berzins (1994). The system is

$$u_t = -u_x - I(t)u, x_L = 0 \le x \le a = x_R, t \ge 0$$

$$I(t) = \int_0^a u(x,t)\,dx$$

$$u(x,0) = \frac{\exp(-x)}{2-\exp(-a)}$$

$$u(0,t) = g\left(\int_0^a b(x,I(t))u(x,t)\,dx, t\right), \text{ where}$$

$$b(x,y) = \frac{xy\exp(-x)}{(y+1)^2}, \text{ and}$$

$$g(z,t) =$$

$$\frac{4z(2-2\exp(-a)+\exp(-t))^2}{(1-\exp(-a))(1-(1+2a)\exp(-2a)(1-\exp)(-a)+\exp(-t))}$$

This is a notable problem because it involves the unknown

$$u(x,t) = \frac{\exp(-x)}{1-\exp(-a)+\exp(-t)}$$

across the entire domain. The software can solve the problem by introducing two dependent algebraic equations:

$$v_1(t) = \int_0^a u(x,t)\,dx,$$
$$v_2(t) = \int_0^a x\exp(-x)\,u(x,t)\,dx$$

This leads to the modified system

$$u_t = -u_x - v_1 u,\ \ 0 \le x \le a,\ t \ge 0$$
$$u(0,t) = \frac{g(1,t)v_1 v_2}{(v_1+1)^2}$$

In the interface to the evaluation of the differential equation and boundary conditions, it is necessary to evaluate the integrals, which are computed with the values of $u(x,t)$ on the grid. The integrals are approximated using the trapezoid rule, commensurate with the truncation error in the integrator.

This is a non-linear integro-differential problem involving non-local conditions for the differential equation and boundary conditions. Access to evaluation of these conditions is provided using the optional arguments `IMSL_PDE_SYS_W_DATA` and `IMSL_BOUNDARY_COND_W_DATA`. Optional changes are made to use an absolute error tolerance and non-zero time-smoothing. The time-smoothing value $\tau = 1$ prevents grid lines from crossing.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>


/* prototypes */

static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
                         double full_u[], double grid_u[],
                         double dudx[], double *c, double q[],
                         double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                                 double gamma[], double full_u[],
                                 double grid_u[], double dudx[],
                                 int npdes, int ngrids, int left,
                                 int *ires);
static double fcn_g (double z, double t);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 1
#define N 101
#define U(I_,J_)        u[I_ * ngrids + J_]
```

```
FILE *file1;
int main ()
{
    int i, j, nframes;
    double u[(NPDE + 1) * N], mid[N - 1];
    int npdes = NPDE, ngrids = N;
    double t0 = 0.0, tout;
    double delta_t = 1e-1, tend = 5.0, a = 5.0;
    char *state;
    double xl = 0.0, xr = 5.0;
    double *ptr_u;
    double tau = 1.0;
    double atol = 1e-2;
    double rtol = 0.0;

    file1 = fopen ("pde_ex03.out", "w");
    imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
    nframes = (int) (tend + delta_t) / delta_t;
    fprintf (file1, "   %d\t%d\t%d", npdes, ngrids, nframes);
    fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

    ptr_u = u;

    imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
        IMSL_TIME_SMOOTHING, tau,
        IMSL_INITIAL_CONDITIONS, initial_conditions, 0);


    tout = delta_t;
    fprintf (file1, "%f\n", t0);
    do
    {
        imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
            xr, state, pde_systems, boundary_conditions,
            IMSL_RELATIVE_TOLERANCE, rtol,
            IMSL_ABSOLUTE_TOLERANCE, atol, 0);

        t0 = tout;
        if (t0 <= tend)
        {
            fprintf (file1, "%f\n", tout);
            for (i = 0; i < npdes + 1; i++)
            {
                for (j = 0; j < ngrids; j++)
                {
                    fprintf (file1, "%16.10f    ", U (i, j));
                    if (((j + 1) % 4) == 0)
                        fprintf (file1, "\n");
                }
                fprintf (file1, "\n");
            }
```

```
        }
        tout = MIN (tout + delta_t, tend);
    }
    while (t0 < tend);
    imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);

    fclose (file1);

#undef MIN
#undef NPDE
#undef N
#undef XL
#undef XR
#undef U
}


static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)        u[I_ * ngrids + J_]
#define XL 0.0
#define XR 5.0


    int i, j;
    double dx, xi;

    dx = (XR - XL) / (ngrids - 1);
    for (i = 0; i < ngrids; i++)
    {
        U (0, i) = exp (-U (1, i)) / (2.0 - exp (-XR));
    }

    for (i = 0; i < npdes + 1; i++)
    {
        for (j = 0; j < ngrids; j++)
        {
            fprintf (file1, "%16.10f    ", U (i, j));
            if (((j + 1) % 4) == 0)
                fprintf (file1, "\n");
        }
        fprintf (file1, "\n");
    }

#undef U
#undef XL
#undef XR
}

static void
```

```
pde_systems (double t, double x, int npdes, int ngrids,
             double full_u[], double grid_u[], double dudx[],
             double *c, double q[], double r[], int *ires)
{
#define U(I_,J_)        full_u[I_ * ngrids + J_]

    double v1;
    double sum = 0.0;
    int i;

    c[0] = 1.0;
    r[0] = -1 * grid_u[0];

    for (i = 0; i < ngrids - 1; i++)
    {
        sum += (U (0, i) + U (0, i + 1)) * (U (1, i + 1) - U (1, i));
    }

    v1 = 0.5 * sum;
    q[0] = v1 * grid_u[0];


    return;
#undef U
}

static void
boundary_conditions (double t, double beta[], double gamma[],
                     double full_u[], double grid_u[], double dudx[],
                     int npdes, int ngrids, int left, int *ires)
{
#define U(I_,J_)        full_u[I_ * ngrids + J_]
    double v1, v2, mid;
    double sum = 0.0;
    double sum1 = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0;
    int i;


    for (i = 0; i < ngrids - 1; i++)
    {
        sum += (U (0, i) + U (0, i + 1)) * (U (1, i + 1) - U (1, i));
        mid = 0.5 * (U (1, i) + U (1, i + 1));
        sum1 += mid * exp (-mid) *
            ((U (0, i) + U (0, i + 1)) * (U (1, i + 1) - U (1, i)));
    }

    if (left)
    {
        v1 = 0.5 * sum;
        v2 = 0.5 * sum1;
        beta[0] = 0.0;
```

```
        gamma[0] =  fcn_g (1.0, t) * v1 * v2 /
                   ((v1 + 1.0) * (v1 + 1.0)) - grid_u[0];
    }
    else
    {
        beta[0] = 0.0;
        gamma[0] = dudx[0];
    }
    return;
#undef  U
}

static double
fcn_g (double z, double t)
{
    double g, a = 5.0;

    g = 4.0 * z * (2.0 - 2.0 * exp (-a) + exp (-t)) *
        (2.0 - 2.0 * exp (-a) + exp (-t));
    g = g / ((1.0 - exp (-a)) * (1.0 - (1.0 + 2.0 * a) *
        exp (-2.0 * a)) * (1.0 - exp (-a) + exp (-t)));
    return g;
}
```

## Example 4 - A Model in Cylindrical Coordinates

This example is from Blom and Zegeling (1994). The system models a reactor-diffusion problem:

$$
\begin{aligned}
&T_z = r^{-1}\frac{\partial\left(\beta r T_r\right)}{\partial r} + \gamma\exp\left(\frac{T}{1+\varepsilon T}\right)\\
&T_r\left(0,z\right) = 0,\ T\left(1,z\right) = 0,\ z > 0\\
&T\left(r,0\right) = 0,\ 0 \le r < 1\\
&\beta = 10^{-4}, \gamma = 1, \varepsilon = 0.1
\end{aligned}
$$

The axial direction $z$ is treated as a time coordinate. The radius $r$ is treated as the single space variable.

This is a non-linear problem in cylindrical coordinates. Our example illustrates assigning $m = 1$ in Equation 2. We provide the optional argument IMSL_CYL_COORDINATES that resets this value from its default, $m = 0$.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>


/* prototypes */
static void initial_conditions (int npdes, int ngrids, double t[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
           double u[], double grid_u[], double dudx[], double *c,
```

```
                  double q[], double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                  double gamma[], double u[], double grid_u[], double dudx[],
                  int npdes, int ngrids, int left, int *ires);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 1
#define N 41
#define T(I_,J_)       t[I_ * ngrids + J_]
int main ()
{
  int i, j, ido;
  int nframes;
  double t[(NPDE + 1) * N];
  double z0 = 0.0, zout;
  double dx1, dx2, diff;
  double delta_z = 1e-1, zend = 1.0, zmax = 1.0;
  double beta = 1e-4, gamma = 1.0, eps = 1e-1;
  char *state;
  int npdes = NPDE, ngrids = N;
  double xl = 0.0, xr = 1.0;
  FILE *file1;
  int m = 1;


  file1 = fopen ("pde_ex04.out", "w");
  imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
  nframes = (int) ((zend + delta_z) / delta_z) - 1;
  fprintf (file1, "  %d\t%d\t%d", npdes, N, nframes);
  fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, z0, zend);

  imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state, IMSL_CYL_COORDINATES,
0);
  initial_conditions (npdes, ngrids, t);

  zout = delta_z;
  do
    {
      imsl_d_pde_1d_mg (npdes, ngrids, &z0, zout, t, xl,
       xr, state, pde_systems, boundary_conditions, 0);

      z0 = zout;
      if (z0 <= zend)
       {
         fprintf (file1, "%f\n", zout);
         for (i = 0; i < npdes + 1; i++)
           {
             for (j = 0; j < ngrids; j++)
               {
                 fprintf (file1, "%16.10f   ", T (i, j));
                 if (((j + 1) % 4) == 0)
```

```
                    fprintf (file1, "\n");
                }
              fprintf (file1, "\n");
            }

        }
      zout = MIN ((zout + delta_z), zend);
    }
  while (z0 < zend);

  imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);
  fclose (file1);
#undef MIN
#undef NPDE
#undef N
#undef T
}

static void
initial_conditions (int npdes, int ngrids, double t[])
{
#define T(I_,J_)       t[I_ * ngrids + J_]
  int i;

  for (i = 0; i < ngrids; i++)
    {
       T (0, i) = 0.0;
    }
#undef T
}

static void
pde_systems (double t, double x, int npdes, int ngrids, double u[],
       double grid_u[], double dudx[], double *c,
       double q[], double r[], int *ires)
{
#define C(I_,J_)       c[I_ * npdes + J_]
  static double beta = 01e-4, gamma = 1.0, eps = 1e-1;

  C (0, 0) = 1.0;

  r[0] = beta * dudx[0];
  q[0] = -1.0 * gamma * exp (grid_u[0] / (1.0 + eps * grid_u[0]));
  return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
         double u[], double grid_u[], double dudx[],
         int npdes, int ngrids, int left, int *ires)
```

```
{


  if (left)
    {
      beta[0] = 1.0;
      gamma[0] = 0.0;
    }
  else
    {
      beta[0] = 0.0;
      gamma[0] = grid_u[0];
    }
  return;
}
```

## Example 5 - A Flame Propagation Model

This example is presented more fully in Verwer, ***et al.***, (1989). The system is a normalized problem relating mass density $u(x, t)$ and temperature $v(x, t)$:

$$
\begin{aligned}
&u_t = u_{xx} - uf(v) \\
&v_t = v_{xx} + uf(v), \\
&\text{where } f(z) = \gamma \exp(-\beta / z), \beta = 4, \gamma = 3.52 \times 10^6 \\
&0 \le x \le 1, 0 \le t \le 0.006 \\
&u(x,0) = 1, v(x,0) = 0.2 \\
&u_x = v_x = 0, x = 0 \\
&u_x = 0, v = b(t), x = 1, \text{ where} \\
&b(t) = 1.2, \text{ for } t \ge 2 \times 10^{-4}, \text{ and} \\
&\qquad = 0.2 + 5 \times 10^3 t, \text{ for } 0 \le t \le 2 \times 10^{-4}
\end{aligned}
$$

This is a non-linear problem. The example shows the model steps for replacing the banded solver in the software with one of the user’s choice.  Following the computation of the matrix factorization in imsl_lin_sol_gen_band (see Chapter 1, ***Linear Systems***), we declare the system to be singular when the reciprocal of the condition number is smaller than the working precision. This choice is not suitable for all problems. Attention must be given to detecting a singularity when this option is used.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <imsl.h>

/* prototypes */
static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
```

```
                        double u[], double grid_u[], double dudx[],
                        double *c, double q[], double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                              double gamma[], double u[],
                              double grid_u[], double dudx[],
                              int npdes, int ngrids, int left,
                              int *ires);
static int fac (int neq, int iband, double *a);
static void sol (int neq, int iband, double *g, double *y);
static double fcn (double z);


int *ipvt = NULL;
double *factor = NULL;

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 2
#define N 40
#define NEQ ((NPDE+1)*N)
#define U(I_,J_)     u[I_ * ngrids + J_]
int main ()
{
   int i, j, nframes;
   double u[(NPDE + 1) * N];
   double t0 = 0.0, tout;
   double delta_t = 1e-4, tend = 6e-3;
   char *state;
   int npdes = NPDE, ngrids = N;
   double xl = 0.0, xr = 1.0;
   FILE *file1;
   double work[NEQ], rcond;
   double xmax = 1.0, beta = 4.0, gamma = 3.52e6;
   int max_bdf_order = 5;

   file1 = fopen ("pde_ex05.out", "w");
   imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
   nframes = (int) ((tend + delta_t) / delta_t) - 1;
   fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, nframes);
   fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

   initial_conditions (npdes, ngrids, u);
   imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
       IMSL_MAX_BDF_ORDER, max_bdf_order,
       IMSL_USER_FACTOR_SOLVE, fac, sol, 0);
   tout = delta_t;
   do
   {
       imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
           xr, state, pde_systems, boundary_conditions, 0);
       t0 = tout;
       if (t0 <= tend)
```

```
        {
            fprintf (file1, "%f\n", tout);
            for (i = 0; i < npdes + 1; i++)
            {
                for (j = 0; j < ngrids; j++)
                {
                    fprintf (file1, "%16.10f   ", U (i, j));
                    if (((j + 1) % 4) == 0)
                        fprintf (file1, "\n");
                }
            }

        }
        tout = MIN ((tout + delta_t), tend);

    }
    while (t0 < tend);

    imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);

    fclose (file1);

    if (factor != NULL)
    {
        imsl_free (factor);
    }
    if (ipvt != NULL)
    {
        imsl_free (ipvt);
    }
}
#undef MIN
#undef U

static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)     u[I_ * ngrids + J_]


    int i;

    for (i = 0; i < ngrids; i++)
    {
        U (0, i) = 1.0;
        U (1, i) = 2e-1;
    }

#undef U
}
static void
```

```
pde_systems (double t, double x, int npdes, int ngrids, double u[],
            double grid_u[], double dudx[], double *c,
            double q[], double r[], int *ires)
{
#define C(I_,J_)        c[I_ * npdes + J_]

   C (0, 0) = 1.0;
   C (0, 1) = 0.0;
   C (1, 0) = 0.0;
   C (1, 1) = 1.0;
   r[0] = dudx[0];
   r[1] = dudx[1];

   q[0] = grid_u[0] * fcn (grid_u[1]);
   q[1] = -1.0 * q[0];
   return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
                    double u[], double grid_u[], double dudx[],
                    int npdes, int ngrids, int left, int *ires)
{
   if (left)
   {
       beta[0] = 0.0;
       beta[1] = 0.0;
       gamma[0] = dudx[0];
       gamma[1] = dudx[1];
   }
   else
   {
       beta[0] = 1.0;
       gamma[0] = 0.0;
       beta[1] = 0.0;
       if (t >= 2e-4)
       {
           gamma[1] = 12e-1;
       }
       else
       {
           gamma[1] = 2e-1 + 5e3 * t;
       }
       gamma[1] -= grid_u[1];
   }
   return;
}

/* Factor the banded matrix. This is the same solver used
* internally but that is not required. A user can substitute
```

```
* one of their own.
* Note: Allowing lin_sol_gen_band to allocate ipvt and factor
* variables, then use in sol function.
*/
static int
fac (int neq, int iband, double *a)
{
   double rcond, panic_flag;
   int i, j;
   double b[NEQ];

   /* Free factor and pivot sequence if previously allocated. */
   if (factor != NULL)
   {
       imsl_free (factor);
       factor = NULL;
   }
   if (ipvt != NULL)
   {
       imsl_free (ipvt);
       ipvt = NULL;
   }

   imsl_d_lin_sol_gen_band (neq, a, iband, iband, b,
       IMSL_FACTOR, &ipvt, &factor,
       IMSL_FACTOR_ONLY, IMSL_CONDITION, &rcond, 0);

   panic_flag = 0;
   if (1.0 / rcond <= imsl_d_machine (4))
       panic_flag = 3;
   return panic_flag;
}

static void
sol (int neq, int iband, double *g, double *y)
{
   imsl_d_lin_sol_gen_band (neq, (double *) NULL, iband, iband, g,
       IMSL_SOLVE_ONLY,
       IMSL_FACTOR_USER, ipvt, factor,
       IMSL_RETURN_USER, y, 0);
   return;
}

static double
fcn (double z)
{
   double beta = 4.0, gamma = 3.52e6;

   return gamma * exp (-1.0 * beta / z);
}
```

## Example 6 - A 'Hot Spot' Model

This example is presented more fully in Verwer, ***et al.***, (1989). The system is a normalized problem relating the temperature $u(x, t)$, of a reactant in a chemical system. The formula for $h(z)$ is equivalent to their example.

$$
\begin{aligned}
&u_t = u_{xx} + h(u),\\
&\text{where } h(z) = \tfrac{R}{a\delta}(1 + a - z)\exp(-\delta(1/z - 1)),\\
&a = 1, \delta = 20, R = 5\\
&0 \le x \le 1, 0 \le t \le 0.29\\
&u(x,0) = 1\\
&u_x = 0, x = 0\\
&u = 1, x = 1
\end{aligned}
$$

This is a non-linear problem. The output shows a case where a rapidly changing front, or hot-spot, develops after a considerable way into the integration. This causes rapid change to the grid. An option sets the maximum order BDF formula from its default value of 2 to the theoretical stable maximum value of 5.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>

/* prototypes */

static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
                        double full_u[], double grid_u[],
                        double dudx[], double *c, double q[],
                        double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                                double gamma[], double full_u[],
                                double grid_u[], double dudx[],
                                int npdes, int ngrids, int left,
                                int *ires);
static double fcn_h (double z);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 1
#define N 80
#define U(I_,J_)     u[I_ * ngrids + J_]
int main ()
{
   int i, j, nframes;
   double u[(NPDE + 1) * N];
   double t0 = 0.0, tout;
   double delta_t = 1e-2, tend = 29e-2;
   double u0 = 1.0, u1 = 0.0, tdelta = 1e-1, tol = 29e-2;
   double a = 1.0, delta = 20.0, r = 5.0;
   char *state;
```

```
    int npdes = NPDE, ngrids = N;
    double xl = 0.0, xr = 1.0;
    FILE *file1;
    int max_bdf_order = 5;


    file1 = fopen ("pde_ex06.out", "w");
    imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
    nframes = (int) ((tend + delta_t) / delta_t) - 1;
    fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, nframes);
    fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

    initial_conditions (npdes, ngrids, u);

    imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
        IMSL_MAX_BDF_ORDER, max_bdf_order, 0);
    tout = delta_t;
    do
    {
        imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
            xr, state, pde_systems, boundary_conditions, 0);

        t0 = tout;
        if (t0 <= tend)
        {
            fprintf (file1, "%f\n", tout);
            for (i = 0; i < npdes + 1; i++)
            {
                for (j = 0; j < ngrids; j++)
                {
                    fprintf (file1, "%16.10f   ", U (i, j));
                    if (((j + 1) % 4) == 0) fprintf (file1, "\n");
                }
            }

        }
        tout = MIN ((tout + delta_t), tend);
    }
    while (t0 < tend);

    imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);
    fclose (file1);
#undef MIN
#undef NPDE
#undef N
#undef U
}

static void
initial_conditions (int npdes, int ngrids, double u[])
{
```

```
#define U(I_,J_)       u[I_ * ngrids + J_]
   int i;

   for (i = 0; i < ngrids; i++)
   {
       U (0, i) = 1.0;
   }
#undef U
}

static void
pde_systems (double t, double x, int npdes, int ngrids,
            double full_u[], double grid_u[], double dudx[],
            double *c, double q[], double r[], int *ires)
{
#define C(I_,J_)       c[I_ * npdes + J_]

   c[0] = 1.0;
   r[0] = dudx[0];
   q[0] = -fcn_h (grid_u[0]);

   return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
                    double full_u[], double grid_u[], double dudx[],
                    int npdes, int ngrids, int left, int *ires)
{
   if (left)
   {
       beta[0] = 0.0;
       gamma[0] = dudx[0];
   }
   else
   {
       beta[0] = 0.0;
       gamma[0] = grid_u[0] - 1.0;
   }
   return;
}
static double
fcn_h (double z)
{
   double a = 1.0, delta = 2e1, r = 5.0;

   return (r / (a * delta)) * (1.0 + a - z) *
               exp (-delta * (1.0 / z - 1.0));
}
```

## Example 7 - Traveling Waves

This example is presented more fully in Verwer, ***et al.***, (1989). The system is a normalized problem relating the interaction of two waves, $u(x,t)$ and $v(x,t)$ moving in opposite directions. The waves meet and reduce in amplitude, due to the non-linear terms in the equation. Then they separate and travel onward, with reduced amplitude.

$$
\begin{aligned}
&u_t = -u_x - 100uv,\\
&v_t = v_x - 100uv,\\
&-0.5 \le x \le 0.5, 0 \le t \le 0.5\\
&u(x,0) = 0.5(1+\cos(10\pi x)), x \in [-0.3, -0.1], \text{ and}\\
&\qquad\quad = 0, \text{ otherwise},\\
&v(x,0) = 0.5(1+\cos(10\pi x)) x \in [0.1, 0.3], \text{ and}\\
&\qquad\quad = 0, \text{ otherwise},\\
&u = v = 0 \text{ at both ends, } t \ge 0
\end{aligned}
$$

This is a non-linear system of first order equations.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>

/* prototypes */
static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
                        double full_u[], double grid_u[],
                        double dudx[], double *c, double q[],
                        double r[], int *ires);
static void boundary_conditions (double t, double beta[],
                                double gamma[], double full_u[],
                                double grid_u[], double dudx[],
                                int npdes, int ngrids, int left,
                                int *ires);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 2
#define N 50
#define XL (-0.5)
#define XR 0.5
#define U(I_,J_)     u[I_ * ngrids + J_]
FILE *file1;
int main ()
{
   int i, j, nframes;
   double u[(NPDE + 1) * N];
   double t0 = 0.0, tout;
   double delta_t = 5e-2, tend = 5e-1;
   char *state;
```

```
int npdes = NPDE, ngrids = N;
double xl = XL, xr = XR;
double tau = 1e-3;
double atol = 1e-3;
double rtol = 0.0;
int max_bdf_order = 3;

file1 = fopen ("pde_ex07.out", "w");
imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
nframes = (int) ((tend + delta_t) / delta_t);
fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, nframes);
fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
    IMSL_TIME_SMOOTHING, tau,
    IMSL_MAX_BDF_ORDER, max_bdf_order,
    IMSL_INITIAL_CONDITIONS, initial_conditions, 0);
fprintf (file1, "%f\n", t0);
tout = delta_t;
do
{
    imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
        xr, state, pde_systems, boundary_conditions,
        IMSL_RELATIVE_TOLERANCE, rtol,
        IMSL_ABSOLUTE_TOLERANCE, atol, 0);

    t0 = tout;
    if (t0 <= tend)
    {
        fprintf (file1, "%f\n", tout);
        for (i = 0; i < npdes + 1; i++)
        {
            for (j = 0; j < ngrids; j++)
            {
                fprintf (file1, "%16.10f   ", U (i, j));
                if (((j + 1) % 4) == 0)
                    fprintf (file1, "\n");
            }
            fprintf (file1, "\n");
        }


    }
    tout = MIN ((tout + delta_t), tend);
}
while (t0 < tend);

imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);


fclose (file1);
```

```
#undef MIN
#undef NPDE
#undef N
#undef XL
#undef XR
#undef U
}


static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)      u[I_ * ngrids + J_]
#define XL -0.5
#define XR 0.5


   int i, j;
   double _pi, pulse;
   double dx, xi;


   _pi = imsl_d_constant("pi",0);
   for (i = 0; i < ngrids; i++)
   {
       pulse = (0.5 * (1.0 + cos (10.0 * _pi * U (npdes, i))));
       U (0, i) = pulse;
       U (1, i) = pulse;
   }
   for (i = 0; i < ngrids; i++)
   {
       if ((U (npdes, i) < -3e-1) || (U (npdes, i) > -1e-1))
       {
           U (0, i) = 0.0;
       }
       if ((U (npdes, i) < 1e-1 || U (npdes, i) > 3e-1))
       {
           U (1, i) = 0.0;
       }
   }
   for (i = 0; i < npdes + 1; i++)
   {
       for (j = 0; j < ngrids; j++)
       {
           fprintf (file1, "%16.10f   ", U (i, j));
           if (((j + 1) % 4) == 0)
               fprintf (file1, "\n");
       }
       fprintf (file1, "\n");
   }
```

```
#undef XL
#undef XR
#undef U
}

static void
pde_systems (double t, double x, int npdes, int ngrids,
            double full_u[], double grid_u[], double dudx[],
            double *c, double q[], double r[], int *ires)
{
#define C(I_,J_)       c[I_ * npdes + J_]

   C (0, 0) = 1.0;
   C (0, 1) = 0.0;
   C (1, 0) = 0.0;
   C (1, 1) = 1.0;

   r[0] = -1.0 * grid_u[0];
   r[1] = grid_u[1];
   q[0] = 100.0 * grid_u[0] * grid_u[1];
   q[1] = q[0];
   return;
#undef C
}

static void
boundary_conditions (double t, double beta[], double gamma[],
                    double full_u[], double grid_u[], double dudx[],
                    int npdes, int ngrids, int left, int *ires)
{

   beta[0] = 0.0;
   beta[1] = 0.0;
   gamma[0] = grid_u[0];
   gamma[1] = grid_u[1];

   return;
}
```

### Example 8 - Black-Scholes

The value of a European “call option,” $c(s,t)$, with exercise price $e$ and expiration date $T$, satisfies the “asset-or-nothing payoff” $c(s,T) = s,\, s \ge e; = 0,\, s < e$. Prior to expiration $c(s,t)$ is estimated by the Black-Scholes differential equation $c_t + \frac{\sigma^2}{2}s^2 c_{ss} + rsc_s - rc \equiv c_t + \frac{\sigma^2}{2}\left(s^2 c_s\right)_s + \left(r - \sigma^2\right)sc_s - rc = 0$. The parameters in the model are the risk-free interest rate, $r$, and the stock volatility, $\sigma$. The boundary conditions are $c(0,t) = 0$ and $c_s(s,t) \approx 1,\, s \to \infty$. This development is described in Wilmott, *et al.* (1995), pages 41-57. There are explicit solutions for this equation based on the Normal Curve of Probability. The normal curve, and the solution itself, can be efficiently computed with the IMSL function `imsl_f_normal_cdf`, see Chapter 9, “Special Functions." With numerical integration the equation itself or the payoff can be readily changed to include other formulas, $c(s, T)$, and corresponding boundary conditions. We use $e = 100, r = 0.08, T - t = 0.25, \sigma^2 = 0.04, s_L = 0$ and $s_R = 150$.

This is a linear problem but with initial conditions that are discontinuous. It is necessary to use a positive time-smoothing value to prevent grid lines from crossing. We have used an absolute tolerance of $10^{-3}$. In $US, this is one-tenth of a cent.

```
#include <stdio.h>
#include <imsl.h>

/* prototypes */
static void initial_conditions (int npdes, int ngrids, double u[]);
static void pde_systems (double t, double x, int npdes, int ngrids,
             double full_u[], double grid_u[], double dudx[], double *c,
             double q[], double r[], int *ires);
static void boundary_conditions (double t, double beta[], double gamma[],
            double full_u[], double grid_u[], double dudx[], int npdes,
            int ngrids, int left, int *ires);

#define MIN(X,Y) (X<Y)?X:Y
#define NPDE 1
#define N 100
#define XL 0.0
#define XR 150.0
#define U(I_,J_)     u[I_ * ngrids + J_]
int main ()
{
  int i, j, nframes;
  double u[(NPDE + 1) * N];
  double t0 = 0.0, tout, xval;
  double delta_t = 25e-3, tend = 25e-2;
  double xmax = 150.0;
  char *state;
  int npdes = NPDE, ngrids = N;
  double xl = XL, xr = XR;
  FILE *file1;
  double tau = 5e-3;
  double atol = 1e-2;
  double rtol = 0.0;
  int max_bdf_order = 5;

  file1 = fopen ("pde_ex08.out", "w");
  imsl_output_file (IMSL_SET_OUTPUT_FILE, file1, 0);
  nframes = (int) ((tend + delta_t) / delta_t);
  fprintf (file1, "  %d\t%d\t%d", npdes, ngrids, nframes);
  fprintf (file1, "\t%f\t%f\t%f\t%f\n", xl, xr, t0, tend);

  initial_conditions (npdes, ngrids, u);
```

```
  imsl_d_pde_1d_mg_mgr (IMSL_PDE_INITIALIZE, &state,
              IMSL_TIME_SMOOTHING, tau,
              IMSL_MAX_BDF_ORDER, max_bdf_order, 0);
  tout = delta_t;
  do
    {
      imsl_d_pde_1d_mg (npdes, ngrids, &t0, tout, u, xl,
              xr, state, pde_systems, boundary_conditions,
              IMSL_RELATIVE_TOLERANCE, rtol,
              IMSL_ABSOLUTE_TOLERANCE, atol, 0);

     t0 = tout;
     if (t0 <= tend)
       {
         fprintf (file1, "%f\n", tout);
         for (i = 0; i < npdes + 1; i++)
           {
             for (j = 0; j < ngrids; j++)
               {
                 fprintf (file1, "%16.10f   ", U (i, j));
                 if (((j + 1) % 4) == 0)
                   fprintf (file1, "\n");
               }
           }

       }
      tout = MIN ((tout + delta_t), tend);
    }
  while (t0 < tend);

  imsl_d_pde_1d_mg_mgr (IMSL_PDE_RESET, &state, 0);

  fclose (file1);

#undef MIN
#undef NPDE
#undef N
#undef XL
#undef XR
#undef U
}

static void
initial_conditions (int npdes, int ngrids, double u[])
{
#define U(I_,J_)     u[I_ * ngrids + J_]
#define XL 0.0
#define XR 150.0
```

```
  int i;
  double dx, xi, xval, e = 100.0;

  dx = (XR - XL) / (ngrids - 1);
  for (i = 0; i < ngrids; i++)
    {
      xi = XL + i * dx;
      if (xi <= e)
        {
          U (0, i) = 0.0;
        }
     else
       {
         U (0, i) = xi;
       }
    }
#undef U
#undef XL
#undef XR
}
static void
pde_systems (double t, double x, int npdes, int ngrids,
       double full_u[], double grid_u[], double dudx[], double *c,
       double q[], double r[], int *ires)
{
 double sigsq, sigma = 2e-1, rr = 8e-2;
 sigsq = sigma * sigma;

 c[0] = 1.0;
 r[0] = dudx[0] * x * x * sigsq * 0.5;
 q[0] = -(rr - sigsq) * x * dudx[0] + rr * grid_u[0];

 return;
}
static void
boundary_conditions (double t, double beta[], double gamma[],
       double full_u[], double grid_u[], double dudx[],
       int npdes, int ngrids, int left, int *ires)
{
  if (left)
    {
      beta[0] = 0.0;
      gamma[0] = grid_u[0];
    }
  else
    {
      beta[0] = 0.0;
      gamma[0] = dudx[0] - 1.0;
    }
  return;
}
```

## Code for PV-WAVE Plotting

```
PRO PDE_1d_mg_plot, FILENAME = filename, PAUSE = pause
;
  if keyword_set(FILENAME) then file = filename else file = "res.dat"
  if keyword_set(PAUSE) then twait = pause else twait = .1
;
;     Define floating point variables that will be read
;     from the first line of the data file.
  xl = 0D0
  xr = 0D0
  t0 = 0D0
  tlast = 0D0
;
;     Open the data file and read in the problem parameters.
  openr, lun, filename, /get_lun
  readf, lun, npde, np, nt, xl, xr, t0, tlast

;     Define the arrays for the solutions and grid.
  u = dblarr(nt, npde, np)
  g = dblarr(nt, np)
  times = dblarr(nt)
;
;     Define a temporary array for reading in the data.
  tmp = dblarr(np)
  t_tmp = 0D0
;
;     Read in the data.
  for i = 0, nt-1 do begin    ; For each step in time
   readf, lun, t_tmp
   times(i) = t_tmp

   for k = 0, npde-1 do begin ;   For each PDE:
      rmf, lun, tmp
      u(i,k,*) = tmp          ;   Read in the components.
   end

   rmf, lun, tmp
   g(i,*) = tmp               ;   Read in the grid.
  end
;
;     Close the data file and free the unit.
  close, lun
  free_lun, lun
;
;     We now have all of the solutions and grids.
;
;     Delete any window that is currently open.
  while (!d.window NE -1) do WDELETE
;
;     Open two windows for plotting the solutions
```

```
;     and grid.
  window, 0, xsize = 550, ysize = 420
  window, 1, xsize = 550, ysize = 420
;
;      Plot the grid.
  wset, 0
  plot, [xl, xr], [t0, tlast], /nodata, ystyle = 1, $
        title = "Grid Points", xtitle = "X", ytitle = "Time"
  for i = 0, np-1 do begin
     oplot, g(*, i), times, psym = -1
  end
;
;     Plot the solution(s):
  wset, 1
  for k = 0, npde-1 do begin
     umin = min(u(*,k,*))
     umax = max(u(*,k,*))
     for i = 0, nt-1 do begin
        title = strcompress("U_"+string(k+1), /remove_all)+ $
                " at time "+string(times(i))
        plot, g(i, *), u(i,k,*), ystyle = 1, $
              title = title, xtitle = "X", $
              ytitle = strcompress("U_"+string(k+1), /remove_all), $
              xr = [xl, xr], yr = [umin, umax], $
              psym = -4
        wait, twait
     end
  end

end
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# pde_method_of_lines

**_Note_**: This function is deprecated and has been replaced by `imsl_f_modified_method_of_lines`. To view the deprecated documentation, see pde_method_of_lines.pdf on the Rogue Wave website. You can also access a local copy in your IMSL installation directory at `pdf\deprecated_routines\math\pde_method_of_lines.pdf`.

# modified_method_of_lines

Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

**Note**: imsl_f_modified_method_of_lines replaces deprecated function imsl_f_pde_method_of_lines.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_modified_method_of_lines_mgr (*int* task, *void* **state, …, 0)

*void* imsl_f_modified_method_of_lines (*int* npdes, *float* *t, *float* tend, *int* nx, *float* xbreak[], *float* y[], *void* *state, *void* fcn_ut(), *void* fcn_bc())

The type ***double*** functions are imsl_d_modified_method_of_lines_mgr and imsl_d_modified_method_of_lines.

## Required Arguments for imsl_f_modified_method_of_lines_mgr

*int* task (Input)
This function must be called with task set to IMSL_PDE_INITIALIZE to set up memory and default values prior to solving a problem and with task equal to IMSL_PDE_RESET to clean up after it has solved.

*void* **state (Input/Output)
The current state of the PDE solution is held in a structure pointed to by state. It cannot be directly manipulated.

## Required Arguments for imsl_f_modified_method_of_lines

*int* npdes (Input)
Number of differential equations.

*float* *t (Input/Output)
Independent variable. On input, ***t*** supplies the initial time, $t_0$. On output, ***t*** is set to the value to which the integration has been updated. Normally, this new value is tend.

*float* tend (Input)
Value of ***t*** = tend at which the solution is desired.

*int* nx (Input)
Number of mesh points or lines.

*float* `xbreak[]` (Input)

Array of length `nx` containing the breakpoints for the cubic Hermite splines used in the $x$ discretization. The points in `xbreak` must be strictly increasing. The values `xbreak[0]` and `xbreak[nx – 1]` are the endpoints of the interval.

*float* `y[]` (Input/Output)

Array of length `npdes` by `nx` containing the solution. The array `y` contains the solution as $y$[`k,i`] = $u_k$(`x, tend`) at $x$ = `xbreak[i]`. On input, $y$ contains the initial values. It must satisfy the boundary conditions. On output, `y` contains the computed solution.

*void* `*state` (Input/Output)

The current state of the PDE solution is held in a structure pointed to by state. It must be initialized by a call to `imsl_f_modified_method_of_lines_mgr`. It cannot be directly manipulated.

*void* `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`)

User-supplied function to evaluate $u_t$.

*int* `npdes` (Input)

Number of equations.

*float* `x` (Input)

Space variable, $x$.

*float* `t` (Input)

Time variable, $t$.

*float* `u[]` (Input)

Array of length `npdes` containing the dependent values, $u$.

*float* `ux[]` (Input)

Array of length `npdes` containing the first derivatives, $u_x$.

*float* `uxx[]` (Input)

Array of length `npdes` containing the second derivative, $u_{xx}$.

*float* `ut[]` (Output)

Array of length `npdes` containing the computed derivatives $u_t$.

*void* `fcn_bc`(*int* `npdes`, *float* `x`, *float* `t`, *float* `alpha[]`, *float* `beta[]`, *float* `gamma[]`)

User-supplied function to evaluate the boundary conditions. The boundary conditions accepted by `imsl_f_modified_method_of_lines` are

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k$$

**NOTE**: Users must supply the values $\alpha_k$, $\beta_k$, and $\gamma_k$.

*int* `npdes` (Input)
Number of equations.

*float* `x` (Input)
Space variable, *x*.

*float* `t` (Input)
Time variable, *t*.

*float* `alpha[]` (Output)
Array of length `npdes` containing the $\alpha_k$ values.

*float* `beta[]` (Output)
Array of length `npdes` containing the $\beta_k$ values.

*float* `gamma[]` (Output)
Array of length `npdes` containing the values of $\gamma_k$.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*void* `imsl_f_modified_method_of_lines_mgr` (*int* `task`, *void* `**state`,

`IMSL_TOL`, *float* `tol`,

`IMSL_HINIT`, *float* `hinit`,

`IMSL_INITIAL_VALUE_DERIVATIVE`, *float* `initial_deriv[]`,

`IMSL_HTRIAL`, *float* `*htrial`,

`IMSL_FCN_UT_W_DATA`, *void* `fcn_ut()`, *void* `*data`,

`IMSL_FCN_BC_W_DATA`, *void* `fcn_bc()`, *void* `*data`,

0)

## Optional Arguments

`IMSL_TOL`, *float* `tol` (Input)

Differential equation error tolerance. An attempt is made to control the local error in such a way that the global relative error is proportional to `tol`.

Default: `tol` = 100.0*`imsl_f_machine`(4)

`IMSL_HINIT`, *float* `hinit` (Input)

Initial step size in the *t* integration. This value must be nonnegative. If `hinit` is zero, an initial step size of 0.001|`tend` - $t_0$| will be arbitrarily used. The step will be applied in the direction of integration.

Default: `hinit` = 0.0

`IMSL_INITIAL_VALUE_DERIVATIVE`, *float* `initial_deriv[]` (Input/Output)

Supply the derivative values $u_x(x, t_0)$ in `initial_deriv`, an array of length `npdes` by `nx`. This derivative information is input as

$$\texttt{initial\_deriv[k,i]} = \frac{\partial u_k}{\partial x}(x,t_0) \text{ at } x = x[i]$$

The array `initial_deriv` contains the derivative values as output:

$$\texttt{initial\_deriv[k,i]} = \frac{\partial u_k}{\partial x}(x, \texttt{tend}) \text{ at } x = x[i]$$

Default: Derivatives are computed using cubic spline interpolation.

`IMSL_HTRIAL`, *float* `*htrial` (Output)

Return the current trial step size.

`IMSL_FCN_UT_W_DATA`, *void* `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`, *void* `*data`), *void* `*data` (Input)

User-supplied function to evaluate $u_t$, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user–supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

`IMSL_FCN_BC_W_DATA`, *void* `fcn_bc`(*int* `npdes`, *float* `x`, *float* `t`, *float* `alpha[]`, *float* `beta[]`, *float* `gamma[]`, *void* `*data`), *void* `*data` (Input)

User-supplied function to evaluate the boundary conditions, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

Let $M$ = `npdes`, $N$ = `nx` and $x_i$ = `xbreak(I)`. The function `imsl_f_modified_method_of_lines` uses the method of lines to solve the partial differential equation system

$$\text{initial\_deriv}\left[k,i\right] = \frac{\partial u_k}{\partial x}\left(x,t_0\right) \text{ at } x = x\left[i\right]$$

$$\text{initial\_deriv}\left[k,i\right] = \frac{\partial u_k}{\partial x}\left(x,\text{tend}\right) \text{ at } x = x\left[i\right]$$

$$\frac{\partial u_k}{\partial t} = f_k\left(x,t,u_1,\ldots u_M,\frac{\partial u_1}{\partial x},\ldots\frac{\partial u_M}{\partial x},\frac{\partial^2 u_1}{\partial x^2},\ldots\frac{\partial^2 u_M}{\partial x^2}\right)$$

with the initial conditions

$$u_k = u_k\left(x,t\right) \text{ at } t = t_0$$

and the boundary conditions

for $k$ = 1, …, $M$.

Cubic Hermite polynomials are used in the $x$ variable approximation so that the trial solution is expanded in the series

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k \text{ at } x = x_1 \text{ and at } x = x_N$$

$$\hat{u}_k\left(x,t\right) = \sum_{i=1}^{N}\left(a_{k,i}\left(t\right)\phi_i\left(x\right) + b_{k,i}\left(t\right)\psi_i\left(x\right)\right)$$

where $\phi_i(x)$ and $\psi_i(x)$ are the standard basis functions for the cubic Hermite polynomials with the knots $x_1 < x_2 < \ldots < x_N$. These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\phi_i\left(x_l\right) = \delta_{il} \quad \psi_i\left(x_l\right) = 0$$

$$\frac{d\phi_i}{dx}\left(x_l\right) = 0 \quad \frac{d\psi_i}{dx}\left(x_l\right) = \delta_{il}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equation at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} - x_j\right)$$

$$p_{2j} = x_j + \frac{3+\sqrt{3}}{6}\left(x_{j+1} + x_j\right)$$

for $j$ = 1, …, $N$. The collocation approximation to the differential equation is

$$\sum_{i=1}^{N} \frac{da_{k,i}}{dt}\phi_i\left(p_j\right) + \frac{db_{k,i}}{dt}\psi_i\left(p_j\right) =$$
$$f_k\left(p_j,t,\hat{u}_1\left(p_j\right), \ldots, \hat{u}_M\left(p_j\right), \ldots, \left(\hat{u}_1\right)_{xx}\left(p_j\right), \ldots, \left(\hat{u}_M\right)_{xx}\left(p_j\right)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N - 1)$.

This is a system of $2M(N - 1)$ ordinary differential equations in $2M N$ unknown coefficient functions, $a_{k,i}$ and $b_{k,i}$. This system can be written in the matrix−vector form as $A\, dc/dt = F(t, c)$ with $c(t_0) = c_0$ where $c$ is a vector of coefficients of length $2M N$ and $c_0$ holds the initial values of the coefficients. The last $2M$ equations are obtained from the boundary conditions.

If $\alpha_k = \beta_k = 0$, it is assumed that no boundary condition is desired for the $k$−th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

The input/output array `Y` contains the values of the $a_{k,i}$. The initial values of the $b_{k,i}$ are obtained by using the IMSL cubic spline function `imsl_f_cub_spline_interp_e_cnd` (Interpolation and Approximation) to construct functions

$$\hat{u}_k\left(x_i, t_0\right)$$

such that

$$d\hat{u}_k\left(x_i,\ t_0\right) = a_{ki}$$

The IMSL function `imsl_f_cub_spline_value` (Interpolation and Approximation) is used to approximate the values

$$\hat{u}_k\left(x_i,\ t_0\right) = a_{ki}$$

$$\frac{d\hat{u}_k}{dx}\left(x_i,\ t_0\right) \equiv b_{k,\ i}$$

Optional argument `IMSL_INITIAL_VALUE_DERIVATIVE` allows the user to provide the initial values of $b_{k,i}$.

The order of matrix $A$ is $2M N$ and its maximum bandwidth is $6M - 1$. The band structure of the Jacobian of $F$ with respect to $c$ is the same as the band structure of $A$. This system is solved using a modified version of `imsl_f_ode_adams_gear`. Numerical Jacobians are used exclusively. Gear's BDF method is used as the default because the system is typically stiff. For more details, see Sewell (1982).

Four examples of PDEs are now presented that illustrate how users can interface their problems with IMSL PDE solving software. The examples are small and not indicative of the complexities that most practitioners will face in their applications. A set of seven sample application problems, some of them with more than one equation, is given in Sincovec and Madsen (1975). Two further examples are given in Madsen and Sincovec (1979).

# Examples

## Example 1

The normalized linear diffusion PDE, $u_t = u_{xx}$, $0 \le x \le 1$, $t > t_0$, is solved. The initial values are $t_0 = 0$, $u(x, t_0) = u_0 = 1$. There is a "zero-flux" boundary condition at $x = 1$, namely $u_x(1, t) = 0$, $(t > t_0)$. The boundary value of $u(0, t)$ is abruptly changed from $u_0$ to the value 0, for $t > 0$.

When the boundary conditions are discontinuous, or incompatible with the initial conditions such as in this example, it may be important to use double precision.

```
#include <imsl.h>
#include <stdio.h>

void fcnut(int, float, float, float *, float *, float *, float *);
void fcnbc(int, float, float, float *, float *, float *);

int main() {

   int npdes = 1, nx = 8, i, j, nstep = 10;
   float hinit, tol, t = 0.0, tend, xbreak[8], y[8];
   char title[50];
   void *state;

   /* Set breakpoints and initial conditions */

   for (i = 0; i < nx; i++) {
       xbreak[i] = (float) i / (float) (nx - 1);
       y[i] = 1.0;
   }

   /* Initialize the solver */

   tol = 10.e-4;
   hinit = 0.01 * tol;
   imsl_f_modified_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
       IMSL_TOL, tol,
       IMSL_HINIT, hinit,
       0);
   for (j = 1; j <= nstep; j++) {
       tend = (float) j / (float) nstep;
       tend *= tend;

       /* Solve the problem */

       imsl_f_modified_method_of_lines(npdes, &t, tend, nx, xbreak, y,
           state, fcnut, fcnbc);

       /* Print results at current t=tend */
```

```
        sprintf(title, "solution at t = %4.2f", t);
        imsl_f_write_matrix(title, npdes, nx, y, 0);
    }
}


void fcnut(int npdes, float x, float t, float *u, float *ux,
   float *uxx, float *ut) {
   /* Define the PDE */
   *ut = *uxx;
}


void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
   float *gam) {
   float delta = 0.09, u0 = 1.0, u1 = 0.1;

   /* Define boundary conditions */

   if (x == 0.0) {

       /* These are for x = 0 */

       *alpha = 1.0;
       *beta = 0.0;
       *gam = u1;

       /* If in the boundary layer, compute

       nonzero gamma */
       if (t <= delta)
           *gam = u0 +(u1 - u0)* t/delta;
   } else {

       /* These are for x = 1 */

       *alpha = 0.0;
       *beta = 1.0;
       *gam = 0.0;
   }
}
```

## Output

```
                          solution at t = 0.01
         1            2            3            4            5            6
     0.900        0.985        0.999        1.000        1.000        1.000

         7            8
     1.000        1.000
```

```
                         solution at t = 0.04
      1          2          3          4          5          6
  0.600      0.834      0.941      0.982      0.996      0.999

      7          8
  1.000      1.000

                         solution at t = 0.09
      1          2          3          4          5          6
 0.1003     0.4906     0.7304     0.8673     0.9395     0.9743

      7          8
 0.9891     0.9931

                         solution at t = 0.16
      1          2          3          4          5          6
 0.1000     0.3145     0.5094     0.6702     0.7905     0.8709

      7          8
 0.9159     0.9303

                         solution at t = 0.25
      1          2          3          4          5          6
 0.1000     0.2571     0.4052     0.5361     0.6434     0.7228

      7          8
 0.7713     0.7876

                         solution at t = 0.36
      1          2          3          4          5          6
 0.1000     0.2178     0.3295     0.4296     0.5129     0.5754

      7          8
 0.6142     0.6273

                         solution at t = 0.49
      1          2          3          4          5          6
 0.1000     0.1852     0.2661     0.3387     0.3993     0.4449

      7          8
 0.4732     0.4827

                         solution at t = 0.64
      1          2          3          4          5          6
 0.1000     0.1587     0.2144     0.2644     0.3061     0.3375

      7          8
 0.3570     0.3636

                         solution at t = 0.81
```

```
         1           2           3           4           5           6
    0.1000      0.1385      0.1752      0.2080      0.2354      0.2561

         7           8
    0.2689      0.2732

                          solution at t = 1.00
         1           2           3           4           5           6
    0.1000      0.1243      0.1475      0.1682      0.1855      0.1985

         7           8
    0.2066      0.2093
```

### Example 2

Here, Problem C is solved from Sincovec and Madsen (1975). The equation is of diffusion-convection type with discontinuous coefficients. This problem illustrates a simple method for programming the evaluation routine for the derivative, $u_t$. Note that the weak discontinuities at $x$ = 0.5 are not evaluated in the expression for $u_t$. The problem is defined as

$$u_t = \partial u / \partial t = \partial / \partial x\left(D\left(x\right)\partial u / \partial x\right) - v(x)\partial u / \partial x$$
$$x \in [0,1], t > 0$$

$$D\left(x\right) = \begin{cases} 5 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$v\left(x\right) = \begin{cases} 1000.0 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$u\left(x,0\right) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$$

$$u\left(0,t\right) = 1, u\left(1,t\right) = 0$$

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

void fcnut(int, float, float, float *, float *, float *, float *);
void fcnbc(int, float, float, float *, float *, float *);

int main()
{
   int   i, j, npdes = 1, nstep = 10, nx = 100;
   float hinit, t = 0.0, tend, tol, xbreak[100], y[100];
   char  title[50];
   void  *state;

   /* Set breakpoints and initial conditions */
```

```
    for (i = 0; i < nx; i++) {
        xbreak[i] = (float) i / (float) (nx - 1);
        y[i] = 0.0;
    }
    y[0] = 1.0;

    /* Initialize the solver */

    tol = sqrt(imsl_f_machine(4));
    hinit = 0.01*tol;
    imsl_f_modified_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
        IMSL_TOL, tol,
        IMSL_HINIT, hinit,
        0);

    for (j = 1; j <= nstep; j++) {
        tend = (float) j / (float) nstep;

        /* Solve the problem */

        imsl_f_modified_method_of_lines(npdes, &t, tend, nx, xbreak, y,
            state, fcnut, fcnbc);
    }

    /* Print results at t=tend */

    sprintf(title, "solution at t = %4.2f", t);
    imsl_f_write_matrix(title, npdes, nx, y, 0);
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
    float *ut)
{
    /* Define the PDE */

    float d = 1.0, v = 1.0;

    if (x <= 0.5) {
        d = 5.0;
        v = 1000.0;
    }
    ut[0] = d*uxx[0] - v*ux[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
    float *gam)
{
    if (x == 0.0) {
        *alpha = 1.0;
        *beta = 0.0;
        *gam = 1.0;
```

```
    } else {
        *alpha = 1.0;
        *beta = 0.0;
        *gam = 0.0;
    }
}
```

### Output

```
                        solution at t = 1.00
        1           2           3           4           5           6
    1.000       1.000       1.000       1.000       1.000       1.000

        7           8           9          10          11          12
    1.000       1.000       1.000       1.000       1.000       1.000

       13          14          15          16          17          18
    1.000       1.000       1.000       1.000       1.000       1.000

       19          20          21          22          23          24
    1.000       1.000       1.000       1.000       1.000       1.000

       25          26          27          28          29          30
    1.000       1.000       1.000       1.000       1.000       1.000

       31          32          33          34          35          36
    1.000       1.000       1.000       1.000       1.000       1.000

       37          38          39          40          41          42
    1.000       1.000       1.000       1.000       1.000       1.000

       43          44          45          46          47          48
    1.000       1.000       1.000       1.000       1.000       1.000

       49          50          51          52          53          54
    1.000       0.997       0.984       0.969       0.953       0.937

       55          56          57          58          59          60
    0.921       0.905       0.888       0.872       0.855       0.838

       61          62          63          64          65          66
    0.821       0.804       0.786       0.769       0.751       0.733

       67          68          69          70          71          72
    0.715       0.696       0.678       0.659       0.640       0.621

       73          74          75          76          77          78
    0.602       0.582       0.563       0.543       0.523       0.502

       79          80          81          82          83          84
    0.482       0.461       0.440       0.419       0.398       0.376
```

```
      85          86          87          88          89          90
   0.354       0.332       0.310       0.288       0.265       0.242

      91          92          93          94          95          96
   0.219       0.196       0.172       0.148       0.124       0.100

      97          98          99         100
   0.075       0.050       0.025       0.000
```

## Example 3

In this example, using `imsl_f_modified_method_of_lines`, the linear normalized diffusion PDE $u_t = u_{xx}$ is solved but with an optional use that provides values of the derivatives, $u_x$, of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$, $n > 1$. The boundary conditions are "zero flux" conditions $u_x(0, t) = u_x(1, t) = 0$ for $t > 0$.

This optional usage signals that the derivative of the initial data is passed by the user. The values $u(x, tend)$ and $u_x(x, tend)$ are output at the breakpoints with the optional usage.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

void fcnut(int, float, float, float *, float *, float *, float *);
void fcnbc(int, float, float, float *, float *, float *);

int main()
{
   int i, j, npdes = 1, nstep = 10, nx = 10;
   float arg, hinit, tol, pi, t = 0.0, tend = 0.0;
   float deriv[10], xbreak[10], y[10];
   char  title[50];
   void  *state;

   pi = imsl_f_constant("pi", 0);
   arg = 9.0 * pi;

   /* Set breakpoints and initial conditions */

   for (i = 0; i < nx; i++) {
       xbreak[i] = (float) i / (float) (nx - 1);
       y[i] = 1.0 + cos(arg * xbreak[i]);
       deriv[i] = -arg * sin(arg * xbreak[i]);
   }

   /* Initialize the solver */

   tol = sqrt(imsl_f_machine(4));
   imsl_f_modified_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
       IMSL_TOL, tol,
       IMSL_INITIAL_VALUE_DERIVATIVE, deriv,
       0);

   for (j = 1; j <= nstep; j++) {
       tend += 0.001;

       /* Solve the problem */

       imsl_f_modified_method_of_lines(npdes, &t, tend, nx, xbreak, y,
           state, fcnut, fcnbc);

       /* Print results at every other t=tend */

       if (!(j % 2)) {
           sprintf(title, "\nsolution at t = %5.3f", t);
           imsl_f_write_matrix(title, npdes, nx, y, 0);
           sprintf(title, "\nderivative at t = %5.3f", t);
           imsl_f_write_matrix(title, npdes, nx, deriv, 0);
       }
```

```
    }
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
    float *ut)
{
    /* Define the PDE */

    *ut = *uxx;
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
    float *gam)
{
    /* Define the boundary conditions */

    alpha[0] = 0.0;
    beta[0] = 1.0;
    gam[0] = 0.0;
}
```

### Output

```
                          solution at t = 0.002
         1           2           3           4           5           6
     1.234       0.766       1.234       0.766       1.234       0.766

         7           8           9          10
     1.234       0.766       1.234       0.766


                         derivative at t = 0.002
         1           2           3           4           5           6
0.000e+000   8.983e-007 -3.682e-008   1.772e-006   4.368e-008   2.619e-006

         7           8           9          10
-1.527e-006   4.956e-006 -3.003e-006 -3.009e-011


                          solution at t = 0.004
         1           2           3           4           5           6
     1.054       0.946       1.054       0.946       1.054       0.946

         7           8           9          10
     1.054       0.946       1.054       0.946


                         derivative at t = 0.004
         1           2           3           4           5           6
0.000e+000   2.646e-007   4.763e-007   1.009e-006 -5.439e-007 -8.247e-007
```

```
         7           8           9          10
 3.142e-007  1.750e-006 -1.019e-006  4.300e-012


                     solution at t = 0.006
         1           2           3           4           5           6
     1.012       0.988       1.012       0.988       1.012       0.988

         7           8           9          10
     1.012       0.988       1.012       0.988


                    derivative at t = 0.006
         1           2           3           4           5           6
 0.000e+000  4.923e-007  4.082e-008  6.763e-007 -3.347e-007 -7.026e-007

         7           8           9          10
 4.525e-007  3.456e-007 -5.008e-007  1.327e-012


                     solution at t = 0.008
         1           2           3           4           5           6
     1.003       0.997       1.003       0.997       1.003       0.997

         7           8           9          10
     1.003       0.997       1.003       0.997


                    derivative at t = 0.008
         1           2           3           4           5           6
 0.000e+000 -1.323e-007  1.079e-006  2.271e-007 -7.651e-007  4.554e-007

         7           8           9          10
 7.479e-007 -5.015e-009 -3.918e-007  2.261e-013


                     solution at t = 0.010
         1           2           3           4           5           6
     1.001       0.999       1.001       0.999       1.001       0.999

         7           8           9          10
     1.001       0.999       1.001       0.999


                    derivative at t = 0.010
         1           2           3           4           5           6
 0.000e+000  9.523e-008  1.043e-006  3.912e-007 -6.791e-007  2.734e-008

         7           8           9          10
 4.506e-007  2.447e-007 -2.414e-008 -1.161e-015
```

## Example 4

In this example, consider the linear normalized hyperbolic PDE, $u_{tt} = u_{xx}$, the "vibrating string" equation. This naturally leads to a system of first order PDEs. Define a new dependent variable $u_t = v$. Then, $v_t = u_{xx}$ is the second equation in the system. Take as initial data $u(x, 0) = \sin(\pi x)$ and $u_t(x, 0) = v(x, 0) = 0$. The ends of the string are fixed so $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$. The exact solution to this problem is $u(x, t) = \sin(\pi x) \cos(\pi t)$. Residuals are computed at the output values of $t$ for $0 < t \le 2$. Output is obtained at 200 steps in increments of 0.01.

Even though the sample code `imsl_f_modified_method_of_lines` gives satisfactory results for this PDE, users should be aware that for ***nonlinear problems***, "shocks" can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pp 488-490, for an introductory discussion of shocks in hyperbolic systems.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

void fcnut(int, float, float, float *, float *, float *, float *);
void fcnbc(int, float, float, float *, float *, float *);

int main()
{
   int  i, j, npdes = 2, nstep = 200, nx = 10;
   float error[10], erru = 0.0, err, hinit, pi, t = 0.0, tend = 0.0;
   float deriv[20], tol, y[20], xbreak[10];
   void  *state;

   pi = imsl_f_constant("pi", 0);

   /* Set breakpoints and initial conditions */

   for (i = 0; i < nx; i++) {
       xbreak[i] = (float) i / (float) (nx - 1);
       y[i] = sin(pi * xbreak[i]);
       y[nx + i] = 0.0;
       deriv[i] = pi * cos(pi * xbreak[i]);
       deriv[nx + i] = 0.0;
   }

   /* Initialize the solver */

   tol = sqrt(imsl_f_machine(4));
   imsl_f_modified_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
       IMSL_TOL, tol,
       IMSL_INITIAL_VALUE_DERIVATIVE, deriv,
       0);

   for ( j=1; j <= nstep; j++) {
       tend += 0.01;

       /* Solve the problem */

       imsl_f_modified_method_of_lines(npdes, &t, tend, nx, xbreak, y,
           state, fcnut, fcnbc);


       /* Look at output at steps of 0.01 and compute errors */


       for (i = 0; i < nx; i++) {
           error[i] = y[i] - sin(pi * xbreak[i]) * cos(pi *tend);
           err = fabs(error[i]);
           if (err > erru) erru = err;
```

```
        }
    }
    printf("Maximum error in u(x,t) = %e\n", erru);
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
    float *ut)
{
    /* Define the PDE */

    ut[0] = u[1];
    ut[1] = uxx[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
    float *gam)
{
    /* Define the boundary conditions */

    alpha[0] = 1.0;
    beta[0] = 0.0;
    gam[0] = 0.0;
    alpha[1] = 1.0;
    beta[1] = 0.0;
    gam[1] = 0.0;
}
```

### Output

```
Maximum error in u(x,t) = 5.757928e-003
```

## Fatal Errors

| | |
|---|---|
| `IMSL_REPEATED_ERR_TEST_FAILURE` | After some initial success, the integration was halted by repeated error test failures. |
| `IMSL_INTEGRATION_HALTED_1` | Integration halted after failing to pass the error test, even after reducing the stepsize by a factor of 1.0E+10 to H = #. `TOL` = # may be too small. |
| `IMSL_INTEGRATION_HALTED_2` | Integration halted after failing to achieve corrector convergence, even after reducing the stepsize by a factor of 1.0E+10 to H = #. `TOL` = # may be too small. |
| `IMSL_INTEGRATION_HALTED_3` | After some initial success, the integration was halted by a test on `TOL` = #. |
| `IMSL_TOL_TOO_SMALL_OR_STIFF` | On the next step X+H will equal X, with X = # and H = #. Either `TOL` = # is too small or the problem is stiff. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# feynman_kac

more...

This function solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are satisfied. The solution is represented by a series of $C^2$ Hermite quintic splines.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_feynman_kac (*int* nxgrid, *int* ntgrid, *int* nlbcd, *int* nrbcd, *float* xgrid[], *float* tgrid[], *void* fcn_fkcfiv(), *void* fcn_fkbcp(), *float* y[], *float* y_prime[],..., 0)

The type *double* function is imsl_d_feynman_kac.

## Required Arguments

*int* nxgrid (Input)
Number of grid lines in the *x*-direction. nxgrid must be at least 2.

*int* ntgrid (Input)
Number of time points where an approximate solution is returned. The value ntgrid must be at least 1.

*int* nlbcd (Input)
Number of left boundary conditions. It is required that 1 ≤ nlbcd ≤ 3.

*int* nrbcd (Input)
Number of right boundary conditions. It is required that 1 ≤ nrbcd ≤ 3.

*float* xgrid[] (Input)
Array of length nxgrid containing the breakpoints for the Hermite quintic splines used in the *x* discretization. The points in xgrid must be strictly increasing. The values xgrid[0] and xgrid[nxgrid-1] are the endpoints of the interval.

*float* tgrid[] (Input)
Array of length ntgrid containing the set of time points (in time-remaining units) where an approximate solution is returned. The points in tgrid must be positive and strictly increasing.

*void* `fcn_fkcfiv` (*float* `x`, *float* `t`, *int* `*iflag`, *float* `*value`)

User-supplied function to compute the values of the coefficients $\sigma, \sigma', \mu, \kappa$ for the Feynman-Kac PDE and the initial data function $p(x), x_{\min} \le x \le x_{\max}$.

*float* `x` (Input)

Space variable.

*float* `t` (Input)

Time variable.

*int* `*iflag` (Input/Output)

On entry, `iflag` indicates which coefficient or data function is to be computed. The following table shows which value has to be returned by `fcn_fkcfiv` for all possible values of `iflag`:

| `iflag` | **Computed coefficient** |
|---|---|
| -1 | $\sigma' = \frac{\partial \sigma(x, t)}{\partial x}$ |
| 0 | $p(x)$ |
| 1 | $\sigma$ |
| 2 | $\mu$ |
| 3 | $\kappa$ |

For non-zero input values of `iflag`, note when a coefficient does not depend on *t*. This is done by setting `iflag = 0` after the coefficient is defined. If there is time dependence, the value of `iflag` should not be changed. This action will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each *t* value.

*float* `*value` (Output)

Value of the coefficient or initial data function. Which value is computed depends on the input value for `iflag`, see description of `iflag`.

*void* `fcn_fkbcp` (*int* `nbc`, *float* `t`, *int* `*iflag`, *float* `values[]`)

User-supplied function to define boundary values that the solution of the differential equation must satisfy. There are `nlbcd` conditions specified at the left end, $x_{\min}$, and `nrbcd` conditions at the right end, $x_{\max}$. The boundary conditions are

$$a(x,t) f + b(x,t) f_x + c(x,t) f_{xx} = d(x,t), \quad x = x_{\min} \text{ or } x = x_{\max}$$

*int* `nbc` (Input)

Number of boundary conditions. At $x_{\min}$, `nbc=nlbcd`, at $x_{\max}$, `nbc = nrbcd`.

*float* `t` (Input)

Time point of the boundary conditions.

*int* `*iflag` (Input/Output)

On entry, `iflag` indicates whether the coefficients for the left or right boundary conditions are to be computed:

| `iflag` | **Computed boundary conditions** |
|---|---|
| 1 | Left end, x = $x_{min}$ |
| 0 | Right end, x = $x_{max}$ |

If there is no time dependence for one of the boundaries then set `iflag = 0` after the array `values` is defined for either end point. This flag can avoid unneeded continued computation of the finite element matrices.

*float* `values[]` (Output)

Array of length 4 * max (`nlbcd`, `nrbcd`) containing the values of the boundary condition coefficients in its first `4*nbc` locations. The coefficients for $x_{min}$ are stored row-wise according to the following scheme:

$$\begin{pmatrix} a_1(x_{\min}, t), b_1(x_{\min}, t), c_1(x_{\min}, t), d_1(x_{\min}, t) \\ \vdots \\ a_{nlbcd}(x_{\min}, t), b_{nlbcd}(x_{\min}, t), c_{nlbcd}(x_{\min}, t), d_{nlbcd}(x_{\min}, t) \end{pmatrix}$$

The coefficients for $x_{max}$ are stored similarly.

*float* `y[]` (Output)

An array of size (`ntgrid`+1) by (3 × `nxgrid`) containing the coefficients of the Hermite representation of the approximate solution for the Feynman-Kac PDE at time points 0, `tgrid[0]`, …, `tgrid[ntgrid-1]`. The approximate solution is given by

$$f(x,t) = \sum_{j=0}^{3*\texttt{nxgrid}-1} y_{ij}\beta_j(x)$$

for

$$t = \texttt{tgrid}\left[i-1\right], i = 1, \ldots, \texttt{ntgrid}$$

The representation for the initial data at $t$ = 0 is

$$p(x) = \sum_{j=0}^{3*\mathtt{nxgrid}-1} y_{0j}\beta_j(x)$$

The (`ntgrid` + 1) by (3 * `nxgrid`) matrix

$$(y_{ij})_{i=0,\ldots,\mathtt{ntgrid}}^{j=0,\ldots,3*\mathtt{nxgrid}-1}$$

is stored row-wise in array `y`.

After the integration, use row `i` of array `y` as input argument `coef` in function `imsl_f_feynman_kac_evaluate` to obtain an array of values for $f(x, t)$ or its partials $f_x, f_{xx}, f_{xxx}$ at time point `t=0,tgrid[i-1]`, `i=1,...,ntgrid`.

The expressions for the basis functions $\beta_j(x)$ are represented piece-wise and can be found in Hanson, R. (2008) Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements.

*float* `y_prime[]` (Output)

An array of size (`ntgrid` + 1) by (3 × `nxgrid`) containing the first derivatives of `y` at time points `0`, `tgrid[0]`,...,`tgrid[ntgrid - 1]`, i.e.

$$f_t(x,\bar{t}) = \sum_{j=0}^{3*\mathtt{nxgrid}-1} y'_{ij}\beta_j(x)$$

for

$$\bar{t} = \mathtt{tgrid}[i-1], i = 1, \ldots, \mathtt{ntgrid}$$

and

$$f_t(x,\bar{t}) = \sum_{j=0}^{3*\mathtt{nxgrid}-1} y'_{0j}\beta_j(x) \quad \text{for } \bar{t} = 0.$$

The (`ntgrid` + 1) by (3 *`nxgrid`) matrix

$$(y'_{ij})_{i=0,\ldots,\mathtt{ntgrid}}^{j=0,\ldots,3*\mathtt{nxgrid}-1}$$

is stored row-wise in array `y_prime`.

After the integration, use row `i` of array `y_prime` as input argument `coef` in function `imsl_f_feynman_kac_evaluate` to obtain an array of values for the partials $f_t, f_{tx}, f_{txx}, f_{txxx}$ at time point `t=tgrid[i-1]`, `i=1,...,ntgrid`, and row `0` for the partials at `t=0`.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_f_feynman_kac (*int* nxgrid, *int* ntgrid, *int* nlbcd, *int* nrbcd, *float* xgrid[],
*float* tgrid[], *void* fcn_fkcfiv(), *void* fcn_fkbcp(), *float* y[], *float* y_prime[],

IMSL_FCN_FKCFIV_W_DATA, *void* fcn_fkcfiv(), *void* *data,

IMSL_FCN_FKBCP_W_DATA, *void* fcn_fkbcp(), *void* *data,

IMSL_FCN_INIT, *void* fcn_fkinit(),

IMSL_FCN_INIT_W_DATA, *void* fcn_fkinit(), *void* *data,

IMSL_FCN_FORCE, *void* fcn_fkforce(),

IMSL_FCN_FORCE_W_DATA, *void* fcn_fkforce(), *void* *data,

IMSL_ATOL_RTOL_SCALARS, *float* atol, *float* rtol,

IMSL_ATOL_RTOL_ARRAYS, *float* atol[], *float* rtol[]

IMSL_NDEGREE, *int* ndeg,

IMSL_TDEPEND, *int* tdepend[],

IMSL_MAX_STEP, *float* max_stepsize,

IMSL_INITIAL_STEPSIZE, *float* init_stepsize,

IMSL_MAX_NUMBER_STEPS, *int* max_steps,

IMSL_STEP_CONTROL, *int* step_control,

IMSL_MAX_BDF_ORDER, *int* max_bdf_order,

IMSL_T_BARRIER, *float* t_barrier,

IMSL_ISTATE, *int* istate[],

IMSL_EVALS, *int* nval[],

0)

## Optional Arguments

IMSL_FCN_FKCFIV_W_DATA, *void* fcn_fkcfiv(*float* x, *float* t, *int* *iflag, *float* *value,
*void* *data), *void* *data (Input)

User-supplied function to compute the values of the coefficients $\sigma, \sigma', \mu, \kappa$ for the Feynman-Kac PDE and the initial data function $p(x), x_{\min} \le x \le x_{\max}$. This function also accepts a pointer to the object data supplied by the user.

IMSL_FCN_FKBCP_W_DATA, *void* fcn_fkbcp (*int* nbc, *float* t,*int* *iflag, *float* values[], *void* *data), *void* *data (Input)

User-supplied function to define boundary values that the solution of the differential equation must satisfy. This function also accepts a pointer to data supplied by the user.

IMSL_FCN_INIT, *void* fcn_fkinit(*int* nxgrid, *int* ntgrid, *float* xgrid[], *float* tgrid[], *float* time, *float* yprime[], *float* y[], *float* atol[], *float* rtol[]) (Input)

User-supplied function for adjustment of initial data or as an opportunity for output during the integration steps. The solution values of the model parameters are presented in the arrays y and yprime at the integration points time = 0, tgrid[j], j=0,...,ntgrid-1. At the initial point, integral least-squares estimates are made for representing the initial data $p(x)$. If this is not satisfactory, fcn_fkinit can change the contents of y[] to match data for any reason.

*int* nxgrid (Input)

Number of grid lines in the *x*-direction.

*int* ntgrid (Input)

Number of time points where an approximate solution is returned.

*float* xgrid[] (Input)

Vector of length nxgrid containing the breakpoints for the Hermite quintic splines used in the *x* discretization.

*float* tgrid[] (Input)

Vector of length ntgrid containing the set of time points (in time-remaining units) where an approximate solution is returned.

*float* time (Input)

Time variable.

*float* yprime[] (Input)

Vector of length 3*nxgrid containing the derivative of the coefficients of the Hermite quintic spline at time point time.

*float* y[] (Input/Output)

Vector of length 3 × nxgrid containing the coefficients of the Hermite quintic spline at time point time.

*float* atol[] (Input/Output)

Array of length 3 × nxgridcontaining absolute error tolerances.

*float* rtol[] (Input/Output)

Array of length 3 × nxgrid containing relative error tolerances.

IMSL_FCN_INIT_W_DATA, *void* fcn_fkinit(*int* nxgrid, *int* ntgrid, *float* xgrid[], *float* tgrid[], *float* time, *float* yprime[], *float* y[], *float* atol[], *float* rtol[], *void* *data), *void* *data (Input)

User-supplied function for adjustment of initial data or as an opportunity for output during the integration steps which also accepts a pointer to data supplied by the user. For an explanation of the other arguments of function `fcn_fkinit`, see optional argument `IMSL_FCN_INIT`.

`IMSL_FCN_FORCE`, *void* `fcn_fkforce`(*int* `interval`, *int* `ndeg`, *int* `nxgrid`, *float* `y[]`, *float* `time`, *float* `width`, *float* `xlocal[]`, *float* `qw[]`, *float* `u[]`, *float* `phi[]`, *float* `dphi[]`) `(Input)`

Function `fcn_fkforce` is used in case there is a non-zero term $\phi\left(f, x, t\right)$ in the Feynman-Kac differential equation. If function `fcn_fkforce` is not used, it is assumed that $\phi\left(f, x, t\right)$ is identically zero.

*int* `interval` (Input)

Index indicating the bounds `xgrid[interval-1]` and `xgrid[interval]` of the integration interval, `1≤interval ≤ nxgrid-1`.

*int* `ndeg` (Input)

The degree used for the Gauss-Legendre formulas.

*int* `nxgrid` (Input)

Number of grid lines in the x-direction.

*float* `y[]` (Input)

Vector of length `3*nxgrid` containing the coefficients of the Hermite quintic spline representing the solution of the Feynman-Kac equation at time point `time`.

*float* `time` (Input)

Time variable.

*float* `width` (Input)

The interval width, `width = xgrid[interval] - xgrid[interval-1]`.

*float* `xlocal[]` (Input)

Array of length `ndeg` containing the Gauss-Legendre points translated and normalized to the interval `[xgrid[interval-1], xgrid[interval]]`.

*float* `qw[]` `(Input)`

Vector of length `ndeg` containing the Gauss-Legendre weights.

*float* `u[]` (Input)

Array of dimension 12 by `ndeg` containing the basis function values that define $\tilde{\beta}\left(x\right)$ at the Gauss-Legendre points `xlocal`. Setting $u_{k,i} := $ `u[i+k*ndeg]` and $x_i := $ `xlocal[i]`, $\tilde{\beta}\left(x_i\right)$ is defined as

$$\tilde{\beta}(x_i) := \left(\beta_{3*(\text{interval}-1)}(x_i), \ldots, \beta_{3*(\text{interval}+2)}(x_i)\right)^T$$
$$= \left(u_{0,\mathrm{i}}, u_{1,\mathrm{i}}, u_{2,\mathrm{i}}, u_{6,\mathrm{i}}, u_{7,\mathrm{i}}, u_{8,\mathrm{i}}\right)^T.$$

*float* `phi[]` (Output)

Vector of length 6 containing Gauss-Legendre approximations for the local contributions

$$\varphi_{\mathrm{t}} := \int_{\text{xgrid}[\text{interval}-1]}^{\text{xgrid}[\text{interval}]} \Phi(f,x,t)\tilde{\beta}(x)\,dx,$$

where $t =$ time and

$$\tilde{\beta}(x) := \left(\beta_{3*(\text{interval}-1)}(x), \ldots, \beta_{3*(\text{interval}+2)}(x)\right)^T$$

Vector phi contains elements

$$\text{phi}[i] = \text{width} * \sum_{\mathrm{j}=0}^{\text{ndegGL}-1} qw[j]\,\tilde{\beta}_{\mathrm{i}}(x_{\mathrm{j}})\,\phi(f, \text{xlocal}[j], \text{time})$$

for `i`=0,...,5.

*float* `dphi[]` (Output)

Array of dimension 6 `by` 6 containing a Gauss-Legendre approximation for the Jacobian of the local contributions $\varphi_t$ at $t =$ time,

$$\frac{\partial \varphi_{\mathrm{t}}}{\partial y} = \int_{\text{xgrid}[\text{interval}-1]}^{\text{xgrid}[\text{interval}]} \frac{\partial \Phi(f,x,t)}{\partial f}\tilde{\beta}(x)\tilde{\beta}^{\mathrm{T}}(x)\,dx.$$

The approximation to this symmetric matrix is stored row-wise, i.e.

$$\text{dphi}[j + i*6] = \text{width} * \sum_{k=0}^{ndegGL-1} \text{qw}[k]\;\; \tilde{\beta}_i(x_k)\tilde{\beta}_j(x_{\mathrm{k}})\frac{\partial \Phi}{\partial f}\Big|_{x=xlocal[\mathrm{k}],t=time}$$

for i, j = 0, ..., 5.

`IMSL_FCN_FORCE_W_DATA`, *void* `fcn_fkforce`(*int* `interval`, *int* `ndeg`, *int* `nxgrid`, *float* `y[]`, *float* `time`, *float* `width`, *float* `xlocal[]`, *float* `qw[]`, *float* `u[]`, *float* `phi[]`, *float* `dphi[]`, *void* `*data`), *void* `*data` (Input)

Function `fcn_fkforce` is used in case there is a non-zero term $\phi(f, x, t)$ in the Feynman-Kac differential equation. This function also accepts a data pointer to data supplied by the user. For an explanation of the other arguments of function `fcn_fkforce`, see optional argument `IMSL_FCN_FORCE`.

`IMSL_ATOL_RTOL_SCALARS`, *float* `atol`, *float* `rtol` (Input)

Scalar values that apply to the error estimates of all components of the solution `y` in the differential equation solver `SDASLX`. See optional argument `IMSL_ATOL_RTOL_ARRAYS` if separate tolerances are to be applied to each component of `y`.

Default: atol and rtol are set to $10^{-3}$ in single precision and $10^{-5}$ in double precision.

IMSL_ATOL_RTOL_ARRAYS, *float* atol[], *float* rtol[], (Input)

Componentwise tolerances are used for the computation of solution y in the differential equation solver SDASLX. Arguments atol and rtol are pointers to arrays of length $3 \times$ nxgrid to be used for the absolute and relative tolerance and to be applied to each component of the solution, y. See optional argument IMSL_ATOL_RTOL_SCALARS if scalar values of absolute and relative tolerances are to be applied to all components.

Default: All elements of atol and rtol are set to $10^{-3}$ in single precision and $10^{-5}$ in double precision.

IMSL_NDEGREE, *int* ndeg (Input)

The degree used for the Gauss-Legendre formulas for constructing the finite element matrices. It is required that ndeg ≥ 6.

Default: ndeg = 6.

IMSL_TDEPEND, *int* tdepend[] (Output)

Vector of length 7 indicating time dependence of the coefficients, boundary conditions and function $\phi$ in the Feynman-Kac equation. If tdepend[i] = 0 then argument i is not time dependent, if tdepend[i]=1 then argument i is time dependent.

| i | Computed coefficient |
|---|---|
| 0 | $\sigma'$ |
| 1 | $\sigma$ |
| 2 | $\mu$ |
| 3 | $\kappa$ |
| 4 | Left boundary conditions |
| 5 | Right boundary conditions |
| 6 | $\phi$ |

IMSL_MAX_STEP, *float* max_stepsize (Input)

This is the maximum step size the integrator may take.

Default: max_stepsize = imsl_f_machine(2), the largest possible machine number.

IMSL_INITIAL_STEPSIZE, *float* init_stepsize (Input)

The starting step size for the integration. Must be less than zero since the integration is internally done from t=0 to t=tgrid[ntgrid-1] in a negative direction.

Default: init_stepsize = $-\varepsilon$, where $\varepsilon$ is the machine precision.

`IMSL_MAX_NUMBER_STEPS`, *int* `max_steps` (Input)

The maximum number of steps between each output point of the integration.

Default: `maxsteps = 500000`.

`IMSL_STEP_CONTROL`, *int* `step_control` (Input)

Indicates which step control algorithm is used for the integration. If `step_control` = 0, then the step control method of Söderlind is used. If `step_control` = 1, then the method used by the original Petzold code `SASSL` is used.

Default: `step_control = 0.`

`IMSL_MAX_BDF_ORDER`, *int* `max_bdf_order` (Input)

Maximum order of the backward differentiation formulas used in the integrator. It is required that 1≤ `max_bdf_order` ≤ 5.

Default: `max_bdf_order = 5`.

`IMSL_T_BARRIER`, *float* `t_barrier` (Input)

This optional argument controls whether the code should integrate past a special point, `t_barrier`, and then interpolate to get $y$ and $y'$ at the points in `tgrid[]`. If this optional argument is present, the integrator assumes the equations either change on the alternate sides of `t_barrier` or they are undefined there. In this case, the code creeps up to `t_barrier` in the direction of integration. It is required that `t_barrier` ≥`tgrid[ntgrid-1].`

Default: `t_barrier = tgrid[ntgrid-1]`.

`IMSL_ISTATE`, *int* `istate[]` (Output)

An array of length 7 whose entries flag the state of computation for the matrices and vectors required in the integration. For each entry, a zero indicates that no computation has been done or there is a time dependence. A one indicates that the entry has been computed and there is no time dependence. The `istate` entries are as follows:

| **i** | **Entry in istate** |
|---|---|
| 0 | Mass Matrix, M |
| 1 | Stiffness matrix, N |
| 2 | Bending matrix, R |
| 3 | Weighted mass, K |
| 4 | Left boundary conditions |
| 5 | Right- boundary conditions |
| 6 | Forcing term |

Default: `istate[i]` = 0 for `I` = 0,...,6.

`IMSL_EVALS`, *int* `nval[]` (Output)

Array of length 3 summarizing the number of evaluations required during the integration.

| i | nval[i] |
|---|---|
| 0 | Number of residual function evaluations of the DAE used in the model. |
| 1 | Number of factorizations of the differential matrix associated with solving the DAE. |
| 2 | Number of linear system solve steps using the differential matrix. |

## Description

The generalized Feynman-Kac differential equation has the form

$$f_t + \mu(x,t) f_x + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t) f = \phi(f,x,t),$$

where the initial data satisfies $f(x,T) = p(x)$. The derivatives are

$$f_t = \frac{\partial f}{\partial t}$$

The function `imsl_f_feynman_kac` uses a finite element Galerkin method over the rectangle

$$\left[x_{\min}, x_{\max}\right] \times \left[\hat{T}, T\right]$$

in $(x,t)$ to compute the approximate solution. The interval $\left[x_{\min}, x_{\max}\right]$ is decomposed with a grid

$$\left(x_{\min} = \right) x_1 < x_2 < \ \ldots \ < x_m \left( = x_{\max}\right).$$

On each subinterval the solution is represented by

$$\begin{aligned} f(x,t) = f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f_i' b_1(z) - h_i f'_{i+1} b_1(1-z) \\ + h_i^2 f_i'' b_2(z) + h_i^2 f''_{i+1} b_2(1-z). \end{aligned}$$

The values

$$f_i, f'_i, f''_i, f_{i+1}, f'_{i+1}, f''_{i+1}$$

are time-dependent coefficients associated with each interval. The basis functions $b_0, b_1, b_2$ are given for

$$x \in \left[x_i, x_{i+1}\right], h_i = x_{i+1} - x_i, z = \left(x - x_i\right) / h_i, z \in [0, 1],$$

by

$$b_0(z) = -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3(6z^2 + 3z + 1),$$
$$b_1(z) = -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z+1),$$
$$b_2(z) = \tfrac{1}{2}(-z^5 + 3z^4 - 3z^3 + z^2) = \tfrac{1}{2}(1-z)^3 z^2.$$

The Galerkin principle is then applied. Using the provided initial and boundary conditions leads to an Index 1 differential-algebraic equation for the time-dependent coefficients

$$f_i, f'_i, f''_i, f_{i+1}, f'_{i+1}, f''_{i+1}\left[ = y_i, y_{i+1}, y_{i+2}, \dots \right],\ i = 1, \dots, m-1$$

This system is integrated using the variable order, variable step algorithm `DDASLX/SDASLX` noted in Hanson and Krogh, R. (2008) Solving Constrained Differential-Algebraic Systems Using Projections. Solution values and their time derivatives are returned at a grid preceding time *T*, expressed in units of time remaining. For further details of deriving and solving the system see Hanson, R. (2008) Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements. To evaluate *f* or its partial derivatives at any time point in the grid, use the function `imsl_f_feynman_kac_evaluate`.

# Examples

## Example 1

The value of the American Option on a Vanilla Put can be no smaller than its European counterpart. That is due to the American Option providing the opportunity to exercise at any time prior to expiration. This example compares this difference – or premium value of the American Option – at two time values using the Black-Scholes model. The example is based on Wilmott et al. (1996, p. 176), and uses the non-linear forcing or weighting term described in Hanson, R. (2008), Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements, for evaluating the price of the American Option. The coefficients, payoff, boundary conditions and forcing term for American and European options are defined by the user functions `fkcfiv_put`, `fkbcp_put` and `fkforce_put`, respectively. One breakpoint is set exactly at the strike price.

The sets of parameters in the computation are:

1. Strike price $K = \{10.0\}$.
2. Volatility $\sigma = \{0.4\}$.
3. Times until expiration = $\{1/4, 1/2\}$.
4. Interest rate $r = 0.1$.
5. $x_{\min} = 0.0, \quad x_{\max} = 30.0$.
6. $nx = 61, \quad n = 3 \times nx = 183$.

The payoff function is the "vanilla option", $p(x) = \max(K - x, 0)$.

```
#include <imsl.h>
#include <stdio.h>
```

```
#include <math.h>

#define max(A,B)  ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 9

void fkcfiv_put(float, float, int *, float *);
void fkbcp_put(int, float, int *, float[]);
void fkforce_put(int, int, int, float[], float, float, float[],
   float[], float[], float[],float[], void *);

int main()
{
   /* Compute American Option Premium for Vanilla Put */
   /* The strike price */
   float KS = 10.0;

   /* The sigma value */
   float sigma = 0.4;

   /* Time values for the options */
   int nt = 2;
   float time[] = { 0.25, 0.5};

   /* Values of the underlying where evaluations are made */
   float xs[] = { 0.0, 2.0, 4.0, 6.0, 8.0, 10.0,
       12.0, 14.0, 16.0 };

   /* Value of the interest rate */
   float r = 0.1;

   /* Values of the min and max underlying values modeled */
   float x_min=0.0, x_max=30.0;

   /* Define parameters for the integration step. */
   int nint=NXGRID-1, n=3*NXGRID;
   float xgrid[NXGRID];
   float ye[(NTGRID+1)*3*NXGRID],yeprime[(NTGRID+1)*3*NXGRID];
   float ya[(NTGRID+1)*3*NXGRID], yaprime[(NTGRID+1)*3*NXGRID];
   float fe[NTGRID*NV], fa[NTGRID*NV];
   float dx;
   int i;
   int nlbcd = 2, nrbcd = 3;
   float atol = 0.2e-2;

   /* Array for user-defined data */
   float usr_data[3];

   /* Define an equally-spaced grid of points for the
   underlying price */
```

```
   dx = (x_max-x_min)/((float) nint);
   xgrid[0]=x_min;
   xgrid[NXGRID-1]=x_max;

   for (i=2; i<=NXGRID-1; i++) xgrid[i-1]=xgrid[i-2]+dx;
   usr_data[0] = KS;
   usr_data[1] = r;
   usr_data[2] = atol;

   imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
       fkcfiv_put, fkbcp_put, ye, yeprime,
       IMSL_ATOL_RTOL_SCALARS, 0.5e-2, 0.5e-2,
       0);

   imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
       fkcfiv_put, fkbcp_put, ya, yaprime,
       IMSL_FCN_FORCE_W_DATA, fkforce_put, usr_data,
       IMSL_ATOL_RTOL_SCALARS, 0.5e-2, 0.5e-2,
       0);

   /* Evaluate solutions at vector of points XS(:), at each
   time value prior to expiration. */
   for (i=0; i<nt; i++)
   {
       imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &ye[(i+1)*n],
           IMSL_RETURN_USER, &fe[i*NV], 0);
       imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &ya[(i+1)*n],
           IMSL_RETURN_USER, &fa[i*NV], 0);
   }

   printf("\nAmerican Option Premium for Vanilla Put, "
       "3 and 6 Months Prior to Expiry\n");
   printf("%7sNumber of equally spaced spline knots:%4d\n", " ",
       NXGRID);
   printf("%7sNumber of unknowns:%4d\n", " ", n);
   printf("%7sStrike=%6.2f, sigma=%5.2f, Interest Rate=%5.2f\n\n",
       " ",KS,sigma,r);
   printf("%7s%10s%20s%20s\n", " ","Underlying","European","American");
   for (i=0; i<NV; i++)
       printf("%7s%10.4f%10.4f%10.4f%10.4f%10.4f\n", " ", xs[i], fe[i],
           fe[i+NV], fa[i], fa[i+NV]);
}

/* These functions define the coefficients, payoff, boundary conditions
and forcing term for American and European Options. */
void fkcfiv_put(float x, float tx, int *iflag, float *value)
{
   /* The sigma value */
   float sigma=0.4;

   /* Value of the interest rate and continuous dividend */
```

```
   float strike_price=10.0, interest_rate=0.1, dividend=0.0;
   float zero=0.0;

   switch (*iflag)
   {
   case 0:
       /* The payoff function */
       *value = max(strike_price - x, zero);
       break;
   case -1:
       /* The coefficient derivative d sigma/ dx */
       *value = sigma;
       break;
   case 1:
       /* The coefficient sigma(x) */
       *value = sigma*x;
       break;
   case 2:
       /* The coefficient mu(x) */
       *value = (interest_rate - dividend) * x;
       break;
   case 3:
       /* The coefficient kappa(x) */
       *value = interest_rate;
       break;
   }

   /* Note that there is no time dependence */
   *iflag = 0;
   return;
}

void fkbcp_put(int nbc, float tx, int *iflag, float val[])
{
   switch (*iflag)
   {
   case 1:
       val[0] = 0.0; val[1] = 1.0; val[2] = 0.0;
       val[3] = -1.0; val[4] = 0.0; val[5] = 0.0;
       val[6] = 1.0; val[7] = 0.0;
       break;
   case 2:
       val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
       val[3] = 0.0; val[4] = 0.0; val[5] = 1.0;
       val[6] = 0.0; val[7] = 0.0; val[8] = 0.0;
       val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
       break;
   }
   /* Note no time dependence */
   *iflag = 0;
   return;
```

```
}

void fkforce_put(int interval, int ndeg, int nxgrid,
   float y[], float time, float width, float xlocal[],
   float qw[], float u[], float phi[],float dphi[],
   void *data_ptr)
{
   int i, j, k, l;
   const int local=6;
   float yl[6], bf[6];
   float value, strike_price, interest_rate, zero=0.0, one=1.0;
   float rt, mu;
   float *data = NULL;

   data = (float *) data_ptr;

   for (i=0; i<local; i++)
   {
       yl[i] = y[3*interval-3+i];
       phi[i] = zero;
   }

   strike_price = data[0];
   interest_rate = data[1];
   value = data[2];
   mu=2.0;

   /* This is the local definition of the forcing term */
   for (j=1; j<=local; j++){
       for (l=1; l<=ndeg; l++)
       {
           bf[0] = u[(l-1)];
           bf[1] = u[(l-1)+ndeg];
           bf[2] = u[(l-1)+2*ndeg];
           bf[3] = u[(l-1)+6*ndeg];
           bf[4] = u[(l-1)+7*ndeg];
           bf[5] = u[(l-1)+8*ndeg];

           rt = 0.0;
           for (k=0; k<local; k++)
               rt += yl[k]*bf[k];
           rt = value/(rt + value - (strike_price-xlocal[l-1]));
           phi[j-1] += qw[l-1] * bf[j-1] * pow(rt,mu);
       }
   }

   for (i=0; i<local; i++)
       phi[i] = -phi[i]*width*interest_rate*strike_price;

   /* This is the local derivative matrix for the forcing term */
   for (i=0; i<local*local; i++)
```

```
        dphi[i] = zero;

    for (j=1; j<=local; j++){
        for (i=1; i<=local; i++){
            for (l=1; l<=ndeg; l++)
            {
                bf[0] = u[(l-1)];
                bf[1] = u[(l-1)+ndeg];
                bf[2] = u[(l-1)+2*ndeg];
                bf[3] = u[(l-1)+6*ndeg];
                bf[4] = u[(l-1)+7*ndeg];
                bf[5] = u[(l-1)+8*ndeg];

                rt = 0.0;
                for (k=0; k<local; k++)
                    rt += yl[k]*bf[k];
                rt = one/(rt + value-(strike_price-xlocal[l-1]));
                dphi[i-1+(j-1)*local] += qw[l-1] * bf[i-1] * bf[j-1]
                * pow(rt, mu+1.0);
            }
        }
    }

    for (i=0; i<local*local; i++)
        dphi[i] = mu * dphi[i] * width * pow(value, mu) *
        interest_rate * strike_price;
    return;
}
```

### Output

```
American Option Premium for Vanilla Put, 3 and 6 Months Prior to Expiry
      Number of equally spaced spline knots: 61
      Number of unknowns: 183
      Strike= 10.00, sigma= 0.40, Interest Rate= 0.10

      Underlying           European             American
          0.0000    9.7534   9.5136  10.0000  10.0000
          2.0000    7.7536   7.5138   8.0000   8.0000
          4.0000    5.7537   5.5156   6.0000   6.0000
          6.0000    3.7615   3.5683   4.0000   4.0000
          8.0000    1.9070   1.9168   2.0170   2.0865
         10.0000    0.6529   0.8548   0.6771   0.9030
         12.0000    0.1632   0.3371   0.1680   0.3519
         14.0000    0.0372   0.1270   0.0373   0.1321
         16.0000    0.0088   0.0483   0.0084   0.0501
```

## Example 2

In Beckers (1980) there is a model for a Stochastic Differential Equation of option pricing. The idea is a “constant elasticity of variance diffusion (or CEV) class”

$$dS = \mu S dt + \sigma S^{\alpha/2} dW, \quad 0 \le \alpha < 2$$

The Black-Scholes model is the limiting case $\alpha \to 2$. A numerical solution of this diffusion model yields the price of a call option. Various values of the strike price $K$, time values, $\sigma$ and power coefficient $\alpha$ are used to evaluate the option price at values of the underlying price. The sets of parameters in the computation are:

1. power $\alpha = \{2.0, 1.0, 0.0\}$.
2. strike price $K = \{15.0, 20.0, 25.0\}$.
3. volatility $\sigma = \{0.2, 0.3, 0.4\}$.
4. times until expiration $= \{1/12, 4/12, 7/12\}$.
5. underlying prices $= \{19.0, 20.0, 21.0\}$.
6. interest rate $r = 0.05$.
7. $x_{\min} = 0, x_{\max} = 60$.
8. $nx = 121, \quad n = 3 \times nx = 363$.

With this model the Feynman-Kac differential equation is defined by identifying:

$$x: \quad S$$

$$\sigma(x,t): \quad \sigma x^{\alpha/2}; \quad \frac{\partial \sigma}{\partial x} = \frac{a\sigma}{2} x^{\alpha/2-1}$$

$$\mu(x,t): \quad rx$$

$$\kappa(x,t): \quad r$$

$$\phi(f,x,t) \equiv 0$$

The payoff function is the "vanilla option", $p(x) = \max(x-K,0)$.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define max(A,B)  ((A) >= (B) ? (A) : (B))
#define NXGRID 121
#define NTGRID 3
#define NV 3

void fcn_fkcfiv(float, float, int *, float *, void *);
void fcn_fkbcp(int, float, int *, float[], void *);

int main()
{
```

```
/* Compute Constant Elasticity of Variance Model for Vanilla Call */

/* The set of strike prices */
float KS[] = { 15.0, 20.0, 25.0};

/* The set of sigma values */
float sigma[] = { 0.2, 0.3, 0.4};

/* The set of model diffusion powers */
float alpha[] = { 2.0, 1.0, 0.0};

/* Time values for the options */
int nt = 3;
float time[] = { 1.0/12.0, 4.0/12.0, 7.0/12.0 };

/* Values of the underlying where evaluations are made */
float xs[] = { 19.0, 20.0, 21.0};

/* Value of the interest rate and continuous dividend */
float r=0.05, dividend=0.0;

/* Values of the min and max underlying values modeled */
float x_min=0.0, x_max=60.0;

/* Define parameters for the integration step. */
int nint = NXGRID-1, n=3*NXGRID;
float xgrid[NXGRID], y[(NTGRID+1)*3*NXGRID];
float yprime[(NTGRID+1)*3*NXGRID], f[NTGRID*NV];
float dx;

/* Number of left/right boundary conditions */
int nlbcd = 3, nrbcd = 3;

float usr_data[6];

int i,i1,i2,i3,j;

/* Define equally-spaced grid of points for the underlying price */

dx = (x_max-x_min)/((float) nint);
xgrid[0] = x_min;
xgrid[NXGRID-1] = x_max;

for (i=2; i<=NXGRID-1; i++)
    xgrid[i-1] = xgrid[i-2]+dx;

printf("%2sConstant Elasticity of Variance Model for Vanilla Call\n",
       " ");
printf("%7sInterest Rate:%7.3f\tContinuous Dividend:%7.3f\n",
       " ", r, dividend);
printf("%7sMinimum and Maximum Prices of Underlying:%7.2f%7.2f\n",
```

```
           " ",x_min, x_max);
 printf("%7sNumber of equally spaced spline knots:%4d \n", " ",
        NXGRID-1);
 printf("%7sNumber of unknowns:%4d\n\n", " ",n);
 printf("%7sTime in Years Prior to Expiration:%7.4f%7.4f%7.4f\n",
        " ",time[0], time[1], time[2]);
 printf("%7sOption valued at Underlying Prices:%7.2f%7.2f%7.2f\n\n",
        " ", xs[0], xs[1], xs[2]);

 for (i1=1; i1<=3; i1++)       /* Loop over power */
   for (i2=1; i2<=3; i2++)     /* Loop over volatility */
     for (i3=1; i3<=3; i3++)  /* Loop over strike price */
     {
     /* Pass data through into evaluation functions. */
        usr_data[0] = KS[i3-1];
        usr_data[1] = x_max;
        usr_data[2] = sigma[i2-1];
        usr_data[3] = alpha[i1-1];
        usr_data[4] = r;
        usr_data[5] = dividend;

        imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
                   NULL, NULL, y, yprime,
                       IMSL_FCN_FKCFIV_W_DATA, fcn_fkcfiv, usr_data,
                       IMSL_FCN_FKBCP_W_DATA, fcn_fkbcp, usr_data, 0);

        /* Evaluate solution at vector of points xs, at each time
           Value prior to expiration. */
        for (i=0; i<NTGRID; i++)
           imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs,
                        &y[(i+1)*n], IMSL_RETURN_USER, &f[i*NV], 0);

        printf("%2sStrike=%5.2f, Sigma=%5.2f, Alpha=%5.2f\n",
                        " ", KS[i3-1], sigma[i2-1], alpha[i1-1]);

        for (i=0; i<NV; i++)
        {
          printf("%23sCall Option Values%2s", " ", " ");
          for (j=0; j<nt; j++) printf("%7.4f ", f[j*NV+i]);
                       printf("\n");
        }
        printf("\n");
     }
}

void fcn_fkcfiv(float x, float tx, int *iflag, float *value,
      void *data_ptr)
{
 float sigma, strike_price, interest_rate;
 float alpha, dividend, zero=0.0, half=0.5;
 float *data = NULL;
```

```
 data = (float *)data_ptr;

 strike_price = data[0];
 sigma = data[2];
 alpha = data[3];
 interest_rate = data[4];
 dividend = data[5];

 switch (*iflag)
 {
   case 0:
     /* The payoff function */
     *value = max(x - strike_price, zero);
   break;

   case -1:
     /* The coefficient derivative d sigma/ dx */
     *value = half * alpha * sigma * pow(x, alpha*half-1.0);
   break;

   case 1:
     /* The coefficient sigma(x) */
     *value = sigma * pow(x, alpha*half);
   break;

   case 2:
     /* The coefficient mu(x) */
     *value = (interest_rate - dividend) * x;
   break;

   case 3:
     /* The coefficient kappa(x) */
     *value = interest_rate;
   break;
 }

 data = NULL;

 /* Note that there is no time dependence */
 *iflag = 0;

 return;
}


void fcn_fkbcp(int nbc, float tx, int *iflag, float val[],
              void *data_ptr)
{
 float x_max, df, interest_rate, strike_price;
 float *data = NULL;
```

```
 data = (float *)data_ptr;

 strike_price = data[0];
 x_max = data[1];
 interest_rate = data[4];

 switch (*iflag)
 {
   case 1:
     val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
     val[3] = 0.0; val[4] = 0.0; val[5] = 1.0;
     val[6] = 0.0; val[7] = 0.0; val[8] = 0.0;
     val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;

     /* Note no time dependence at left end */
     *iflag = 0;
   break;

   case 2:
     df = exp(interest_rate*tx);
     val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
     val[3] = x_max - df*strike_price; val[4] = 0.0;
     val[5] = 1.0; val[6] = 0.0; val[7] = 1.0;
     val[8] = 0.0; val[9] = 0.0; val[10] = 1.0;
     val[11] = 0.0;
   break;
 }

 data = NULL;
 return;
}
```

## Output

```
 Constant Elasticity of Variance Model for Vanilla Call
      Interest Rate: 0.050  Continuous Dividend: 0.000
      Minimum and Maximum Prices of Underlying:  0.00 60.00
      Number of equally spaced spline knots: 120
      Number of unknowns: 363

      Time in Years Prior to Expiration: 0.0833 0.3333 0.5833
      Option valued at Underlying Prices: 19.00 20.00 21.00

 Strike=15.00, Sigma= 0.20, Alpha= 2.00
                      Call Option Values  4.0624  4.2577  4.4730
                      Call Option Values  5.0624  5.2507  5.4491
                      Call Option Values  6.0624  6.2487  6.4386

 Strike=20.00, Sigma= 0.20, Alpha= 2.00
                      Call Option Values  0.1310  0.5956  0.9699
```

```
                    Call Option Values  0.5028  1.0889  1.5100
                    Call Option Values  1.1979  1.7485  2.1751

Strike=25.00, Sigma= 0.20, Alpha= 2.00
                    Call Option Values  0.0000  0.0113  0.0742
                    Call Option Values  0.0000  0.0372  0.1614
                    Call Option Values  0.0007  0.1025  0.3132

Strike=15.00, Sigma= 0.30, Alpha= 2.00
                    Call Option Values  4.0637  4.3399  4.6623
                    Call Option Values  5.0626  5.2945  5.5788
                    Call Option Values  6.0624  6.2709  6.5241

Strike=20.00, Sigma= 0.30, Alpha= 2.00
                    Call Option Values  0.3103  1.0274  1.5500
                    Call Option Values  0.7315  1.5422  2.1024
                    Call Option Values  1.3758  2.1689  2.7385

Strike=25.00, Sigma= 0.30, Alpha= 2.00
                    Call Option Values  0.0006  0.1112  0.3547
                    Call Option Values  0.0038  0.2170  0.5552
                    Call Option Values  0.0184  0.3857  0.8225

Strike=15.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values  4.0759  4.5136  4.9673
                    Call Option Values  5.0664  5.4199  5.8321
                    Call Option Values  6.0635  6.3577  6.7294

Strike=20.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values  0.5116  1.4645  2.1286
                    Call Option Values  0.9623  1.9957  2.6944
                    Call Option Values  1.5815  2.6110  3.3230

Strike=25.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values  0.0083  0.3288  0.7790
                    Call Option Values  0.0285  0.5169  1.0657
                    Call Option Values  0.0813  0.7688  1.4103

Strike=15.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values  4.0624  4.2479  4.4311
                    Call Option Values  5.0624  5.2479  5.4311
                    Call Option Values  6.0624  6.2479  6.4311

Strike=20.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values  0.0000  0.0241  0.1061
                    Call Option Values  0.1498  0.4102  0.6483
                    Call Option Values  1.0832  1.3313  1.5772

Strike=25.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values  0.0000 -0.0000  0.0000
                    Call Option Values  0.0000  0.0000  0.0000
```

```
                    Call Option Values -0.0000  0.0000  0.0000

Strike=15.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values  4.0624  4.2477  4.4310
                    Call Option Values  5.0624  5.2477  5.4310
                    Call Option Values  6.0624  6.2477  6.4310

Strike=20.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values  0.0016  0.0812  0.2214
                    Call Option Values  0.1981  0.4981  0.7535
                    Call Option Values  1.0836  1.3441  1.6018

Strike=25.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values -0.0000  0.0000  0.0000
                    Call Option Values -0.0000  0.0000  0.0000
                    Call Option Values -0.0000  0.0000  0.0005

Strike=15.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values  4.0624  4.2479  4.4312
                    Call Option Values  5.0624  5.2479  5.4312
                    Call Option Values  6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values  0.0072  0.1556  0.3445
                    Call Option Values  0.2501  0.5919  0.8720
                    Call Option Values  1.0867  1.3783  1.6577

Strike=25.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values -0.0000  0.0000  0.0001
                    Call Option Values  0.0000  0.0000  0.0007
                    Call Option Values  0.0000  0.0002  0.0059

Strike=15.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values  4.0625  4.2479  4.4311
                    Call Option Values  5.0625  5.2479  5.4312
                    Call Option Values  6.0623  6.2479  6.4312

Strike=20.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values  0.0001  0.0001  0.0002
                    Call Option Values  0.0816  0.3316  0.5748
                    Call Option Values  1.0818  1.3308  1.5748

Strike=25.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values  0.0000 -0.0000 -0.0000
                    Call Option Values  0.0000 -0.0000 -0.0000
                    Call Option Values -0.0000  0.0000 -0.0000

Strike=15.00, Sigma= 0.30, Alpha= 0.00
                    Call Option Values  4.0624  4.2479  4.4311
                    Call Option Values  5.0625  5.2479  5.4311
                    Call Option Values  6.0623  6.2479  6.4311
```

```
Strike=20.00, Sigma= 0.30, Alpha= 0.00
                     Call Option Values  0.0000 -0.0000  0.0029
                     Call Option Values  0.0895  0.3326  0.5753
                     Call Option Values  1.0826  1.3306  1.5749

Strike=25.00, Sigma= 0.30, Alpha= 0.00
                     Call Option Values  0.0000 -0.0000 -0.0000
                     Call Option Values  0.0000 -0.0000 -0.0000
                     Call Option Values  0.0000 -0.0000 -0.0000

Strike=15.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values  4.0624  4.2479  4.4312
                     Call Option Values  5.0624  5.2479  5.4312
                     Call Option Values  6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values -0.0000  0.0001  0.0111
                     Call Option Values  0.0985  0.3383  0.5781
                     Call Option Values  1.0830  1.3306  1.5749

Strike=25.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values  0.0000  0.0000 -0.0000
                     Call Option Values  0.0000 -0.0000 -0.0000
                     Call Option Values  0.0000 -0.0000 -0.0000
```

## Example 3

This example evaluates the price of a European Option using two payoff strategies: ***Cash-or-Nothing*** and ***Vertical Spread***. In the first case the payoff function is

$$p(x) = \begin{cases} 0, & x \le K \\ B, & x > K \end{cases}$$

The value $B$ is regarded as the bet on the asset price, see Wilmott et al. (1995, p. 39-40). The second case has the payoff function

$$p(x) = \max(x - K_1) - \max(x - K_2), \quad K_2 > K_1$$

Both problems use the same boundary conditions. Each case requires a separate integration of the Black-Scholes differential equation, but only the payoff function evaluation differs in each case. The sets of parameters in the computation are:

1. Strike and bet prices $K_1$={10.0}, $K_2$ = {15.0}, and $B$ = {2.0}.
2. Volatility $\sigma$ = {0.4}.
3. Times until expiration = {1/4, 1/2}.
4. Interest rate $r$ = 0.1.
5. $x_{min}$ = 0, $x_{max}$ = 30.

6. $nx$ =61, $n = 3 \times nx = 183$.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define max(A,B)  ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 12

void fkcfiv_call(float, float, int *, float *, void *);
void fkbcp_call(int, float, int *, float[], void *);

int main()
{
 int i;
 /* The strike price */
 float KS1 = 10.0;
 /* The spread value */
 float KS2 = 15.0;
 /* The Bet for the Cash-or-Nothing Call */
 float bet = 2.0;
 /* The sigma value */
 float sigma = 0.4;

 /* Time values for the options */
 int nt = 2;
 float time[] = {0.25, 0.5};

 /* Values of the underlying where evaluations are made */
 float xs[NV];

 /* Value of the interest rate and continuous dividend */
 float r = 0.1, dividend=0.0;

 /* Values of the min and max underlying values modeled */
 float x_min=0.0, x_max=30.0;

 /* Define parameters for the integration step. */
 int nint=NXGRID-1, n=3*NXGRID;

 float xgrid[NXGRID];
 float yb[(NTGRID+1)*3*NXGRID], ybprime[(NTGRID+1)*3*NXGRID];
 float yv[(NTGRID+1)*3*NXGRID], yvprime[(NTGRID+1)*3*NXGRID];
 float fb[NTGRID*NV], fv[NTGRID*NV];
 float dx;

 /* Number of left/right boundary conditions. */
 int nlbcd = 3, nrbcd = 3;
```

```
/* Structure for the evaluation functions. */
struct {
     int idope[1];
     float rdope[7];
} usr_data;

/* Define an equally-spaced grid of points for the underlying
   price */
dx = (x_max-x_min)/((float)(nint));
xgrid[0]=x_min;
xgrid[NXGRID-1]=x_max;
for (i=2; i<=NXGRID-1; i++)
   xgrid[i-1] = xgrid[i-2]+dx;

for (i=1; i<=NV; i++)
   xs[i-1] = 2.0+(i-1)*2.0;

usr_data.rdope[0] = KS1;
usr_data.rdope[1] = bet;
usr_data.rdope[2] = KS2;
usr_data.rdope[3] = x_max;
usr_data.rdope[4] = sigma;
usr_data.rdope[5] = r;
usr_data.rdope[6] = dividend;

/* Flag the difference in payoff functions */
/* 1 for the Bet, 2 for the Vertical Spread */

usr_data.idope[0] = 1;

imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
               NULL, NULL, yb, ybprime,
               IMSL_FCN_FKCFIV_W_DATA, fkcfiv_call, &usr_data,
               IMSL_FCN_FKBCP_W_DATA, fkbcp_call, &usr_data,
               0);

usr_data.idope[0] = 2;

imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
               NULL, NULL, yv, yvprime,
               IMSL_FCN_FKCFIV_W_DATA, fkcfiv_call, &usr_data,
               IMSL_FCN_FKBCP_W_DATA, fkbcp_call, &usr_data,
           0);

/* Evaluate solutions at vector of points XS(:), at each time value
   prior to expiration. */

for (i=0; i<NTGRID; i++)
{
   imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &yb[(i+1)*n],
                     IMSL_RETURN_USER, &fb[i*NV], 0);
```

```
    imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &yv[(i+1)*n],
                               IMSL_RETURN_USER, &fv[i*NV], 0);
 }

 printf("%2sEuropean Option Value for A Bet\n", " ");
 printf("%3sand a Vertical Spread, 3 and 6 Months Prior to Expiry\n",
        " ");
 printf("%5sNumber of equally spaced spline knots:%4d\n", " ",
        NXGRID);
 printf("%5sNumber of unknowns:%4d\n", " ", n);
 printf("%5sStrike=%5.2f, Sigma=%5.2f, Interest Rate=%5.2f\n",
        " ", KS1, sigma, r);
 printf("%5sBet=%5.2f, Spread Value=%5.2f\n\n", " ", bet, KS2);
 printf("%17s%18s%18s\n", "Underlying", "A Bet", "Vertical Spread");
 for (i=0; i<NV; i++)
   printf("%8s%9.4f%9.4f%9.4f%9.4f%9.4f\n", " ", xs[i], fb[i],
          fb[i+NV], fv[i], fv[i+NV]);
}


/* These functions define the coefficients, payoff, boundary conditions
        and forcing term for American and European Options. */

void fkcfiv_call(float x, float tx, int *iflag, float *value,
       void *data_ptr)
{
 float sigma, strike_price, interest_rate;
 float spread, bet, dividend, zero=0.0;
 float *data_real = NULL;
 int *data_int = NULL;

 struct struct_data {
       int idope[1];
        float rdope[7];
 };

 struct struct_data *data = NULL;

 data = data_ptr;

 data_real = data->rdope;
 data_int = data->idope;

 strike_price = data_real[0];
 bet = data_real[1];
 spread = data_real[2];
 sigma = data_real[4];
 interest_rate = data_real[5];
 dividend = data_real[6];

 switch (*iflag)
```

```
  {
    case 0:
      /* The payoff function - Use flag passed to decide which */
      switch (data_int[0])
      {
        case 1:
          /* After reaching the strike price the payoff jumps
             from zero to the bet value. */
          *value = zero;
          if (x > strike_price) *value = bet;
        break;
        case 2:
        /* Function is zero up to strike price.
             Then linear between strike price and spread.
             Then has constant value Spread-Strike Price after
             the value Spread. */
          *value = max(x-strike_price, zero)-max(x-spread, zero);
        break;
      }
    break;

    case -1:
      /* The coefficient derivative d sigma/ dx */
      *value = sigma;
    break;

    case 1:
      /* The coefficient sigma(x) */
      *value = sigma*x;
    break;

    case 2:
      /* The coefficient mu(x) */
      *value = (interest_rate - dividend)*x;
    break;

    case 3:
      /* The coefficient kappa(x) */
      *value = interest_rate;
    break;
  }
  /* Note that there is no time dependence */
  *iflag = 0;

  data_real = NULL;
  data_int = NULL;
  data = NULL;

  return;
}
```

```
void fkbcp_call(int nbc, float tx, int *iflag, float val[],
                void *data_ptr)
{
  float strike_price, spread, bet, interest_rate, df;
  int *data_int = NULL;
  float *data_real = NULL;

  struct struct_data {
          int idope[1];
          float rdope[7];
  };

  struct struct_data *data = NULL;

  data = data_ptr;

  data_int = data->idope;
  data_real = data->rdope;
  strike_price = data_real[0];
  bet = data_real[1];
  spread = data_real[2];
  interest_rate = data_real[5];

  switch (*iflag)
  {
     case 1:
       val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
       val[3] = 0.0; val[4] = 0.0; val[5] = 1.0;
       val[6] = 0.0; val[7] = 0.0; val[8] = 0.0;
       val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
    /* Note no time dependence in case (1) for IFLAG */
       *iflag = 0;
     break;

     case 2:
       /* This is the discount factor using the risk-free
          interest rate */
       df = exp(interest_rate*tx);
       /* Use flag passed to decide on boundary condition */
       switch (data_int[0])
       {
          case 1:
            val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
            val[3] = bet*df;
          break;
          case 2:
    val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
    val[3] = (spread-strike_price)*df;
     break;
       }
```

```
        val[4] = 0.0; val[5] = 1.0; val[6] = 0.0;
        val[7] = 0.0; val[8] = 0.0; val[9] = 0.0;
        val[10] = 1.0; val[11] = 0.0;
      break;
   }

   data_real = NULL;
   data_int = NULL;
   data = NULL;

   return;
}
```

## Output

```
 European Option Value for A Bet
  and a Vertical Spread, 3 and 6 Months Prior to Expiry
    Number of equally spaced spline knots: 61
    Number of unknowns: 183
    Strike=10.00, Sigma= 0.40, Interest Rate= 0.10
    Bet= 2.00, Spread Value=15.00

      Underlying            A Bet  Vertical Spread
          2.0000  0.0000  0.0000  0.0000  0.0000
          4.0000  0.0000  0.0014  0.0000  0.0006
          6.0000  0.0110  0.0722  0.0039  0.0446
          8.0000  0.2690  0.4304  0.1479  0.3831
         10.0000  0.9948  0.9781  0.8909  1.1927
         12.0000  1.6095  1.4287  2.1911  2.2274
         14.0000  1.8654  1.6924  3.4255  3.1551
         16.0000  1.9337  1.8177  4.2264  3.8263
         18.0000  1.9476  1.8700  4.6264  4.2492
         20.0000  1.9501  1.8903  4.7911  4.4922
         22.0000  1.9505  1.8979  4.8497  4.6232
         24.0000  1.9506  1.9007  4.8685  4.6909
```

## Example 4

This example evaluates the price of a convertible bond. Here, convertibility means that the bond may, at any time of the holder's choosing, be converted to a multiple of the specified asset. Thus a convertible bond with price $x$ returns an amount $K$ at time $T$ *unless* the owner has converted the bond to $\nu x, \nu \geq 1,$ units of the asset at some time prior to $T$. This definition, the differential equation and boundary conditions are given in Chapter 18 of Wilmott et al. (1996). Using a constant interest rate and volatility factor, the parameters and boundary conditions are:

1. Bond face value $K = \{1\}$, conversion factor $\nu = 1.125$
2. Volatility $\sigma = \{0.25\}$
3. Times until expiration = $\{1/2, 1\}$

4. Interest rate $r = 0.1$ , dividend $D = 0.02$

5. $x_{\min} = 0, \quad x_{\max} = 4$

6. $nx = 61, \quad n = 3 \times nx = 183$

7. Boundary conditions $f(0,t) = K\exp(-r(T-t)), f(x_{\max},t) = vx_{\max}$

8. Terminal data $f(x, T) = \max(K, vx)$

9. Constraint for bond holder $f(x, t) \geq vx$

Note that the error tolerance is set to a pure absolute error of value $10^{-3}$ . The free boundary constraint $f(x, t) \geq vx$ is achieved by use of a non-linear forcing term in the function `fkforce_cbond`. The terminal conditions are provided with the user function `fkinit_cbond`.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define max(A,B)  ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 13

void fkcfiv_cbond(float, float, int *, float *, void *);
void fkbcp_cbond(int, float, int *, float[], void *);
void fkforce_cbond(int, int, int, float[], float, float,
               float[], float[], float[], float[], float[], void *);
void fkinit_cbond(int, int, float[], float[], float,
                  float[], float[], float[], float[], void *);

int main()
{
 int i;
 /* Compute value of a Convertible Bond */

 /* The face value */
 float KS = 1.0e0;

 /* The sigma or volatility value */
 float sigma = 0.25e0;

 /* Time values for the options */
 float time[] = { 0.5, 1.0};

 /* Values of the underlying where evaluation are made */
 float xs[NV];

 /* Value of the interest rate, continuous dividend and factor */
 float r = 0.1, dividend=0.02, factor = 1.125;
```

```
/* Values of the min and max underlying values modeled */
float x_min = 0.0, x_max = 4.0;

/* Define parameters for the integration step. */
int nint = NXGRID-1, n=3*NXGRID;

float xgrid[NXGRID];
float y[(NTGRID+1)*3*NXGRID], yprime[(NTGRID+1)*3*NXGRID];
float f[(NTGRID+1)*NV], dx;

/* Array for user-defined data */
float usr_data[8];
float atol;

/* Number of left/right boundary conditions. */
int nlbcd = 3, nrbcd = 3;

/*
 * Define an equally-spaced grid of points for the
 * underlying price
 */
dx = (x_max-x_min)/((float) nint);
xgrid[0] = x_min;
xgrid[NXGRID-1] = x_max;

for (i=2; i<=NXGRID-1; i++) xgrid[i-1] = xgrid[i-2] + dx;
for (i=1; i<=NV; i++) xs[i-1] = (i-1)*0.25;

/* Pass the data for evaluation */
usr_data[0] = KS;
usr_data[1] = x_max;
usr_data[2] = sigma;
usr_data[3] = r;
usr_data[4] = dividend;
usr_data[5] = factor;

/* Use a pure absolute error tolerance for the integration */
atol = 1.0e-3;
usr_data[6] = atol;

/* Compute value of convertible bond */
imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
          NULL, NULL, y, yprime,
          IMSL_FCN_FKCFIV_W_DATA, fkcfiv_cbond, usr_data,
          IMSL_FCN_FKBCP_W_DATA, fkbcp_cbond, usr_data,
          IMSL_FCN_INIT_W_DATA, fkinit_cbond, usr_data,
          IMSL_FCN_FORCE_W_DATA, fkforce_cbond, usr_data,
          IMSL_ATOL_RTOL_SCALARS, 1.0e-3, 0.0e0,
          0);
```

```
  /*
   * Evaluate and display solutions at vector of points XS(:),
   * at each time value prior to expiration.
   */

   for (i=0; i<=NTGRID; i++)
       imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &y[i*n],
                         IMSL_RETURN_USER, &f[i*NV], 0);

   printf("%2sConvertible Bond Value, 0+, 6 and 12 Months Prior "
          "to Expiry\n", " ");
   printf("%5sNumber of equally spaced spline knots:%4d\n", " ",
          NXGRID);
   printf("%5sNumber of unknowns:%4d\n", " ",n);
   printf("%5sStrike=%5.2f, Sigma=%5.2f\n", " ", KS, sigma);
   printf("%5sInterest Rate=%5.2f, Dividend=%5.2f, Factor=%6.3f\n\n",
          " ", r, dividend, factor);
   printf("%15s%18s\n", "Underlying", "Bond Value");

   for (i=0; i<NV; i++)
      printf("%7s%8.4f%8.4f%8.4f%8.4f\n",
        " ", xs[i], f[i], f[i+NV], f[i+2*NV]);
}

/*
* These functions define the coefficients, payoff, boundary conditions
* and forcing term.
*/

void fkcfiv_cbond(float x, float tx, int *iflag, float *value,
          void *data_ptr)
{
  float sigma, strike_price, interest_rate;
  float dividend, factor, zero=0.0;
  float *data = NULL;

  data = (float *) data_ptr;

  strike_price = data[0];
  sigma = data[2];
  interest_rate = data[3];
  dividend = data[4];
  factor = data[5];

  switch(*iflag)
  {
     case 0:
       /* The payoff function - */
       *value = max(factor * x, strike_price);
     break;
     case -1:
```

```
        /* The coefficient derivative d sigma/ dx */
        *value = sigma;
      break;
      case 1:
        /* The coefficient sigma(x) */
        *value = sigma*x;
      break;
      case 2:
        /* The coefficient mu(x) */
        *value = (interest_rate - dividend) * x;
      break;
      case 3:
        /* The coefficient kappa(x) */
        *value = interest_rate;
      break;
  }
  /* Note that there is no time dependence */
  *iflag = 0;
}

void fkbcp_cbond(int nbc, float tx, int *iflag, float val[],
              void *data_ptr)
{
  float interest_rate, strike_price, dp, factor, x_max;
  float *data = NULL;

  data = (float *) data_ptr;
  switch (*iflag)
  {
      case 1:
        strike_price = data[0];
        interest_rate = data[3];
        dp = strike_price * exp(tx*interest_rate);
        val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
        val[3] = dp; val[4] = 0.0; val[5] = 1.0;
        val[6] = 0.0; val[7] = 0.0; val[8] = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
      break;

      case 2:
        x_max = data[1];
        factor = data[5];
        val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
        val[3] = factor*x_max; val[4] = 0.0; val[5] = 1.0;
        val[6] = 0.0; val[7] = factor; val[8] = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
        /* Note no time dependence */
        *iflag = 0;
      break;
  }
```

```
  return;
}

void fkforce_cbond(int interval, int ndeg, int nxgrid, float y[],
             float time, float width, float xlocal[], float qw[],
             float u[], float phi[], float dphi[], void *data_ptr)
{
  int i, j, k, l;
  const int local=6;

  float yl[6], bf[6];
  float value, strike_price, interest_rate, zero=0.e0;
  float one=1.0e0, rt, mu, factor;

  float *data = NULL;

  data = (float *) data_ptr;

  for (i=0; i<local; i++)
  {
     yl[i] = y[3*interval-3+i];
     phi[i] = zero;
  }

  for (i=0; i<local*local; i++)
     dphi[i] = zero;

  value = data[6];
  strike_price = data[0];
  interest_rate = data[3];
  factor = data[5];

  mu = 2.0;

  /*
   * This is the local definition of the forcing term -
   * It "forces" the constraint f >= factor*x.
   */
  for (j=1; j<=local; j++)
    for (l=1; l<=ndeg; l++)
    {
       bf[0] = u[(l-1)];
       bf[1] = u[(l-1)+ndeg];
       bf[2] = u[(l-1)+2*ndeg];
       bf[3] = u[(l-1)+6*ndeg];
       bf[4] = u[(l-1)+7*ndeg];
       bf[5] = u[(l-1)+8*ndeg];

       rt = 0.0;
       for (k=0; k<local; k++)
             rt += yl[k]*bf[k];
```

```
        rt = value/(rt + value - factor * xlocal[l-1]);
        phi[j-1] += qw[l-1] * bf[j-1] * pow(rt,mu);
     }

     for (i=0; i<local; i++)
       phi[i] = -phi[i]*width*factor*strike_price;

  /*
   * This is the local derivative matrix for the forcing term
   */
   for (j=1; j<=local; j++)
     for (i=1; i<=local; i++)
        for (l=1; l<=ndeg; l++)
        {
           bf[0] = u[(l-1)];
           bf[1] = u[(l-1)+ndeg];
           bf[2] = u[(l-1)+2*ndeg];
           bf[3] = u[(l-1)+6*ndeg];
           bf[4] = u[(l-1)+7*ndeg];
           bf[5] = u[(l-1)+8*ndeg];

           rt = 0.0;
           for (k=0; k<local; k++) rt += yl[k]*bf[k];

           rt = one/(rt + value - factor * xlocal[l-1]);
           dphi[i-1+(j-1)*local] += qw[l-1] * bf[i-1] *
                             bf[j-1] * pow(value*rt, mu) * rt;
         }

   for (i=0; i<local*local; i++)
        dphi[i] = -mu * dphi[i] * width * factor * strike_price;

   return;
}

void fkinit_cbond(int nxgrid, int ntgrid, float xgrid[], float tgrid[],
                 float time, float yprime[], float y[], float atol[],
                 float rtol[], void *data_ptr)
{
  int i;
  float *data = NULL;

  data = (float *) data_ptr;

  if (time == 0.0)
  {
    /* Set initial data precisely. */
    for (i=1; i<=nxgrid; i++)
    {
       if (xgrid[i-1] * data[5] < data[0])
```

```
      {
         y[3*i-3] = data[0];
         y[3*i-2] = 0.0;
         y[3*i-1] = 0.0;
      }
      else
      {
         y[3*i-3] = xgrid[i-1] * data[5];
         y[3*i-2] = data[5];
         y[3*i-1] = 0.0;
      }
   }
 }
 return;
}
```

### Output

```
 Convertible Bond Value, 0+, 6 and 12 Months Prior to Expiry
    Number of equally spaced spline knots: 61
    Number of unknowns: 183
    Strike= 1.00, Sigma= 0.25
    Interest Rate= 0.10, Dividend= 0.02, Factor= 1.125

    Underlying       Bond Value
        0.0000 1.0000 0.9512 0.9048
        0.2500 1.0000 0.9512 0.9049
        0.5000 1.0000 0.9513 0.9065
        0.7500 1.0000 0.9737 0.9605
        1.0000 1.1250 1.1416 1.1464
        1.2500 1.4063 1.4117 1.4121
        1.5000 1.6875 1.6922 1.6922
        1.7500 1.9688 1.9731 1.9731
        2.0000 2.2500 2.2540 2.2540
        2.2500 2.5312 2.5349 2.5349
        2.5000 2.8125 2.8160 2.8160
        2.7500 3.0938 3.0970 3.0970
        3.0000 3.3750 3.3781 3.3781
```

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm. User flag = "#". |

# feynman_kac_evaluate

Computes the value of a Hermite quintic spline or the value of one of its derivatives. In particular, computes solutions to the Feynman-Kac PDE handled by function `imsl_f_feynman_kac`.

## Synopsis

*#include* `<imsl.h>`

*float* `*imsl_f_feynman_kac_evaluate` (*int* `nw`, *int* `m`, *float* `breakpoints[]`, *float* `w[]`, *float* `coef[]`, …, 0)

The type *double* function is `imsl_d_feynman_kac_evaluate`.

## Required Arguments

*int* `nw` (Input)
Length of the array containing the evaluation points of the spline.

*int* `m` (Input)
Number of breakpoints for the Hermite quintic spline interpolation. It is required that `m` ≥ 2. When applied to `imsl_f_feynman_kac`, `m` is identical to argument `nxgrid`.

*float* `breakpoints[]` (Input)
Array of length `m` containing the breakpoints for the Hermite quintic spline interpolation. The breakpoints must be in strictly increasing order. When applied to `imsl_f_feynman_kac`, `breakpoints[]` is identical to array `xgrid[]`.

*float* `w[]` (Input)
Vector of length `nw` containing the evaluation points for the spline. It is required that `breakpoints[0]` ≤ `w[i]` ≤ `breakpoints[m-1]` for `i`=0,…,`nw`-1.

*float* `coef[]` (Input)
Vector of length `3*m` containing the coefficients of the Hermite quintic spline.
When applied to `imsl_f_feynman_kac`, this vector is one of the rows of output arrays `y` or `y_prime` related to the spline coefficients at time points `t=tgrid[j]`, `j=1`, …,`ntgrid`.

## Return Value

A pointer to an array of length `nw` containing the values of the Hermite quintic spline or one of its derivatives at the evaluation points in array `w[]`. If no values can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*float* `*imsl_f_feynman_kac_evaluate`(*int* nw, *int* `m`, *float* `breakpoints[]`, *float* `w[]`,
*float* `coef[]`,

`IMSL_DERIV`, *int* `deriv`,

`IMSL_RETURN_USER`, *float* `value[]`,

0)

## Optional Arguments

`IMSL_DERIV`, *int* `deriv` (Input)
Let ***d*** = `deriv` and let ***H(w)*** be the given Hermite quintic spline. Then, this option produces the ***d***-th derivative of ***H(w)*** at ***w***, ***H***$^{d}$***(w)***. It is required that `deriv = 0,1,2 or 3`.
Default: `deriv = 0`.

`IMSL_RETURN_USER`, *float* value[] (Output)
A user defined array of length `nw` to receive the ***d***-th derivative of $H(x)$ at the evaluation points in `w[]`. When using this option, the return value of the function is `NULL`.

## Description

The Hermite quintic spline interpolation is done over the composite interval $\left[x_{\min}, x_{\max}\right]$, where `-breakpoints[i-1]` = $x_i$ are given by $\left(x_{\min} = \right)x_1 < x_2 < \ \ldots \ < x_m\left( = x_{\max}\right)$.

The Hermite quintic spline function is constructed using three primary functions, defined by

$$
\begin{aligned}
b_0(z) &= -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3\left(6z^2 + 3z + 1\right),\\
b_1(z) &= -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z+1),\\
b_2(z) &= \tfrac{1}{2}\left(-z^5 + 3z^4 - 3z^3 + z^2\right) = \tfrac{1}{2}(1-z)^3 z^2.
\end{aligned}
$$

For each

$$x \in \left[x_i, x_{i+1}\right], h_i = x_{i+1} - x_i, z_i = \left(x - x_i\right)/h_i, i = 1, \ \ldots\ , m-1,$$

the spline is locally defined by

$$
\begin{aligned}
&H(x) = y_{3i-2}b_0(z) + y_{3i+1}b_0(1-z) + h_i y_{3i-1}b_1(z)\\
&-h_i y_{3i+2}b_1(1-z) + h_i^2 y_{3i}b_2(z) + h_i^2 y_{3i+3}b_2(1-z),
\end{aligned}
$$

where

$$y_{3i-2} = f\left(x_i\right), y_{3i-1} = \left(\partial f / \partial x\right)\left(x_i\right) = f'\left(x_i\right), y_{3i} = \left(\partial^2 f / \partial x^2\right)\left(x_i\right)$$
$$= f''\left(x_i\right), i = 1, \ldots, m-1$$

are the values of a given twice continuously differentiable function $f$ and its first two derivatives at the breakpoints.

The approximating function $H(x)$ is twice continuously differentiable on $\left[x_{\min}, x_{\max}\right]$, whereas the third derivative is in general only continuous within the interior of the intervals $\left[x_i, x_{i+1}\right]$. From the local representation of $H(x)$ it follows that

$$H\left(x_i\right) = f\left(x_i\right) = y_{3i-2}, H'\left(x_i\right) = f'\left(x_i\right) = y_{3i-1}, H''\left(x_i\right) = y_{3i}, i = 1, \ldots, m$$

The spline coefficients $y_i, i = 1, \ldots, 3m,$ are stored as successive triplets in array `coef[]`. For a given $w \in \left[x_{\min}, x_{\max}\right]$, function `imsl_f_feynman_kac_evaluate` uses the information in `coef[]` together with the values of $b_0, b_1, b_2$ and its derivatives at $w$ to compute $H^{(d)}(w), d = 0, \ldots, 3$ using the local representation on the particular subinterval containing $w$.

## Example

Consider function $f(x) = x^5$, a polynomial of degree 5, on the interval $[-1, 1]$ with breakpoints $\pm 1$. Then, the end derivative values are

$$y_1 = f(-1) = -1, y_2 = f'(-1) = 5, y_3 = f''(-1) = -20$$

and

$$y_4 = f(1) = 1, y_5 = f'(1) = 5, y_6 = f''(1) = 20$$

Since the Hermite quintic interpolates all polynomials up to degree 5 exactly, the spline interpolation on $[-1, 1]$ must agree with the exact function value up to rounding errors.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

/* Define function */
#define F(x) pow(x,5.0)

int main()
{
   int i;
   int nw = 7;
   int m = 2;
   float breakpoints[] = { -1.0, 1.0 };
```

```
   float w[] = { -0.75, -0.5, -0.25, 0.0,
       0.25, 0.5, 0.75 };
   float coef[] = { -1.0, 5.0, -20.0,
       1.0, 5.0, 20.0 };
   float *result = NULL;

   result = imsl_f_feynman_kac_evaluate(nw, m, breakpoints, w, coef, 0);

   /* Print results */
   printf("    x         F(x)  Interpolant  Error\n\n");
   for (i=0; i<=6; i++)
       printf(" %6.3f %10.3f %10.3f %10.7f\n", w[i], F(w[i]),
           result[i], fabs(F(w[i])-result[i]));
}
```

## Output

```
    x         F(x)  Interpolant  Error

 -0.750     -0.237     -0.237  0.0000000
 -0.500     -0.031     -0.031  0.0000000
 -0.250     -0.001     -0.001  0.0000000
  0.000      0.000      0.000  0.0000000
  0.250      0.001      0.001  0.0000000
  0.500      0.031      0.031  0.0000000
  0.750      0.237      0.237  0.0000000
```

# fast_poisson_2d

more...

Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_fast_poisson_2d (*float* rhs_pde(), *float* rhs_bc(), *float* coeff_u, *int* nx, *int* ny, *float* ax, *float* bx, *float* ay, *float* by, *Imsl_bc_type* bc_type[], …, 0)

The type ***double*** function is imsl_d_fast_poisson_2d.

## Required Arguments

*float* rhs_pde (*float* x, *float* y)
User-supplied function to evaluate the right-hand side of the partial differential equation at x and y.

*float* rhs_bc(*Imsl_pde_side* side, *float* x, *float* y)
User-supplied function to evaluate the right-hand side of the boundary conditions, on side side, at x and y. The value of side will be one of the following: IMSL_RIGHT, IMSL_BOTTOM, IMSL_LEFT, or IMSL_TOP.

*float* coeff_u (Input)
Value of the coefficient of ***u*** in the differential equation.

*int* nx (Input)
Number of grid lines in the ***x***-direction. nx must be at least 4. See the Description section for further restrictions on nx.

*int* ny (Input)
Number of grid lines in the ***y***-direction. ny must be at least 4. See the Description section for further restrictions on ny.

*float* ax (Input)
The value of x along the left side of the domain.

*float* bx (Input)
The value of x along the right side of the domain.

*float* ay (Input)

The value of y along the bottom of the domain.

*float* by (Input)

The value of y along the top of the domain.

*Imsl_bc_type* bc_type[4] (Input)

Array of size 4 indicating the type of boundary condition on each side of the domain or that the solution is periodic. The sides are numbered as follows:

| **Side** | **Location** |
|---|---|
| IMSL_RIGHT_SIDE(0) | x = bx |
| IMSL_BOTTOM_SIDE(1) | y = ay |
| IMSL_LEFT_SIDE(2) | x = ax |
| IMSL_TOP_SIDE(3) | y = by |
| **The three possible boundary condition types are as follows:** | |
| **Type** | **Location** |
| IMSL_DIRICHLET_BC | Value of ***u*** is given. |
| IMSL_NEUMANN_BC | Value of ***du***/***dx*** is given (on the right or left sides) or ***du***/***dy*** (on the bottom or top of the domain). |
| IMSL_PERIODIC_BC | Periodic |

## Return Value

An array of size nx by ny containing the solution at the grid points.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_fast_poisson_2d (*float* rhs_pde (), *float* rhs_bc (), *float* coeff_u, *int* nx, *int* ny, *float* ax, *float* bx, *float* ay, *float* by, *Imsl_bc_type* bc_type [],

IMSL_RETURN_USER, *float* u_user [],

IMSL_ORDER, *int* order,

IMSL_RHS_PDE_W_DATA, *float* rhs_pde (), *void* *data,

IMSL_RHS_BC_W_DATA, *float* rhs_bc (), *void* *data,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* u_user[] (Output)
User-supplied array of size nx by ny containing the solution at the grid points.

IMSL_ORDER, *int* order (Input)
Order of accuracy of the finite-difference approximation. It can be either 2 or 4.
Default: order = 4

IMSL_RHS_PDE_W_DATA, *float* rhs_pde (*float* x, *float* y, *void* *data), *void* *data, (Input)
User-supplied function to evaluate the right-hand side of the partial differential equation at x and y, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

IMSL_RHS_BC_W_DATA, *float* rhs_bc(*Imsl_pde_side* side, *float* x, *float* y, *void* *data), *void* *data, (Input)
User-supplied function to evaluate right-hand side of the boundary conditions, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

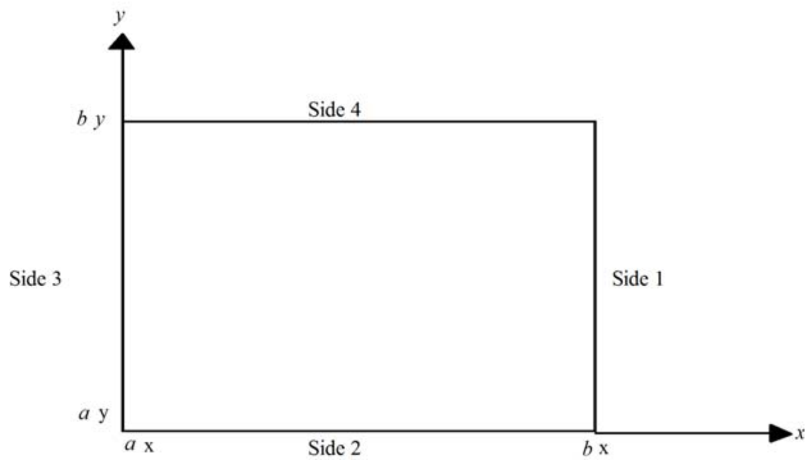
## Description

Let $c$ = coeff_u, $a_x$ = ax, $b_x$ = bx, $a_y$ = ay, $b_y$ = by, $n_x$ = nx and $n_y$ = ny.

imsl_f_fast_poisson_2d is based on the code HFFT2D by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = p$$

on the rectangular domain $(a_x, b_x) \times (a_y, b_y)$ with a user-specified combination of Dirichlet (solution prescribed), Neumann (first-derivative prescribed), or periodic boundary conditions. The sides are numbered clockwise, starting with the right side.




When $c$ = 0 and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum ∞-norm is returned.

The solution is computed using either a second-or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The algorithm relies on the fact that $n_x - 1$ is highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If $n_x - 1$ is highly composite then the execution time of `imsl_f_fast_poisson_2d` is proportional to $n_x n_y \log_2 n_x$. If evaluations of $p(x, y)$ are inexpensive, then the difference in running time between `order` = 2 and `order` = 4 is small.

The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas hx = (`bx` − `ax`)/(`nx` −1) and hy = (`by` − `ay`)/(`ny` − 1). The grid spacings in the x and y directions must be the same, i.e., `nx` and `ny` must be such that hx is equal to hy. Also, as noted above, `nx` and `ny` must be at least 4. To increase the speed of the fast Fourier transform, `nx` − 1 should be the product of small primes. Good choices are 17, 33, and 65.

If `-coeff_u` is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

On some platforms, `imsl_f_fast_poisson_2d` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

## Example

In this example, the equation

$$\frac{\delta^2 u}{\delta x_2} + \frac{\delta^2 u}{\delta y^2} + 3u = -2\sin\left(x + 2y\right) + 16e^{2x+3y}$$

with the boundary conditions

$$\frac{du}{dy} = 2\cos\left(x + 2y\right) + 3e^{2x+3y}$$

on the bottom side and

$$u = \sin\left(x + 2y\right) + e^{2x+3y}$$

on the other three sides is solved. The domain is the rectangle [0, ¼] ×[0, ½]. The output of `imsl_f_fast_poisson_2d` is a 17 × 33 table of values. The functions `imsl_f_spline_2d_value` are used to print a different table of values.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
   float            rhs_pde(float, float);
   float            rhs_bc(Imsl_pde_side, float, float);
   int              nx = 17;
   int              nxtabl = 5;
   int              ny = 33;
   int              nytabl = 5;

   int              i;
   int              j;
   Imsl_f_spline *sp;
   Imsl_bc_type     bc_type[4];

   float            ax, ay, bx, by;
   float            x, y, xdata[17], ydata[33];
   float            coefu, *u;
   float            u_table;
   float            abs_error;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Set rectangle size */
   ax = 0.0;
   bx = 0.25;
   ay = 0.0;
   by = 0.50;

   /* Set boundary conditions */
   bc_type[IMSL_RIGHT_SIDE] = IMSL_DIRICHLET_BC;
   bc_type[IMSL_BOTTOM_SIDE] = IMSL_NEUMANN_BC;
   bc_type[IMSL_LEFT_SIDE] = IMSL_DIRICHLET_BC;
   bc_type[IMSL_TOP_SIDE] = IMSL_DIRICHLET_BC;
```

```
    /* Coefficient of u */
    coefu = 3.0;

    /* Solve the PDE */
    u = imsl_f_fast_poisson_2d(rhs_pde, rhs_bc, coefu, nx, ny,
        ax, bx, ay, by, bc_type,
        0);

    /* Set up for interpolation */
    for (i = 0; i < nx; i++)
        xdata[i] = ax + (bx - ax) * (float) i / (float) (nx - 1);

    for (i = 0; i < ny; i++)
        ydata[i] = ay + (by - ay) * (float) i / (float) (ny - 1);

    /* Compute interpolant */
    sp = imsl_f_spline_2d_interp(nx, xdata, ny, ydata, u,
        0);

    printf("    x         y         u       error\n\n");
    for (i = 0; i < nxtabl; i++)
        for (j = 0; j < nytabl; j++) {
            x = ax + (bx - ax) * (float) j / (float) (nxtabl - 1);
            y = ay + (by - ay) * (float) i / (float) (nytabl - 1);
            u_table = imsl_f_spline_2d_value(x, y, sp,
                0);
            abs_error = fabs(u_table - sin(x + 2.0 * y) -
                exp(2.0 * x + 3.0 * y));

            /* Print computed answer and absolute on
            nxtabl by nytabl grid */
            printf(" %6.4f    %6.4f    %6.4f    %8.2e\n",
                x, y, u_table, abs_error);
        }
}

float rhs_pde(float x, float y)
{
    /* Define the right side of the PDE */
    return (-2.0 * sin(x + 2.0 * y) + 16.0 * exp(2.0 * x + 3.0 * y));
}

float rhs_bc(Imsl_pde_side side, float x, float y)
{
    /* Define the boundary conditions */
    if (side == IMSL_BOTTOM_SIDE)
        return (2.0 * cos(x + 2.0 * y) + 3.0 * exp(2.0 * x + 3.0 *
        y));
    else
        return (sin(x + 2.0 * y) + exp(2.0 * x + 3.0 * y));
}
```

### Output

```
   x          y          u        error
0.0000     0.0000     1.0000     0.00e+00
0.0625     0.0000     1.1956     5.12e-06
0.1250     0.0000     1.4087     7.19e-06
0.1875     0.0000     1.6414     5.10e-06
0.2500     0.0000     1.8961     8.67e-08
0.0000     0.1250     1.7024     1.73e-07
0.0625     0.1250     1.9562     6.39e-06
0.1250     0.1250     2.2345     9.50e-06
0.1875     0.1250     2.5407     6.36e-06
0.2500     0.1250     2.8783     1.66e-07
0.0000     0.2500     2.5964     2.60e-07
0.0625     0.2500     2.9322     9.25e-06
0.1250     0.2500     3.3034     1.34e-05
0.1875     0.2500     3.7148     9.27e-06
0.2500     0.2500     4.1720     9.40e-08
0.0000     0.3750     3.7619     4.84e-07
0.0625     0.3750     4.2163     9.16e-06
0.1250     0.3750     4.7226     1.36e-05
0.1875     0.3750     5.2878     9.44e-06
0.2500     0.3750     5.9199     5.72e-07
0.0000     0.5000     5.3232     5.93e-07
0.0625     0.5000     5.9520     9.84e-07
0.1250     0.5000     6.6569     1.34e-06
0.1875     0.5000     7.4483     4.55e-07
0.2500     0.5000     8.3380     2.27e-06
```

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm. User flag = "#". |

# Chapter 6 Transforms

## Functions

# Usage Notes

## Fast Fourier Transforms

A fast Fourier transform (FFT) is simply a discrete Fourier transform that is computed efficiently. The straightforward method for computing the Fourier transform takes approximately $n^2$ operations where $n$ is the number of points in the transform, while the FFT (which computes the same values) takes approximately $n \log n$ operations. It uses the system's high performance library for the computation, if available. The algorithms in this chapter are modeled on the Cooley-Tukey (1965) algorithm. Hence, these functions are most efficient for integers that are highly composite; that is, integers that are a product of small primes.

For the two functions `imsl_f_fft_real` and `imsl_c_fft_complex` there is a corresponding initialization function. Use these functions *only* when repeatedly transforming sequences of the same length. In this situation, the initialization function computes the initial setup once. Subsequently, the user calls the corresponding main function with the appropriate option. This may result in substantial computational savings. For more information on the use of these functions, consult the documentation under the appropriate function name.

In addition to the one-dimensional transformations described above, we also provide a complex two-dimensional FFT and its inverse.

## Continuous Versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham 1974) as

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i\omega t} dt$$

We begin by making the following approximation:

$$\begin{aligned}\hat{f}(\omega) &\approx \int_{-T/2}^{T/2} f(t) e^{-2\pi i\omega t} dt \\ &= \int_0^T f(t - T/2) e^{-2\pi i\omega(t - T/2)} dt \\ &= e^{\pi i\omega T} \int_0^T f(t - T/2) e^{-2\pi i\omega t} dt\end{aligned}$$

If we approximate the last integral using the rectangle rule with spacing $h = T/n$, we have

$$\hat{f}(\omega) \approx e^{\pi i\omega T} h \sum_{k=0}^{n-1} e^{-2\pi i\omega kh} f(kh - T/2)$$

Finally, setting $\omega = j/T$ for $j = 0, \ldots, n - 1$ yields

$$\hat{f}\left(j/T\right) \approx e^{\pi i j} h \sum_{k=0}^{n-1} e^{-2\pi ijk/n} f\left(kh - T/2\right) = \left(-1\right)^j \sum_{k=0}^{n-1} e^{-2\pi ijk/n} f_k^h$$

where the vector $f^{h} = (f(-T/2), \ldots, f((n-1)h - T/2))$. Thus, after scaling the components by $(-1)^j h$, the discrete Fourier transform, as computed in `imsl_c_fft_complex` (with input $f^{h}$) is related to an approximation of the continuous Fourier transform by the above formula.

If the function $f$ is expressed as a C function, then the continuous Fourier transform

$$\hat{f}$$

can be approximated using the IMSL function `imsl_f_int_fcn_fourier` (Quadrature).

# fft_real

more...

Computes the real discrete Fourier transform of a real sequence.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_fft_real (*int* n, *float* p[], …, 0)

The type ***double*** function is imsl_d_fft_real.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed.

*float* p[] (Input)
Array with n components containing the periodic sequence.

## Return Value

A pointer to the transformed sequence. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_fft_real (*int* n, *float* p[],

IMSL_BACKWARD,

IMSL_PARAMS, *float* params[],

IMSL_RETURN_USER, *float* q[],

0)

## Optional Arguments

`IMSL_BACKWARD`

Compute the backward transform. If `IMSL_BACKWARD` is used, the return value of the function is the backward transformed sequence.

`IMSL_PARAMS`, *float* `params[]` (Input)

Pointer returned by a previous call to `imsl_f_fft_real_init`. If `imsl_f_fft_real` is used repeatedly with the same value of `n`, then it is more efficient to compute these parameters only once.

`IMSL_RETURN_USER`, *float* `q[]` (Output)

Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_f_fft_real` returns `q`. The array `q` must be at least `n` long.

## Description

The function `imsl_f_fft_real` computes the discrete Fourier transform of a real vector of size $n$. It uses the system's high performance library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $n$ is a product of small prime factors. If $n$ satisfies this condition, then the computational effort is proportional to $n \log n$. The Cooley-Tukey algorithm is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

By default, `imsl_f_fft_real` computes the forward transform. If $n$ is even, then the forward transform is

$$q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos\frac{2\pi km}{n} \quad m = 1, \ldots, n/2$$

$$q_{2m} = -\sum_{k=0}^{n-1} p_k \sin\frac{2\pi km}{n} \quad m = 1, \ldots, n/2-1$$

$$q_0 = \sum_{k=0}^{n-1} p_k$$

If $n$ is odd, $q_m$ is defined as above for $m$ from 1 to $(n - 1)/2$.

Let $f$ be a real valued function of time. Suppose we sample $f$ at $n$ equally spaced time intervals of length $\Delta$ seconds starting at time $t_0$. That is, we have

$$p_i := f(t_0 + i\Delta) \qquad i = 0, 1, \ldots, n-1$$

We will assume that $n$ is odd for the remainder of this discussion. The function `imsl_f_fft_real` treats this sequence as if it were periodic of period $n$. In particular, it assumes that $f(t_0) = f(t_0 + n\Delta)$. Hence, the period of the function is assumed to be $T = n\Delta$. We can invert the above transform for $p$ as follows:

$$p_m = \frac{1}{n}\left[ q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi(k+1)m}{n} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi(k+1)m}{n}\right]$$

This formula is very revealing. It can be interpreted in the following manner. The coefficients *q* produced by `imsl_f_fft_real` determine an interpolating trigonometric polynomial to the data. That is, if we define

$$\begin{aligned} g(t) &= \tfrac{1}{n}\left[ q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi(k+1)(t-t_0)}{n\Delta} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi(k+1)(t-t_0)}{n\Delta}\right] \\ &= \tfrac{1}{n}\left[ q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi(k+1)(t-t_0)}{T} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi(k+1)(t-t_0)}{T}\right] \end{aligned}$$

then we have

$$f(t_0 + i\Delta) = g(t_0 + i\Delta)$$

Now suppose we want to discover the dominant frequencies, forming the vector *P* of length (*n* + 1)/2 as follows:

$$\begin{aligned} &P_0 := |q_0| \\ &P_k := \sqrt{q_{2k-1}^2 + q_{2k}^2} \quad k = 1,2, \ldots ,(n-1)/2 \end{aligned}$$

These numbers correspond to the energy in the spectrum of the signal. In particular, $P_k$ corresponds to the energy level at frequency

$$\frac{k}{T} = \frac{k}{n\Delta} \quad k = 0,\ 1,\ \ldots,\ \frac{n-1}{2}$$

Furthermore, note that there are only (*n* + 1)/2 ≈ *T*/(2Δ) resolvable frequencies when *n* observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal. Similar relations hold for the case when *n* is even.

If the optional argument `IMSL_BACKWARD` is specified, then the backward transform is computed. If *n* is even, then the backward transform is

$$q_m = p_0 + (-1)^m p_{n-1} + 2\sum_{k=0}^{n/2-2} p_{2k+1}\cos\frac{2\pi(k+1)m}{n} - 2\sum_{k=0}^{n/2-2} p_{2k+2}\sin\frac{2\pi(k+1)m}{n}$$

If *n* is odd,

$$q_m = p_0 + 2\sum_{k=0}^{(n-3)/2} p_{2k+1}\cos\frac{2\pi(k+1)m}{n} - 2\sum_{k=0}^{(n-3)/2} p_{2k+2}\sin\frac{2\pi(k+1)m}{n}$$

The backward Fourier transform is the unnormalized inverse of the forward Fourier transform.

# Examples

## Example 1

In this example, a pure cosine wave is used as a data vector, and its Fourier series is recovered. The Fourier series is a vector with all components zero except at the appropriate frequency where it has an *n*.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int         k, n = 7;
   float       two_pi = 2*imsl_f_constant("pi", 0);
   float       p[7], *q;
                                 /* Fill q with a pure exponential signal */
   for (k = 0; k < n; k++)
       p[k] = cos(k*two_pi/n);

   q = imsl_f_fft_real (n, p, 0);

   printf("       index     p        q\n");
   for (k = 0; k < n; k++)
       printf("%11d%10.2f%10.2f\n", k, p[k], q[k]);
}
```

### Output

```
       index     p        q
         0      1.00      0.00
         1      0.62      3.50
         2     -0.22      0.00
         3     -0.90     -0.00
         4     -0.90     -0.00
         5     -0.22      0.00
         6      0.62     -0.00
```

## Example 2

This example computes the Fourier transform of the vector $x$, where $x_j = (-1)^j$ for $j = 0$ to $n - 1$. The backward transform of this vector is now computed by using the optional argument `IMSL_BACKWARD`. Note that $s = nx$, that is, $s_j = (-1)^j n$, for $j = 0$ to $n - 1$.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
```

```
    int         k, n = 7;
    float       *q, *s, x[7];
                                /* Fill data vector */
    x[0] = 1.0;
    for (k = 1; k<n; k++)
        x[k] = -x[k-1];
                                /* Compute the forward transform of x */
    q = imsl_f_fft_real (n, x, 0);
                               /* Compute the backward transform of x */
    s = imsl_f_fft_real (n, q,
                         IMSL_BACKWARD,
                         0);
    printf("       index     x        q        s\n");
    for (k = 0; k < n; k++)
        printf("%11d%10.2f%10.2f%10.2f\n", k, x[k], q[k], s[k]);
}
```

### Output

```
       index     x        q        s
         0      1.00      1.00      7.00
         1     -1.00      1.00     -7.00
         2      1.00      0.48      7.00
         3     -1.00      1.00     -7.00
         4      1.00      1.25      7.00
         5     -1.00      1.00     -7.00
         6      1.00      4.38      7.00
```

# fft_real_init

more...

Computes the parameters for imsl_f_fft_real.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_fft_real_init (*int* n)

The type ***double*** function is imsl_d_fft_real_init.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed.

## Return Value

A pointer to the internal parameter vector that can then be used by imsl_f_fft_real when the optional argument IMSL_PARAMS is specified. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Description

The function imsl_f_fft_real_init should be used when many calls are to be made to imsl_f_fft_real without changing the sequence length *n*. This function computes the parameters that are necessary for the real Fourier transform.

It uses the system's high performance library for the computation, if available. Otherwise, the function imsl_f_fft_real_init is based on the routine RFFTI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example computes three distinct real FFTs by calling imsl_f_fft_real_init once and then calling imsl_f_fft_real three times.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int        k, j, n = 7;
   float      two_pi = 2*imsl_f_constant("pi", 0);
   float      p[7], *q, *work;
   work = imsl_f_fft_real_init (n);
   for (j = 0; j < 3; j++){
                          /* Fill p with a pure sinusoidal signal */
      for (k = 0; k < n; k++)
          p[k] = cos(k*two_pi*j/n);

      q = imsl_f_fft_real (n, p,
                       IMSL_PARAMS, work, 0);

      printf("      index     p       q\n");
      for (k = 0; k < n; k++)
          printf("%11d%10.2f%10.2f\n", k, p[k], q[k]);
   }
}
```

## Output

```
index      p       q
 0      1.00     7.00
 1      1.00     0.00
 2      1.00     0.00
 3      1.00     0.00
 4      1.00     0.00
 5      1.00    -0.00
 6      1.00     0.00
index      p       q
 0      1.00     0.00
 1      0.62     3.50
 2     -0.22     0.00
 3     -0.90    -0.00
 4     -0.90    -0.00
 5     -0.22     0.00
 6      0.62    -0.00
index      p       q
 0      1.00    -0.00
 1     -0.22     0.00
 2     -0.90    -0.00
 3      0.62     3.50
 4      0.62    -0.00
 5     -0.90     0.00
```

```
6     -0.22      0.00
```

# fft_complex

more...

Computes the complex discrete Fourier transform of a complex sequence.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_fft_complex (*int* n, *f_complex* p[], …, 0)

The type *d_complex* function is imsl_z_fft_complex.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed.

*f_complex* p[] (Input)
Array with n components containing the periodic sequence.

## Return Value

If no optional arguments are used, imsl_c_fft_complex returns a pointer to the transformed sequence. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_fft_complex (*int* n, *f_complex* p[],

IMSL_BACKWARD,

IMSL_PARAMS, *float* params[],

IMSL_RETURN_USER, *f_complex* q[],

0)

## Optional Arguments

`IMSL_BACKWARD`

Compute the backward transform. If `IMSL_BACKWARD` is used, the return value of the function is the backward transformed sequence.

`IMSL_PARAMS`, *float* `params[]` (Input)

Pointer returned by a previous call to `imsl_c_fft_complex_init`. If `imsl_c_fft_complex` is used repeatedly with the same value of `n`, then it is more efficient to compute these parameters only once.

`IMSL_RETURN_USER`, *f_complex* `q[]` (Output)

Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_c_fft_complex` returns `q`. The array `q` must be of length at least `n`.

## Description

The function `imsl_c_fft_complex` computes the discrete Fourier transform of a real vector of size $n$. It uses the system's high performance library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $n$ is a product of small prime factors. If $n$ satisfies this condition, then the computational effort is proportional to $n \log n$. The Cooley-Tukey algorithm is based on the complex FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

By default, `imsl_c_fft_complex` computes the forward transform below.

$$q_j = \sum_{m=0}^{n-1} p_m e^{-2\pi imj/n}$$

Note that we can invert the Fourier transform as follows below.

$$p_m = \frac{1}{n} \sum_{j=0}^{n-1} q_j e^{2\pi ijm/n}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric interpolating polynomial to the data.

If the option `IMSL_BACKWARD` is selected, then the following computation is performed.

$$q_j = \sum_{m=0}^{n-1} p_m e^{2\pi imj/n}$$

Furthermore, the relation between the forward and backward transforms is that they are unnormalized inverses of each other. That is, the following code fragment begins with a vector $p$ and concludes with a vector $p_2 = np$.

```
q  = imsl_c_fft_complex(n, p, 0);
p2 = imsl_c_fft_complex(n, q, IMSL_BACKWARD, 0);
```

# Examples

## Example 1

This example inputs a pure exponential data vector and recovers its Fourier series, which is a vector with all components zero except at the appropriate frequency where it has an *n*.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int           k, n = 7;
   float         two_pi = 2*imsl_f_constant("pi", 0);
   f_complex     p[7], *q, z;

   /* Fill p with a pure exponential signal */
   for (k = 0; k < n; k++) {
       z.re = 0.;
       z.im = k*two_pi/n;
       p[k] = imsl_c_exp(z);
   }
   q = imsl_c_fft_complex (n, p, 0);

   printf("       index   p.re     p.im     q.re     q.im\n");
   for (k = 0; k < n; k++)
       printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
       q[k].re, q[k].im);
}
```

### Output

```
       index   p.re      p.im      q.re      q.im
         0     1.00      0.00      0.00      0.00
         1     0.62      0.78      7.00      0.00
         2    -0.22      0.97     -0.00      0.00
         3    -0.90      0.43     -0.00      0.00
         4    -0.90     -0.43      0.00     -0.00
         5    -0.22     -0.97      0.00     -0.00
         6     0.62     -0.78      0.00      0.00
```

## Example 2

The backward transform is used to recover the original sequence. Notice that the forward transform followed by the backward transform multiplies the entries in the original sequence by the length of the sequence.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int            k, n = 7;
   float          two_pi = 2*imsl_f_constant("pi", 0);
   f_complex      p[7], *q, *pp;
                                /* Fill p with an increasing signal */
   for (k = 0; k < n; k++) {
        p[k].re = (float) k;
        p[k].im = 0.;
   }
   q = imsl_c_fft_complex (n, p, 0);
   pp = imsl_c_fft_complex (n, q,
                           IMSL_BACKWARD,
                           0);
   printf("       index   p.re     p.im     pp.re    pp.im \n");
   for (k = 0; k < n; k++)
       printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
                pp[k].re , pp[k].im);
}
```

## Output

```
       index    p.re      p.im     pp.re     pp.im
         0      0.00      0.00      0.00      0.00
         1      1.00      0.00      7.00      0.00
         2      2.00      0.00     14.00      0.00
         3      3.00      0.00     21.00      0.00
         4      4.00      0.00     28.00      0.00
         5      5.00      0.00     35.00      0.00
         6      6.00      0.00     42.00      0.00
```

# fft_complex_init

more...

Computes the parameters for imsl_c_fft_complex.

## Synopsis

*#include* <imsl.h>

*float* *imsl_c_fft_complex_init (*int* n)

The type ***double*** function is imsl_z_fft_complex_init.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed.

## Return Value

A pointer to the internal parameter vector that can then be used by imsl_c_fft_complex when the optional argument IMSL_PARAMS is specified. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Description

The routine imsl_c_fft_complex_init should be used when many calls are to be made to imsl_c_fft_complex without changing the sequence length ***n***. This routine computes constants which are necessary for the real Fourier transform.

It uses the system's high performance library for the computation, if available. Otherwise, the function imsl_c_fft_complex_init is based on the routine CFFTI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example computes three distinct complex FFTs by calling imsl_c_fft_complex_init once, then calling imsl_c_fft_complex 3 times.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int            k, j, n = 7;
   float          two_pi = 2*imsl_f_constant("pi", 0), *work;
   f_complex      p[7], *q, z;
   work = imsl_c_fft_complex_init (n);
   for (j = 0; j < 3; j++){
                              /* Fill p with a pure exponential signal */
     for (k = 0; k < n; k++) {
          z.re = 0.;
          z.im = k*two_pi*j/n;
          p[k] = imsl_c_exp(z);
      }
   q = imsl_c_fft_complex (n, p,
                        IMSL_PARAMS, work, 0);

   printf("\n       index   p.re     p.im     q.re     q.im\n");
   for (k = 0; k < n; k++)
       printf("%11d%10.2f%10.2f%10.2f%10.2f\n", k, p[k].re, p[k].im,
               q[k].re, q[k].im);
   }
}
```

## Output

```
      index   p.re     p.im     q.re     q.im
        0     1.00     0.00     7.00     0.00
        1     1.00     0.00     0.00     0.00
        2     1.00     0.00     0.00     0.00
        3     1.00     0.00     0.00     0.00
        4     1.00     0.00     0.00     0.00
        5     1.00     0.00     0.00     0.00
        6     1.00     0.00     0.00     0.00

      index   p.re     p.im     q.re     q.im
        0     1.00     0.00     0.00     0.00
        1     0.62     0.78     7.00     0.00
        2    -0.22     0.97    -0.00     0.00
        3    -0.90     0.43    -0.00     0.00
        4    -0.90    -0.43     0.00    -0.00
        5    -0.22    -0.97     0.00    -0.00
        6     0.62    -0.78     0.00     0.00

      index   p.re     p.im     q.re     q.im
        0     1.00     0.00     0.00     0.00
        1    -0.22     0.97     0.00     0.00
```

```
2     -0.90     -0.43      7.00      0.00
3      0.62     -0.78     -0.00      0.00
4      0.62      0.78     -0.00      0.00
5     -0.90      0.43     -0.00     -0.00
6     -0.22     -0.97      0.00     -0.00
```

# fft_cosine

more...

Computes the discrete Fourier cosine transformation of an even sequence.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_fft_cosine (*int* n, *float* p[], ..., 0)

The type ***double*** procedure is imsl_d_fft_cosine.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed. It must be greater than 1.

*float* p[] (Input)
Array of size n containing the sequence to be transformed.

## Return Value

A pointer to the transformed sequence. To release this space, use imsl_free. If no solution was computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_fft_cosine (*int* n, *float* p[],

IMSL_RETURN_USER, *float* q[],

IMSL_PARAMS, *float* params[],

0)

## Optional Arguments

`IMSL_RETURN_USER`, *float* `q[]` (Output)

Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_f_fft_cosine` returns `q`. The length of array must be at least `n`.

`IMSL_PARAMS`, *float* `params[]` (Input)

Pointer returned by a previous call to `imsl_f_fft_cosine_init`. If `imsl_f_fft_cosine` is used repeatedly with the same value of `n`, then it is more efficient to compute these parameters only once.

Default: Initializing parameters computed each time `imsl_f_fft_cosine` is entered

## Description

The function `imsl_f_fft_cosine` computes the discrete Fourier cosine transform of a real vector of size *N*. It uses the system's high performance library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* – 1 is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log*N*. Specifically, given an *N*-vector `p`, `imsl_f_fft_cosine` returns in `q`

$$q_m = 2\sum_{n=1}^{N-2} p_n \sin\left(\frac{mn\pi}{N-1}\right) + s_0 + s_{N-1}(-1)^m$$

Finally, note that the Fourier cosine transform is its own (unnormalized) inverse. The Cooley-Tukey algorithm is based on the cosine FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example inputs a pure cosine wave as a data vector and recovers its Fourier cosine series, which is a vector with all components zero, except `n` – 1 at the appropriate frequency.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int            n = 7;
   int            i;
   float          p[7];
   float         *q;
   float          pi;

   pi = imsl_f_constant("pi", 0);
```

```
                                   /* Fill p with a pure cosine wave */

   for (i=0; i<n; i++)
           p[i] = cos((float)(i)*pi/(float)(n-1));

   q = imsl_f_fft_cosine (n, p, 0);

   printf ("     index\t  p\t  q\n");
   for (i=0; i<n; i++)
           printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);
}
```

### Output

```
index    p       q
    0    1.00   -0.00
    1    0.87    6.00
    2    0.50    0.00
    3   -0.00    0.00
    4   -0.50   -0.00
    5   -0.87   -0.00
    6   -1.00   -0.00
```

# fft_cosine_init

more...

Computes the parameters needed for `imsl_f_fft_cosine`.

## Synopsis

*#include* <imsl.h>

*float* *`imsl_f_fft_cosine_init` (*int* `n`)

The type ***double*** procedure is `imsl_d_fft_cosine_init`.

## Required Arguments

*int* `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

## Return Value

A pointer to the internal parameter vector that can then be used by `imsl_f_fft_cosine` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `imsl_free`. If no solution was computed, then `NULL` is returned.

## Description

The function `imsl_f_fft_cosine_init` should be used when many calls must be made to `imsl_f_fft_cosine` without changing the sequence length `n`. It uses the system's high performance library for the computation, if available. Otherwise, the function `imsl_f_fft_cosine_init` is based on the routine COSTI in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example computes three distinct sine FFTs by calling `imsl_f_fft_cosine_init` once, then calling `imsl_f_fft_cosine` three times. The internal parameter initialization in `imsl_f_fft_cosine` is now skipped.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int           n = 7;
   int           i, k;
   float         p[7];
   float         q[7];
   float         pi;
   float        *params;

   pi = imsl_f_constant("pi", 0);

                        /* Compute parameters for transform of
                           length n */

   params = imsl_f_fft_cosine_init (n);

                        /* Different frequencies of the same
                           wave will be transformed */
   for (k=0; k<3; k++)
      {
          printf("\n");

                       /* Fill p with a pure cosine wave */

          for (i=0; i<n; i++)
                  p[i] = cos((float)((k+1)*i)*pi/(float)(n-1));

                       /* Compute the transform of p */

          imsl_f_fft_cosine (n, p,
                  IMSL_PARAMS, params,
                  IMSL_RETURN_USER, q,
                  0);

          printf ("     index\t  p\t  q\n");
          for (i=0; i<n; i++)
                  printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);

      }
}
```

## Output

```
index   p      q
    0   1.00  -0.00
```

```
     1   0.87   6.00
     2   0.50   0.00
     3  -0.00   0.00
     4  -0.50  -0.00
     5  -0.87  -0.00
     6  -1.00  -0.00

 index    p       q
     0   1.00   0.00
     1   0.50  -0.00
     2  -0.50   6.00
     3  -1.00   0.00
     4  -0.50   0.00
     5   0.50   0.00
     6   1.00  -0.00

 index    p       q
     0   1.00  -0.00
     1  -0.00   0.00
     2  -1.00  -0.00
     3   0.00   6.00
     4   1.00   0.00
     5  -0.00  -0.00
     6  -1.00   0.00
```

# fft_sine

more...

Computes the discrete Fourier sine transformation of an odd sequence.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_fft_sine (*int* n, *float* p[], ..., 0)

The type ***double*** procedure is imsl_d_fft_sine.

## Required Arguments

*int* n (Input)
Length of the sequence to be transformed. It must be greater than 1.

*float* p[] (Input)
Array of size n containing the sequence to be transformed.

## Return Value

A pointer to the transformed sequence. To release this space, use imsl_free. If no solution was computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_fft_sine (*int* n, *float* p[],
IMSL_RETURN_USER, *float* q[],
IMSL_PARAMS, *float* params[],
0)

## Optional Arguments

IMSL_RETURN_USER, *float* q[] (Output)
: Store the result in the user-provided space pointed to by q. Therefore, no storage is allocated for the solution, and imsl_f_fft_sine returns q. The array must be of length at least n + 1.

IMSL_PARAMS, *float* params[] (Input)
: Pointer returned by a previous call to imsl_f_fft_sine_init. If imsl_f_fft_sine is used repeatedly with the same value of n, then it is more efficient to compute these parameters only once.
Default: Initializing parameters computed each time imsl_f_fft_sine is entered

## Description

The function imsl_f_fft_sine computes the discrete Fourier sine transform of a real vector of size *N*. It uses the system's high performance library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *N* + 1 is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to *N* log*N*. Specifically, given an *N*-vector p, imsl_f_fft_sine returns in q

$$q_m = 2\sum_{n=0}^{N-1} p_n \sin\left(\frac{(m+1)(n+1)\pi}{N+1}\right)$$

Finally, note that the Fourier sine transform is its own (unnormalized) inverse. The Cooley-Tukey algorithm is based on the sine FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example inputs a pure sine wave as a data vector and recovers its Fourier sine series, which is a vector with all components zero, except n at the appropriate frequency.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int           n = 7;
   int           i;
   float         p[7];
   float         *q;
   float         pi;

   pi = imsl_f_constant("pi", 0);

                   /* fill p with a pure sine wave */
```

```
    for (i=0; i<n; i++)
            p[i] = sin((float)(i+1)*pi/(float)(n+1));

    q = imsl_f_fft_sine (n, p, 0);

    printf ("     index\t  p\t  q\n");
    for (i=0; i<n; i++)
            printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);
}
```

## Output

```
index    p       q
   0   0.38   8.00
   1   0.71   0.00
   2   0.92   0.00
   3   1.00   0.00
   4   0.92   0.00
   5   0.71  -0.00
   6   0.38   0.00
```

# fft_sine_init


more...

Computes the parameters needed for `imsl_f_fft_sine`.

## Synopsis

*#include* `<imsl.h>`

*float* `*imsl_f_fft_sine_init` (*int* `n`)

The type ***double*** procedure is `imsl_d_fft_sine_init`.

## Required Arguments

*int* `n` (Input)
Length of the sequence to be transformed. It must be greater than 1.

## Return Value

A pointer to the internal parameter vector that can then be used by `imsl_f_fft_sine` when the optional argument `IMSL_PARAMS` is specified. To release this space, use `imsl_free`. If no solution was computed, then `NULL` is returned.

## Description

The function `imsl_f_fft_sine_init` should be used when many calls must be made to `imsl_f_fft_sine` without changing the sequence length `n`. It uses the system's high performance library for the computation, if available. Otherwise, the function `imsl_f_fft_sine_init` is based on the routine `SINTI` in FFTPACK. The package FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Example

This example computes three distinct sine FFTs by calling `imsl_f_fft_sine_init` once, then calling `imsl_f_fft_sine_init` three times. The internal parameter initialization in `imsl_f_fft_sine` is now skipped.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

int main()
{
   int          n = 7;
   int          i, k;
   float        p[7];
   float        q[7];
   float        pi;
   float       *params;

   pi = imsl_f_constant("pi", 0);

                        /* Compute parameters for transform of
                           length n */

   params = imsl_f_fft_sine_init (n);

                        /* Different frequencies of the same
                           wave will be transformed */
   for (k=0; k<3; k++)
       {
          printf("\n");

                        /* Fill p with a pure sine wave */

          for (i=0; i<n; i++)
                 p[i] = sin((float)((k+1)*(i+1))*pi/(float)(n+1));

                        /* Compute the transform of p */

          imsl_f_fft_sine (n, p,
                 IMSL_PARAMS, params,
                 IMSL_RETURN_USER, q,
                 0);

          printf ("     index\t  p\t  q\n");
          for (i=0; i<n; i++)
                 printf("\t%1d\t%5.2f\t%5.2f\n", i, p[i], q[i]);

     }
}
```

## Output

```
index   p     q
  0   0.38   8.00
```

```
  1    0.71    0.00
  2    0.92    0.00
  3    1.00    0.00
  4    0.92    0.00
  5    0.71   -0.00
  6    0.38    0.00

index    p       q
  0    0.71   -0.00
  1    1.00    8.00
  2    0.71    0.00
  3   -0.00   -0.00
  4   -0.71    0.00
  5   -1.00   -0.00
  6   -0.71    0.00

index    p       q
  0    0.92    0.00
  1    0.71   -0.00
  2   -0.38    8.00
  3   -1.00    0.00
  4   -0.38    0.00
  5    0.71    0.00
  6    0.92    0.00
```

# fft_2d_complex

more...

Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_fft_2d_complex (*int* n, *int* m, *f_complex* p[], …, 0)

The type *d_complex* function is imsl_z_fft_2d_complex.

## Required Arguments

*int* n (Input)
Number of rows in the two-dimensional transform.

*int* m (Input)
Number of columns in the two-dimensional transform.

*f_complex* p[] (Input)
Two-dimensional array of size n × m containing the sequence that is to be transformed.

## Return Value

A pointer to the transformed array. To release this space, use imsl_free. If no value can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_fft_2d_complex (*int* n, *int* m, *f_complex* p[],

IMSL_P_COL_DIM, *int* p_col_dim,

IMSL_BACKWARD,

IMSL_RETURN_USER, *f_complex* q[],

`IMSL_Q_COL_DIM`, *int* `q_col_dim`,

0)

## Optional Arguments

`IMSL_P_COL_DIM`, *int* `p_col_dim` (Input)
The column dimension of `p`.
Default: `p_col_dim` = `m`

`IMSL_BACKWARD`
Compute the backward transform. If `IMSL_BACKWARD` is used, the return value of the function is the backward transformed sequence.

`IMSL_RETURN_USER`, *f_complex* `q[]` (Output)
Store the result in the user-provided space pointed to by `q`. Therefore, no storage is allocated for the solution, and `imsl_c_fft_2d_complex` returns `q`. The array must be of length at least `n` × `m`.

`IMSL_Q_COL_DIM`, *int* `q_col_dim` (Input)
The column dimension of `q`.
Default: `q_col_dim` = `m`

## Description

The function `imsl_c_fft_2d_complex` computes the discrete Fourier transform of a two-dimensional complex array of size $n \times m$. It uses the system's high performance library for the computation, if available. Otherwise, the method used is a variant of the Cooley-Tukey algorithm, which is most efficient when both $n$ and $m$ are a product of small prime factors. If $n$ and $m$ satisfy this condition, then the computational effort is proportional to $nm \log nm$. The Cooley-Tukey algorithm is based on the complex FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research

By default, `imsl_c_fft_2d_complex` computes the forward transform below.

$$q_{jk} = \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} p_{st} e^{-2\pi ijs/n} e^{-2\pi ikt/m}$$

Note that we can invert the Fourier transform as follows.

$$p_{jk} = \frac{1}{nm} \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric interpolating polynomial to the data. The function `imsl_c_fft_2d_complex` is based on the complex FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

If the option IMSL_BACKWARD is selected, then the following computation is performed.

$$p_{jk} = \sum_{s=0}^{n-1}\sum_{t=0}^{m-1} q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

The relation between the forward and backward transforms is that they are unnormalized inverses of each other. That is, the following code fragment begins with a vector *p* and concludes with a vector $p_2 = nmp$.

```
q  = imsl_c_fft_2d_complex(n, m, p, 0);
p2 = imsl_c_fft_2d_complex(n, m, q, IMSL_BACKWARD, 0);
```

# Examples

## Example 1

This example computes the Fourier transform of the pure frequency input for a 5 × 4 array

$$p_{st} = e^{2\pi i2s/S} e^{2\pi it3/4}$$

for $0 \le n \le 4$ and $0 \le m \le 3$. The result, $\hat{p} = q$, has all zeros except in the [2][3] position.

```
#include <imsl.h>

int main()
{
   int            s, t, n = 5, m =4;
   float          two_pi = 2*imsl_f_constant("pi", 0);
   f_complex      p[5][4], *q, z, w;

   /* Fill p with a pure exponential signal */
   for (s = 0; s < n; s++) {
       z.re = 0.;
       z.im = s*two_pi*2./n;
       for(t =0; t < m; t++){
           w.re = 0.;
           w.im = t*two_pi*3./m;
           p[s][t] = imsl_c_mul(imsl_c_exp(z),imsl_c_exp(w));
       }
   }
   q = imsl_c_fft_2d_complex (n, m, (f_complex*)p,
       0);
   /* Write the input */
   imsl_c_write_matrix ("The input matrix is ", 5, 4, (f_complex*)p,
       IMSL_ROW_NUMBER_ZERO,
       IMSL_COL_NUMBER_ZERO,
       0);
   imsl_c_write_matrix ("The output matrix is ", 5, 4, q,
       IMSL_ROW_NUMBER_ZERO,
```

```
        IMSL_COL_NUMBER_ZERO,
        0);
}
```

## Output

```
                The input matrix is
                        0                       1
0 (     1.000,      0.000) (     0.000,     -1.000)
1 (    -0.809,      0.588) (     0.588,      0.809)
2 (     0.309,     -0.951) (    -0.951,     -0.309)
3 (     0.309,      0.951) (     0.951,     -0.309)
4 (    -0.809,     -0.588) (    -0.588,      0.809)

3 (    -0.309,     -0.951) (    -0.951,      0.309)
4 (     0.809,      0.588) (     0.588,     -0.809)
                        2                       3
0 (    -1.000,     -0.000) (    -0.000,      1.000)
1 (     0.809,     -0.588) (    -0.588,     -0.809)
2 (    -0.309,      0.951) (     0.951,      0.309)

                The output matrix is
                        0                       1
0 (        -0,         -0) (         0,         -0)
1 (         0,          0) (         0,         -0)
2 (        -0,         -0) (         0,         -0)
3 (         0,          0) (         0,         -0)
4 (        -0,         -0) (         0,         -0)

                        2                       3
0 (         0,         -0) (         0,         -0)
1 (        -0,          0) (         0,         -0)
2 (         0,         -0) (        20,          0)
3 (        -0,          0) (        -0,         -0)
4 (         0,         -0) (        -0,         -0)
```

## Example 2

This example uses the backward transform to recover the original sequence. Notice that the forward transform followed by the backward transform multiplies the entries in the original sequence by the product of the lengths of the two dimensions.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int           s, t, n = 5, m =4;
   f_complex     p[5][4], *q, *p2;
                            /* Fill p with a pure exponential signal */
```

```
   for (s = 0; s < n; s++) {
        for(t =0; t < m; t++){
        p[s][t].re = s + 5*t;
        p[s][t].im = 0.;
        }
    }                        /* Forward transform */
   q = imsl_c_fft_2d_complex (n, m, (f_complex*)p, 0);
                              /* Backward transform */
   p2 = imsl_c_fft_2d_complex (n, m, q,
                                 IMSL_BACKWARD, 0);
                              /* Write the input */
   imsl_c_write_matrix ("The input matrix is ", 5, 4, (f_complex*)p,
                                 IMSL_ROW_NUMBER_ZERO,
                                 IMSL_COL_NUMBER_ZERO, 0);
   imsl_c_write_matrix ("The output matrix is ", 5, 4, p2,
                                 IMSL_ROW_NUMBER_ZERO,
                                 IMSL_COL_NUMBER_ZERO, 0);
}
```

### Output

```
                    The input matrix is
                              0                        1
0 (          0,          0) (          5,          0)
1 (          1,          0) (          6,          0)
2 (          2,          0) (          7,          0)
3 (          3,          0) (          8,          0)
4 (          4,          0) (          9,          0)

                              2                        3
0 (         10,          0) (         15,          0)
1 (         11,          0) (         16,          0)
2 (         12,          0) (         17,          0)
3 (         13,          0) (         18,          0)
4 (         14,          0) (         19,          0)

                    The output matrix is
                              0                        1
0 (          0,          0) (        100,          0)
1 (         20,          0) (        120,          0)
2 (         40,          0) (        140,          0)
3 (         60,          0) (        160,          0)
4 (         80,          0) (        180,          0)

                              2                        3
0 (        200,          0) (        300,          0)
1 (        220,          0) (        320,          0)
2 (        240,          0) (        340,          0)
3 (        260,          0) (        360,          0)
4 (        280,          0) (        380,          0)
```

## Warning Erors

| | |
|---|---|
| `IMSL_CUFFT_PLAN_ERROR` | An error was encountered setting up the problem using the CUDA Toolkit algorithm. Using the IMSL CPU algorithm instead. |
| `IMSL_CUFFT_COPY_TO_GPU` | An error was encountered copying the data from the CPU to the GPU. Using the IMSL CPU algorithm instead. |
| `IMSL_CUFFT_ERROR` | An error was encountered executing the problem on the GPU. Using the IMSL CPU algorithm instead. |
| `IMSL_CUFFT_COPY_TO_CPU` | An error was encountered copying the data from the GPU to the CPU. Using the IMSL CPU algorithm instead. |
| `IMSL_CUFFT_FREE` | An error was encountered when freeing memory from the GPU. Using the IMSL CPU algorithm instead. |

| | |
|---|---|
| IMSL_CUFFT_ALLOCATION | An error was encountered allocating memory on the GPU. Using the IMSL CPU algorithm instead. |
| IMSL_CUDA_OUT_OF_MEMORY_2 | An error was encountered allocating memory required by the CUDA implementation with "m" = # and "n" = #. Using the IMSL CPU algorithm instead. |

# convolution

Computes the convolution, and optionally, the correlation of two real vectors.

## Synopsis

*#include* <imsl.h>

*float* **imsl_f_convolution* (*int* nx, *float* x[], *int* ny, *float* y[], *int* *nz, …, 0)

The type *double* function is imsl_d_convolution.

## Required Arguments

*int* nx (Input)
Length of the vector x.

*float* x[] (Input)
Real vector of length nx.

*int* ny (Input)
Length of the vector y.

*float* y[] (Input)
Real vector of length ny.

*int* *nz (Output)
Length of the output vector.

## Return Value

A pointer to an array of length nz containing the convolution of x and y. To release this space, use imsl_free. If no zeros are computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_convolution (*int* nx, *float* x[], *int* ny, *float* y[], *int* *nz,

IMSL_PERIODIC,

IMSL_CORRELATION,

IMSL_FIRST_CALL,

IMSL_CONTINUE_CALL,

IMSL_LAST_CALL,

IMSL_RETURN_USER, *float* z[],

IMSL_Z_TRANS, *float* **zhat

IMSL_Z_TRANS_USER, *float* *zhat,

0)

## Optional Arguments

IMSL_PERIODIC
The input is periodic.

IMSL_CORRELATION
Return the correlation of x and y.

IMSL_FIRST_CALL
If the function is called multiple times with the same nx and ny, select this option on the first call.

IMSL_CONTINUE_CALL
If the function is called multiple times with the same nx and ny, select this option on intermediate calls.

IMSL_LAST_CALL
If the function is called multiple times with the same nx and ny, select this option on the final call.

IMSL_RETURN_USER, *float* z[] (Output)
User-supplied array of length at least nz containing the convolution or correlation of x and y.

IMSL_Z_TRANS, *float* **zhat[] (Output)
Address of a pointer to an array of length at least nz containing on output the discrete Fourier transform of z.

IMSL_Z_TRANS_USER, *float* zhat[] (Output)
User-supplied array of length at least nz containing on output the discrete Fourier transform of z.

## Description

The function imsl_f_convolution, by default, computes the discrete convolution of two sequences *x* and *y*. More precisely, let $n_x$ be the length of *x*, and $n_y$ denote the length of *y*. If a circular convolution is desired, the optional argument IMSL_PERIODIC must be selected. We set

$$n_z = \max\{n_y, n_x\},$$

and we pad out the shorter vector with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on $x$ is interpreted as a positive number between 1 and $n_z$, modulo $n_z$.

The technique used to compute the $z_i$'s is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where the following equation is true.

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of $x$ and $y$, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that $n_z$ is the product of small primes if option `IMSL_PERIODIC` is selected. If $n_z$ is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If option `IMSL_PERIODIC` is not selected, then a good value is chosen for $n_z$ so that the Fourier transforms are efficient and $n_z \geq n_x + n_y - 1$. This will mean that both vectors will be padded with zeros.

We point out that no complex transforms of $x$ or $y$ are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Optionally, the function `imsl_f_convolution` computes the discrete correlation of two sequences $x$ and $y$. More precisely, let $n$ be the length of $x$ and $y$. If a circular correlation is desired, then option `IMSL_PERIODIC` must be selected. We set (on output)

| | |
|---|---|
| $n_z = n$ | if `IMSL_PERIODIC` is chosen |
| $(n_z = 2^{\alpha}3^{\beta}5^{\gamma} \geq 2n - 1)$ | if `IMSL_PERIODIC` is not chosen |

where α, β, and γ are nonnegative integers yielding the smallest number of the type $2^{\alpha}3^{\beta}5^{\gamma}$ satisfying the inequality. Once $n_z$ is determined, we pad out the vectors with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on $x$ is interpreted as a positive number between one and $n_z$, modulo $n_z$. Note that this means that

$$z_{n_z-k}$$

contains the correlation of $x(k-1)$ with $y$ as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k-1) = y(k)$ for all $k$, then we would expect

$$z_{n_z}$$

to be the largest component of *z*. The technique used to compute the $z_i$'s is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of *z* is given by

$$\hat{z}_j = \hat{x}_j \overline{y}_j$$

where the following equation is true.

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of *x* and the conjugate of the discrete Fourier transform of *y*, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that $n_z$ is the product of small primes if `IMSL_PERIODIC` is selected. If $n_z$ is the product of small primes, then the computational effort will be proportional to $n_z \log (n_z)$. If `IMSL_PERIODIC` is not chosen, then a good value is chosen for $n_z$ so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeros.

We point out that no complex transforms of *x* or *y* are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

# Examples

## Example 1

This example computes a nonperiodic convolution. The idea here is that you can compute a moving average of the type found in digital filtering using this function. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. We try to recover the values of an exponential function contaminated by noise. The large error for the last value has to do with the fact that the convolution is averaging the zeros in the "pad" rather than the function values. Notice that the signal size is 100, but only reports the errors at 10 points.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NFLTR 5
#define NY    100
/* Define function */
#define F1(A)  exp(A)

int main()
{
   int         i, k, nz;
```

```
float      fltr[NFLTR], fltrer, origer, total1, total2, twopi,
    x, y[NY], *z, *noise;

/* Set up the filter */
for (i = 0; i < NFLTR; i++)
    fltr[i] = 0.2;

/* Set up y-vector for the nonperiodic casE. */

twopi = 2.0*imsl_f_constant ("Pi",
    0);
imsl_random_seed_set(1234579);
noise = imsl_f_random_uniform(NY,
    0);

for (i = 0; i < NY; i++) {
    x = (float)(i) / (NY - 1);
    y[i] = F1(x) + 0.5 *noise[i] - 0.25;
}

/* Call the convolution routine for the nonperiodic case. */
z = imsl_f_convolution(NFLTR, fltr, NY, y, &nz,
    0);

/* Call test routines to check z & zhat here. Print results */
printf("\n Nonperiodic Case\n");
printf("       x         F1(x)       Original Error");
printf("  Filtered Error\n");
total1 = 0.0;
total2 = 0.0;

for (i = 0; i < NY; i++) {
    if (i >= NY-2)
        k = i - NY + 2;
    else
        k = i + 2;
    x = (float)(i) / (float) (NY - 1);
    origer = fabs(y[i] - F1(x));
    fltrer = fabs(z[i+2] - F1(x));
    if ((i % 11) == 0) {
        printf(" %10.4f%13.4f%18.4f%18.4f\n",
            x, F1(x), origer, fltrer);
    }
    total1 += origer;
    total2 += fltrer;
}

printf(" Average absolute error before filter:%10.5f\n",
    total1 / (NY));
printf(" Average absolute error after filter:%11.5f\n",
    total2 / (NY));
```

```
}
```

## Output

```
Nonperiodic Case
       x           F1(x)        Original Error  Filtered Error
    0.0000       1.0000            0.0811            0.3523
    0.1111       1.1175            0.0226            0.0754
    0.2222       1.2488            0.1526            0.0488
    0.3333       1.3956            0.0959            0.0161
    0.4444       1.5596            0.1747            0.0276
    0.5556       1.7429            0.1035            0.0250
    0.6667       1.9477            0.0402            0.0562
    0.7778       2.1766            0.0673            0.0835
    0.8889       2.4324            0.1044            0.0050
    1.0000       2.7183            0.0154            1.1255
Average absolute error before filter:  0.12481
Average absolute error after filter:   0.06785
```

## Example 2

This example computes both a periodic correlation between two distinct signals *x* and *y.* There are 100 equally spaced points on the interval [0, 2π] and $f_1(x) = \sin(x)$. Define *x* and *y* as follows:

$$x_i = f_1\left(\frac{2\pi i}{n-1}\right) \quad i = 0, \ \ldots, \quad n-1$$
$$y_i = f_1\left(\frac{2\pi i}{n-1} + \frac{\pi}{2}\right) \quad i = 0, \ \ldots, \quad n-1$$

Note that the maximum value of *z* (the correlation of *x* with*)* occurs at *i* = 25, which corresponds to the offset.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define N   100
/* Define function */
#define F1(A)  sin(A)

int main()
{
   int   i, k, nz;
   float pi, max, x[N], y[N], *z, xnorm, ynorm;

   /* Set up y-vector for the nonperiodic case. */
   pi = imsl_f_constant ("Pi", 0);

   for (i = 0; i < N; i++) {
       x[i] = F1(2.0*pi*(float)(i) / (N-1));
       y[i] = F1(2.0*pi*(float)(i) / (N-1) + pi/2.0);
   }

   /* Call the correlation function for the nonperiodic case. */
   z = imsl_f_convolution(N, x, N, y, &nz,
       IMSL_CORRELATION,
       IMSL_PERIODIC,
       0);

   xnorm = imsl_f_vector_norm (N, x, 0);
   ynorm = imsl_f_vector_norm (N, y, 0);
   for (i = 0; i < N; i++) {
       z[i] /= xnorm*ynorm;
   }

   max = z[0];
   k = 0;
   for (i = 1; i < N; i++) {
       if (max < z[i]) {
           max = z[i];
           k = i;
       }
   }

   printf("The element of Z with the largest normalized\n");
   printf("value is Z(%2d).\n", k);
   printf("The normalized value of Z(%2d) is %6.3f\n", k, z[k]);
}
```

### Output

```
The element of Z with the largest normalized
```

```
value is Z(25).
The normalized value of Z(25) is 1.000
```

# convolution (complex)

Computes the convolution, and optionally, the correlation of two complex vectors.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_convolution (*int* nx, *f_complex* x[], *int* ny, *f_complex* y[], *int* *nz, …, 0)

The type *double* function is imsl_d_convolution.

## Required Arguments

*int* nx (Input)
Length of the vector *x*.

*f_complex* x[] (Input)
Real vector of length *nx*.

*int* ny (Input)
Length of the vector *y*.

*f_complex* y[] (Input)
Real vector of length *ny*.

*int* *nz (Output)
Length of the output vector.

## Return Value

A pointer to an array of length nz containing the convolution of *x* and *y*. To release this space, use imsl_free. If no zeros are computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_convolution (*int* nx, *f_complex* x[], *int* ny, *f_complex* y[], *int* *nz,

IMSL_PERIODIC,

IMSL_CORRELATION,

IMSL_FIRST_CALL,

IMSL_CONTINUE_CALL,

IMSL_LAST_CALL,

IMSL_RETURN_USER, *f_complex* z[],

IMSL_Z_TRANS, *f_complex* **zhat

IMSL_Z_TRANS_USER, *f_complex* *zhat,

0)

## Optional Arguments

IMSL_PERIODIC
The input is periodic.

IMSL_CORRELATION
Return the correlation of *x* and *y*.

IMSL_FIRST_CALL
If the function is called multiple times with the same nx and ny, select this option on the first call.

IMSL_CONTINUE_CALL
If the function is called multiple times with the same nx and ny, select this option on intermediate calls.

IMSL_LAST_CALL
If the function is called multiple times with the same nx and ny, select this option on the final call.

IMSL_RETURN_USER, *f_complex* z[] (Output)
User-supplied array of length at least nz containing the convolution or correlation of x and y.

IMSL_Z_TRANS, *f_complex* **zhat[] (Output)
Address of a pointer to an array of length at least nz containing on output the discrete Fourier transform of z.

IMSL_Z_TRANS_USER, *f_complex* zhat[] (Output)
User-supplied array of length at least nz containing on output the discrete Fourier transform of z.

## Description

The function imsl_c_convolution, by default, computes the discrete convolution of two sequences *x* and *y*. More precisely, let $n_x$ be the length of *x*, and $n_y$ denote the length of *y*. If a circular convolution is desired, the optional argument IMSL_PERIODIC must be selected. We set

$$n_z = \max\{n_y, n_x\}$$

and we pad out the shorter vector with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j$$

where the index on $x$ is interpreted as a positive number between 1 and $n_z$, modulo $n_z$.

The technique used to compute the $z_i$'s is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where the following equation is true.

$$\hat{z}(n) = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of $x$ and $y$, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that $n_z$ is the product of small primes if option IMSL_PERIODIC is selected. If $n_z$ is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If option IMSL_PERIODIC is not selected, then a good value is chosen for $n_z$ so that the Fourier transforms are efficient and $n_z \geq n_x + n_y - 1$. This will mean that both vectors will be padded with zeros.

Optionally, the function imsl_c_convolution computes the discrete correlation of two sequences $x$ and $y$. More precisely, let $n$ be the length of $x$ and $y$. If a circular correlation is desired, then option IMSL_PERIODIC must be selected. We set (on output)

| | |
|---|---|
| $n_z = n$ | if IMSL_PERIODIC is chosen |
| $(n_z = 2^{\alpha}3^{\beta}5^{\gamma} \geq 2n - 1)$ | if IMSL_PERIODIC is not chosen |

where $\alpha$, $\beta$, and $\gamma$ are nonnegative integers yielding the smallest number of the type $2^{\alpha}3^{\beta}5^{\gamma}$ satisfying the inequality. Once $n_z$ is determined, we pad out the vectors with zeros. Then, we compute

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1} y_j$$

where the index on $x$ is interpreted as a positive number between one and $n_z$, modulo $n_z$. Note that this means that

$$z_{n_z-k}$$

contains the correlation of $x(k-1)$ with $y$ as $k = 0, 1, \ldots, n_z/2$. Thus, if $x(k-1) = y(k)$ for all $k$, then we would expect

$$\Re\, z_{n_z}$$

to be the largest component of $\Re z$. The technique used to compute the $z_i$'s is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication.

Thus, the Fourier transform of *z* is given by

$$\hat{z}_j = \hat{x}_j \overline{\hat{y}}_j$$

where the following equation is true.

$$\hat{z}_j = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(j-1)/n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of *x* and the conjugate of the discrete Fourier transform of *y*, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that $n_z$ is the product of small primes if `IMSL_PERIODIC` is selected. If $n_z$is the product of small primes, then the computational effort will be proportional to $n_z\log(n_z)$. If `IMSL_PERIODIC` is not chosen, then a good value is chosen for $n_z$ so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors will be padded with zeros.

No complex transforms of *x* or *y* are taken since both sequences are real, and real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

# Examples

## Example 1

This example computes a nonperiodic convolution. The purpose is to compute a moving average of the type found in digital filtering. The averaging operator in this case is especially simple and is given by averaging five consecutive points in the sequence. We try to recover the values of an exponential function contaminated by noise. The large error for the last value has to do with the fact that the convolution is averaging the zeros in the "pad" rather than the function values. Notice that the signal size is 100, but only report the errors at ten points.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NFLTR 5
#define NY 100

#define F1(A) (imsl_c_mul(imsl_cf_convert(exp(A),0.0), \
   imsl_cf_convert(cos(A),sin(A)) ))

int main()
```

```
{
   int i, nz;
   f_complex fltr[NFLTR], temp,
       y[NY], *z;
   float x, twopi, total1, total2, *noise, origer, fltrer;

   /* Set up the filter */
   for (i = 0; i < NFLTR; i++) fltr[i] = imsl_cf_convert(0.2,0.0);

   /* Set up y-vector for the periodic case */
   twopi = 2.0*imsl_f_constant ("Pi", 0);
   imsl_random_seed_set(1234579);
   noise = imsl_f_random_uniform(2*NY, 0);

   for (i = 0; i < NY; i++) {
       x = (float)(i) / (NY - 1);
       temp = imsl_cf_convert(0.5*noise[i]-0.25, 0.5*noise[NY+i]-0.25);
       y[i] = imsl_c_add(F1(x), temp);
   }

   /* Call the convolution routine for the periodic case */
   z = imsl_c_convolution(NFLTR, fltr, NY, y, &nz, 0);

   /* Print results */
   printf(" Periodic Case\n");
   printf(" x F1(x) Original Error");
   printf(" Filtered Error\n");

   total1 = 0.0;
   total2 = 0.0;
   for (i = 0; i < NY; i++) {
       x = (float)(i) / (NY - 1);
       origer = imsl_c_abs(imsl_c_sub(y[i],F1(x)));
       fltrer = imsl_c_abs(imsl_c_sub(z[i+2],F1(x)));
       if ((i % 11) == 0)
           printf(" %10.4f (%6.4f,%6.4f) %12.4f %15.4f\n",
           x, (F1(x)).re, (F1(x)).im, origer, fltrer);
       total1 += origer;
       total2 += fltrer;
   }

   printf(" Average absolute error before filter:%10.5f\n",
       total1 / (NY));
   printf(" Average absolute error after filter:%11.5f\n",
       total2 / (NY));
}
```

### Output

```
Periodic Case
       x         F1(x)        Original Error   Filtered Error
```

```
    0.0000  (1.0000,0.0000)      0.1684        0.3524
    0.1111  (1.1106,0.1239)      0.0582        0.0822
    0.2222  (1.2181,0.2752)      0.1991        0.1054
    0.3333  (1.3188,0.4566)      0.1487        0.1001
    0.4444  (1.4081,0.6706)      0.2381        0.1004
    0.5556  (1.4808,0.9192)      0.1037        0.0708
    0.6667  (1.5307,1.2044)      0.1312        0.0904
    0.7778  (1.5508,1.5273)      0.1695        0.0856
    0.8889  (1.5331,1.8885)      0.1851        0.0698
    1.0000  (1.4687,2.2874)      0.2130        1.0760
Average absolute error before filter:  0.19057
Average absolute error after filter:   0.10024
```

## Example 2

This example computes both a periodic correlation between two distinct signals *x* and *y.* There are 100 equally spaced points on the interval [0, 2π] and $f_1(x) = \cos(x) + i\sin(x)$. Define *x* and *y* as follows:

$$x_i = f_1\left(\frac{2\pi(i-1)}{n-1}\right) \quad i = 1, \ \dots, n$$

$$y_i = f_1\left(\frac{2\pi(i-1)}{n-1} + \frac{\pi}{2}\right) \quad i = 1, \ \dots, n$$

Note that the maximum value of *z* (the correlation of *x* with*)* occurs at *i* = 25, which corresponds to the offset.

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

#define N   100

/* Define function */
#define F1(A)  imsl_cf_convert(cos(A),sin(A))

int main()
{
   int        i, k, nz;
   float      zreal[4*N], pi, max, xnorm, ynorm, sumx, sumy;
   f_complex  x[N], y[N], *z;

   /* Set up y-vector for the nonperiodic case */
   pi = imsl_f_constant ("Pi", 0);

   for (i = 0; i < N; i++) {
       x[i] = F1(2.0*pi*(float)(i) / (N-1));
       y[i] = F1(2.0*pi*(float)(i) / (N-1) + pi/2.0);
   }

   /* Call the correlation function for the
   nonperidic case */
```

```
    z = imsl_c_convolution(N, x, N, y, &nz,
        IMSL_CORRELATION, IMSL_PERIODIC,0);

    sumx = sumy = 0.0;
    for (i = 0; i < N; i++) {
        sumx += imsl_c_abs(imsl_c_mul(x[i], x[i]));
        sumy += imsl_c_abs(imsl_c_mul(y[i], y[i]));
    }

    xnorm = sqrt((sumx));
    ynorm = sqrt((sumy));
    for (i = 0; i < N; i++) {
        zreal[i] = (z[i].re/(xnorm*ynorm));
    }

    max = zreal[0];
    k = 0;

    for (i = 1; i < N; i++) {
        if (max < zreal[i]) {
            max = zreal[i];
            k = i;
        }
    }

    printf("The element of Z with the largest normalized\n");
    printf("value is Z(%2d).\n", k);
    printf("The normalized value of Z(%2d) is %6.3f\n", k, zreal[k]);
}
```

### Output

```
The element of Z with the largest normalized
value is Z(25).
The normalized value of Z(25) is 1.000
```

# inverse_laplace

Computes the inverse Laplace transform of a complex function.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_inverse_laplace (*f_complex* fcn (), *float* sigma0, *int* n, *float* t [], …, 0)

The type *double* procedure is imsl_d_inverse_laplace.

## Required Arguments

*f_complex* fcn(*f_complex* z) (Input)
User-supplied function for which the inverse Laplace transform will be computed.

*float* sigma0 (Input)
An estimate for the maximum of the real parts of the singularities of fcn. If unknown, set sigma0 = 0.0.

*int* n (Input)
The number of points at which the inverse Laplace transform is desired.

*float* t [] (Input)
Array of size n containing the points at which the inverse Laplace transform is desired.

## Return Value

A pointer to the array of length n whose *i*-th component contains the approximate value of the inverse Laplace transform at the point t[*i*]. To release this space, use imsl_free. If no solution was computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_inverse_laplace (*f_complex* fcn (), *float* sigma0, *int* n, *float* t [],

IMSL_RETURN_USER, *float* x [],

IMSL_PSEUDO_ACCURACY, *float* pseudo_accuracy,

IMSL_FIRST_LAGUERRE_PARAMETER, *float* sigma,

IMSL_SECOND_LAGUERRE_PARAMETER, *float* bvalue, IMSL_MAXIMUM_COEFFICIENTS, *int* mtop,

IMSL_ERROR_EST, *float* *error_est,

IMSL_DISCRETIZATION_ERROR_EST, *float* *disc_error_est,

IMSL_TRUNCATION_ERROR_EST, *float* *trunc_error_est,

IMSL_CONDITION_ERROR_EST, *float* *cond_error_est,

IMSL_DECAY_FUNCTION_COEFFICIENT, *float* *k,

IMSL_DECAY_FUNCTION_BASE, *float* *r,

IMSL_LOG_LARGEST_COEFFICIENTS, *float* *log_largest_coefs,

IMSL_LOG_SMALLEST_COEFFICIENTS, *float* *log_smallest_coefs,

IMSL_UNDER_OVERFLOW_INDICATORS, *Imsl_laplace_flow* **indicators,

IMSL_FCN_W_DATA, *f_complex* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_RETURN_USER, *float* x[] (Output)
A user-allocated array of length n containing the approximate value of the inverse Laplace transform.

IMSL_PSEUDO_ACCURACY, *float* pseudo_accuracy (Input)
The required absolute uniform pseudo accuracy for the coefficients and inverse Laplace transform values.

Default: pseudo_accuracy = $\sqrt{\varepsilon}$, where **ε** is machine epsilon

IMSL_FIRST_LAGUERRE_PARAMETER, *float* sigma (Input)
The first parameter of the Laguerre expansion. If sigma is not greater than sigma0, it is reset to sigma0 + 0.7.
Default: sigma = sigma0 + 0.7

IMSL_SECOND_LAGUERRE_PARAMETER, *float* bvalue (Input)
The second parameter of the Laguerre expansion. If bvalue is less than 2.0*(sigma − sigma0), it is reset to 2.5*(sigma − sigma0).
Default: bvalue = 2.5*(sigma − sigma0)

IMSL_MAXIMUM_COEFFICIENTS, *int* mtop (Input)
An upper limit on the number of coefficients to be computed in the Laguerre expansion. Argument mtop must be a multiple of four.
Default: mtop = 1024

`IMSL_ERROR_EST`, *float* `*error_est` (Output)

Overall estimate of the pseudo error, `disc_error_est` + `trunc_error_est` + `cond_error_est`. See the Description section for details.

`IMSL_DISCRETIZATION_ERROR_EST`, *float* `*disc_error_est` (Output)

Estimate of the pseudo discretization error.

`IMSL_TRUNCATION_ERROR_EST`, *float* `*trunc_error_est` (Output)

Estimate of the pseudo truncation error.

`IMSL_CONDITION_ERROR_EST`, *float* `*cond_error_est` (Output)

Estimate of the pseudo condition error on the basis of minimal noise levels in the function values.

`IMSL_DECAY_FUNCTION_COEFFICIENT`, *float* `*k` (Output)

The coefficient of the decay function. See the Description section for details.

`IMSL_DECAY_FUNCTION_BASE`, *float* `*r` (Output)

The base of the decay function. See the Description section for details.

`IMSL_LOG_LARGEST_COEFFICIENTS`, *float* `*log_largest_coefs` (Output)

The logarithm of the largest `coefficient` in the decay function. See the Descriptionsection for details.

`IMSL_LOG_SMALLEST_COEFFICIENTS`, *float* `*log_smallest_coefs` (Output)

The logarithm of the smallest nonzero `coefficient` in the decay function. See the Description section for details.

`IMSL_UNDER_OVERFLOW_INDICATORS`, *Imsl_laplace_flow* `**indicators` (Output)

The address of a pointer initialized by `imsl_f_inverse_laplace` to point to an array of length `n` containing the overflow/underflow indicators for the computed approximate inverse Laplace transform. For the *i*th point at which the transform is computed, indicators[i] signifies the following:

| **indicators [i]** | **Meaning** |
|---|---|
| `IMSL_NORMAL_TERMINATION` | Normal termination. |
| `IMSL_TOO_LARGE` | The value of the inverse Laplace transform is too large to be representable. This component of the result is set to NaN. |
| `IMSL_TOO_SMALL` | The value of the inverse Laplace transform is found to be too small to be representable. This component of the result is set to 0.0. |
| `IMSL_TOO_LARGE_BEFORE_EXPANSION` | The value of the inverse Laplace transform is estimated to be too large, even before the series expansion, to be representable. This component of the result is set to NaN. |
| `IMSL_TOO_SMALL_BEFORE_EXPANSION` | The value of the inverse Laplace transform is estimated to be too small, even before the series expansion, to be representable. This component of the result is set to 0.0. |

`IMSL_FCN_W_DATA`, *f_complex* `fcn`(*f_complex* `z`, *void* `*data`) ,*void* `*data`, (Input)

User supplied function for which the inverse Laplace transform will be computed, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function `imsl_f_inverse_laplace` computes the inverse Laplace transform of a complex-valued function. Recall that if *f* is a function that vanishes on the negative real axis, then the Laplace transform of *f* is defined by

$$L[f](s) = \int_0^{\infty} e^{-sx} f(x)\,dx$$

It is assumed that for some value of *s* the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on a modification of Weeks' method (see Weeks (1966)) due to Garbow et al. (1988). This method is suitable when *f* has continuous derivatives of all orders on [0, ∞). In particular, given a complex-valued function *F*(*s*) = *L*[*f*] (*s*), *f* can be expanded in a Laguerre series whose coefficients are determined by *F*. This is fully described in Garbow et al. (1988) and Lyness and Giunta (1986).

The algorithm attempts to return approximations *g*(*t*) to *f*(*t*) satisfying

$$\left| \frac{g(t) - f(t)}{e^{\sigma t}} \right| < \varepsilon$$

where $\varepsilon$ = `pseudo_accuracy` and $\sigma$= `sigma` > `sigma0`. The expression on the left is called the ***pseudo error***. An estimate of the pseudo error in available in `error_est`.

The first step in the method is to transform *F* to $\phi$ where

$$\phi(z) = \frac{b}{1-z} F\left(\frac{b}{1-z} - \frac{b}{2} + \sigma\right)$$

Then, if *f* is smooth, it is known that $\phi$ is analytic in the unit disc of the complex plane and hence has a Taylor series expansion

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s$$

which converges for all *z* whose absolute value is less than the radius of convergence $R_C$. This number is estimated in *r*, obtained through the optional argument `IMSL_DECAY_FUNCTION_BASE`. Using optional argument `IMSL_DECAY_FUNCTION_COEFFICIENT`, the smallest number *K* is estimated which satisfies

$$|a_s| < \frac{K}{R^s}$$

for all $R < R_C$.

The coefficients of the Taylor series for $\phi$ can be used to expand $f$ in a Laguerre series

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt)$$

On some platforms, `imsl_f_inverse_laplace` can evaluate the user-supplied function `fcn` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

# Examples

## Example 1

This example computes the inverse Laplace transform of the function $(s - 1)^{-2}$, and prints the computed approximation, true transform value, and difference at five points. The correct inverse transform is $xe^{x}$. From Abramowitz and Stegun (1964).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
   f_complex f(f_complex);
   int n = 5;
   float t[5];
   float true_inverse[5];
   float relative_diff[5];
   int i;
   float *inverse;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);

   /* Initialize t and compute inverse */
   for (i=0; i<n; i++)
       t[i] = (float)i + 0.5;

   inverse = imsl_f_inverse_laplace(f, 1.5, n, t,
       0);

   /* Compute true inverse, relative difference */
   for (i=0; i<n; i++) {
       true_inverse[i] = t[i]*exp(t[i]);
       relative_diff[i] = fabs(inverse[i] - true_inverse[i])/
           true_inverse[i];
```

```
    }

    printf("\t  t\t\t f_inv\t\t true\t\t diff\n");
    for (i=0; i<n; i++)
        printf ("\t%5.1f\t\t%7.3f\t\t%7.3f\t\t%6.1e\n", t[i],
        inverse[i], true_inverse[i], relative_diff[i]);
}

f_complex f(f_complex s)
{
    /* Return 1/(s-1)**2 */
    f_complex one = {1.0, 0.0};

    return (imsl_c_div(one,
        imsl_c_mul(imsl_c_sub(s, one), imsl_c_sub(s, one))));
}
```

### Output

```
          t            f_inv           true           diff
        0.5            0.824          0.824        1.5e-05
        1.5            6.722          6.723        1.0e-05
        2.5           30.456         30.456        5.6e-07
        3.5          115.906        115.904        1.8e-05
        4.5          405.054        405.077        5.8e-05
```

## Example 2

This example computes the inverse Laplace transform of the function $e^{-1/s}/s$, and prints the computed approximation, true transform value, and difference at five points. Additionally, the inverse is returned in user-supplied space, and a required accuracy for the inverse transform values is specified. The correct inverse transform is

$$J_0\left(2\sqrt{x}\right)$$

from Abramowitz and Stegun (1964).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
   f_complex f(f_complex);
   int n = 5;
   int i;
   float t[5];
   float true_inverse[5];
   float relative_diff[5];
   float inverse[5];
   Imsl_laplace_flow *indicators;

   /* Initialize t and compute inverse */
   for (i=0; i<n; i++)
       t[i] = (float)i + 0.5;

   imsl_f_inverse_laplace(f, 0.0, n, t,
       IMSL_PSEUDO_ACCURACY, 1.0e-6,
       IMSL_UNDER_OVERFLOW_INDICATORS, &indicators,
       IMSL_RETURN_USER, inverse,
       0);

   /* Compute true inverse, relative
   difference */
   for (i=0; i<n; i++) {
       true_inverse[i] = imsl_f_bessel_J0(2.0*sqrt(t[i]));
       relative_diff[i] = fabs((inverse[i] - true_inverse[i])/
           true_inverse[i]);
   }

   /* Print results, noting if any results
   overflowed or underflowed */
   printf("\t  T\t\t f_inv\t\t true\t\t diff\n");
   for (i=0; i<n; i++)
       if (indicators[i] == IMSL_NORMAL_TERMINATION)
           printf ("\t%5.1f\t\t%7.3f\t\t%7.3f\t\t%6.1e\n",
           t[i],
           inverse[i], true_inverse[i],
           relative_diff[i]);
       else
           printf("Overflow or underflow noted.\n");
}

f_complex f(f_complex s)
{
   /* Return (1/s)(exp(-1/s) */
   f_complex one = {1.0, 0.0};
```

```
    f_complex s_inverse;

    s_inverse = imsl_c_div(one, s);
    return (imsl_c_mul(s_inverse, imsl_c_exp(imsl_c_neg(s_inverse))));
}
```

### Output

```
          T          f_inv          true         diff
         0.5         0.559          0.559       2.1e-07
         1.5        -0.023         -0.023       8.5e-06
         2.5        -0.310         -0.310       9.6e-08
         3.5        -0.401         -0.401       7.4e-08
         4.5        -0.370         -0.370       6.4e-07
```

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# Chapter 7 Nonlinear Equations

## Functions

# Usage Notes

## Zeros of a Polynomial

A polynomial function of degree *n* can be expressed as follows:

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where $a_n \neq 0$. The function `imsl_f_zeros_poly` finds zeros of a polynomial with real coefficients using the Jenkins-Traub method.

## Zeros of a Function

The function `imsl_f_zeros_function` finds the real zeros of a real, continuous, univariate function. It uses a meta-algorithm based on partitioning the interval using a low-discrepancy sequence and a combination of Müller's method and Brent's method. This algorithm can find roots without requiring the user to bracket the root in an interval over which the function changes sign, as required by Brent's method, or give good guesses for the roots, as required by Müller's method.

The function `imsl_f_zero_univariate` finds a real zero of a real, continuous, univariate function. It uses an algorithm attributed to Dr. Fred T. Krogh, JPL, 1972. Tests have shown this algorithm to require fewer function evaluations, on average, than a number of other algorithms for finding a zero of a continuous function.

## Root of System of Equations

A system of equations can be stated as follows:

$$f_i(x) = 0, \text{ for } i = 1, 2, \ldots, n$$

where $x \in \mathfrak{R}_n$, and $f_i : \mathfrak{R}_n \rightarrow \mathfrak{R}$.The function `imsl_f_zeros_sys_eqn` uses a modified hybrid method due to M.J.D. Powell to find the zero of a system of nonlinear equations.

# zeros_poly

Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub, three-stage algorithm.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_f_zeros_poly (*int* ndeg, *float* coef[], ..., 0)

The type *d_complex* function is imsl_d_zeros_poly.

## Required Arguments

*int* ndeg (Input)
Degree of the polynomial.

*float* coef[] (Input)
Array with ndeg + 1 components containing the coefficients of the polynomial in increasing order by degree. The polynomial is coef[$n$] $z^n$ + coef [$n - 1$] $z_n n^{-1}$ + … + coef[0], where $n$ = ndeg.

## Return Value

A pointer to the complex array of zeros of the polynomial. To release this space, use imsl_free. If no zeros are computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_f_zeros_poly (*int* ndeg, *float* coef[],
IMSL_RETURN_USER, *f_complex* root[],
0)

## Optional Arguments

IMSL_RETURN_USER, *f_complex* root[] (Output)
Array with ndeg components containing the zeros of the polynomial.

## Description

The function `imsl_f_zeros_poly` computes the *n* zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ \dots \ + a_1 z + a_0$$

where the coefficients $a_i$ for $i = 0, 1, \dots, n$ are real and *n* is the degree of the polynomial.

The function `imsl_f_zeros_poly` uses the Jenkins-Traub, three-stage algorithm (Jenkins and Traub 1970; Jenkins 1975). The zeros are computed one at a time for real zeros or two at a time for a complex conjugate pair. As the zeros are found, the real zero, or quadratic factor, is removed by polynomial deflation.

## Examples

### Example 1

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where *z* is a complex variable.

```
#include <imsl.h>

#define NDEG   3

int main()
{
   f_complex      *zeros;
   static float   coeff[NDEG + 1] = {-2.0, 4.0, -3.0, 1.0};

   zeros = imsl_f_zeros_poly(NDEG, coeff, 0);

   imsl_c_write_matrix ("The complex zeros found are", 1, 3,
           zeros, 0);
}
```

### Output

```
                     The complex zeros found are
                   1                     2                     3
(        1,        0)  (        1,        1)  (        1,       -1)
```

### Example 2

The same problem is solved with the return option.

```
#include <imsl.h>

#define NDEG   3

int main()
{
   f_complex      zeros[3];
   static float   coeff[NDEG + 1] = {-2.0, 4.0, -3.0, 1.0};

   imsl_f_zeros_poly(NDEG, coeff,
                         IMSL_RETURN_USER, zeros, 0);

   imsl_c_write_matrix ("The complex zeros found are", 1, 3,
           zeros, 0);
}
```

### Output

```
                      The complex zeros found are
                    1                      2                      3
(         1,          0) (         1,          1) (         1,         -1)
```

## Warning Errors

| | |
|---|---|
| IMSL_ZERO_COEFF | The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem. |
| IMSL_FEWER_ZEROS_FOUND | Fewer than `ndeg` zeros were found. The root vector will contain the value for machine infinity in the locations that do not contain zeros. |

# zeros_poly (complex)

Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub, three-stage algorithm.

## Synopsis

*#include* <imsl.h>

*f_complex* *imsl_c_zeros_poly (*int* ndeg, *f_complex* coef[], ..., 0)

The type ***d_complex*** function is imsl_z_zeros_poly.

## Required Arguments

*int* ndeg (Input)
Degree of the polynomial.

*f_complex* coef[] (Input)
Array with ndeg + 1 components containing the coefficients of the polynomial in increasing order by degree. The degree of the polynomial is

$$\mathtt{coef}\,[n]\, z^{n} + \mathtt{coef}\,[n-1]\, z^{n-1} + \ldots + \mathtt{coef}\,[0]$$

where $n$ = ndeg.

## Return Value

A pointer to the complex array of zeros of the polynomial. To release this space, use imsl_free. If no zeros are computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*f_complex* *imsl_c_zeros_poly (*int* ndeg, *f_complex* coef[],

IMSL_RETURN_USER, *f_complex* root[],

0)

## Optional Arguments

IMSL_RETURN_USER, *f_complex* root[] (Output)
Array with ndeg components containing the zeros of the polynomial.

# Description

The function `imsl_c_zeros_poly` computes the $n$ zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients $a_i$ for $i = 0, 1, \ldots, n$ are complex and $n$ is the degree of the polynomial.

The function `imsl_c_zeros_poly` uses the Jenkins-Traub, three-stage complex algorithm (Jenkins and Traub 1970, 1972). The zeros are computed one at a time in roughly increasing order of modulus. As each zero is found, the polynomial is deflated to one of lower degree.

# Examples

## Example 1

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - (3 + 6i)\, z^2 - (8 - 12i)\, z + 10$$

where $z$ is a complex variable.

```
#include <imsl.h>

#define NDEG   3

int main()
{
   f_complex      *zeros;
   f_complex coeff[NDEG + 1] = { {10.0, 0.0},
                                 {-8.0, 12.0},
                                 {-3.0, -6.0},
                                 { 1.0, 0.0} };

   zeros = imsl_c_zeros_poly(NDEG, coeff, 0);

   imsl_c_write_matrix ("The complex zeros found are", 1, 3,
          zeros, 0);
}
```

### Output

```
                      The complex zeros found are
                    1                      2                      3
(         1,         1) (         1,         2) (         1,         3)
```

## Example 2

The same problem is solved with the return option.

```
#include <imsl.h>

#define NDEG   3

int main()
{
   f_complex      zeros[3];
   f_complex coeff[NDEG + 1] = { {10.0, 0.0},
                                  {-8.0, 12.0},
                                  {-3.0, -6.0},
                                  { 1.0, 0.0} };

   imsl_c_zeros_poly(NDEG, coeff, IMSL_RETURN_USER, zeros, 0);

   imsl_c_write_matrix ("The complex zeros found are", 1, 3,
       zeros, 0);
}
```

## Output

```
                       The complex zeros found are
                     1                      2                      3
(        1,         1)  (        1,         2)  (        1,         3)
```

## Warning Errors

| | |
|---|---|
| `IMSL_ZERO_COEFF` | The first several coefficients of the polynomial are equal to zero. Several of the last roots will be set to machine infinity to compensate for this problem. |
| `IMSL_FEWER_ZEROS_FOUND` | Fewer than `ndeg` zeros were found. The root vector will contain the value for machine infinity in the locations that do not contain zeros. |

# zero_univariate

Finds a zero of a real univariate function.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_zero_univariate (*float* fcn (), *float* *a, *float* *b, …, 0)

The type *double* function is imsl_d_zero_univariate.

## Required Arguments

*float* fcn (*float* x) (Input/Output)
User-supplied function to compute the value of the function of which the zero will be found.

**Arguments**

*float* x (Input)
The point at which the function is evaluated.

**Return Value**

The computed function value at the point x.

*float* *a (Input/Output)
See b.

*float* *b (Input/Output)
Two points at which the user-supplied function can be evaluated.
On input, if fcn(a) and fcn(b) are of opposite sign, the zero will be found in the interval [a, b] and on output b will contain the value of x at which fcn(x) = 0. If fcn(a) × fcn(b) > 0, and a ≠ b then a search along the *x* number line is initiated for a point at which there is a sign change and |b − a| will be used in setting the step size for the initial search. If a = b on entry then the search is started as described in the description section below.
On output, b is the abscissa at which |fcn(x)| had the smallest value. If fcn(b) ≠ 0 on output, a will contain the nearest abscissa to output b at which fcn(x) was evaluated and found to have the opposite sign from fcn(b).

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_f_zero_univariate (*float* fcn (), *float* *a, *float* *b, IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

IMSL_ERR_TOL, *float* err_tol,

IMSL_MAX_EVALS, *int* maxfn,

IMSL_N_EVALS, *int* *n_evals,

0)

## Optional Arguments

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)

*float* fcn (*float* x, *void* *data) (Input)

User supplied function to compute the value of the function of which the zero will be found, which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* x (Input)

The point at which the function is evaluated.

*void* *data (Input)

A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point x.

IMSL_ERR_TOL, *float* err_tol (Input)

Error tolerance. If err_tol > 0.0, the zero is to be isolated to an interval of length less than err_tol. If err_tol < 0.0, an x is desired for which |fcn(x)| is ≤ |err_tol|. If err_tol = 0.0, the iteration continues until the zero of fcn(x) is isolated as accurately as possible.

Default: err_tol = 0.0

IMSL_MAX_EVALS, *int* maxfn (Input)

An upper bound on the number of function evaluations required for convergence. Set maxfn to 0 if the number of function evaluations is to be unbounded.

Default: maxfn = 0

IMSL_N_EVALS, *int* *n_evals (Output)

The actual number of function evaluations used.

## Description

The function `imsl_f_zero_univariate` is based on the JPL Library routine `SZERO`. The algorithm used is attributed to Dr. Fred T. Krogh, JPL, 1972. Tests have shown `imsl_f_zero_univariate` to require fewer function evaluations, on average, than a number of other algorithms for finding a zero of a continuous function. Let $f$ be a continuous univariate function. `imsl_f_zero_univariate` will accept any two points `a` and `b` and initiate a search on the number line for an $x$ such that $f(x) = 0$ when there is no sign difference between $f$(`a`) and $f$(`b`). In either case, `b` is updated with a new value on each successive iteration. The algorithm description follows.

When $f$(`a`) × $f$(`b`) >0 at the initial point, iterates for $x$ are generated according to the formula $x = x_{min} + (x_{min} - x_{max}) \times \rho$, where the subscript "min" is associated with the $(f, x)$ pair that has the smallest value for $|f|$, the subscript "max" is associated with the $(f, x)$ pair that has the largest value for $|f|$, and $\rho$ is 8 if $r = f_{min}/(f_{max} - f_{min}) \geq 8$, else $\rho = \max(\kappa/4, r)$, where $\kappa$ is a count of the number of iterations that have been taken without finding $f$s with opposite signs. If `a` and `b` have the same value initially, then the next $x$ is a distance $0.008 + |x_{min}|/4$ from $x_{min}$ taken toward 0. (If `a` = `b` = 0, the next $x$ is -.008.)

Let $x_1$ and $x_2$ denote the first two $x$ values that give $f$ values with different signs. Let $\alpha < \beta$ be the two values of $x$ that bracket the zero as tightly as is known. Thus $\alpha = x_1$ or $\alpha = x_2$ and $\beta$ is the other when computing $x_3$. The next point, $x_3$, is generated by treating $x$ as the linear function $q(f)$ that interpolates the points $(f(x_1), x_1)$ and $(f(x_2), x_2)$, and then computing $x_3 = q(0)$, subject to the condition that $\alpha + \varepsilon \leq x_3 \leq \beta - \varepsilon$, where $\varepsilon = 0.875 \times max($`err_tol`, machine precision). (This condition on $x_3$ with updated values for $\alpha$ and $\beta$ is also applied to future iterates.)

Let $x_4, x_5, \ldots, x_m$ denote the abscissae on the following iterations. Let $a = x_m$, $b = x_{m-1}$, and $c = x_{m-2}$. Either $\alpha$ or $\beta$ (defined as above) will coincide with $a$, and $\beta$ will frequently coincide with either $b$ or $c$. Let $p(x)$ be the quadratic polynomial in $x$ that passes through the values of $f$ evaluated at $a$, $b$, and $c$. Let $q(f)$ be the quadratic polynomial in $f$ that passes through the points $(f(a), a)$, $(f(b), b)$, and $(f(c), c)$.

Let $\zeta = \alpha$ or $\beta$, selected so that $\zeta \neq a$. If the sign of $f$ has changed in the last 4 iterations and $p'(a) \times q'(f(a))$ and $p'(\zeta) \times q'(f(\zeta))$ are both in the interval [1/4, 4], then $x$ is set to $q(0)$. (Note that if $p$ is replaced by $f$ and $q$ is replaced by $x$, then both products have the value 1.) Otherwise $x$ is set to $a - (a - \zeta)(\phi/(1+\phi))$, where $\phi$ is selected based on past behavior and is such that $0 < \phi$. If the sign of $f$() does not change for an extended period, $\phi$ gets large.

## Example

This example finds a zero of the function

$$f(x) = x^2 + x - 2$$

in the interval [ − 10.0, 0.0].

```
#include <imsl.h>
#include <stdio.h>
```

```
float fcn (float x);

int main() {
   int n_evals;
   float a=-10.0, b=0.0;

   imsl_f_zero_univariate(fcn, &a, &b, IMSL_N_EVALS, &n_evals, 0);

   printf("The best approximation to the zero of f is");
   printf(" equal to %6.3f\n", b);
   printf("The number of function evaluations required");
   printf(" was %d\n",n_evals);
}


float fcn (float x) {
   return x*x + x - 2.0;
}
```

### Output

```
The best approximation to the zero of f is equal to -2.0
The number of function evaluations required was 10
```

## Fatal Errors

| | |
|---|---|
| `IMSL_ERROR_TOL_NOT_SATISFIED` | The error tolerance criteria was not satisfied. "`b`" contains the abscissa at which "`|fcn(x)|`" had the smallest value. |
| `IMSL_DISCONTINUITY_IDENTIFIED` | Apparently "`fcn`" has a discontinuity between "`a`" = # and "`b`" = #. No zero has been identified. |
| `IMSL_ZERO_NOT_FOUND` | "`fcn(a)*fcn(b)`" > 0 where "`a`" = #" and "`b`" = #, but the algorithm is unable to identify function values with opposite signs. |
| `IMSL_MAX_FCN_EVAL_EXCEEDED` | The maximum number of function evaluations, "`maxfn`" = #, has been exceeded. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# zeros_function

Finds the real zeros of a real, continuous, univariate function.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_zeros_function (*float* fcn(), …, 0)

The type *double* function is imsl_d_zeros_function.

## Required Arguments

*float* fcn (*float* x) (Input/Output)
User-supplied function to compute the value of the function of which the zeros will be found.

**Argument**

*float* x (Input)
The point at which the function is evaluated.

**Return Value**
The computed function value at the point x.

## Return Value

A pointer to an array containing the zeros of the function. The zeros are in increasing order. If fewer than the requested number of zeros were found, the final entries are set to NaN. To release this space, use imsl_free. If there is a fatal error, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_zeros_function (*float* fcn (),

IMSL_NUM_ROOTS, *int* num_roots,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

IMSL_NUM_ROOTS_FOUND, *int* *num_roots_found,

IMSL_N_EVALS, *int* *n_evals,

IMSL_BOUND, *float* a, *float* b,

IMSL_MAX_EVALS, *int* max_evals,

IMSL_XGUESS, *float* xguess[],

IMSL_ERR_ABS, *float* err_abs,

IMSL_ERR_X, *float* err_x,

IMSL_TOLERANCE_MULLER, *float* tolerance_muller,

IMSL_MIN_SEPARATION, *float* min_separation,

IMSL_XSCALE, *float* xscale,

IMSL_RETURN_USER, *float* x[],

0)

## Optional Arguments

IMSL_NUM_ROOTS, *int* num_roots (Input)
The number of zeros to be found.
Default: num_roots = 1.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)
User supplied function to compute the value of the function of which the zeros will be found, which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float* x (Input)
The point at which the function is evaluated.

*void* *data (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point x.

IMSL_NUM_ROOTS_FOUND, *int* *num_roots_found (Output)
The number of zeros actually found.

IMSL_N_EVALS, *int* *n_evals (Output)
The actual number of function evaluations used.

IMSL_BOUND, *float* a, *float* b (Input)
The closed interval in which to search for the roots. The function must be defined for all values in this interval.
Default: The search for the roots is not bounded.

`IMSL_MAX_EVALS`, *int* `max_evals` (Input)
The maximum number of function evaluations allowed. Once this limit is reached, the roots found are returned.
Default: `max_evals` = 100

`IMSL_XGUESS`, *float* `xguess[]` (Input)
Array with `num_roots` components containing initial guesses for the zeros. If a bound on the zeros is also given, the guesses must satisfy the bound condition.

`IMSL_ERR_ABS`, *float* `err_abs` (Input)
A convergence criterion. A root is accepted if the absolute value of the function at the point is less than or equal to `err_abs`.
Default: `err` _abs= 100 $\varepsilon$, where $\varepsilon$ is the machine precision.

`IMSL_ERR_X`, *float* `err_x` (Input)
A convergence criterion. A root is accepted if it is bracketed within an interval of length `err_x`.
Default: `err_x` = 100 $\varepsilon$ / `xscale`, where $\varepsilon$ is the machine precision.

`IMSL_TOLERANCE_MULLER`, *float* `tolerance_muller` (Input)
Müller's method is started if, during refinement, a point is found for which the absolute value of the function is less than `tolerance_muller` and the point is not near an already discovered root. If `tolerance_muller` is less than or equal to zero Müller's method is never used.
Default: `tolerance_muller` = $\varepsilon$ / `err_abs`, where $\varepsilon$ is the machine precision. With the default value of `err_abs`, this equals 0.01.

`IMSL_MIN_SEPARATION`, *float* `min_separation` (Input)
The minimum separation between accepted roots. If two points both satisfy the convergence criteria, but are within `min_separation` of each other, only one of the roots is accepted.
Default: `min_separation` = $\varepsilon^{1/2}$/`xscale`, where $\varepsilon$ is the machine precision.

`IMSL_XSCALE`, *float* `xscale` (Input)
The scaling in the *x*-coordinate. The absolute value of the roots divided by `xscale` should be about one.
Default: `xscale` = 1.0

`IMSL_RETURN_USER`, *float* `x[]` (Output)
Array with `num_root` components containing the computed zeros.

# Description

The function `imsl_f_zeros_function` computes `num_roots` real zeros of a real, continuous, univariate function. The search for the zeros of the function can be limited to a specified interval, or extended over the entire real line. The code is generally more efficient if an interval is specified. The user supplied function must return valid results for all values in the specified interval. If no interval is given, the user-supplied function must return valid results for all real numbers.

The function has two convergence criteria. The first criterion accepts a root, $x$, if

$$|f(x)| \le \tau$$

where $\tau$ = `err_x`.

The second criterion accepts a root if it is known to be inside of an interval of length at most `err_abs`.

A root is accepted if it satisfies either criteria and is not within `min_separation` of another accepted root.

If initial guesses for the roots are given, Müller's method (Müller 1956) is used for each of these guesses. For each guess, the Müller iteration is stopped if the next step would be outside of the bound, if given. The iteration is also stopped if it cannot make further progress in finding a root.

If no guess for the zeros were given, or if Müller's method with the guesses did not find the requested number of roots, a meta-algorithm, combining Müller's and Brent's methods, is used. Müller's method is used primarily to find the roots of functions, such as $f(x) = x^2$, where the function does not cross the $y$=0 line. Brent's method is used to find other types of roots.

The meta-algorithm successively refines the interval using a one-dimensional Faure low-discrepancy sequence. If the optional argument `IMSL_BOUND` is used to specify a bounded interval, [a,b], the Faure sequence is scaled from (0,1) to (a,b).

If no bound on the function's domain is given, the entire real line must be searched for roots. In this case the Faure sequence is scaled from (0, 1) to (-∞,+∞) using the mapping

$$h(u) = \text{xscale} \cdot \tan(\pi((u - 1/2))$$

where `xscale` is given by the optional argument `IMSL_XSCALE`.

At each step of the iteration the next point in the Faure sequence is added to the list of breakpoints defining the subintervals. Call the points $x_0=a$, $x_1=b$, $x_2$, $x_3$, .... The new point, $x_s$ splits an existing subinterval, [$x_p$, $x_q$].

The function is evaluated at $x_s$. If its value is small enough, specifically if

$$|f(x_s)| < \text{tolerance\_muller}$$

then Müller's method is used with $x_p$, $x_q$ and $x_s$ as starting values. If a root is found, it is added to the list of roots. If more roots are required, the new Faure point is used.

If Müller's method did not find a root using the new point, the function value at the point is compared with the function values at the endpoints of the subinterval it divides. If $f(x_p)f(x_s) < 0$ and no root has previously been found in $[x_p, x_s]$ then Brent's method is used to find a root in this interval. Similarly, if the function changes sign over the interval $[x_s, x_q]$, and a root has not already been found in the subinterval, Brent's method is used there.

# Examples

## Example 1

This example finds a real zero of the function.

$$f(x) = e^x - 3$$

```
#include <imsl.h>
#include <math.h>
#include <stdio.h>

float          fcn(float x);

int main()
{
   float      *x;

   x = imsl_f_zeros_function (fcn, 0);
   printf("fcn(%6.3f) = %12.3e\n",
       x[0], fcn(x[0]));
}


float fcn(float x)
{
   return exp(x) - 3.0;
}
```

### Output

```
fcn( 1.099) = -6.378e-006
```

## Example 2

This example finds two real zeros of function

$$f(x) = e^{-x}\sqrt{x} - 0.3$$

on the interval [0,20].

```
#include <imsl.h>
#include <math.h>

float          fcn(float x);

int main()
{
   float      *x;
```

```
   int         n_found;

   x = imsl_f_zeros_function (fcn,
                         IMSL_NUM_ROOTS, 2,
                         IMSL_BOUND, 0.0, 20.0,
                         IMSL_NUM_ROOTS_FOUND, &n_found,
                         0);
   imsl_f_write_matrix ("x", 1, n_found, x, 0);
}


float fcn(float x)
{
   return sqrt(x)*exp(-x) - 0.3;
}
```

### Output

```
           x
        1            2
    0.113        1.356
```

### Warning Errors

| | |
|---|---|
| IMSL_ZEROS_MAX_EVALS_EXCEEDED | The maximum number of function evaluations allowed has been exceeded. Any zeros found are returned. |

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm. User flag = "#". |

# zeros_sys_eqn

Solves a system of *n* nonlinear equations *f*(*x*) = 0 using a modified Powell hybrid algorithm.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_zeros_sys_eqn (*void* fcn (), *int* n, …, 0)

The type *double* function is imsl_d_zeros_sys_eqn.

## Required Arguments

*void* fcn (*int* n, *float* x[], *float* f[]) (Input/Output)
User-supplied function to evaluate the system of equations to be solved, where n is the size of x and f, x is the point at which the functions are evaluated, and f contains the computed function values at the point x.

*int* n (Input)
The number of equations to be solved and the number of unknowns.

## Return Value

A pointer to the vector *x* that is a solution of the system of equations. To release this space, use imsl_free. If no solution can be computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_zeros_sys_eqn (*void* fcn (), *int* n,

IMSL_XGUESS, *float* xguess[],

IMSL_JACOBIAN, *void* jacobian(),

IMSL_ERR_REL, *float* err_rel,

IMSL_MAX_ITN, *int* max_itn,

IMSL_RETURN_USER, *float* x[],

IMSL_FNORM, *float* *fnorm,

IMSL_FCN_W_DATA, *void* fcn (), *void* *data, IMSL_JACOBIAN_W_DATA, *void* jacobian(), *void* *data,

0)

## Optional Arguments

IMSL_XGUESS, *float* xguess[] (Input)

Array with *n* components containing the initial estimate of the root.

Default: xguess = 0

IMSL_JACOBIAN, *void* jacobian (*int* n, *float* x[], *float* fjac[]) (Input/Output)

User-supplied function to evaluate the Jacobian, where n is the number of components in x, x is the point at which the Jacobian is evaluated, and fjac is the computed $n \times n$ Jacobian matrix at the point x. Note that each derivative $\partial f_i / \partial x_j$ should be returned in fjac[(i-1)×n+j-1].

IMSL_ERR_REL, *float* err_rel (Input)

Stopping criterion. The root is accepted if the relative error between two successive approximations to this root is less than err_rel.

Default: $err\_rel = \sqrt{\varepsilon}$ where ε is the machine precision

IMSL_MAX_ITN, *int* max_itn (Input)

The maximum allowable number of iterations.

Default: max_itn = 200

IMSL_RETURN_USER, *float* x[] (Output)

Array with n components containing the best estimate of the root found by f_zeros_sys_eqn.

IMSL_FNORM, *float* *fnorm (Output)

Scalar with the value

$$f_1^2 + \ldots + f_n^2$$

at the point x.

IMSL_FCN_W_DATA, *void* fcn (*int* n, *float* x[], *float* f[], *void* *data), *void* *data (Input)

User supplied function to evaluate the system of equations to be solved, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to this manual for more details.

## Description

The function `imsl_f_zeros_sys_eqn` is based on the MINPACK subroutine HYBRDJ, which uses a modification of the hybrid algorithm due to M.J.D. Powell. This algorithm is a variation of Newton's method, which takes precautions to avoid undesirable large steps or increasing residuals. For further description, see Moré et al. (1980).

## Examples

### Example 1

The following 2 × 2 system of nonlinear equations

$$f_1(x) = x_1 + x_2 - 3$$
$$f_2(x) = x_1^2 + x_2^2 - 9$$

is solved.

```
#include <imsl.h>
#include <stdio.h>

#define N       2

void            fcn(int, float[], float[]);

int main()
{
   float      *x;

   x = imsl_f_zeros_sys_eqn(fcn, N, 0);
   imsl_f_write_matrix("The solution to the system is", 1, N, x, 0);
}


void fcn(int n, float x[], float f[])
{
   f[0] = x[0] + x[1] - 3.0;
   f[1] = x[0]*x[0] + x[1] * x[1] - 9.0;
}
```

### Output

```
The solution to the system is
           1          2
           0          3
```

## Example 2

The following 3 × 3 system of nonlinear equations

$$f_1(x) = x_1 + e^{x_1 - 1} + (x_2 + x_3)^2 - 27$$
$$f_2(x) = e^{x_2 - 2} / x_1 + x_3^2 - 10$$
$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7$$

is solved with the initial guess (4.0, 4.0, 4.0).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define N       3

void            fcn(int, float[], float[]);

int main()
{
   int        maxitn = 100;
   float      *x, err_rel = 0.0001, fnorm;
   float      xguess[N] = {4.0, 4.0, 4.0};

   x = imsl_f_zeros_sys_eqn(fcn, N,
                            IMSL_ERR_REL, err_rel,
                            IMSL_MAX_ITN, maxitn,
                            IMSL_XGUESS, xguess,
                            IMSL_FNORM, &fnorm,
                            0);
   imsl_f_write_matrix("The solution to the system is", 1, N, x, 0);
   printf("\nwith fnorm = %5.4f\n", fnorm);
}


void fcn(int n, float x[], float f[])
{
   f[0] = x[0] + exp(x[0] - 1.0) + (x[1] + x[2]) * (x[1] + x[2]) - 27.0;
   f[1] = exp(x[1] - 2.0) / x[0] + x[2] * x[2] - 10.0;
   f[2] = x[2] + sin(x[1] - 2.0) + x[1] * x[1] - 7.0;
}
```

### Output

```
The solution to the system is
     1          2          3
     1          2          3

with fnorm = 0.0000
```

## Warning Errors

| | |
|---|---|
| `IMSL_TOO_MANY_FCN_EVALS` | The number of function evaluations has exceeded `max_itn`. A new initial guess may be tried. |
| `IMSL_NO_BETTER_POINT` | Argument `err_rel` is too small. No further improvement in the approximate solution is possible. |
| `IMSL_NO_PROGRESS` | The iteration has not made good progress. A new initial guess may be tried. |

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# Chapter 8 Optimization

## Functions

# Usage Notes

## Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in R^n} f(x)$$

where $f : \Re^n \rightarrow \Re$ 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:

$$\begin{aligned} &\min_{\mathrm{x} \in \mathrm{R}^n} f(x) \\ &\text{subject to} \quad \begin{aligned} A_1 x &= b_1 \\ A_2 x &\le b_2 \end{aligned} \end{aligned}$$

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_linear_programming uses an active set strategy to solve linear programming problems, and is intended as a replacement for the function imsl_f_lin_prog. The two functions have similar interfaces, which should help facilitate migration from imsl_f_lin_prog to imsl_f_linear_programming. In general, the function imsl_f_linear_programming should be expected to perform more efficiently than imsl_f_lin_prog. Both imsl_f_linear_programming and imsl_f_lin_prog are intended for use with small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

Function imsl_d_sparse_lin_prog uses an infeasible primal-dual interior-point method to solve sparse linear programming problems of all sizes. The constraint matrix is stored in sparse coordinate storage format.

The function imsl_f_quadratic_prog is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then imsl_f_quadratic_prog modifies it to be positive definite. In this case, output should be interpreted with care because the problem has been changed slightly. Here, the Hessian of $f(x)$ is defined to be the $n \times n$ matrix

$$\nabla^2 f(x) = \left[ \frac{\partial^2}{\partial x_i \partial x_j} f(x) \right]$$

Function imsl_d_sparse_quadratic_prog uses an infeasible primal-dual interior-point method to solve sparse convex quadratic programming problems of all sizes. The constraint matrix and the Hessian are stored in sparse coordinate storage format.

## Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\begin{aligned}
&\min_{x \in R^n} f(x) \\
&\text{subject to } g_i(x) = 0 \text{ for } i = 1,2,\ldots m_1 \\
&g_i(x) \geq 0 \text{ for } i = m_1 + 1,\ldots m
\end{aligned}$$

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 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.

## Return Values from User-Supplied Functions

All values returned by user-supplied functions must be valid real numbers. It is the user's responsibility to check that the values returned by a user-supplied function do not contain NaN, infinity, or negative infinity values.

```
#include <imsl.h>
#include <math.h>
void fcn(int, int, float[], float[]);
void main()
{
int m=3, n=1;
float *result, fx[3];
float xguess[]={1.0};
result = imsl_f_nonlin_least_squares(fcn, m, n, IMSL_XGUESS, xguess, 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++)
{
/*      check for x=0
         do not want to return infinity to nonlin_least_squares    */
  if (x[0] == 0.0) {
     f[i] = 10000.;
  } else {
     f[i] = t[i]/x[0] - y[i];
  }
}

} /* End of function */
```

# 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 Passing Data to User-Supplied Functions in the introduction to 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,

$$x_1, x_2, x_3, \text{ with } x_1 < x_2 < x_3, f(x_1) \geq f(x_2), \text{ and } f(x_2) \leq f(x_3).$$

There are three main rules in the technique for choosing the new $x$ from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter $\eta$, 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 $\varepsilon$ 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 $\varepsilon$ 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 $\delta$ of x, where $\delta$ = 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);

int 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);
int 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. |

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# 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: 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: 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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to 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 sof 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 $\varepsilon_c = \max\{1.0, |x_c|\}\, \varepsilon$, $\varepsilon$ is an error tolerance, and $\varepsilon_g$ 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);

int 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);

int 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. |

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# min_uncon_golden

Finds the minimum point of a nonsmooth function of a single variable using the golden section search method.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_min_uncon_golden (*float* fcn (), *float* a, *float* b, …, 0)

The type *double* function is imsl_d_min_uncon_golden.

## Required Arguments

*float* fcn (*float* x) (Input)
User-supplied function, *f*(*x*), to be minimized.

*float x* (Input)
The point at which the function is evaluated.

**Return Value**

The computed function value at the point x.

*float* a (Input)
The lower endpoint of the interval in which the minimum of *f* is to be located.

*float* b (Input)
The upper endpoint of the interval in which the minimum of *f* is to be located.

## Return Value

The approximate minimum point of the function *f* on the original interval [a,b]. If no value can be computed, NaN is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_min_uncon_golden (*float* fcn (), *float* a, *float* b,

IMSL_TOLERANCE, *float* tol,

IMSL_FCN_W_DATA, *float* fcn (), *void* *data,

IMSL_LOWER_ENDPOINT, *float* *lower,

IMSL_UPPER_ENDPOINT, *float* *upper,

0)

## Optional Arguments

IMSL_TOLERANCE, *float* tol (Input)
The allowable length of the final subinterval containing the minimum point.
Default: tol = $\sqrt{\varepsilon}$, where ε is the machine precision.

IMSL_FCN_W_DATA, *float* fcn (*float* x, *void* *data), *void* *data (Input)

*float* fcn (*float* x, *void* *data)
User-supplied function, *f*(*x*), to be minimized, which also accepts a pointer to data that is supplied by the user. See Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Arguments**

*float x* (Input)
The point at which the function is evaluated.

*void* *data (Input)
A pointer to the data to be passed to the user-supplied function.

**Return Value**

The computed function value at the point x.

*void* data (Input)
A pointer to the data to be passed to the user-supplied function.

IMSL_LOWER_ENDPOINT, *float* *lower (Output)
The lower endpoint of the interval in which the minimum of *f* is located.

IMSL_UPPER_ENDPOINT, *float* *upper (Output)
The upper endpoint of the interval in which the minimum of *f* is located.

## Description

The function imsl_f_min_uncon_golden uses the ***golden section search*** technique to compute to the desired accuracy the independent variable value that minimizes a function of one independent variable, where a known finite interval contains the minimum and where the function is unimodal in the same known finite interval.

Let $\tau$ =tol. The number of iterations required to compute the minimizing value to accuracy $\tau$ is the greatest integer less than or equal to

$$\frac{\ln(\tau / (b-a))}{\ln(1-c)} + 1$$

where $a$ and $b$ define the interval and

$$c = (3 - \sqrt{5}) / 2$$

The first two test points are $v_1$ and $v_2$ that are defined as

$$v_1 = a + c(b - a), \text{ and } v_2 = b - c(b - a)$$

If $f(v_1) < f(v_2)$, then the minimizing value is in the interval $(a, v_2)$. In this case, $b \leftarrow v_2$, $v_2 \leftarrow v_1$, and $v_1 \leftarrow a + c(b - a)$. If $f(v_1) \geq f(v_2)$, the minimizing value is in $(v_1, b)$. In this case, $a \leftarrow v_1$, $v_1 \leftarrow v_2$, and $v_2 \leftarrow b - c(b - a)$.

The algorithm continues in an analogous manner where only one new test point is computed at each step. This process continues until the desired accuracy $\tau$ is achieved. The point, `xmin`, producing the minimum value for the current iteration is returned.

Mathematically, the algorithm always produces the minimizing value to the desired accuracy, however, numerical problems may be encountered. If $f$ is too flat in part of the region of interest, the function may appear to be constant to the computer in that region. The user may rectify the problem by relaxing the requirement on $\tau$, modifying (scaling, etc.) the form of $f$ or executing the program in a higher precision.

# Remarks

1. On exit from `imsl_f_min_uncon_golden` without any error messages, the following conditions hold:

   $$\texttt{(upper-lower)} \leq \texttt{tol}$$
   $$\texttt{lower} \leq \texttt{xmin} \text{ and } \texttt{xmin} \leq \texttt{upper}$$
   $$\texttt{f(xmin)} \leq \texttt{f(lower)} \text{ and } \texttt{f(xmin)} \leq \texttt{f(upper)}$$

2. On exit from `imsl_f_min_uncon_golden` with `IMSL_NOT_UNIMODAL` error, the following conditions hold:

   $$\texttt{lower} \leq \texttt{xmin} \text{ and } \texttt{xmin} \leq \texttt{upper}$$
   $$\texttt{f(xmin)} \geq \texttt{f(lower)} \text{ and } \texttt{f(xmin)} \geq \texttt{f(upper)} \text{ (only one equality can hold)}$$

   Further analysis of the function $f$ is necessary in order to determine whether it is not unimodal in the mathematical sense or whether it appears to be not unimodal to the routine due to rounding errors, in which case the `lower`, `upper`, and `xmin` returned may be acceptable.

# Example

A minimum point of $3x^2 - 2x + 4$ is found.

```
#include <imsl.h>
#include <stdio.h>

float fcn(float);
```

```
int main () {
  float a = 0.0e0, b = 5.0e0, tol = 1.0e-3, lower,
    upper, xmin, fx;
  xmin = imsl_f_min_uncon_golden (fcn, a, b,
    IMSL_TOLERANCE, tol,
    IMSL_LOWER_ENDPOINT, &lower,
    IMSL_UPPER_ENDPOINT, &upper,
    0);
  fx = fcn(xmin);
  printf ("The minimum is at: %8.3f\n", xmin);
  printf ("The function value is: %8.3f\n", fx);
  printf ("The final interval is: (%8.3f, %8.3f)\n",
    lower, upper);
}

float fcn(float x) {
  return 3.0e0*x*x - 2.0e0*x + 4.0e0;
}
```

### Output

```
The minimum is at:    0.333
The function value is:    3.667
The final interval is: (   0.333,    0.334)
```

## Warning Errors

| | |
|---|---|
| `IMSL_TOL_TOO_SMALL` | `tol` is too small to be satisfied.. |

## Fatal Errors

| | |
|---|---|
| `IMSL_NOT_UNIMODAL` | Due to rounding errors, the function does not appear to be unimodal. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# min_uncon_multivar

more...

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 imsl_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_, *float* xscale [],

IMSL_FSCALE, *float* fscale,

IMSL_GRAD_TOL, *float* grad_tol,

IMSL_STEP_TOL, *float* step_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

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\max\left(|f(x)|, f_s\right)}$$

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

$$\frac{|x_i - y_i|}{\max\left(|x_i|, 1/s_i\right)}$$

where $s$ = xscale.

Default: step_tol = $\varepsilon^{2/3}$

IMSL_MAX_STEP, *float* max_step (Input)

Maximum allowable step size.

Default: max_step = 1000 max ($\varepsilon_1$, $\varepsilon_2$) where,

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} \left(s_i t_i\right)^2}$$

$\varepsilon_2 = \|s\|^2$, $s$ = 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

$$\max\left(|f(t)|, f_s\right) * s_i^2$$

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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to 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:

$$\min_{x \in R^n} f(x)$$

Given a starting point $x_c$, the search direction is computed according to the formula

$$d = -B - 1gc$$

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

$$f(x_n) \le f(x_c) + \alpha g^T d, \alpha \in (0, 0.5)$$

Finally, the optimality condition $\|g(x)\| \le \varepsilon$ is checked where $\varepsilon$ 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.

On some platforms, `imsl_f_min_uncon_multivar` can evaluate the user-supplied functions `fcn` and `grad` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables

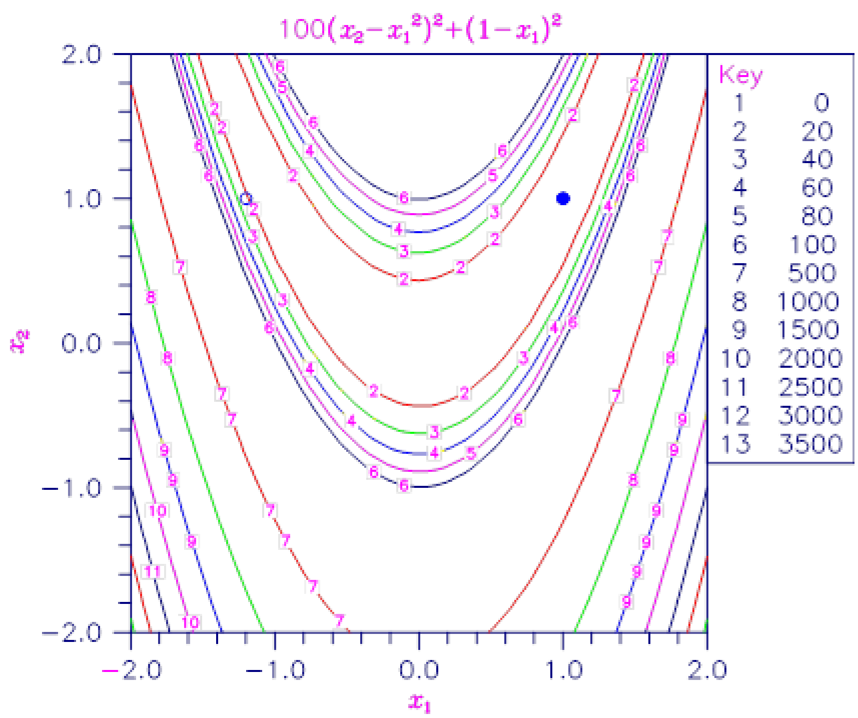



Figure 8.6 — Plot of the Rosenbrock Function

# 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>

int main()
{
   int             i, n=2;
   float           *result, fx;
   static float    rosbrk(int, float[]);

   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
   /* 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 Plot of the Rosenbrock Function.

```
#include <stdio.h>
#include <imsl.h>

int main()
{
    int              i, n=2;
    float            *result, fx;
    float   rosbrk(int, float[]);
    void    rosgrd(int, float[], float[]);
    static float   xguess[2] = {-1.2e0, 1.0e0};
    static float   grad_tol = .0001;

    imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    /* 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_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. |
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# min_uncon_polytope

Minimizes a function of n variables using a direct search polytope algorithm.

## Synopsis

*#include* <imsl.h>

*float* *imsl_f_min_uncon_polytope (*void* fcn(), *int* n, …, 0)

The type *double* function is imsl_d_min_uncon_polytope.

## Required Arguments

*void* fcn (*int* n, *float* x[], *float* *f) (Input)
User-supplied function to evaluate the function to be minimized.

**Arguments**

*int* n (Input)
Length of x.

*float* x[] (Input)
Array of length n at which point the function is evaluated.

*float* *f (Output)
The computed function value at the point x.

*int* n (Input)
Dimension of the problem.

## Return Value

An array of length n containing the best estimate of the minimum found. To release this space, use imsl_free. If no solution was computed, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_min_uncon_polytope (*void* fcn(), *int* n,

IMSL_XGUESS, *float* xguess[],

IMSL_TOLERANCE, *float* ftol,

IMSL_MAX_FCN, *int* *maxfcn,

IMSL_SIDE_LENGTH, *float* *s,

IMSL_FVALUE, *float* *fvalue,

IMSL_RETURN_USER, *float* x[],

IMSL_FCN_W_DATA, *void* fcn(), *void* *data,

0)

## Optional Arguments

IMSL_XGUESS, *float* xguess[] (Input)

An array of length n which contains an initial guess to the minimum.

Default: xguess = 0.0

IMSL_TOLERANCE, *float* ftol (Input)

The error tolerance used is based on two convergence criteria.

**First convergence criterion**: The algorithm stops when a relative error in the function values is less than ftol, i.e. when (fcn(worst) − fcn(best)) < ftol * (1 + fabs(fcn(best))) where fcn(worst) and fcn(best) are the function values of the current worst and best points, respectively.

**Second convergence criterion**: The algorithm stops when the standard deviation of the function values at the n + 1 current points is less than ftol.

If the routine terminates prematurely, try again with a smaller value for ftol.

Default: ftol = 1.e-5 in single precision and 1.e-10 in double precision.

IMSL_MAX_FCN, *int* *maxfcn (Input/Output)

On input, maximum allowed number of function evaluations. On output, actual number of function evaluations needed.

Default: maxfcn = 300

IMSL_SIDE_LENGTH, *float* *s (Input/Output)

On input, real scalar containing the length of each side of the initial simplex. If no reasonable information about s is known, s could be set to a number less than or equal to zero and imsl_f_min_uncon_polytope will generate the starting simplex from the initial guess with a random number generator. On output, the average distance from the final vertices to their centroid; see Remark 2.

Default: s = 0.0

IMSL_FVALUE, *float* *fvalue (Output)

Function value at the computed solution.

IMSL_RETURN_USER, *float* x[] (Output)

Array with n 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 evaluate 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 routine `imsl_f_min_uncon_polytope` uses the polytope algorithm to find a minimum point of a function $f(x)$ of $n$ variables. The polytope method is based on function comparison; no smoothness is assumed. It starts with $n + 1$ points $x_1, x_2, \ldots, x_{n+1}$. At each iteration, a new point is generated to replace the worst point $x_j$, which has the largest function value among these $n + 1$ points. The new point is constructed by the following formula:

$$x_k = c + \alpha(c - x_j)$$

where

$$c = \frac{1}{n}\sum_{i \neq j} x_i$$

and $\alpha$ ($\alpha > 0$) is the ***reflection coefficient***.

When $x_k$ is a best point, that is $f(x_k) \le f(x_i)$ for $i = 1, \ldots, n + 1$, an expansion point is computed $x_e = c + \beta(x_k - c)$ where $\beta(\beta > 1)$ is called the ***expansion coefficient***. If the new point is a worst point, then the polytope would be contracted to get a better new point. If the contraction step is unsuccessful, the polytope is shrunk by moving the vertices halfway toward the current best point. This procedure is repeated until one of the following stopping criteria is satisfied:

**Criterion 1**:

$$f_{worst} - f_{best} \le \varepsilon_f (1. + |f_{best}|)$$

**Criterion 2**:

$$\sqrt{\frac{1}{n+1}\sum_{i=1}^{n+1}(f_i - \bar{f})^2} \le \varepsilon_f$$

where $f_i = f(x_i)$, $f_j = f(x_j)$, $\varepsilon_f$ is a given tolerance and

$$\bar{f} = \frac{\sum_{j=1}^{n+1} f_j}{n+1}$$

For a complete description, see Nelder and Mead (1965) or Gill et al. (1981).

## Remarks

1. Since imsl_f_min_uncon_polytope uses only function value information at each step to determine a new approximate minimum, it could be quite inefficient on smooth problems compared to other methods, such as those implemented in routine imsl_f_min_uncon_multivar that takes into account derivative information at each iteration. Hence, routine imsl_f_min_uncon_polytope should be used only as a last resort. Briefly, a set of n + 1 points in an n-dimensional space is called a simplex. The minimization process iterates by replacing the point with the largest function value with a new point with a smaller function value. The iteration continues until all the points cluster sufficiently close to a minimum.

2. The value returned in s is useful for assessing the flatness of the function near the computed minimum. The larger its value for a given value of ftol, the flatter the function tends to be in the neighborhood of the returned point.

## Example

The function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2$$

is minimized and the solution is printed.

```
#include <imsl.h>
#include <stdio.h>

void fcn(int n, float x[], float *f);

#define N 2
int main() {
    float xguess[N] = {-1.2, 1.0};
    float *x, fvalue;
    float ftol = 1.0e-7, s = 1.0;

    x = imsl_f_min_uncon_polytope(fcn, N,
        IMSL_XGUESS, xguess,
        IMSL_TOLERANCE, ftol,
        IMSL_FVALUE, &fvalue,
        IMSL_SIDE_LENGTH, &s,
        0);

    printf("The best estimate for the minimum value of the\n");
    printf("function is x = (%4.2f, %4.2f) with\n", x[0], x[1]);
    printf("function value fvalue = %12.6e\n", fvalue);

}
```

```
void fcn(int n, float x[], float *f)
{
    float t1, t2;
    t1 = x[0]*x[0]-x[1];
    t2 = 1.0-x[0];
    *f = 100.0*t1*t1 + t2*t2;

}
```

### Output

```
The best estimate for the minimum value of the
function is x = (1.00, 1.00) with
function value fvalue = 2.126065e-007
```

## Fatal Errors

| | |
|---|---|
| `IMSL_FCN_EVAL_EXCEEDED_MAXFCN` | Maximum number of function evaluations exceeded. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm. User flag = "#". |

# nonlin_least_squares

more...

more...

Solves 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 imsl_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[(i-1)*fjac_col_dim+j-1]

IMSL_XSCALE, *float* xscale[] (Input)

Array with n components containing the scaling vector for the variables. xscale is used mainly in scaling the gradient and the distance between two points. See keywords IMSL_GRAD_TOL and IMSL_STEP_TOL for more detail.

Default: xscale[] = 1.

IMSL_FSCALE, *float* fscale[] (Input)

Array with m components containing the diagonal scaling matrix for the functions. The *i*-th component of fscale is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem.

Default: fscale[] = 1.

IMSL_GRAD_TOL, *float* grad_tol (Input)

Scaled gradient tolerance. The *i*-th component of the scaled gradient at x is calculated as

$$\frac{|g_i| * \max\left(|x_i|, 1/s_i\right)}{\frac{1}{2}\|F(x)\|_2^2}$$

where $\boldsymbol{g} = \nabla F(x)$, $\boldsymbol{s}$ = 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 $\varepsilon$ 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 $\varepsilon$ 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 $\varepsilon$ 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,

$$\varepsilon_1 = \left(\sum_{i=1}^{n}\left(s_i t_i\right)^2\right)^{1/2}, \varepsilon_2 = \|s\|_2$$

$s$ = `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 $J^TJ$. For imsl_f_nonlin_least_squares, if IMSL_JACOBIAN is specified, then tolerance = 100 × imsl_f_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_d_machine(4) is the default. Otherwise, the square root of imsl_d_machine(4) is the default. See imsl_f_machine.

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 × 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 × 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 × n containing the inverse matrix of $J^TJ$ where the $J$ is the final Jacobian. If $J^TJ$ is singular, the inverse is a symmetric $g_2$ inverse of $J^TJ$. (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 $J^TJ$ 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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to 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 $m \geq n$, $F : \Re^n \rightarrow \Re^m$, and $f_i(x)$ is the $i$-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach,

$$\min_{x \in R^n} \| F(x_c) + J(x_c)(x_n - x_c) \|_2$$
$$\text{subject to } \|x_n - x_c\|_2 \leq \delta_c$$

to get a new point $x_n$, which is computed as

$$x_n = x_c - (J(x_c)^T J(x_c) + \mu_c I)^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \geq \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c)\|_2$ and $\mu_c > 0$, otherwise. The value $\mu_c$ 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 `afcn_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).

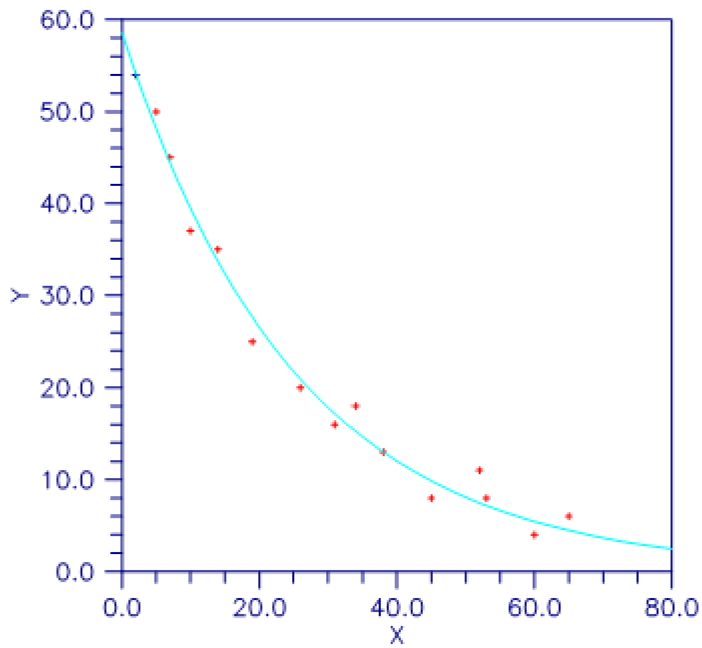



Figure 8.7 — Plot of the Nonlinear Fit

On some platforms, imsl_f_nonlin_least_squares can evaluate the user-supplied functions fcn and jacobian in parallel. This is done only if the function imsl_omp_options is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables

# 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[]);

int main()
{
   int              m=3, n=1;
   float            *result, fx[3];

   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

   result = imsl_f_nonlin_least_squares(fcn, m, n, 0);
   fcn(m, n, result, fx);


   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
   /* 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):

$$y_i = \theta_1 e^{\theta_2 x_i} + \varepsilon_i \;\; i = 1,2,...15$$

where the $\varepsilon_i$'s are independently distributed each normal with mean zero and variance $\sigma^2$. The estimate of $\sigma^2$ is then computed as

$$s^2 = \frac{\sum_{i=1}^{15} e_i^2}{15 - \text{rank}\left(J\right)}$$

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

$$est.asy.var\left(\hat{\theta}\right) = s^2\left(J^T J\right)^{-1}$$

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 $\theta_1$ and $\theta_2$.

```
#include <math.h>
#include <imsl.h>

void            exampl(int, int, float[], float[]);

int 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;

    imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    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. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# read_mps

Reads an MPS file containing a linear programming problem or a quadratic programming problem.

## Synopsis

*#include* <imsl.h>

*imsl_f_mps* *imsl_f_read_mps(*char* *filename, ..., 0)

*void* imsl_f_mps_free(*imsl_f_mps* *mps)

The type ***double*** function is imsl_d_read_mps.

## Required Argument

*char* *filename (Input)
Name of the MPS file to be read. It may be NULL if the optional argument IMSL_FILE is used.

## Return Value

A pointer to a structure containing the data read from the MPS file. To release this space use imsl_f_mps_free.

The returned structure contains the following fields.

| **Field** | **Description** |
|---|---|
| *char** filename | Name of the MPS file. |
| *char* name[9] | Name of the problem. |
| *int* nrows | Number of rows in the constraint matrix. |
| *int* ncolumns | Number of columns in the constraint matrix. This is also the number of variables. |
| *int* nonzero | Number of non-zeros in the constraint matrix. |
| *int* nhessian | Number of non-zeros in the Hessian matrix. If zero, then there is no Hessian matrix. |
| *int* ninteger | Number of variables required to be integer. This includes binary variables. |
| *int* nbinary | Number of variables required to be binary (0 or 1). |
| *float** objective | A float array of length ncolumns containing the objective vector. |
| *Imsl_f_sparse_elem** constraint | A imsl_f_sparse_elem array of length nonzeros containing the sparse matrix representation of the constraint matrix. See below for details. |

| Field | Description |
|---|---|
| *Imsl_f_sparse_elem** `hessian` | A `imsl_f_sparse_elem` array of length `nhessian` containing the sparse matrix representation of the Hessian matrix. If `nhessian` is zero, then this field is `NULL`. |
| *float** `lower_range` | A `float` array of length `nrows` containing the lower constraint bounds. If a constraint is unbounded below, the corresponding entry in `lower_range` is set to `negative_infinity`, defined below. |
| *float** `upper_range` | A `float` array of length `nrows` containing the upper constraint bounds. If a constraint is unbounded above, the corresponding entry in `upper_range` is set to `positive_infinity`, defined below. |
| *float** `lower_bound` | A `float` array of length `ncolumns` containing the lower variable bounds. If a variable is unbounded below, the corresponding entry in `lower_bound` is set to `negative_infinity`, defined below. |
| *float** `upper_bound` | A `float` array of length `ncolumns` containing the upper variable bounds. If a variable is unbounded above, the corresponding entry in `upper_bound` is set to `positive_infinity`, defined below. |
| *int** `variable_type` | An `int` array of length `ncolumns` containing the type of each variable. Variable types are: |
| | 0 Continuous |
| | 1 Integer |
| | 2 Binary (0 or 1) |
| | 4 Semicontinuous |
| *char* `name_objective`[9] | Name of the set in ROWS used for the objective row. |
| *char* `name_rhs`[9] | Name of the RHS set used. |
| *char* `name_ranges`[9] | Name of the RANGES set used or the empty string if no RANGES section in the file. |
| *char* `name_bounds`[9] | Name of the BOUNDS set used or the empty string if no BOUNDS section in the file. |
| *char*** `name_row` | Array of length `nrows` containing the row names. The name of the *i*-th constraint row is `name_row[i]`. |
| *char*** `name_column` | Array of length `ncolumns` containing the column names. The name of the *i*-th column and variable is `name_column[i]`. |
| *float* `negative_infinity` | Value used for a constraint or bound upper limit when the constraint or bound is unbounded above. This can be set using an optional argument. Default is 1.0e+30. |
| *float* `objective_constant` | Value of the constant in the objective. |

This structure stores the constraint and Hessian matrices in a simple sparse matrix format. For each non-zero element in the matrix, a row index, a column index and a value are given. The following code fragment expands the sparse constraint matrix in the structure pointed to by `mps` into a dense matrix:

```
/* allocate a matrix */
int nr =mps->nrows;
int nc =mps->ncolumns;
float* matrix =(float*)calloc(nr*nc, sizeof(float));

/* expand the sparse matrix */
```

```
for (k =0;  k < mps->nonzeros;  k++) {
        i =mps->constraint[k].row;
        j =mps->constraint[k].col;
        matrix[nc*i+j] =mps->constraint[k].val;
}
```

## Synopsis with Optional Arguments

*#include* <imsl.h>

*imsl_f_mps* *imsl_f_read_mps(*char* *filename,

IMSL_FILE, *FILE* *file,

IMSL_NAME_OBJECTIVE, *char* *name_objective,

IMSL_NAME_RHS, *char* *name_rhs,

IMSL_NAME_RANGES, *char* *name_ranges,

IMSL_NAME_BOUNDS, *char* *name_bounds,

IMSL_POSITIVE_INFINITY, *float* positive_infinity,

IMSL_NEGATIVE_INFINITY, *float* negative_infinity,

0)

## Optional Arguments

IMSL_FILE, *FILE* *file, (Input)
Handle for MPS file. The file is read but not closed. This option overrides the filename required argument.

IMSL_NAME_OBJECTIVE, *char* *name_ojective (Input)
Name of the set in ROWS used for the objective row. An MPS file can contain multiple objective function sets.
By default, the first objective function set in the MPS file is used. This name is case sensitive.

IMSL_NAME_RHS, *char* *name_rhs (Input)
Name of the RHS set to be used. An MPS file can contain multiple RHS sets.
By default, the first RHS set in the MPS file is used. This name is case sensitive.

IMSL_NAME_RANGES, *char* *name_ranges (Input)
Name of the RANGES set to be used. An MPS file can contain multiple RANGES sets.
By default, the first RANGES set in the MPS file is used. This name is case sensitive.

IMSL_NAME_BOUNDS, *char* *name_bounds (Input)
Name of the BOUNDS set to be used. An MPS file can contain multiple BOUNDS sets.
By default, the first BOUNDS set in the MPS file is used. This name is case sensitive.

IMSL_POSITIVE_INFINITY, *float* positive_infinity (Input)
Value used for a constraint or bound upper limit when the constraint or bound is unbounded above. Default: 1.0e+30.

IMSL_NEGATIVE_INFINITY, *float* negative_infinity (Input)
Value used for a constraint or bound lower limit when the constraint or bound is unbounded below. Default: -1.0e+30.

# Description

An MPS file defines a linear or quadratic programming problem.

A linear programming problem is assumed to have the form:

$$\min_{x \in R^n} c^T x$$

$$b_l \le Ax \le b_u$$

$$x_l \le x \le x_u$$

A quadratic programming problem is assumed to have the form:

$$\min_x \frac{1}{2} x^T Q x + c^T x$$

$$b_l \le Ax \le b_u$$

$$x_l \le x \le x_u$$

The following table maps this notation into the fields in the structure returned by the reader:

| | |
|---|---|
| $C$ | Objective |
| $A$ | Constraint |
| $Q$ | Hessian |
| $b_l$ | lower_range |
| $b_u$ | upper_range |
| $x_l$ | lower_bound |
| $x_u$ | upper_bound |

If the MPS file specifies an equality constraint or bound, the corresponding lower and upper values in the returned structure will be exactly equal.

The problem formulation assumes that the constraints and bounds are two-sided. If a particular constraint or bound has no lower limit, then the corresponding entry in the structure is set to −1.0e+30. If the upper limit is missing, then the corresponding entry in the structure is set to +1.0e+30.

# MPS File Format

There is some variability in the MPS format. This section describes the MPS format accepted by this reader.

An MPS file consists of a number of sections. Each section begins with a name in column 1. With the exception of the NAME section, the rest of this line is ignored. Lines with a '*'or '$'in column 1 are considered comment lines and are ignored.

The body of each section consists of lines divided into fields, as follows:

| Field Number | Columns | Contents |
|---|---|---|
| 1 | 2-3 | Indicator |
| 2 | 5-12 | Name |
| 3 | 15-22 | Name |
| 4 | 25-36 | Value |
| 5 | 40-47 | Name |
| 6 | 50-61 | Value |

The format limits MPS names to 8 characters and values to 12 characters. The names in fields 2, 3 and 5 are case sensitive. Leading and trailing blanks are ignored, but internal spaces are significant.

The sections in an MPS file are as follows.

- NAME
- ROWS
- COLUMNS
- RHS
- RANGES (optional)
- BOUNDS (optional)
- QUADRATIC (optional)
- ENDATA

Sections must occur in the above order.

MPS keywords (defined by the user in MPS data files), section names and indicator values, are case insensitive. Row, column and set names are case sensitive.

## NAME Section

The NAME section contains the single line. A problem name can occur anywhere on the line after NAME and before columns 62. The problem name is truncated to 8 characters.

## ROWS Section

The ROWS section defines the name and type for each row. Field 1 contains the row type and field 2 contains the row name. Row type values are not case sensitive. Row names are case sensitive. The following row types are allowed:

| Row Type | Meaning |
|---|---|
| E | Equality Constraint. |
| L | Less than or equal constraint. |
| G | Greater than or equal constraint. |
| N | Objective or a free row. |

## COLUMNS Section

The COLUMNS section defines the nonzero entries in the objective and the constraint matrix. The row names here must have been defined in the ROWS section.

| Field | Contents |
|---|---|
| 2 | Column name. |
| 3 | Row name. |
| 4 | Value for the entry whose row and column are given by fields. |
| 5 | Row name. |
| 6 | Value for the entry whose row and column are given by fields 5 and 2. |

**NOTE:** Fields 5 and 6 are optional.

The COLUMNS section can also contain markers. These are indicated by the name 'MARKER' (with the quotes) in field 3 and the marker type in field 4 or 5.

Marker type 'INTORG' (with the quotes) begins an integer group. The marker type 'INTEND' (with the quotes) ends this group. The variables corresponding to the columns defined within this group are required to be integer.

## RHS Section

The RHS section defines the right-hand side of the constraints. An MPS file can contain more than one RHS set, distinguished by the RHS set name. The row names here must be defined in the ROWS section.

| Field | Contents |
|---|---|
| 2 | RHS set name. |
| 3 | Row name. |
| 4 | Value for the entry whose set and row are given by fields 2 and 3. |

| Field | Contents |
|---|---|
| 5 | Row name. |
| 6 | Value for the entry whose set and row are given by fields 2 and 5. |

If the row name is identical with the name of the objective, then the negative of the value in field 6 is the constant in the objective function.

**NOTE**: Fields 5 and 6 are optional.

## RANGES Section

The optional RANGES section defines two-sided constraints. An MPS file can contain more than one range set, distinguished by the range set name. The row names here must have been defined in the ROWS section.

| Field | Contents |
|---|---|
| 2 | Range set name. |
| 3 | Row name. |
| 4 | Value for the entry whose set and row are given by fields 2 and 3. |
| 5 | Row name. |
| 6 | Value for the entry whose set and row are given by fields 2 and 5. |

**NOTE**: Fields 5 and 6 are optional.

Ranges change one-sided constraints, defined in the RHS section, into two-sided constraints. The two-sided constraint for row *i* depends on the range value, $r_i$, defined in this section. The right-hand side value, $b_i$, is defined in the RHS section. The two-sided constraints for row *i* are given in the following table:

| Row Type | Lower Constraint | Upper Constraint |
|---|---|---|
| G | $b_i$ | $b_i + \|r_i\|$ |
| L | $b_i - \|r_i\|$ | $b_i$ |
| E | $b_i + \min(0, r_i)$ | $b_i + \max(0, r_i)$ |

## BOUNDS Section

The optional BOUNDS section defines bounds on the variables. By default, the bounds are $0 \leq x_i \leq \infty$. The bounds can also be used to indicate that a variable must be an integer.

More than one bound can be set for a single variable. For example, to set $2 \leq x_i \leq 6$ use a LO bound with value 2 to set $2 \leq x_i$ and an UP bound with value 6 to add the condition $x_i \leq 6$.

An MPS file can contain more than one bounds set, distinguished by the bound set name.

| Field | Contents |
|---|---|
| 1 | Bounds type. |
| 2 | Bounds set name. |
| 3 | Column name |
| 4 | Value for the entry whose set and column are given by fields 2 and 3. |
| 5 | Column name. |
| 6 | Value for the entry whose set and column are given by fields 2 and 5. |

**NOTE**: Fields 5 and 6 are optional.

The bound types are as follows. Here $b_i$ are the bound values defined in this section, the $x_i$ are the variables, and *I* is the set of integers.

| Bounded Type | Definition | Formula |
|---|---|---|
| LO | Lower bound | $b_j \leq x_i$ |
| UP | Upper bound | $x_i \leq b_i$ |
| FX | Fixed variable | $x_i = b_i$ |
| FR | Free variable | $-\infty \leq x_i \leq \infty$ |
| MI | Lower bound is minus infinity | $-\infty \leq x_i$ |
| PL | Upper bound is positive infinity | $x_i \leq \infty$ |
| BV | Binary variable (variable must be 0 or 1). | $x_i \in \{0, 1\}$ |
| UI | Upper bound and integer | $x_i \leq b_i$ and $x_i \in I$ |
| LI | Lower bound and integer | $b_i \leq x_i$ and $x_i \in I$ |
| SC | Semicontinuous | 0 or $b_i \leq x_i$ |

The bound type names are not case sensitive.

If the bound type is UP or UI and $b_j < 0$ then the lower bound is set to $-\infty$.

## QUADRATIC Section

The optional QUADRATIC section defines the Hessian for quadratic programming problems. The names HESSIAN, QUADS, QUADOBJ, QSECTION and QMATRIX are also recognized as beginning the QUADRATIC section.

| Field | Contents |
| --- | --- |
| 2 | Column name. |
| 3 | Column name |
| 4 | Value for the entry whose row and column are given by fields 2 and 3. |
| 5 | Column name. |
| 6 | Value for the entry whose row and column are given by fields 2 and 4. |

**NOTE**: Fields 5 and 6 are optional.

## ENDATA Section

The ENDATA section ends the MPS file.

# linear_programming

Solves a linear programming problem.

---

**NOTE:** For double precision, the function `lin_prog` has generally been superseded by the function `linear_programming`. Function `lin_prog` remains in place to ensure compatibility of existing calls.

---

## Synopsis

*#include* <imsl.h>

*double* *imsl_d_linear_programming (*int* m, *int* n, *double* a[], *double* b[], *double* c[], …, 0)

## Required Arguments

*int* m (Input)
Number of constraints.

*int* n (Input)
Number of variables.

*double* a[] (Input)
Array of size m × n containing a matrix with coefficients of the m constraints.

*double* 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.

*double* 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 `imsl_free`. If no solution can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*double* *imsl_d_linear_programming (*int* m, *int* n, *double* a[], *double* b[], *double* c[],

IMSL_A_COL_DIM, *int* a_col_dim,

IMSL_UPPER_LIMIT, *double* bu[],

IMSL_CONSTR_TYPE, *int* irtype[],

IMSL_LOWER_BOUND, *double* xlb[],

IMSL_UPPER_BOUND, *double* xub[],

IMSL_REFINEMENT,

IMSL_EXTENDED_REFINEMENT,

IMSL_OBJ, *double* *obj,

IMSL_RETURN_USER, *double* x[],

IMSL_DUAL, *double* **y,

IMSL_DUAL_USER, *double* 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, *double* 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 | $b_i \le r_i \le bu_i$ |
| 4 | Ignore this constraint |

Default: irtype = 0

IMSL_LOWER_BOUND, *double* 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, *double* 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: no upper bound

IMSL_REFINEMENT (Input)

The coefficient matrices and other data are saved at the beginning of the computation. When finished this data together with the solution obtained is checked for consistency. If the discrepancy is too large, the solution process is restarted using the problem data just after processing the equalities, but with the final x values and final active set.

Default: Refinement is not performed.

IMSL_EXTENDED_REFINEMENT (Input)

This is similar to IMSL_REFINEMENT, except it iterates until there is a sign that no further progress is possible (recommended if all the accuracy possible is desired) .

Default: Extended refinement is not performed.

IMSL_OBJ, *double* *obj (Output)

Optimal value of the objective function.

IMSL_ITERATION_COUNT, *int* *iterations (Output)

Number of iterations.

IMSL_RETURN_USER, *double* x[] (Output)

Array with n components containing the primal solution.

IMSL_DUAL, *double* **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_d_linear_programming. Typically, *double* *y is declared, and &y is used as an argument.

IMSL_DUAL_USER, *double* y[] (Output)

A user-allocated array of size m. On return, y contains the dual solution.

## Description

The function imsl_d_linear_programming uses an active set strategy to solve linear programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x \quad \text{subject to} \quad b_l \le A_x \le b_u$$
$$x_l \le x \le x_u$$

where $\boldsymbol{c}$ is the objective coefficient vector, $\boldsymbol{A}$ is the coefficient matrix, and the vectors $\boldsymbol{b}_l$, $\boldsymbol{b}_u$, $\boldsymbol{x}_l$, and $\boldsymbol{x}_u$ are the lower and upper bounds on the constraints and the variables, respectively.

# Examples

## Example 1

The linear programming problem in the standard form

$$
\begin{array}{lllllllll}
 & \min\ f(x) = -x_1 - 3x_2 & & & & & & & \\
\text{subject to} & x_1 & +x_2 & +x_3 & & & & = 1.5 \\
 & x_1 & +x_2 & & -x_4 & & & = 0.5 \\
 & x_1 & & & & +x_5 & & = 1.0 \\
 & & x_2 & & & & +x_6 & = 1.0 \\
 & x_i \geq 0, \text{ for } i = 1, \ldots, 6 & & & & & &
\end{array}
$$

is solved.

```
#include <imsl.h>

int main()
{
   int          m = 4;
   int          n = 6;
   double        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};
   double        b[ ] = {1.5, 0.5, 1.0, 1.0};
   double        c[ ] = {-1.0, -3.0, 0.0, 0.0, 0.0, 0.0};
   double        *x;
                               /* Solve the LP problem */

   x = imsl_d_linear_programming (m, n, a, b, c, 0);
                               /* Print x */
   imsl_d_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

This example demonstrates how the function `imsl_d_read_mps` can be used together with `imsl_d_linear_programming` to solve a linear programming problem defined in an MPS file. The MPS file used in this example is an ***uncompressed*** version of the file ‘`afiro`’, available from `http://www.netlib.org/lp/data/`. This example also demonstrates the use of the optional argument `IMSL_REFINEMENT` to activate iterative refinement in `imsl_d_linear_programming`.

```
#include <stdio.h>
#include <malloc.h>
#include <imsl.h>

int main() {
#define A(I, J) a[(I)*problem->ncolumns+J]
   Imsl_d_mps* problem;
   int i, j, k, *irtype;
   double *x, objective, *a, *bl, *bu, *xlb, *xub;

   /* Read the MPS file. */
   problem = imsl_d_read_mps("afiro", 0);

   /* Setup constraint type array. */
   irtype = (int*) malloc(problem->nrows*sizeof(int));
   for (i = 0; i < problem->nrows; i++)
       irtype[i] = 3;

   /* Setup the constraint matrix. */
   a = (double*) calloc(problem->nrows*problem->ncolumns*sizeof(double),
       sizeof(double));
   for (k = 0; k < problem->nonzeros; k++) {
       i = problem->constraint[k].row;
       j = problem->constraint[k].col;
       A(i, j) = problem->constraint[k].val;
   }

   /* Setup constraint bounds. */
   bl = (double*) malloc(problem->nrows*sizeof(double));
   bu = (double*) malloc(problem->nrows*sizeof(double));
   for (i = 0; i < problem->nrows; i++) {
       bl[i] = problem->lower_range[i];
       bu[i] = problem->upper_range[i];
   }

   /* Setup variable bounds. Be sure to account for
      how unbounded variables should be set. */
   xlb = (double*) malloc(problem->ncolumns*sizeof(double));
   xub = (double*) malloc(problem->ncolumns*sizeof(double));
   for (i = 0; i < problem->ncolumns; i++) {
       xlb[i] = (problem->lower_bound[i] == problem->negative_infinity) ?
           1.0e30 : problem->lower_bound[i];
       xub[i] = (problem->upper_bound[i] == problem->positive_infinity) ?
           -1.0e30 : problem->upper_bound[i];
   }

   /* Solve the LP problem. */
   x = imsl_d_linear_programming(problem->nrows, problem->ncolumns,
       a, bl, problem->objective,
       IMSL_UPPER_LIMIT, bu,
```

```
        IMSL_CONSTR_TYPE, irtype,
        IMSL_LOWER_BOUND, xlb,
        IMSL_UPPER_BOUND, xub,
        IMSL_REFINEMENT,
        IMSL_OBJ, &objective,
        0);

    /* Output results. */
    printf("Problem Name: %s\n", problem->name);
    printf("objective : %e\n", objective);
    imsl_d_write_matrix("Solution", problem->ncolumns, 1, x, 0);

    /* Free MPS structure. */
    imsl_d_mps_free(problem);
}
```

### Output

```
Problem Name: AFIRO
objective  : -4.647531e+02

  Solution
1        80.0
2        25.5
3        54.5
4        84.8
5        57.9
6         0.0
7         0.0
8         0.0
9         0.0
10         0.0
11         0.0
12         0.0
13        18.2
14        39.7
15        61.3
16       500.0
17       475.9
18        24.1
19         0.0
20       215.0
21       363.9
22         0.0
23         0.0
24         0.0
25         0.0
26         0.0
27         0.0
28         0.0
29       339.9
```

```
30        20.1
31       156.5
32         0.0
```

# Note Errors

| | |
|---|---|
| IMSL_MULTIPLE_SOLUTIONS | Multiple solutions giving essentially the same minimum exist. |

# Warning Errors

| | |
|---|---|
| IMSL_SOME_CONSTRAINTS_DISCARDED | Some constraints were ignored or discarded because they were too linearly dependent on other active constraints. |
| IMSL_ALL_CONSTR_NOT_SATISFIED | All constraints are not satisfied. If a feasible solution is possible then try using refinement by supplying optional argument IMSL_REFINEMENT. |
| IMSL_CYCLING_OCCURRING | The algorithm appears to be cycling. Using refinement may help. |

# Fatal Errors

| | |
|---|---|
| IMSL_PROB_UNBOUNDED | The problem is unbounded. |
| IMSL_PIVOT_NOT_FOUND | An acceptable pivot could not be found. |

# lin_prog

Solves a linear programming problem using the revised simplex algorithm.

**NOTE:** For double precision, the function `lin_prog` has generally been superseded by the function `linear_programming`. Function `lin_prog` remains in place to ensure compatibility of existing calls.

## 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` × `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 `imsl_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.

`IMSL_USE_UPDATED_LP_ALGORITHM` (Input)

Calls the function `imsl_d_linear_programming` to solve the problem. If this optional argument is present, then the optional argument `IMSL_MAX_ITN` is ignored. This optional argument is only valid in double precision.

# Description

The function `imsl_f_lin_prog` uses a revised simplex method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x \quad \text{subject to} \quad \begin{array}{l} b_l \le A_x \le b_u \\ x_l \le x \le x_u \end{array}$$

where $c$ is the objective coefficient vector, $A$ is the coefficient matrix, and the vectors $b_l$, $b_u$, $x_l$, and $x_u$ are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

# Examples

## Example 1

The linear programming problem in the standard form

$$
\begin{aligned}
& \min\ f(x) = -x_1 - 3x_2 \\
\text{subject to}\quad & x_1 + x_2 + x_3 = 1.5 \\
& x_1 + x_2 - x_4 = 0.5 \\
& x_1 + x_5 = 1.0 \\
& x_2 + x_6 = 1.0 \\
& x_i \geq 0,\ \text{for}\ i = 1, \ldots, 6
\end{aligned}
$$

is solved.

```
#include <imsl.h>

int 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:

$$\min\ f(x) = -x_1 - 3x_2$$
$$\text{subject to}\ \ 0.5 \le x_1 + x_2 \le 1.5$$
$$0 \le x_1 \le 1.0$$
$$0 \le x_2 \le 1.0$$

This problem can be solved more efficiently.

```
#include <imsl.h>
#include <stdio.h>

int 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

more...

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 × 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 × 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 imsl_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_MAX_ITN, *int* max_itn,

IMSL_TOLERANCE, *float* small,

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_MAX_ITN, *int* max_itn (Input)
Maximum number of iterations.If max_itn is set to 0, the iteration count is unbounded.
Default: max_itn = 100000

IMSL_TOLERANCE, *float* small (Input)

This constant is used in the determination of the positive definiteness of the Hessian H. small is also used for the convergence criteria of a constraint violation.

Default: small = 10.0 × machine precision for single precision, and 1000.0 × machine precision for double precision.

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

$$\begin{aligned} &\min_{x \in R^n} \; g^T x + \tfrac{1}{2} x^T H x \\ &\text{subject to} \quad A_1 x = b_1 \\ &\qquad\qquad\quad\;\; A_2 x \ge b_2 \end{aligned}$$

given the vectors $b_1$, $b_2$, and $g$, and the matrices $H$, $A_1$, and $A_2$. $H$ is required to be positive definite. In this case, a unique $x$ solves the problem or the constraints are inconsistent. If $H$ is not positive definite, a positive definite perturbation of $H$ is used in place of $H$. For more details, see Powell (1983, 1985).

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

$$\begin{aligned}
&\min\ f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1 \\
&\text{subject to} \quad x_1 + x_2 + x_3 + x_4 + x_5 = 5 \\
&\qquad\qquad\quad\ x_3 - 2x_4 - 2x_5 = -3
\end{aligned}$$

is solved.

```
#include <imsl.h>

int 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

$$\min f(x) = x_1^2 + x_2^2 + x_3^2 \quad \text{subject to} \quad \begin{aligned} x_1 + 2x_2 - x_3 &= 4 \\ x_1 - x_2 + x_3 &= -2 \end{aligned}$$

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};
int 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. |

# sparse_lin_prog

Solves a sparse linear programming problem by an infeasible primal-dual interior-point method.

---

**NOTE:** Function `sparse_lin_prog` is available in double precision only.

---

## Synopsis

*#include* `<imsl.h>`

*double* `*imsl_d_sparse_lin_prog` (*int* `m`, *int* `n`, *int* `nza`, *Imsl_d_sparse_elem* `a[]`, *double* `b[]`, *double* `c[]`, …, 0)

## Required Arguments

*int* `m` (Input)
Number of constraints.

*int* `n` (Input)
Number of variables.

*int* `nza` (Input)
Number of nonzero entries in the constraint matrix *A*.

*Imsl_d_sparse_elem* `a[]` (Input)
An array of length `nza` containing the location and value of each nonzero coefficient in the constraint matrix *A*.

*double* `b[]` (Input)
An array of length `m` 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.

*double* `c[]` (Input)
An array of length `n` containing the coefficients of the objective function.

## Return Value

A pointer to an array of length `n` containing the solution *x* of the linear programming problem. To release this space, use `imsl_free`. If no solution can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*double* *imsl_d_sparse_lin_prog (*int* m, *int* n, *int* nza, *Imsl_d_sparse_elem* a[], *double* b[], *double* c[],

IMSL_CONSTR_TYPE, *int* irtype[],

IMSL_UPPER_LIMIT, *double* bu[],

IMSL_LOWER_BOUND, *double* xlb[],

IMSL_UPPER_BOUND, *double* xub[],

IMSL_OBJ_CONSTANT, *double* c0,

IMSL_PREORDERING, *int* preorder,

IMSL_MAX_ITERATIONS, *int* max_iterations,

IMSL_OPT_TOL, *double* opt_tol,

IMSL_PRINF_TOL, *double* prinf_tol,

IMSL_DLINF_TOL, *double* dlinf_tol,

IMSL_PRINT, *int* iprint,

IMSL_PRESOLVE, *int* presolve,

IMSL_CSC_FORMAT, *int* a_colptr[], *int* a_rowind[], *double* a_values[],

IMSL_TERMINATION_STATUS, *int* *status,

IMSL_OBJ, *double* *obj,

IMSL_ITERATION_COUNT, *int* *iterations,

IMSL_DUAL, *double* **y,

IMSL_DUAL_USER, *double* y[],

IMSL_PRIMAL_INFEAS, *double* *err_b, *double* *err_u,

IMSL_DUAL_INFEAS, *double* *err_c,

IMSL_CP_RATIO_SMALLEST, *double* *cp_smallest,

IMSL_CP_RATIO_LARGEST, *double* *cp_largest,

IMSL_RETURN_USER, *double* x[],

0)

## Optional Arguments

`IMSL_CONSTR_TYPE`, *int* `irtype[]` (Input)

An array of length `m` containing 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 | $r_i \le b_i$ |
| 2 | $r_i \ge b_i$ |
| 3 | $b_i \le r_i \le bu_i$ |
| 4 | Ignore this constraint |

Note that `irtype[i]` = 3 should only be used for constraints `i` with both a finite lower and a finite upper bound. For one-sided constraints, use `irtype[i]` = 1 or `irtype[i]` = 2. For free constraints, use `irtype[i]` = 4.

Default: `irtype` = 0

`IMSL_UPPER_LIMIT`, *double* `bu[]` (Input)

Array of length `m` containing the upper limit of the constraints that have both a lower and an upper bound. If such a constraint exists, then optional argument `IMSL_CONSTR_TYPE` must be used to define the type of the constraints. If `irtype[i]` ≠ 3, i.e. if constraint `i` is not two-sided, then the corresponding entry in `bu`, `bu[i]`, is ignored.

Default: None of the constraints has an upper bound.

`IMSL_LOWER_BOUND`, *double* `xlb[]` (Input)

An array of length `n` 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`, *double* `xub[]` (Input)

An array of length `n` 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: None of the variables has an upper bound.

`IMSL_OBJ_CONSTANT`, *double* `c0` (Input)

Value of the constant term in the objective function.

Default: `c0` = 0.

`IMSL_PREORDERING`, *int* `preorder` (Input)

The variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix.

| `preorder` | Method |
|---|---|
| 0 | A variant of the MDO algorithm using pivotal cliques. |
| 1 | A variant of George and Liu's Quotient Minimum Degree algorithm. |

Default: `preorder` = 0.

`IMSL_MAX_ITERATIONS`, *int* `max_iterations` (Input)

The maximum number of iterations allowed for the primal-dual solver.

Default: `max_iterations` = 200.

`IMSL_OPT_TOL`, *double* `opt_tol` (Input)

The relative optimality tolerance.

Default: `opt_tol` = 1.0e-10.

`IMSL_PRINF_TOL`, *double* `prinf_tol` (Input)

The primal infeasibility tolerance.

Default: `prinf_tol` = 1.0e-8.

`IMSL_DLINF_TOL`, *double* `dlinf_tol` (Input)

The dual infeasibility tolerance.

Default: `dlinf_tol` = 1.0e-8.

`IMSL_PRINT`, *int* `iprint` (Input)

Printing option.

| `iprint` | Action |
|---|---|
| 0 | No printing is performed. |
| 1 | Prints statistics on the LP problem and the solution process. |

Default: `iprint` = 0.

`IMSL_PRESOLVE`, *int* `presolve` (Input)

Presolve the LP problem in order to reduce the problem size or to detect infeasibility or unboundedness of the problem. Depending on the number of presolve techniques used, different presolve levels can be chosen:

| `presolve` | Description |
|---|---|
| 0 | No presolving. |
| 1 | Eliminate singleton rows |

| presolve | Description |
|---|---|
| 2 | In addition to 1, eliminate redundant (and forcing) rows. |
| 3 | In addition to 1 and 2, eliminate dominated variables. |
| 4 | In addition to 1, 2, and 3, eliminate singleton columns. |
| 5 | In addition to 1, 2, 3, and 4, eliminate doubleton rows. |
| 6 | In addition to 1, 2, 3, 4, and 5, eliminate aggregate columns. |

Default: `presolve =0`.

`IMSL_CSC_FORMAT`, *int* `a_colptr[]`, *int* `a_rowind[]`, *double* `a_values[]` (Input)

Accept the constraint matrix *A* in Harwell-Boeing format. See (Compressed Sparse Column (CSC) Format) in the Introduction to this manual for a discussion of this storage scheme.

If this optional argument is used, then required argument `a` is ignored.

`IMSL_TERMINATION_STATUS`, *int* `*status` (Output)

The termination status for the problem.

| status | Description |
|---|---|
| 0 | Optimal solution found. |
| 1 | The problem is primal infeasible (or dual unbounded). |
| 2 | The problem is primal unbounded (or dual infeasible). |
| 3 | Suboptimal solution found (accuracy problems). |
| 4 | Iterations limit `max_iterations` exceeded. |
| 5 | An error outside of the solution phase of the algorithm, e.g. a user input or a memory allocation error. |

`IMSL_OBJ`, *double* `*obj` (Output)

Optimal value of the objective function.

`IMSL_ITERATION_COUNT`, *int* `*iterations` (Output)

The number of iterations required by the primal-dual solver.

`IMSL_DUAL`, *double* `**y` (Output)

The address of a pointer `y` to an internally allocated array of length `m` containing the dual solution.

`IMSL_DUAL_USER`, *double* `y[]` (Output)

A user-allocated array of length `m` containing the dual solution.

`IMSL_PRIMAL_INFEAS`, *double* `*err_b`, *double* `*err_u` (Output)

The violation of the primal constraints, described by `err_b`, the primal infeasibility of the solution, $\|x + s - u\|$, and by `err_u`, the violation of the variable bounds, $\|b - Ax\|$.

IMSL_DUAL_INFEAS, *double* *err_c (Output)

The violation of the dual constraints, described by err_c, the dual infeasibility of the solution,

$\|c - A^T y - z + w\|$.

IMSL_CP_RATIO_SMALLEST, *double* *cp_smallest (Output)

The ratio of the smallest complementarity product to the average.

IMSL_CP_RATIO_LARGEST, *double* *cp_largest (Output)

The ratio of the largest complementarity product to the average.

IMSL_RETURN_USER, *double* x[] (Output)

A user-allocated array of length n containing the primal solution.

## Description

The function imsl_d_sparse_lin_prog uses an infeasible primal-dual interior-point method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x \quad \text{subject to} \quad \begin{aligned} b_l \le Ax \le b_u \\ x_l \le x \le x_u \end{aligned}$$

where $c$ is the objective coefficient vector, $A$ is the coefficient matrix, and the vectors $b_l$, $b_u$, $x_l$, and $x_u$ are the lower and upper bounds on the constraints and the variables, respectively.

Internally, imsl_d_sparse_lin_prog transforms the problem given by the user into a simpler form that is computationally more tractable. After redefining the notation, the new form reads

$$\begin{array}{llrll} \min c^T x & \text{subject to} & Ax = b & & \\ & & x_i + s_i = u_i, & x_i,\ s_i \ge 0, & i \in I_u \\ & & x_j \ge 0, & & j \in I_s \end{array}$$

Here, $I_u \cup I_s = \{1, \ \dots, n\}$ is a partition of the index set $\{1, \ \dots, n\}$ into upper bounded and standard variables.

In order to simplify the description, it is assumed in the following that the problem above contains only variables with upper bounds, i.e. is of the form

$$(P) \quad \min c^T x \quad \text{subject to} \quad \begin{aligned} Ax &= b, \\ x + s &= u, \\ x, s &\ge 0 \end{aligned}$$

The corresponding dual problem is then

$$(D) \quad \max\ b^T y - u^T w \quad \text{subject to} \quad A^T y + z - w = c, \\ z, w \geq 0$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for (P) and (D) are

$$\begin{aligned} &Ax = b, && (1.1)\\ &x + s = u, && (1.2)\\ &A^T y + z - w = c, && (1.3)\\ &XZe = 0, && (1.4)\\ &SWe = 0, && (1.5)\\ &x,z,s,w \geq 0, && (1.6) \end{aligned}$$

where $X = diag(x)$, $Z = diag(z)$, $S = diag(s)$, $W = diag(w)$ are diagonal matrices and $e = (1,\ \dots,1)^T$ is a vector of ones.

Function `imsl_d_sparse_lin_prog`, like all infeasible interior-point methods, generates a sequence

$$\left(x_k, s_k, y_k, z_k, w_k\right), \quad k = 0, 1, \ \dots$$

of iterates, that satisfy $\left(x_k, s_k, y_k, z_k, w_k\right) > 0$ for all $k$, but are in general not feasible, i.e. the linear constraints (1.1)-(1.3) are only satisfied in the limiting case $k \to \infty$.

The barrier parameter $\mu$, defined by

$$\mu = \frac{x^T z + s^T w}{2n}$$

measures how good the complementarity conditions (1.4), (1.5) are satisfied.

Mehrotra's predictor-corrector algorithm is a variant of Newton's method applied to the KKT conditions (1.1)-(1.5). Function `imsl_d_sparse_lin_prog` uses a modified version of this algorithm to compute the iterates $\left(x_k, s_k, y_k, z_k, w_k\right)$. In every step of the algorithm, the search direction vector

$$\varDelta := \left(\varDelta x, \varDelta s, \varDelta y, \varDelta z, \varDelta w\right)$$

is decomposed into two parts, $\varDelta = \varDelta_a + \varDelta_c^{\omega}$, where $\varDelta_a$ and $\varDelta_c^{\omega}$ denote the affine-scaling and the weighted centering component, respectively. Here,

$$\varDelta_c^{\omega} := \left(\omega_P \varDelta x_c, \omega_P \varDelta s_c, \omega_D \varDelta y_c, \omega_D \varDelta z_c, \omega_D \varDelta w_c\right)$$

where $\omega_P$ and $\omega_D$ denote the primal and dual corrector weights, respectively.

The vectors $\Delta_a$ and $\Delta_c := \left(\Delta x_c, \Delta s_c, \Delta y_c, \Delta z_c, \Delta w_c\right)$ are determined by solving the linear system

$$\begin{bmatrix} A & 0 & 0 & 0 & 0 \\ I & 0 & I & 0 & 0 \\ 0 & A^T & 0 & I & -I \\ Z & 0 & 0 & X & 0 \\ 0 & 0 & W & 0 & S \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \\ \Delta z \\ \Delta w \end{bmatrix} = \begin{bmatrix} r_b \\ r_u \\ r_c \\ r_{xz} \\ r_{ws} \end{bmatrix} \quad (2)$$

for two different right-hand sides.

For $\Delta_a$, the right-hand side is defined as

$$\left(r_b, r_u, r_c, r_{xz}, r_{ws}\right) = \left(b - Ax, u - x - s, c - A^T y - z + w, \ -XZe, \ -WSe\right).$$

Here, $r_b$ and $r_u$ are the violations of the primal constraints and $r_c$ defines the violations of the dual constraints.

The resulting direction $\Delta_a$ is the pure Newton step applied to the system (1.1)-(1.5).

In order to obtain the corrector direction $\Delta_c$, the maximum stepsizes $\alpha_{Pa}$ in the primal and $\alpha_{Da}$ in the dual space preserving nonnegativity of $\left(x, s\right)$ and $\left(z, w\right)$ respectively, are determined, and the predicted complementarity gap,

$$g_a = \left(x + \alpha_{Pa}\Delta x_a\right)^T\left(z + \alpha_{Da}\Delta z_a\right) + \left(s + \alpha_{Pa}\Delta s_a\right)^T\left(w + \alpha_{Da}\Delta w_a\right)$$

is computed. It is then used to determine the barrier parameter

$$\hat{\mu} = \left(\frac{g_a}{g}\right)^2 \frac{g_a}{2n},$$

where $g = x^T z + s^T w$ denotes the current complementarity gap.

The direction $\Delta_c$ is then computed by choosing

$$\left(r_b, r_u, r_c, r_{xz}, r_{sw}\right) = \left(0, 0, 0, \hat{\mu}e - \Delta X_a \Delta Z_a e, \hat{\mu}e - \Delta W_a \Delta S_a e\right)$$

as the right-hand side in the linear system (2).

Function `imsl_d_sparse_lin_prog` now uses a line search to find the optimal weight $\hat{\omega} = \left(\hat{\omega}_P, \hat{\omega}_D\right)$ that maximizes the stepsizes $\left(\alpha_P, \alpha_D\right)$ in the primal and dual directions of $\Delta = \Delta_a + \Delta_c^{\omega}$, respectively.

A new iterate is then computed using a step reduction factor $\alpha_0 = 0.99995$:

$$\left(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}, w_{k+1}\right) = \left(x_k, s_k, y_k, z_k, w_k\right) + \alpha_0\left(\alpha_P \Delta x, \alpha_P \Delta s, \alpha_D \Delta y, \alpha_D \Delta z, \alpha_D \Delta w\right)$$

In addition to the weighted Mehrotra predictor-corrector, `imsl_d_sparse_lin_prog` also uses multiple centrality correctors to enlarge the primal-dual stepsizes per iteration step and to reduce the overall number of iterations required to solve an LP problem. The maximum number of centrality corrections depends on the ratio of the factorization and solve efforts for system (2) and is therefore problem dependent. For a detailed description of multiple centrality correctors, refer to Gondzio(1994).

The linear system (2) can be reduced to more compact forms, the augmented system (AS)

$$\begin{bmatrix} -\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r \\ h \end{bmatrix} \quad (3)$$

or further by elimination of $\Delta x$ to the normal equations (NE) system

$$A\Theta A^T \Delta y = A\Theta r + h, \qquad (4)$$

where

$$\Theta = \left(X^{-1}Z + S^{-1}W\right)^{-1}, \; r = r_c - X^{-1}r_{xz} + S^{-1}r_{ws} - S^{-1}Wr_u, \; h = r_b.$$

The matrix on the left-hand side of (3), which is symmetric indefinite, can be transformed into a symmetric quasidefinite matrix by regularization. Since these types of matrices allow for a Cholesky-like factorization, the resulting linear system can be solved easily for $\left(\Delta x, \Delta y\right)$ by triangular substitutions. For more information on the regularization technique, see Altman and Gondzio (1998). For the NE system, matrix $A\Theta A^T$ is positive definite, and therefore a sparse Cholesky algorithm can be used to factor $A\Theta A^T$ and solve the system for $\Delta y$ by triangular substitutions with the Cholesky factor $L$.

In function `imsl_d_sparse_lin_prog`, both approaches are implemented. The AS approach is chosen if $A$ contains dense columns, if there are a considerable number of columns in $A$ that are much denser than the remaining ones or if there are many more rows than columns in the structural part of $A$. Otherwise, the NE approach is selected.

Function `imsl_d_sparse_lin_prog` stops with optimal termination status if the current iterate satisfies the following three conditions:

$$\frac{\mu}{1 + 0.5\left(|c^T x| + |b^T y - u^T w|\right)} \le \text{opt\_tol}$$

$$\frac{\|\left(b - Ax, x + s - u\right)\|}{1 + \|\left(b, u\right)\|} \le \text{prinf\_tol, and} \frac{\|c - A^T y - z + w\|}{1 + \|c\|} \le \text{dlinf\_tol},$$

where `prinf_tol`, `dlinf_tol` and `opt_tol` are primal infeasibility, dual infeasibility and optimality tolerances, respectively. The default value is 1.0e-10 for `opt_tol` and 1.0e-8 for the two other tolerances.

Function `imsl_d_sparse_lin_prog` is based on the code HOPDM developed by Jacek Gondzio et al.; see the HOPDM User's Guide (1995).

# Examples

## Example 1

The linear programming problem

$$\begin{array}{lll} \min f(x) = 2x_1 - 8x_2 + 3x_3 & & \\ \text{subject to } x_1 + 3x_2 & \le & 3 \\ \qquad 2x_2 + 3x_3 & \le & 6 \\ \qquad x_1 + x_2 + x_3 & \ge & 2 \\ \qquad -1 \le x_1 \le 5 & & \\ \qquad 0 \le x_2 \le 7 & & \\ \qquad 0 \le x_3 \le 9 & & \end{array}$$

is solved.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int m = 3, n = 3, nza = 7;
   double obj, *x = NULL;

   Imsl_d_sparse_elem a[] = { 0, 0, 1.0,
       0, 1, 3.0,
       1, 1, 2.0,
       1, 2, 3.0,
       2, 0, 1.0,
       2, 1, 1.0,
       2, 2, 1.0 };

   double b[] = { 3.0, 6.0, 2.0 };
   double c[] = { 2.0, -8.0, 3.0 };
   double xlb[] = { -1.0, 0.0, 0.0 };
   double xub[] = { 5.0, 7.0, 9.0 };
   int irtype[] = { 1, 1, 2 };

   x = imsl_d_sparse_lin_prog(m, n, nza, a, b, c,
       IMSL_CONSTR_TYPE, irtype,
       IMSL_LOWER_BOUND, xlb,
       IMSL_UPPER_BOUND, xub,
       IMSL_OBJ, &obj,
       0);
```

```
    imsl_d_write_matrix("x", 1, n, x, 0);
    printf("\nObjective: %lf\n", obj);
}
```

## Output

```
                 x
        1            2            3
    -0.375       1.125        1.250

Objective: -6.000000
```

## Example 2

This example demonstrates how the function `imsl_d_read_mps` can be used with `imsl_d_sparse_lin_prog` to solve a linear programming problem defined in an MPS file. The MPS file used in this example is an ***uncompressed*** version of the file '`afiro`', available from `http://www.netlib.org/lp/data/`.

```
#include <imsl.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    Imsl_d_mps *problem;
    int   i, m, n, *irtype, nza;
    double *x, objective, *bl, *bu, *xlb, *xub;
    Imsl_d_sparse_elem *a = NULL;

    /* Read the MPS file. */
    problem = imsl_d_read_mps("afiro", 0);

    m = problem->nrows;
    n = problem->ncolumns;

    /*
    * Setup the constraint matrix.
    */
    nza = problem->nonzeros;
    a = problem->constraint;

    /*
    * Setup constraint bounds and constraint type array.
    */
    irtype = (int*) malloc(m*sizeof(int));
    bl = (double*) malloc(m*sizeof(double));
    bu = (double*) malloc(m*sizeof(double));
```

```
for (i = 0; i < m; i++) {
    if (problem->lower_range[i] == problem->negative_infinity &&
        problem->upper_range[i] == problem->positive_infinity)
    {
        bl[i] = problem->negative_infinity;
        bu[i] = problem->positive_infinity;
        irtype[i] = 4;
    }
    else if (problem->lower_range[i] == problem->negative_infinity)
    {
        irtype[i] = 1;
        bl[i] = problem->upper_range[i];
        bu[i] = problem->positive_infinity;
    }
    else if (problem->upper_range[i] == problem->positive_infinity)
    {
        irtype[i] = 2;
        bl[i] = problem->lower_range[i];
        bu[i] = problem->positive_infinity;
    }
    else
    {
        if (problem->lower_range[i] == problem->upper_range[i])
        {
            irtype[i] = 0;
            bl[i] = problem->lower_range[i];
            bu[i] = problem->positive_infinity;
        }
        else
        {
            irtype[i] = 3;
            bl[i] = problem->lower_range[i];
            bu[i] = problem->upper_range[i];
        }
    }
}

/*
* Set up variable bounds. Be sure to account for
* how unbounded variables should be set.
*/
xlb = (double*) malloc(n*sizeof(double));
xub = (double*) malloc(n*sizeof(double));
for (i = 0; i < n; i++) {
    xlb[i] = (problem->lower_bound[i] == problem->negative_infinity)?
        -1.0e30:problem->lower_bound[i];
    xub[i] = (problem->upper_bound[i] == problem->positive_infinity)?
        1.0e30:problem->upper_bound[i];
}

/*
```

```
    * Solve the LP problem.
    */
    x = imsl_d_sparse_lin_prog(m, n, nza,
        a, bl, problem->objective,
        IMSL_UPPER_LIMIT, bu,
        IMSL_CONSTR_TYPE, irtype,
        IMSL_LOWER_BOUND, xlb,
        IMSL_UPPER_BOUND, xub,
        IMSL_OBJ, &objective,
        IMSL_PRESOLVE, 6,
        0);

    printf("Problem Name: %s\n", problem->name);
    printf("objective  : %15.10e\n", objective);
    imsl_d_write_matrix("Solution", 1, n, x, 0);

    /*
    * Free memory.
    */
    imsl_d_mps_free(problem);
    free(irtype);
    free(bl);
    free(bu);
    free(xlb);
    free(xub);
    imsl_free(x);
}
```

### Output

```
Problem Name: AFIRO
objective   : -4.6475314284e+002

                                Solution
         1           2           3           4           5           6
      80.0        25.5        54.5        84.8        65.4         0.0

         7           8           9          10          11          12
       0.0         0.0         0.0         0.0         0.0         0.0

        13          14          15          16          17          18
      18.2        47.2        69.4       500.0       475.9        24.1

        19          20          21          22          23          24
       0.0       215.0       141.7         0.0         0.0         0.0

        25          26          27          28          29          30
       0.0         0.0         0.0         0.0       339.9       242.3

        31          32
      60.9         0.0
```

## Warning Errors

| | |
|---|---|
| `IMSL_ALL_FEAS_SOLS_OPTIMAL` | The coefficients of the objective function are all equal to zero. Every feasible solution is also optimal. |
| `IMSL_SUBOPTIMAL_SOL_FOUND` | A suboptimal solution was found after #iterations. |
| `IMSL_MAX_ITERATIONS_REACHED_1` | The maximum number of iterations was reached. The best answer will be returned. "#"=#was used, a larger value may help complete the algorithm. |

## Fatal Errors

| | |
|---|---|
| `IMSL_PRIMAL_UNBOUNDED` | The primal problem is unbounded. |
| `IMSL_PRIMAL_INFEASIBLE` | The primal problem is infeasible. |
| `IMSL_DUAL_INFEASIBLE` | The dual problem is infeasible. |
| `IMSL_INIT_SOL_INFEASIBLE` | The initial solution for the one-row linear program is infeasible. |
| `IMSL_PROB_UNBOUNDED` | The problem is unbounded. |
| `IMSL_DIAG_WEIGHT_TOO_SMALL` | The diagonal element #[#]=#of the diagonal weight matrix #is too small. |
| `IMSL_CHOL_FAC_ACCURACY` | The Cholesky factorization failed because of accuracy problems. |

# sparse_quadratic_prog

Solves a sparse convex quadratic programming problem by an infeasible primal-dual interior-point method.

**NOTE:** Function `sparse_quadratic_prog` is available in double precision only.

## Synopsis

*#include* `<imsl.h>`

*double* `*imsl_d_sparse_quadratic_prog` (*int* `m`, *int* `n`, *int* `nza`, *int* `nzq`, *Imsl_d_sparse_elem* `a[]`, *double* `b[]`, *double* `c[]`, *Imsl_d_sparse_elem* `q[]`, …, 0)

## Required Arguments

*int* `m` (Input)
Number of constraints.

*int* `n` (Input)
Number of variables.

*int* `nza` (Input)
Number of nonzero entries in constraint matrix *A*.

*int* `nzq` (Input)
Number of nonzero entries in the lower triangular part of the matrix *Q* of the objective function.

*Imsl_d_sparse_elem* `a[]` (Input)
An array of length `nza` containing the location and value of each nonzero coefficient in the constraint matrix *A*.

*double* `b[]` (Input)
An array of length `m` 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.

*double* `c[]` (Input)
An array of length `n` containing the coefficients of the linear term of the objective function.

*Imsl_d_sparse_elem* `q[]` (Input)
Array of length `nzq` containing the location and value of each nonzero coefficient in the lower triangular part of the matrix *Q* of the objective function. The matrix must be symmetric positive semidefinite.

## Return Value

A pointer to an array of length `n` containing the solution *x* of the convex QP problem. To release this space, use `imsl_free`. If no solution can be computed, then `NULL` is returned.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*double* `*imsl_d_sparse_quadratic_prog` (*int* `m`, *int* `n`, *int* `nza`, *int* `nzq`, *Imsl_d_sparse_elem* `a[]`,
*double* `b[]`, *double* `c[]`, *Imsl_d_sparse_elem* `q[]`,

`IMSL_CONSTR_TYPE`, *int* `irtype[]`,

`IMSL_UPPER_LIMIT`, *double* `bu[]`,

`IMSL_LOWER_BOUND`, *double* `xlb[]`,

`IMSL_UPPER_BOUND`, *double* `xub[]`,

`IMSL_OBJ_CONSTANT`, *double* `c0`,

`IMSL_PREORDERING`, *int* `preorder`,

`IMSL_MAX_ITERATIONS`, *int* `max_iterations`,

`IMSL_OPT_TOL`, *double* `opt_tol`,

`IMSL_PRINF_TOL`, *double* `prinf_tol`,

`IMSL_DLINF_TOL`, *double* `dlinf_tol`,

`IMSL_PRINT`, *int* `iprint`,

`IMSL_PRESOLVE`, *int* `presolve`,

`IMSL_CSC_FORMAT`, *int* `a_colptr[]`, *int* `a_rowind[]`, *double* `a_values[]`,
*int* `q_colptr[]`, *int* `q_rowind[]`, *double* `q_values[]`,

`IMSL_TERMINATION_STATUS`, *int* `*status`,

`IMSL_OBJ`, *double* `*obj`,

`IMSL_ITERATION_COUNT`, *int* `*iterations`,

`IMSL_DUAL`, *double* `**y`,

`IMSL_DUAL_USER`, *double* `y[]`,

`IMSL_PRIMAL_INFEAS`, *double* `*err_b`, *double* `*err_u`,

`IMSL_DUAL_INFEAS`, *double* `*err_c`,

IMSL_CP_RATIO_SMALLEST, *double* *cp_smallest,

IMSL_CP_RATIO_LARGEST, *double* *cp_largest,

IMSL_RETURN_USER, *double* x[],

0)

## Optional Arguments

IMSL_CONSTR_TYPE, *int* irtype[] (Input)

An array of length m 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 | $r_i \le b_i$ |
| 2 | $r_i \ge b_i$ |
| 3 | $b_i \le r_i \le bu_i$ |
| 4 | Ignore this constraint |

Note that irtype[i] = 3 should only be used for constraints i with both finite lower and finite upper bound. For one-sided constraints, use irtype[i] = 1 or irtype[i] = 2. For free constraints, use irtype[i] = 4.

Default: irtype = 0

IMSL_UPPER_LIMIT, *double* bu[] (Input)

An array of length m containing the upper limit of the constraints that have both a lower and an upper bound. If such a constraint exists, then optional argument IMSL_CONSTR_TYPE must be used to define the type of constraints. If irtype[i] $\neq$ 3, i.e. if constraint i is not two-sided, then the corresponding entry in bu, bu[i], is ignored.

Default: None of the constraints has an upper bound.

IMSL_LOWER_BOUND, *double* xlb[] (Input)

An array of length n 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, *double* xub[] (Input)

An array of length n 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: None of the variables has an upper bound.

IMSL_OBJ_CONSTANT, *double* c0 (Input)

Value of the constant term in the objective function.

Default: c0 = 0.

IMSL_PREORDERING, *int* preorder (Input)

The variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix:

| preorder | Method |
|---|---|
| 0 | A variant of the MDO algorithm using pivotal cliques. |
| 1 | A variant of George and Liu's Quotient Minimum Degree algorithm. |

Default: preorder = 0.

IMSL_MAX_ITERATIONS, *int* max_iterations (Input)

The maximum number of iterations allowed for the primal-dual solver.

Default: max_iterations = 200.

IMSL_OPT_TOL, *double* opt_tol (Input)

Relative optimality tolerance.

Default: opt_tol = 1.0e-10.

IMSL_PRINF_TOL, *double* prinf_tol (Input)

The primal infeasibility tolerance.

Default: prinf_tol = 1.0e-8.

`IMSL_DLINF_TOL`, *double* `dlinf_tol` (Input)

The dual infeasibility tolerance.

Default: `dlinf_tol` = 1.0e-8.

`IMSL_PRINT`, *int* `iprint` (Input)

Printing option.

| `iprint` | Action |
|---|---|
| 0 | No printing is performed. |
| 1 | Prints statistics on the QP problem and the solution process. |

Default: `iprint` = 0.

`IMSL_PRESOLVE`, *int* `presolve` (Input)

Presolve the QP problem in order to reduce the problem size or to detect infeasibility or unboundedness of the problem. Depending on the number of presolve techniques used different presolve levels can be chosen:

| `presolve` | Description |
|---|---|
| 0 | No presolving. |
| 1 | Eliminate singleton rows |
| 2 | Additionally to 1, eliminate redundant (and forcing) rows. |
| 3 | Additionally to 2, eliminate dominated variables. |
| 4 | Additionally to 3, eliminate singleton columns. |
| 5 | Additionally to 4, eliminate doubleton rows. |
| 6 | Additionally to 5, eliminate aggregate columns. |

Default: `presolve` = 0.

`IMSL_CSC_FORMAT`, *int* `a_colptr[]`, *int* `a_rowind[]`, *double* `a_values[]`, *int* `q_colptr[]`, *int* `q_rowind[]`, *double* `q_values[]` (Input)

Accept the constraint matrix *A* (via vectors `a_colptr`, `a_rowind` and `a_values)` and the matrix *Q* of the objective function (via vectors `q_colptr`, `q_rowind` and `q_values)` in Harwell-Boeing format. See (Compressed Sparse Column (CSC) Format) in the Introduction to this manual for a discussion of this storage scheme.

If this optional argument is used, then required arguments `a` and `q` are ignored.

IMSL_TERMINATION_STATUS, *int* *status (Output)

The termination status for the problem.

| status | Description |
|---|---|
| 0 | Optimal solution found. |
| 1 | The problem is primal infeasible (or dual unbounded). |
| 2 | The problem is primal unbounded (or dual infeasible). |
| 3 | Suboptimal solution found (accuracy problems). |
| 4 | Iterations limit max_iterations exceeded. |
| 5 | An error outside of the solution phase of the algorithm, e.g. a user input or a memory allocation error. |

IMSL_OBJ, *double* *obj (Output)

Optimal value of the objective function.

IMSL_ITERATION_COUNT, *int* *iterations (Output)

The number of iterations required by the primal-dual solver.

IMSL_DUAL, *double* **y (Output)

The address of a pointer y to an internally allocated array of length m containing the dual solution.

IMSL_DUAL_USER, *double* y[] (Output)

A user-allocated array of size m containing the dual solution.

IMSL_PRIMAL_INFEAS, *double* *err_b, *double* *err_u (Output)

The violation of the primal constraints, described by err_b, the primal infeasibility of the solution, and by err_u, the violation of the variable bounds.

IMSL_DUAL_INFEAS, *double* *err_c (Output)

The violation of the dual constraints, described by err_c, the dual infeasibility of the solution.

IMSL_CP_RATIO_SMALLEST, *double* *cp_smallest (Output)

The ratio of the smallest complementarity product to the average.

IMSL_CP_RATIO_LARGEST, *double* *cp_largest (Output)

The ratio of the largest complementarity product to the average.

IMSL_RETURN_USER, *double* x[] (Output)

A user-allocated array of length n containing the primal solution.

# Description

The function imsl_d_sparse_quadratic_prog uses an infeasible primal-dual interior-point method to solve convex quadratic programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x + \frac{1}{2} x^T Q x$$
$$\text{subject to} \quad b_l \le Ax \le b_u$$
$$x_l \le x \le x_u$$

where $c$ is the objective coefficient vector, $Q$ is the symmetric positive semidefinite coefficient matrix, $A$ is the constraint 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.

Internally, `imsl_d_sparse_quadratic_prog` transforms the problem given by the user into a simpler form that is computationally more tractable. After redefining the notation, the new form reads

$$\begin{array}{lll} \min c^T x + \frac{1}{2} x^T Q x & & \\ \text{subject to} \quad Ax = b & & \\ \qquad x_i + s_i = u_i, & x_i, s_i \ge 0, & i \in I_u \\ \qquad x_j \ge 0, & & j \in I_s. \end{array}$$

Here, $I_u \cup I_s = \{1, \ldots, n\}$ is a partition of the index set $\{1, \ldots, n\}$ into upper bounded and standard variables.

In order to simplify the description it is assumed in the following that the problem above contains only variables with upper bounds, i.e. is of the form

$$(P) \qquad \begin{array}{ll} \min c^T x + \frac{1}{2} x^T Q x & \\ \qquad \text{subject to} & Ax = b, \\ & x + s = u, \\ & x, s \ge 0 \end{array}$$

The corresponding dual problem is then

$$(D) \qquad \begin{array}{ll} \max b^T y - u^T w - \frac{1}{2} x^T Q x & \\ \qquad \text{subject to} & A^T y + z - w - Qx = c, \\ & x, z, w \ge 0 \end{array}$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for (P) and (D) are

$$\begin{aligned}
&Ax = b, && (1.1)\\
&x + s = u, && (1.2)\\
&A^T y + z - w - Qx = c, && (1.3)\\
&XZe = 0, && (1.4)\\
&SWe = 0, && (1.5)\\
&x,z,s,w \geq 0, && (1.6)
\end{aligned}$$

where $X = diag(x)$, $Z = diag(z)$, $S = diag(s)$, $W = diag(w)$ are diagonal matrices and $e = (1, \ldots, 1)^T$ is a vector of ones.

Function `imsl_d_sparse_quadratic_prog`, like all infeasible interior point methods, generates a sequence

$$\left(x_k, s_k, y_k, z_k, w_k\right), \quad k = 0, 1, \ldots$$

of iterates, that satisfy $\left(x_k, s_k, y_k, z_k, w_k\right) > 0$ for all $k$, but are in general not feasible, i.e. the linear constraints (1.1)-(1.3) are only satisfied in the limiting case $k \longrightarrow \infty$.

The barrier parameter $\mu$, defined by

$$\mu = \frac{x^T z + s^T w}{2n}$$

measures how good the complementarity conditions (1.4), (1.5) are satisfied.

Mehrotra's predictor-corrector algorithm is a variant of Newton's method applied to the KKT conditions (1.1)-(1.5). Function `imsl_d_sparse_quadratic_prog` uses a modified version of this algorithm to compute the iterates $\left(x_k, s_k, y_k, z_k, w_k\right)$. In every step of the algorithm, the search direction vector

$$\Delta : = \left(\Delta x, \Delta s, \Delta y, \Delta z, \Delta w\right)$$

is decomposed into two parts, $\Delta = \Delta_a + \Delta_c^{\omega}$, where $\Delta_a$ and $\Delta_c^{\omega}$ denote the affine-scaling and a weighted centering component, respectively. Here,

$$\Delta_c^{\omega} : = \omega\left(\Delta x_c, \Delta s_c, \Delta y_c, \Delta z_c, \Delta w_c\right)$$

where the scalar $\omega$ denotes the corrector weight.

The vectors $\Delta_a$ and $\Delta_c : = \left(\Delta x_c, \Delta s_c, \Delta y_c, \Delta z_c, \Delta w_c\right)$ are determined by solving the linear system

$$\begin{bmatrix} A & 0 & 0 & 0 & 0 \\ I & 0 & I & 0 & 0 \\ -Q & A^T & 0 & I & -I \\ Z & 0 & 0 & X & 0 \\ 0 & 0 & W & 0 & S \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \\ \Delta z \\ \Delta w \end{bmatrix} = \begin{bmatrix} r_b \\ r_u \\ r_c \\ r_{xz} \\ r_{ws} \end{bmatrix} \quad (2)$$

for two different right-hand sides.

For $\Delta_a$, the right-hand side is defined as

$$\left(r_b, r_u, r_c, r_{xz}, r_{ws}\right) = \left(b - Ax, u - x - s, c - A^T y - z + w + Qx, \; - XZe, \; - WSe\right).$$

Here, $r_b$ and $r_u$ are the violations of the primal constraints and $r_c$ defines the violations of the dual constraints.

The resulting direction $\Delta_a$ is the pure Newton step applied to the system (1.1)-(1.5).

In order to obtain the corrector direction $\Delta_c$, the maximum stepsize $\alpha_a$ in the primal and dual space preserving nonnegativity of $(x, s, z, w)$, is determined and the predicted complementarity gap

$$g_a = \left(x + \alpha_a \Delta x_a\right)^T \left(z + \alpha_a \Delta z_a\right) + \left(s + \alpha_a \Delta s_a\right)^T \left(w + \alpha_a \Delta w_a\right)$$

is computed. It is then used to determine the barrier parameter

$$\hat{\mu} = \left(\frac{g_a}{g}\right)^2 \frac{g_a}{2n},$$

where $g = x^T z + s^T w$ denotes the current complementarity gap.

The direction $\Delta_c$ is then computed by choosing

$$\left(r_b, r_u, r_c, r_{xz}, r_{sw}\right) = \left(0, 0, 0, \hat{\mu}e - \Delta X_a \Delta Z_a e, \hat{\mu}e - \Delta W_a \Delta S_a e\right)$$

as the right-hand side in the linear system (2).

Function `imsl_d_sparse_quadratic_prog` now uses a linesearch to find the optimal weight $\hat{\omega}$ that maximizes the stepsize $\alpha_{PD}$ in the primal and dual direction of $\Delta = \Delta_a + \Delta_c^{\omega}$.

A new iterate is then computed using a step reduction factor $\alpha_0 = 0.99995$:

$$\left(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}, w_{k+1}\right) = \left(x_k, s_k, y_k, z_k, w_k\right) + \alpha_0 \alpha_{PD} \left(\Delta x, \Delta s, \Delta y, \Delta z, \Delta w\right)$$

In addition to the weighted Mehrotra predictor-corrector, `imsl_d_sparse_quadratic_prog` also uses multiple centrality correctors to enlarge the primal-dual stepsize per iteration step and to reduce the overall number of iterations required to solve a QP problem. The maximum number of centrality corrections depends on the ratio of the factorization and solve efforts for system (2) and is therefore problem dependent. For a detailed description of multiple centrality correctors, refer to Gondzio(1994).

The linear system (2) can be reduced to more compact forms, the augmented system (AS)

$$\begin{bmatrix} -Q-\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r \\ h \end{bmatrix} \qquad (3)$$

or further by elimination of $\Delta x$ to the normal equations (NE) system

$$A\left(Q+\Theta^{-1}\right)^{-1}A^T\Delta y = A\left(Q+\Theta^{-1}\right)^{-1}r+h, \quad (4)$$

where

$$\Theta = \left(X^{-1}Z+S^{-1}W\right)^{-1},\ r = r_c - X^{-1}r_{xz} + S^{-1}r_{ws} - S^{-1}Wr_u,\ h = r_b.$$

The matrix on the left-hand side of (3), which is symmetric indefinite, can be transformed into a symmetric quasidefinite matrix by regularization. Since these types of matrices allow for a Cholesky-like factorization, the resulting linear system can be solved easily for $(\Delta x, \Delta y)$ by triangular substitutions. For more information on the regularization technique, see Altman and Gondzio (1998). For the NE system, matrix $A\left(Q+\Theta^{-1}\right)A^T$ is positive definite, and therefore a sparse Cholesky algorithm can be used to factor $A\left(Q+\Theta^{-1}\right)A^T$ and solve the system for $\Delta y$ by triangular substitutions with the Cholesky factor $L$.

In function `imsl_d_sparse_quadratic_prog`, both approaches are implemented. The AS approach is chosen if $A$ contains dense columns, if there is a considerable number of columns in $A$ that are much denser than the remaining ones or if there are many more rows than columns in the structural part of $A$. Otherwise, the NE approach is selected.

Function `imsl_d_sparse_quadratic_prog` stops with optimal termination status if the current iterate satisfies the following three conditions:

$$\frac{\mu}{1+0.5\left(|c^Tx| + |b^Ty-u^Tw-0.5x^TQx|\right)} \le \texttt{opt\_tol}$$

$$\frac{\|\left(b-Ax,x+s-u\right)\|}{1+\|\left(b,u\right)\|} \le \texttt{print\_tol}, \text{ and}$$

$$\frac{\|c-A^Ty-z+w+Qx\|}{1+\|c\|} \le \texttt{dling\_tol},$$

where `prinf_tol`, `dlinf_tol` and `opt_tol` are primal infeasibility, dual infeasibility and optimality tolerances, respectively. The default value is 1.0e-10 for `opt_tol` and 1.0e-8 for the two other tolerances.

Function `imsl_d_sparse_quadratic_prog` is based on the code HOPDM developed by Jacek Gondzio et al., see the HOPDM User's Guide (1995).

# Examples

## Example 1

The convex quadratic programming problem

$$\begin{aligned} \min f(x) &= 10x_1 + 3x_3 + 0.5(2x_1^2 + 32x_2^2 + 4x_3^2 - 8x_1x_2) \\ \text{subject to} \quad & 2x_1 + x_2 - 8x_3 \geq 0 \\ & 2x_1 + 3x_2 \leq 6 \\ & 0 \leq x_1 \leq 7 \\ & -3 \leq x_2 \leq 2 \\ & -5 \leq x_3 \leq 20 \end{aligned}$$

is solved.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int m = 2, n = 3, nza = 5, nzq = 4;
   Imsl_d_sparse_elem a[] = { 0, 0, 2.0,
       0, 1, 1.0,
       0, 2, -8.0,
       1, 0, 2.0,
       1, 1, 3.0 };

   Imsl_d_sparse_elem q[] = { 0, 0, 2.0,
       1, 1, 32.0,
       2, 2, 4.0,
       1, 0, -4.0 };

   double b[] = { 0.0, 6.0 };
   double c[] = { 10.0, 0.0, 3.0 };

   double xlb[] = { 0.0, -3.0, -5.0 };
   double xub[] = { 7.0, 2.0, 20.0 };

   int irtype[] = { 2, 1 };
   double *x = NULL;
   double obj;

   x = imsl_d_sparse_quadratic_prog(m, n, nza, nzq, a, b, c, q,
       IMSL_CONSTR_TYPE, irtype,
       IMSL_LOWER_BOUND, xlb,
       IMSL_UPPER_BOUND, xub,
       IMSL_OBJ, &obj,
       0);

   imsl_d_write_matrix("x", 1, n, x, 0);
   printf("\nObjective: %lf\n", obj);
}
```

### Output

```
                x
        1             2             3
     0.00          0.00         -0.75

Objective: -1.125000
```

## Example 2

This example demonstrates how the function imsl_d_read_mps can be used with imsl_d_sparse_quadratic_prog to solve a convex quadratic programming problem defined in an MPS file. The MPS file used in this example is the file ‘qafiro’, available from the QP problems collection QPDATA2 on István Maros’ home page under http://www.doc.ic.ac.uk/~im/#DATA/.

```
#include <imsl.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    Imsl_d_mps *problem;
    int    i, m, n, *irtype, nza, nzq;
    double *x, objective, *bl, *bu, *xlb, *xub;
    Imsl_d_sparse_elem *a = NULL, *q = NULL;

    /* Read the QPS file. */
    problem = imsl_d_read_mps("QAFIRO.QPS", 0);

    m = problem->nrows;
    n = problem->ncolumns;

    /*
    * Setup the constraint matrix.
    */
    nza = problem->nonzeros;
    a = problem->constraint;

    /*
    * Setup the Hessian.
    */
    nzq = problem->nhessian;
    q = problem->hessian;

    /*
    * Setup constraint bounds and constraint type array.
    */
    irtype = (int*) malloc(m*sizeof(int));
    bl = (double*) malloc(m*sizeof(double));
    bu = (double*) malloc(m*sizeof(double));
```

```
for (i = 0; i < m; i++) {
    if (problem->lower_range[i] == problem->negative_infinity &&
        problem->upper_range[i] == problem->positive_infinity)
    {
        bl[i] = problem->negative_infinity;
        bu[i] = problem->positive_infinity;
        irtype[i] = 4;
    }
    else if (problem->lower_range[i] == problem->negative_infinity)
    {
        irtype[i] = 1;
        bl[i] = problem->upper_range[i];
        bu[i] = problem->positive_infinity;
    }
    else if (problem->upper_range[i] == problem->positive_infinity)
    {
        irtype[i] = 2;
        bl[i] = problem->lower_range[i];
        bu[i] = problem->positive_infinity;
    }
    else
    {
        if (problem->lower_range[i] == problem->upper_range[i])
        {
            irtype[i] = 0;
            bl[i] = problem->lower_range[i];
            bu[i] = problem->positive_infinity;
        }
        else
        {
            irtype[i] = 3;
            bl[i] = problem->lower_range[i];
            bu[i] = problem->upper_range[i];
        }
    }
}

/*
* Set up variable bounds. Be sure to account for
* how unbounded variables should be set.
*/
xlb = (double*) malloc(n*sizeof(double));
xub = (double*) malloc(n*sizeof(double));

for (i = 0; i < n; i++) {
    xlb[i] = (problem->lower_bound[i] == problem->negative_infinity)?
        -1.0e30:problem->lower_bound[i];
    xub[i] = (problem->upper_bound[i] == problem->positive_infinity)?
        1.0e30:problem->upper_bound[i];
}
```

```
    /*
    * Solve the QP problem.
    */
    x = imsl_d_sparse_quadratic_prog(m, n, nza, nzq,
        a, bl, problem->objective, q,
        IMSL_UPPER_LIMIT, bu,
        IMSL_CONSTR_TYPE, irtype,
        IMSL_LOWER_BOUND, xlb,
        IMSL_UPPER_BOUND, xub,
        IMSL_OBJ, &objective,
        IMSL_PRESOLVE, 6,
        0);

    /*
    * Output results.
    */
    printf("Problem Name: %s\n", problem->name);
    printf("objective   : %15.10e\n", objective);
    imsl_d_write_matrix("Solution", 1, n, x, 0);

    /*
    * Free memory.
    */
    imsl_d_mps_free(problem);
    free(irtype);
    free(bl);
    free(bu);
    free(xlb);
    free(xub);
    imsl_free(x);
}
```

### Output

```
Problem Name: AFIRO
objective   : -1.5907817909e+000

                               Solution
        1            2            3            4            5            6
     0.38         0.00         0.38         0.40        65.17         0.00

        7            8            9           10           11           12
     0.00         0.00         0.00         0.00         0.00         0.00

       13           14           15           16           17           18
     0.00        65.17        69.08         3.49         3.37         0.11

       19           20           21           22           23           24
     0.00         1.50        12.69         0.00         0.00         0.00

       25           26           27           28           29           30
     0.00         0.00         0.00         0.00         2.41        33.72

       31           32
     5.46         0.00
```

## Warning Errors

| | |
|---|---|
| IMSL_SUBOPTIMAL_SOL_FOUND | A suboptimal solution was found after #iterations. |
| IMSL_MAX_ITERATIONS_REACHED_1 | The maximum number of iterations was reached. The best answer will be returned. "#"=#was used, a larger value may help complete the algorithm. |

## Fatal Errors

| | |
|---|---|
| IMSL_PRIMAL_UNBOUNDED | The primal problem is unbounded. |
| IMSL_PRIMAL_INFEASIBLE | The primal problem is infeasible. |
| IMSL_DUAL_INFEASIBLE | The dual problem is infeasible. |
| IMSL_INIT_SOL_INFEASIBLE | The initial solution for the one-row linear program is infeasible. |
| IMSL_PROB_UNBOUNDED | The problem is unbounded. |
| IMSL_DIAG_WEIGHT_TOO_SMALL | The diagonal element #[#]=#of the diagonal weight matrix #is too small. |
| IMSL_CHOL_FAC_ACCURACY | The Cholesky factorization failed because of accuracy problems. |

# min_con_gen_lin

more...

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 $a_{k,0}x_0 + \ldots + a_{k,\text{nvar}-1}x_{\text{nvar}-1} = b_k$, $k = 0, \ldots,$ neq − 1 and $a_{k,0}x_0 + \ldots + a_{k,\text{nvar}-1}x_{\text{nvar}-1} \le b_k$, $k =$ 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*] $\leq x_i$, for *i* = 1, …, nvar.

*float* xub[] (Input)

Array of length nvar containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. Specifically, these simple bounds are $x_i \leq$ xub[*i*], for *i* = 1, nvar.

# Return Value

A pointer to the solution *x*. To release this space, use imsl_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_CONSTRAINTS, *int* **iact,

IMSL_ACTIVE_CONSTRAINTS_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* gradient(), *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_CONSTRAINTS, *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_CONSTRAINTS_USER, *int* × iact_user (Output)

A user-supplied array of length at least ncon + 2

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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to 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

$$A_1 x = b_1$$
$$A_2 x \le b_2$$
$$x_t \le x \le z_u$$

given the vectors $b_1$, $b_2$, $x_l$ ,and $x_u$ and the matrices $A_1$ and $A_2$.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise $x^0$, the initial guess, to satisfy

$$A_1 x = b_1$$

Next, $x^0$ is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible $x^k$, let $J^k$ be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let $I_k$ be the set of indices of active constraints. The following quadratic programming problem

$$\min f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

subject to

$$a_j d = 0,\ j \in I_k$$
$$a_j d \le 0,\ j \in J_k$$

is solved to get ($d^k$, $\lambda^k$) where $a_j$ is a row vector representing either a constraint in $A_1$ or $A_2$ or a bound constraint on $x$. In the latter case, the $a_j = e_i$ for the bound constraint $x_i \le (x_u)_i$ 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 $\lambda^k$ are the Lagrange multipliers, and $B^k$ is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction $d^k$ is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f\left(x^k + \alpha^k d^k\right) \le f\left(x^k\right) + 0.1\alpha^k \left(d^k\right)^T \nabla f\left(x^k\right)$$

and

$$\left(d^K\right)^T \nabla f\left(x^k + \alpha^k d^k\right) \ge 0.7 \left(d^k\right)^T \nabla f\left(x^K\right)$$

The main idea in forming the set $J_k$ is that, if any of the equality constraints restricts the step-length $\alpha^k$, then its index is not in $J_k$. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation $B^K$, is updated by the BFGS formula, if the condition

$$\left(d^K\right)^T \nabla f\left(x^k + \alpha^k d^k\right) - \nabla f\left(x^K\right) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\| \nabla \mathrm{f}\left(x^k\right) - \mathrm{A}^k \lambda^K \|_2 \le \tau$$

is satisfied. Here $\tau$ 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 non-critical point. In such cases, high precision arithmetic is recommended. Also, if the gradient can be easily provided, the option `IMSL_GRADIENT` should be used.

On some platforms, `imsl_f_min_con_gen_lin` can evaluate the user-supplied functions `fcn` and `gradient` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables

# Examples

## Example 1

In this example, the problem

$$\begin{aligned}
&\min f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1 \\
&\text{subject to} \\
&\quad x_1 + x_2 + x_3 + x_4 + x_5 = 5 \\
&\quad x_3 - 2x_4 - 2x_5 = -3 \\
&\quad 0 \le x \le 10
\end{aligned}$$

is solved.

```
#include <imsl.h>

int 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;

       imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
       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)

$$
\begin{aligned}
&\min f(x) = -x_0 x_1 x_2 \\
&\text{subject to} \\
&\quad -x_0 - 2x_1 - 2x_2 \le 0 \\
&\quad x_0 + 2x_1 + 2x_2 \le 72 \\
&\quad 0 \le x_0 \le 20 \\
&\quad 0 \le x_1 \le 11 \\
&\quad 0 \le x_2 \le 42
\end{aligned}
$$

is solved with an initial guess of $x_0$ =10, $x_1$ =10 and $x_2$ =10.

```
#include <imsl.h>

int 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_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 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
```

## Fatal Errors

| | |
|---|---|
| IMSL_STOP_USER_FCN | Request from user supplied function to stop algorithm. User flag = "#". |

# bounded_least_squares

more...

more...

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 imsl_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 partial derivative $\partial f_i / \partial x_j$ should be returned in fjac[(i-1)*fjac_col_dim+j-1].

IMSL_XSCALE, *float* xscale[] (Input)

Array with n components containing the scaling vector for the variables. 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 second bounded_least_squares stopping criterion occurs when

$\max_{i=1}^{n}[g_{si}(x)] \le$ grad_tol

where $g_{si}(x)$ is component *i* of the scaled gradient of *F* at *x*, defined as:

$$g_{si}(x) = \frac{|g_i(x)| * \max\left(|x_i|, 1/s_i\right)}{\frac{1}{2}||F(x)||_2^2}$$

$$g_i(x) = f_{si}^{\;2} * \nabla_i\left[\frac{1}{2}\|F(x)\|_2^2\right] = f_{si}^{\;2} * (J^T F)_i$$

$$\left\|F(x)\right\|_2^2 = \sum_{j=1}^{m} f_j(x)^2$$

and where ***J*** is the Jacobian matrix for ***m***-component function vector *F*(*x*) with ***n***-component argument *x* with $J_{ji} = \nabla_i f_j(x)$, $s_i$ = `xscale[i-1]`, and $f_{si}$ = `fscale[i-1]`.

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 third `bounded_least_squares` stopping criterion occurs when

$\max_{i=1}^{n}[\Delta x_i] \le$ `step_tol`, where:

*x* and *y*, and *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.

The first `bounded_least_squares` stopping criterion occurs when objective function

$\frac{1}{2}\left\|F(x)\right\|_2^2 \le$ `afcn_tol`.

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

$$\varepsilon_1 = \left(\sum_{i=1}^{n}\left(s_i t_i\right)^2\right)^{1/2}, \varepsilon_2 = \left\|s\right\|_2$$

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` × `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` × `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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to 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 $m \ge n$, $F : \mathfrak{R}^n \rightarrow \mathfrak{R}^m$, and $f_i(x)$ is the $i$-th component function of $F(x)$. From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where $\mu$ is the Levenberg-Marquardt parameter, $F = F(x)$, and $J$ is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are

$$\|g(x_i)\| \le \varepsilon, \ l_i < x_i < u_i$$

$$g(x_i) < 0, \ x_i = u_i$$

$$g(x_i) > 0, \ x_i = l_i$$

where $\varepsilon$ is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detail on the active set strategy, see Gill and Murray (1976).

The first stopping criterion for imsl_f_bounded_least_squares occurs when the norm of the function is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the scaled gradient tolerance. The third stopping criterion occurs when the scaled distance between the last two steps is less than the step tolerance. See options IMSL_ABS_FCN_TOL, IMSL_GRAD_TOL, and IMSL_STEP_TOL for details.

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.

On some platforms, imsl_f_bounded_least_squares can evaluate the user-supplied functions fcn and jacobian in parallel. This is done only if the function imsl_omp_options is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

# 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

$$f_0(x) = 10\left(x_1 - x_0^2\right) \text{ and } f_1(x) = \left(1 - x_0\right)$$

is solved with an initial guess (−1.2, 1.0).

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define M      2
#define N      2
#define LDFJAC 2

int 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];
}
```

## Example 2

This example solves the nonlinear least-squares problem

$$\min \frac{1}{2}\sum_{i=0}^{1} f_i(x)^2$$
$$-2 \le x_0 \le 0.5$$
$$-1 \le x_1 \le 2$$

where

$$f_0(x) = 10\left(x_1 - x_0^2\right) \text{ and } f_1(x) = \left(1 - x_0\right)$$

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 <stdio.h>
#include <math.h>

#define M      2
#define N      2
#define LDFJAC 2

int 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
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# constrained_nlp

more...

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. iact =1 to meq for equality constraints and iact =meq+1 to m for inequality constraints.

*float* result [] (Output)
If iact is zero, then result is the computed objective function at the point x. If iact is non-zero, 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, *int* 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

**Arguments**

*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. iact =1 to meq for equality constraints and iact =meq+1 to m for inequality constraints.

*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. |
| 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.

Default: xguess = X, (with the smallest value of $\|x\|_2$) that satisfies the bounds.

IMSL_ITMAX, *int* itmax (Input)

Maximum number of iterations allowed.

Default: itmax = 200.

IMSL_TAU0, *float* tau0 (Input)

A universal bound describing how much the unscaled penalty-term may deviate from zero.

imsl_f_constrained_nlp assumes that within the region described by

$$\sum_{i=1}^{M_e} |g_i(x)| - \sum_{i=M_e+1}^{M} \min\left(0, g_i(x)\right) \le tau0$$

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

$$\frac{g_i(x)}{\max\left(1, \|\nabla g_i(x)\|\right)} \le del0 \quad i = M_{\mathrm{e}} + 1, \dots, M$$

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 $|g_i(x)| \le$ `delmin` for equality constraints, and $g_i(x) \ge$(−`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_LAGRANGE_MULTIPLIERS`, ***float*** `**lagrange` (Output)

The address of a pointer, which on exit, points to an array containing the Lagrange multiplier estimates of the constraints.

IMSL_LAGRANGE_MULTIPLIERS_USER, *float* lagrange_user[] (Output)

A user-supplied array of length ncon containing the Lagrange multiplier estimates of the constraints.

IMSL_CONSTRAINT_RESIDUALS, *float* **const_res (Output)

The address of a pointer, which on exit, points to an array containing the constraints residuals.

IMSL_CONSTRAINT_RESIDUALS_USER, *float* const_res_user[] (Output)

A user-supplied array of length ncon containing the constraint residuals.

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 Passing Data to User-Supplied Functions in the introduction to 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 Passing Data to User-Supplied Functions in the introduction to this manual for more details.

**Note**: 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]. 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` = $\varepsilon$ where $\varepsilon$ is the machine precision.

`IMSL_EPSFCN`, *float* `epsfcn` (Input)

Relative precision of the function evaluation routine.

Default: `epsfcn` = $\varepsilon$ where $\varepsilon$ 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.

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:

$$\begin{aligned}
&\min_{x\in R^n} f(x) \\
&\text{subject to} \quad g_j(x) = 0, \text{ for } j = 1, \ldots, m_e \\
&\qquad\qquad\quad g_j(x) \geq 0, \text{ for } j = m_e + 1, \ldots, m \\
&\qquad\qquad\quad x_1 \leq x \leq x_u
\end{aligned}$$

Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of optional arguments, `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 perfo-mance 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 $l_i \le g_i(x) \le u_i$ is transformed into two constraints $g_{2i}(x) \ge 0$ and $g_{2i+1}(x) \ge 0$ , then choose del0 < ½$(u_i - l_i)$ /max{1,$\|\nabla g_i(x)\|$}, or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See optional argument `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.

On some platforms, `constrained_nlp` can evaluate the user-supplied functions `fcn` and `grad` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

## Example

The problem

$$\begin{aligned} \min F(x) &= (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to} \quad g_1(x) &= x_1 - 2x_2 + 1 = 0 \\ g_2(x) &= -x_1^2/4 - x_2^2 + 1 \ge 0 \end{aligned}$$

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);

int main()
{
   int ibtype = 0;
   float *x, ans[2];
   static float xlb[N], xub[N];

   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

  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
```

## Fatal Errors

| | |
|---|---|
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# jacobian

Approximates the Jacobian of *m* functions in *n* unknowns using divided differences.

## Synopsis

*#include* <imsl.h>

*void* imsl_f_jacobian (*void* fcn (), *int* m, *int* n, *float* y[], *float* f[], *float* fjac[], …, 0);

The type *double* function is imsl_d_jacobian.

## Required Arguments

*void* fcn (*int* indx, *float* y[], *float* f[]) (Input/Output)
User-supplied function to compute the value of the function whose Jacobian is to be calculated using divided differences and/or the value of the analytic derivative of that function.

**Required Arguments**

*int* indx (Input)
Index of the variable whose derivative is to be computed. imsl_f_jacobian sets this argument to the index of the variable whose derivative is being computed. In those cases where finite differencing and direct analytic computation are combined to calculate a derivative (see optional argument IMSL_ACCUMULATE), imsl_f_jacobian makes an extra call to fcn (with argument indx negative) each time the derivative with respect to variable indx is calculated, in order to calculate the analytic component of that derivative. Note that indx runs from 1 to n, where n is the number of variables.

*float* y[] (Input)
Array of length n containing the point at which the function is to be computed.

*float* f[] (Output)
Array of length m, where m is the number of functions to be evaluated at point y, containing the function values at point y. Normally, the user returns the values of the functions evaluated at point y in f. However, when the function can be broken into two parts, a part which is known analytically and a part to be differenced, fcn is called by imsl_f_jacobian once with indx positive for the portion to be differenced and again with indx negative for the portion which is known analytically. In the case where optional argument IMSL_ACCUMULATE has been specified by the user, fcn must be written to handle the known analytic portion separately from the part to be differenced. (See Example 4 for an example where IMSL_ACCUMULATE is used.)

*int* m (Input)
The number of equations.

*int* `n` (Input)

The number of variables.

*float* `y[]` (Input)

Array of length `n` containing the point at which the Jacobian is to be evaluated.

*float* `f[]` (Output)

Array of length `m` containing the function values at point `y`.

*float* `fjac[]` (Input/Output)

`m` by `n` array which, on output, contains the estimated Jacobian. Note that if optional argument `IMSL_METHOD, method,` is used, then for each variable `i` for which `method[i]` is set to `IMSL_DD_SKIP`, array elements `fjac[j=0,…,m-1][i]` are input arguments and must be set to the analytic derivatives with respect to variable `i` prior to calling `imsl_f_jacobian`. (See description of optional argument `IMSL_METHOD` and Example 3 below).

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*void* `imsl_f_jacobian` (*void* `fcn()`, *int* `m`, *int* `n`, *float* `y[]`, *float* `f[]`, *float* `fjac[]`,

`IMSL_YSCALE`, *float* `scale[]`,

`IMSL_METHOD`, *int* `method[]`,

`IMSL_ACCUMULATE`,

`IMSL_FACTOR`, *float* `factor[]`,

`IMSL_ISTATUS`, *int* `istatus[]`,

`IMSL_FCN_W_DATA`, *void* `fcn_w_data()`,

*void* `*data`,

`0);`

## Optional Arguments

`IMSL_YSCALE`, *float* `scale[]` (Input)

An array of length `n` containing the diagonal scaling matrix for the variables. `scale` can also be used to provide appropriate signs for the increments. If the user sets `scale[0]` to `0.0`, then differencing increment $del_j$ (for variable $j = 1, \ldots, n$) is set to $\sigma_j$ * |`y[j-1]`| * `factor[j-1]`. Otherwise, $del_j$ is set to $\sigma_j$ * |`scale[j-1]`| * `factor[j-1]`. (See the discussion of optional argument `IMSL_FACTOR` below for more information about the calculation of $del_j$.)

Default: `scale[i=0,…,n-1]` = 1.0.

IMSL_METHOD, *int* method[] (Input)

An array of length n containing the methods used to compute the derivatives. method[i] is the method to be used for variable i. method[i] can be one of the values in the following table:

| Value | Description |
|---|---|
| IMSL_DD_ONE_SIDED | Indicates one-sided differences. |
| IMSL_DD_CENTRAL | Indicates central differences. |
| IMSL_DD_SKIP | Indicates that the user has set the input Jacobian fjac[j=0,...,m-1][i] to the exact analytic derivative of the function with respect to variable i at point y[i], and that the calculation of the divided difference approximation is to be skipped. |

See Example 2 and Example 3 below for demonstrations of how this optional argument is used.

Note that if (and only if) IMSL_DD_SKIP is specified for a variable i, the required array elements fjac[j=0,…,m-1][i] must be set to the analytic derivatives with respect to variable i prior to calling imsl_f_jacobian. See Example 3 below.

Default: If optional argument IMSL_METHOD is not used, then one-sided differences are used for all variables.

IMSL_ACCUMULATE (Input)

Indicates that divided differences are to be accumulated with a Jacobian value previously initialized by the user with analytically calculated components of the derivatives via a call to fcn using negative values of fcn argument indx. See the description of indx above and Example 4 below.

IMSL_FACTOR, *float* factor[] (Input)

An array of length n containing the percentage factor for differencing.

For each divided difference for variable $j = 1, \ldots, n$, the differencing increment used is $del_j$. (See the Description below for a discussion of the differencing methods.) The value of $del_j$ is computed as follows: If scale[0] has been set to 0.0 (see the description of optional argument IMSL_YSCALE above), define $y_{a,j}$ = |y[j-1]| and $\sigma_j = 1$. Otherwise, if scale[j-1] {< , >} 0, define $y_{a,j}$ = |scale[j-1]| and $\sigma_j = \{-1, 1\}$. Finally, compute $del_j = \sigma_j \, y_{a,j}$ factor[j-1].

By changing the sign of scale[j-1], the difference $del_j$ can have any desired orientation, such as staying within bounds on variable $j$. For central differences, a reduced factor is used for $del_j$ that normally results in relative errors as small as $\varepsilon^{2/3}$, where $\varepsilon$ == machine precision = {imsl_f_machine(4), imsl_d_machine(4)} for {single, double} precision. The elements of factor must be such that: $\varepsilon^{3/4} \le$ factor[j-1] $\le 0.1$.

Default: All elements of factor are set to $\varepsilon^{1/2}$.

`IMSL_ISTATUS`, *int* `istatus[]` (Output)

Array of length 10 containing status information that might prove useful to a user wanting to gain better control over the differencing parameters. This information can often be ignored. The following table describes the diagnostic information that is returned in each of the entries of array `istatus[]`:

| Index | Description |
|---|---|
| 0 | The number of times a function evaluation was computed. |
| 1 | The number of columns in which three attempts were made to increase a percentage factor for differencing (i.e., a component in the `factor` array) but the computed $del_j$ (for $j$ = 1, …, $n$) remained unacceptably small relative to `y[j-1]` or `scale[j-1]`. In such cases the percentage factor is set to the square root of machine precision. |
| 2 | The number of columns in which the computed $del_j$ was zero to machine precision because `y[j-1]` or `scale[j-1]` was zero. In such cases $del_j$ is set to the square root of machine precision. |
| 3 | The number of Jacobian columns that had to be recomputed because the largest difference formed in the column was close to zero relative to scale, where $$scale = \max\left( \lvert f_i(\mathbf{y}) \rvert \text{ , } \lvert f_i\left(\mathbf{y} + del_j \mathbf{e_j}\right) \rvert \right)$$ and $i$ denotes the row index of the largest difference in the column currently being processed. *index* = 9 gives the last column where this occurred. |
| 4 | The number of columns whose largest difference is close to zero relative to scale after the column has been recomputed. |
| 5 | The number of times scale information was not available for use in the round-off and truncation error tests. This occurs when $$\min\left( \lvert f_i(\mathbf{y}) \rvert \text{ , } \lvert f_i\left(\mathbf{y} + del_j \mathbf{e_j}\right) \rvert \right) = 0$$ where $i$ is the index of the largest difference for the column currently being processed. |
| 6 | The number of times the increment for differencing ($del$) was computed and had to be increased because (`scale[j-1]` + $del_j$) − `scale[j-1]` was too small relative to `y[j-1]` or `scale[j-1]`. |
| 7 | The number of times a component of the `factor` array was reduced because changes in function values were large and excess truncation error was suspected. *index* = 8 gives the last column in which this occurred. |

| Index | Description |
|---|---|
| 8 | The index of the last column where the corresponding component of the `factor` array had to be reduced because excessive truncation error was suspected. |
| 9 | The index of the last column where the difference was small and the column had to be recomputed with an adjusted increment (see *index* = 3). The largest derivative in this column may be inaccurate due to excessive round-off error. |

`IMSL_FCN_W_DATA`, *void* `fcn_w_data()`, *void* `*data[]` (Input/Output)

User supplied function whose Jacobian is being calculated, and which can also accept a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See Example 4 for a demonstration of how this optional argument is used and Passing Data to User-Supplied Functions in the "Introduction" chapter for more details on the use of `IMSL_FCN_W_DATA`.

# Description

The function `imsl_f_jacobian` computes the Jacobian matrix for a function *f*(*y*) with *m* components and *n* independent variables. `imsl_f_jacobian` uses divided finite differences to compute the Jacobian. This function is designed for use in numerical methods for solving nonlinear problems where a Jacobian is evaluated repeatedly at neighboring arguments. For example, this occurs in a Gauss-Newton method for solving non-linear least squares problems, or in a non-linear optimization method.

`imsl_f_jacobian` is suited for applications where the Jacobian is a dense matrix. All cases $m < n$, $m = n$, or $m > n$ are allowed. Both one-sided and central divided differences can be used.

The design allows for computation of derivatives in a variety of contexts. Note that a gradient should be considered as the special case with $m = 1$, $n \geq 1$. A derivative of a single function of one variable is the case $m = 1$, $n = 1$. Any non-linear solving routine that optionally requests a Jacobian or gradient can use `imsl_f_jacobian`. This should be considered if there are special properties or scaling issues associated with *f*(*y*). Use the optional argument `IMSL_METHOD` to specify different differencing options for numerical differentiation. These can be combined with some analytic subexpressions or other known relationships.

Two divided differences methods are implemented in `imsl_f_jacobian` for computing the Jacobian: ***one-sided*** and ***central*** differences.

One-sided differences are computed:

$$\frac{\partial f_i(\mathbf{y})}{\partial y_j} = \frac{f_i(\mathbf{y} + del_j \mathbf{e_j}) - f_i(\mathbf{y})}{del_j}$$

using values of the independent variables at the Jacobian evaluation point $\mathbf{y} = \{ y_j, j=1,\ldots,n \}$ and differenced points $\mathbf{y} + del_j\, \mathbf{e}_j$, where the $\mathbf{e}_j$, $j=1,\ldots,n$ are the unit coordinate vectors.

Central differences are computed:

$$\frac{\partial f_i(\mathbf{y})}{\partial y_j} = \frac{f_i(\mathbf{y} + del_j\mathbf{e_j}) - f_i(\mathbf{y} - del_j\mathbf{e_j})}{2\ del_j}$$

The value for each difference $del_j$ depends on the variable $y_j$ , the differencing method, and the scaling for that variable. This difference is computed internally. See `IMSL_FACTOR` for computational details. $f_i(\mathbf{y})$ is evaluated with user-supplied argument `fcn`, where index $j$, variable $\mathbf{y}$, and output `f` == $f_i(\mathbf{y})$ are arguments to `fcn`.

There are five examples provided that illustrate various ways to use `imsl_f_jacobian`. For a discussion of the expected errors for the difference methods, see Ralston (1965).

Function `imsl_f_jacobian` is based upon the Fortran 77 program SJACG, which was designed and programmed by D. A. Salane, Sandia Labs (1986) and modifed by R. J. Hanson, Rice University (June, 2002) with advice from F. T. Krogh. See Salane (1986).

# Examples

## Example 1

In this example, the Jacobian matrix of

$$f_1(x) = x_1x_2 - 2$$
$$f_2(x) = x_1 - x_1x_2 + 1$$

is estimated by the finite-difference method at the point (1.0, 1.0).

```
#include <imsl.h>
#include <stdio.h>

void fcn(int, float*, float*);

int main()
{
   int  n = 2, m = 2;
   float fjac[4], y[2], f[2];
   char *fmt="%14.5e";

   y[0] = 1.0;
   y[1] = 1.0;

/* Calculate and print
 * Jacobian one-sided difference approximation: */

   imsl_f_jacobian (fcn, m, n, y, f, fjac, 0);

   imsl_f_write_matrix ("The Jacobian is:", m, n, fjac,
       IMSL_WRITE_FORMAT, fmt, 0);
}
```

```
void fcn(int indx, float y[], float f[])
{
   f[0] = y[0]*y[1] - 2.0;
   f[1] = y[0] - y[0]*y[1] + 1.0;
}
```

### Output

```
       The Jacobian is:
               1              2
1   1.00000e+000   1.00000e+000
2   0.00000e+000  -1.00000e+000
```

## Example 2

A simple use of `imsl_f_jacobian` is shown. The gradient of the function

$$f(y_1,y_2) = a\exp(by_1) + cy_1y_2^2$$

is required for values

$$a = 2.5e6,\ b = 3.4,\ c = 4.5,\ y_1 = 2.1,\ y_2 = 3.2$$

The analytic gradient of this function is:

$$grad(f) = [ab\exp(by_1) + cy_2^2,\ 2cy_1y_2]$$

Note that the comparison of the Jacobian estimates using one-sided and central differences with the exact analytic Jacobian results given in this example demonstrates the increased accuracy afforded by use of central differences. However, these estimates require up to twice as many function calculations as do the one-sided differences estimates for Jacobians with a large number of variables.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

void fcn(int, float*, float*);

void fcn_drv(float*, float*);

int main()
{
   int  n = 2, m = 1, j;
   int  method[2];
   float fjac[2], y[2], f[1], scale[2];
   char *fmt="%14.5e";

   y[0] = 2.1;
   y[1] = 3.2;
   scale[0] = 1.0;
```

```
   scale[1] = 8000.0;

/* Use one-sided and central differences
 * to approximate gradient and print results: */

   for (j = 0; j <= 1; j++) {
       if (j == 0) {
           method[0] = IMSL_DD_ONE_SIDED;
           method[1] = IMSL_DD_ONE_SIDED;
       } else {
           method[0] = IMSL_DD_CENTRAL;
           method[1] = IMSL_DD_CENTRAL;
       }

       imsl_f_jacobian (fcn, m, n, y, f, fjac,
           IMSL_YSCALE, scale,
           IMSL_METHOD, method,
           0);

       if (j == 0) {
           imsl_f_write_matrix ("One-Sided Jacobian:",
               m, n, fjac, IMSL_WRITE_FORMAT, fmt, 0);
       } else {
           imsl_f_write_matrix ("Central Jacobian:",
               m, n, fjac, IMSL_WRITE_FORMAT, fmt, 0);
       }
   }

/* Calculate analytic Jacobian: */

   fcn_drv (y, fjac);
   imsl_f_write_matrix ("Analytic Jacobian:",
       m, n, fjac,IMSL_WRITE_FORMAT, fmt, 0);
}

void fcn(int indx, float y[], float f[])
{
   float a, b, c;

   a = 2500000.;
   b = 3.4;
   c = 4.5;

   f[0] = a * exp (b * y[0]) + c * y[0] * y[1] * y[1];
}

void fcn_drv(float y[], float fjac[])
{
   float a, b, c;

   a = 2500000.;
```

```
   b = 3.4;
   c = 4.5;

   fjac[0] = a * b * exp (b * y[0]) + c * y[1] * y[1];
   fjac[1] = 2 * c * y[0] * y[1];
}
```

### Output

```
    One-Sided Jacobian:
           1              2
 1.07285e+010   9.26819e+001

     Central Jacobian:
           1              2
 1.07225e+010   6.17690e+001

    Analytic Jacobian:
           1              2
 1.07221e+010   6.04800e+001
```

## Example 3

This example uses the same data as in Example 2. Here we assume that the second component of the gradient is analytically known. Therefore only the first gradient component needs numerical approximation. The input value `IMSL_DD_SKIP` of array element `method[1]` specifies that numerical differentiation with respect to $y_2$ is skipped.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

void fcn(int, float*, float*);

int main()
{
   int  n = 2, m = 1;
   int  method[2];
   float fjac[2], y[2], f[1], scale[2];
   char *fmt="%14.5e";

   y[0] = 2.1;
   y[1] = 3.2;
   scale[0] = 1.0;
   scale[1] = 8000.0;

/* One-sided differences for fjac[0]: */

   method[0] = IMSL_DD_ONE_SIDED;
```

```
/* Set method[1] to skip differencing for fjac[1]
 * and initialize it to analytic derivative: */

   method[1] = IMSL_DD_SKIP;
   fjac[1] = 2.0 * 4.5 * y[0] * y[1];

/* Get Gradient approximation: */

   imsl_f_jacobian (fcn, m, n, y, f, fjac,
       IMSL_YSCALE, scale,
       IMSL_METHOD, method,
       0);

/* Print results: */

   imsl_f_write_matrix ("The Jacobian is:", m, n, fjac,
       IMSL_WRITE_FORMAT, fmt, 0);
}

void fcn(int indx, float y[], float f[])
{
   float a, b, c;

   a = 2500000.;
   b = 3.4;
   c = 4.5;

   f[0] = a * exp (b * y[0]) + c * y[0] * y[1] * y[1];
}
```

### Output

```
      The Jacobian is:
           1              2
 1.07285e+010   6.04800e+001
```

## Example 4

This example uses the same data as in Example 2, computing the Jacobian (gradient) of the function:

$$f(y_1, y_2) = a\exp(by_1) + cy_1y_2^2$$

For this example, the analytic derivative of the second term with respect to $y_1$, $cy_2^2$, is provided by the user (in the example, see `case -1:` within the user-provided function `fcn`). This leaves only the first term, $a\exp(by_1)$, to be evaluated in order to use direct differencing to calculate the first partial (see `case 1:` within the user-provided function `fcn`). Also, since the first term does not depend on the second variable, $y_2$, it can be left out of the function evaluation when computing the partial with respect to $y_2$ using differencing methods, potentially avoiding cancellation errors (see `case 2:` within the user-provided function `fcn`). Since the code does not specify the analytic derivative with respect to $y_2$ for either of the two terms of $f(y_1, y_2)$, `fcn` returns `f[0]` set to 0 for `case -2:`. The use of optional argument `IMSL_ACCUMULATE` thereby allows `imsl_f_jacobian` to use these facts to obtain greater accuracy using a minimum number of computations of the exponential function.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

void fcn_w_data(int, float*, float*, void*);

int main()
{
   int  n = 2, m = 1;
   float fjac[2], y[2], f[1], scale[2];
   char *fmt="%14.5e";

   float rdata[3];

   y[0] = 2.1;
   y[1] = 3.2;
   scale[0] = 1.0;
   scale[1] = 8000.0;

/* Set up to pass extra information to the function: */

   rdata[0] = 2500000.0;
   rdata[1] = 3.4;
   rdata[2] = 4.5;

/* Use optional argument IMSL_ACCUMULATE so that the
 * user can specify which function components are to be
 * used for divided difference approximation of
 * derivatives and which are to be replaced with exact
 * analytically calculated derivatives. Both components
 * are set to the default one-sided differences method.
 *
 * Calculate and print Jacobian approximation: */

   imsl_f_jacobian (NULL, m, n, y, f, fjac,
       IMSL_YSCALE, scale,
       IMSL_ACCUMULATE,
       IMSL_FCN_W_DATA, fcn_w_data, (void*) rdata,
       0);

   imsl_f_write_matrix ("The Jacobian is:", m, n, fjac,
       IMSL_WRITE_FORMAT, fmt, 0);
}

void fcn_w_data(int indx, float y[], float f[],
        void* data)
```

```
{
   float a, b, c;
   float *rdata = (float*) data;

   a = rdata[0];
   b = rdata[1];
   c = rdata[2];

/* Handle both the differenced part and the part that is
 * known analytically for each dependent variable: */

   switch (indx) {
   case 1:
       f[0] = a * exp(b *y[0]);
       break;
   case -1:
       f[0] = c * y[1] *y[1];
       break;
   case 2:
       f[0] = c * y[0] * y[1] * y[1];
       break;
   case -2:
       f[0] = 0.;
       break;
   }
}
```

### Output

```
       The Jacobian is:

           1             2

1.07285e+010  6.04862e+001
```

## Example 5

This example uses CNL function `imsl_f_bounded_least_squares` to solve the nonlinear least-squares problem:

$$\min \tfrac{1}{2}\sum_{i=0}^{1} f_i(x)^2$$
$$-2 \le x_0 \le 0.5$$
$$-1 \le x_1 \le 2$$

where: $f_0(x) = 10(x_1 - x_0^2)$, $f_1(x) = 1 - x_0$, an initial guess (−1.2,  1.0) is supplied, and the residual at the approximate solution is returned. This example is identical to Example 2 of `imsl_f_bounded_least_squares`, except that Example 2 uses an analytic Jacobian, and this example uses `imsl_f_jacobian` to approximate the Jacobian using the default one-sided differences.

Note that the function vector whose sum of squares is to be minimized, `rosbck`, is supplied directly (as a required argument) to `imsl_f_bounded_least_squares` and indirectly to `imsl_f_jacobian`, wrapped in function `fcn`. Function `fcn` is supplied to `imsl_f_jacobian` via optional argument `IMSL_FCN_W_DATA, fcn, (void*) idata`. `imsl_f_jacobian` is called from within function `jacobian` which is passed to `imsl_f_bounded_least_squares` via optional argument `IMSL_JACOBIAN, jacobian`.

Also note that the array size parameters `m` and `n` are passed to function `rosbck` (which is wrapped in function `fcn` for use by `imsl_f_jacobian`) via integer array `idata`, which is specified in the optional argument `IMSL_FCN_W_DATA, fcn, (void*) idata`. This is an example of how to pass necessary integer data to `imsl_f_jacobian` required argument function `fcn` using `IMSL_FCN_W_DATA`; an example of passing real data using `IMSL_FCN_W_DATA` is given in Example 4 above.

Example 5 demonstrates how `imsl_f_jacobian` can be used to supply estimates of the Jacobian matrix that are necessary for solving many optimization problems when the function to be minimized is complex and its Jacobian cannot be calculated analytically.

```
#include <imsl.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define M      2
#define N      2
#define LDFJAC 2

void   rosbck(int m, int n, float *x, float *f);
void   jacobian(int m, int n, float *x, float *fjac, int fjac_col_dim);
void   fcn(int indx, float *x, float *f, void* data);

void main()
{
   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)
{
   int idata[2];
   float *f = NULL;
   f = (float*)malloc(m*sizeof(float));
   idata[0] = m;
   idata[1] = n;

   imsl_f_jacobian(NULL, m, n, x, f, fjac,
                    IMSL_FCN_W_DATA, fcn, (void*) idata,
                    0);
   if (f) free(f);
}

void fcn (int indx, float *x, float *f, void* data)
{
   int *idata = (int*) data;
   int m = idata[0];
   int n = idata[1];

   rosbck (m, n, x, f);
}
```

### Output

```
x[0] = 0.500000
x[1] = 0.250000

fvec[0] = 0.000000
fvec[1] = 0.500000
```

# Chapter 9 Special Functions

## Functions

### Elliptic Integrals



### Fresnel Integrals



### Airy Functions



### Kelvin Functions



### Statistical Functions

### Basic Financial Functions



### Bond 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

$$\operatorname{erf}(x) = \frac{2}{(\pi)^{1/2}} \int_0^x e^{-t^2} dt$$

All values of `x` are legal.

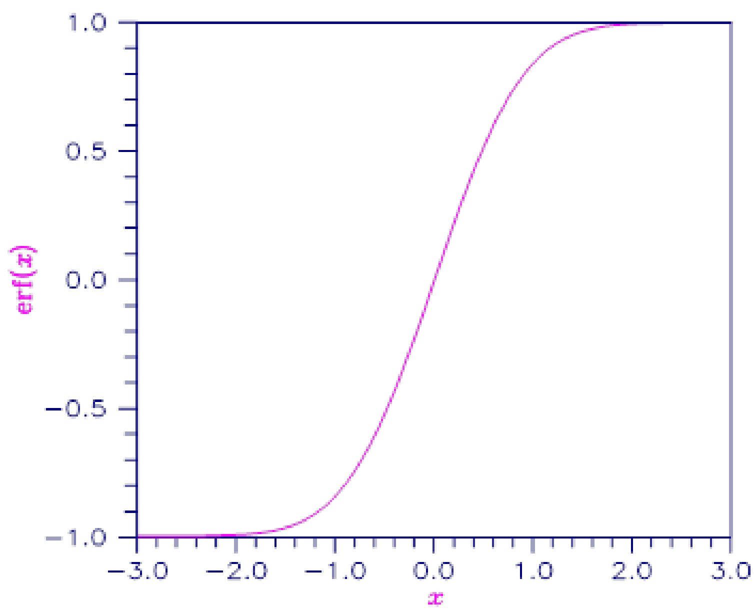



Figure 9.8 — Plot of erf(x)

# Example

Evaluate the error function at *x* = 1/2.

```
#include <imsl.h>
#include <stdio.h>

int 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

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2} dt$$

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.

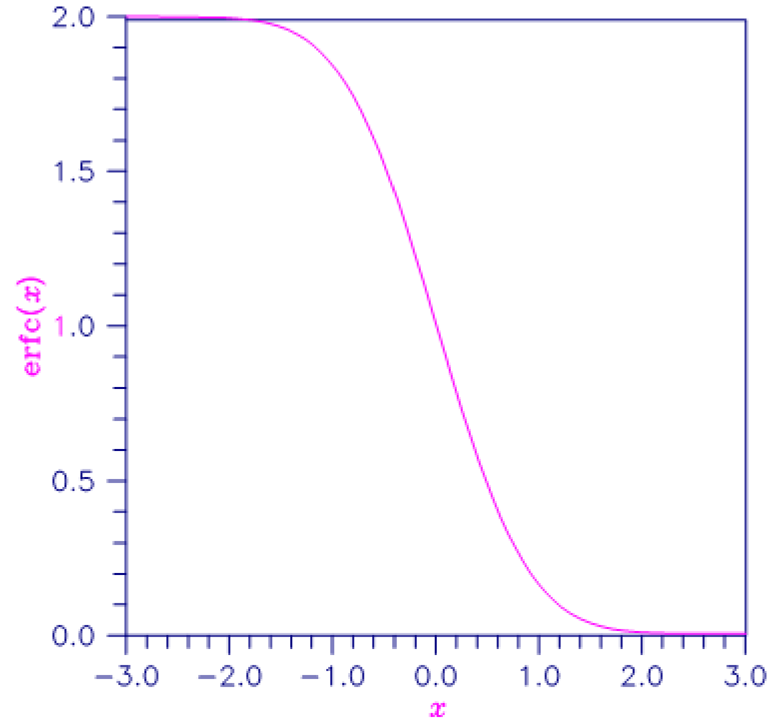



Figure 9.9 — Plot of erfc(x)

## Example

Evaluate the error function at *x* = 1/2.

```
#include <imsl.h>
#include <stdio.h>

int 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

$$e^{x^2}\operatorname{erfc}(x)$$

where erfc(x) is the complementary error function. See imsl_f_erfc 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. For more information, see the description for imsl_f_machine.

## Example

In this example, `imsl_f_erfce`(1.0) is computed and printed.

```
#include <imsl.h>
#include <stdio.h>

int 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).

For more information, see the description for imsl_f_machine.

## Example

In this example, imsl_c_erfe(2.5 +2.5*i*) is computed and printed.

```
#include <imsl.h>
#include <stdio.h>

int 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);
}
```

## Output

```
erfe(2.500 +2.500i) = 0.117 +0.108i
```

# erf_inverse

Evaluates the real inverse error function $\mathrm{erf}^{-1}(x)$.

## 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` = 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 $-1 < x < 1$.

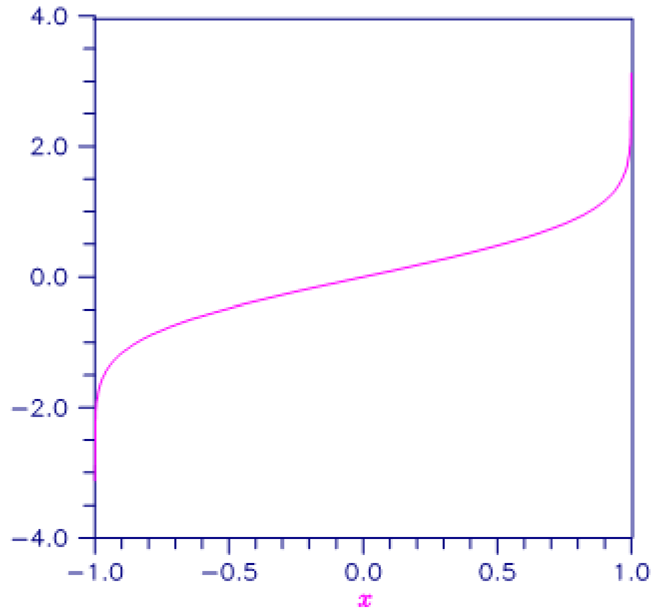



Figure 9.10 — Plot of $erf^{-1}(x)$

## Example

Evaluate the inverse error function at *x* = 1/2.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   float      x = 0.5;
   float      ans;

   ans = imsl_f_erfc_inverse(x);
   printf("inverse erfc(%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 $\text{erfc}^{-1}(x)$.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_erfc_inverse` (*float* `x`)

The type ***double*** procedure is `imsl_d_erfc_inverse`.

## Required Arguments

*float* `x` (Input)
Point at which the inverse complementary error function is to be evaluated. The argument ***x*** must be in the range 0 < `x` < 2.

## Return Value

The value of the inverse complementary error function.

## Description

The inverse complementary error function $y = \text{erfc}^{-1}(x)$ is such that $x = \text{erfc}(y)$ where

$$\mathrm{erfc}(y) = \frac{2}{\sqrt{\pi}} \int_y^{\infty} e^{-t^2} dt$$

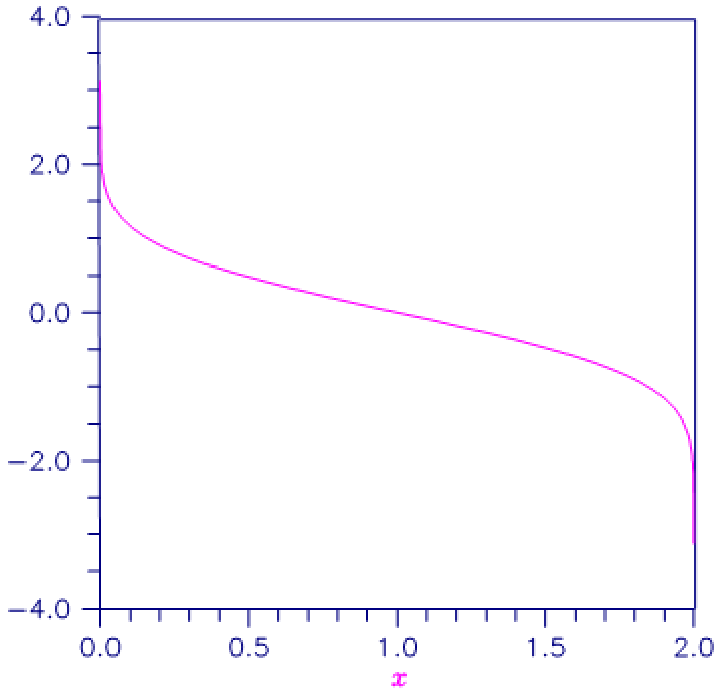



Figure 9.11 — Plot of $\mathrm{erfc}^{-1}(x)$

# Example

Evaluate the inverse complementary error function at $x = 1/2$.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   float      x = 0.5;
   float      ans;

   ans = imsl_f_erf_inverse(x);
   printf("inverse erf(%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 $\varepsilon$ is the machine precision. |

## Warning Errors

| | |
|---|---|
| IMSL_LARGE_ARG_WARN | $\lvert x \rvert$ should be less than $1/\sqrt{\varepsilon}$ where $\varepsilon$ 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.

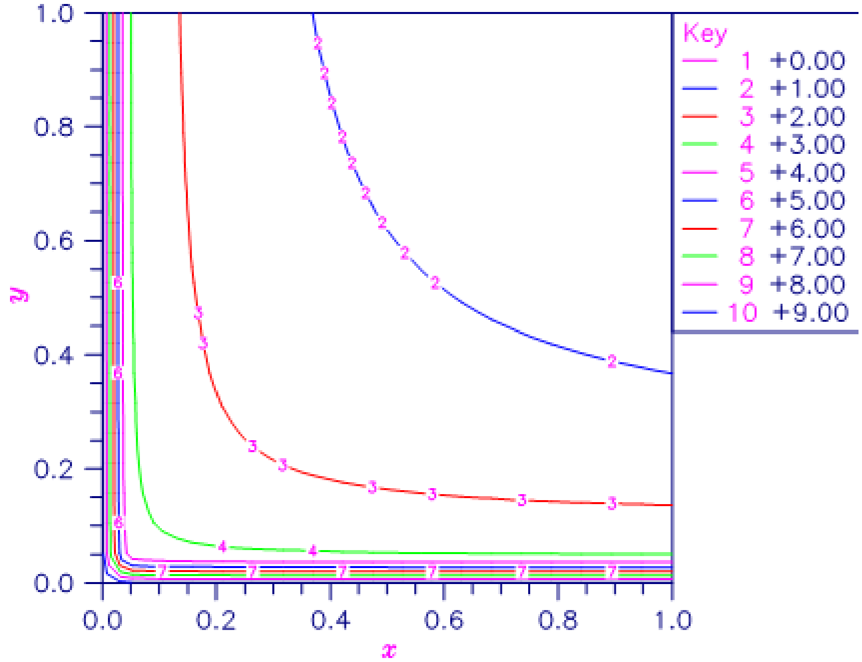



Figure 9.12 — Plot of β(x,y)

## Example

Evaluate the beta function **β** (0.5, 0.2).

```
#include <imsl.h>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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 regularized incomplete beta function.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_beta_incomplete (*float* x, *float* a, *float* b)

The type *double* function is imsl_d_beta_incomplete.

## Required Arguments

*float* x (Input)
Argument at which the regularized incomplete beta function is to be evaluated.

*float* a (Input)
First shape parameter.

*float* b (Input)
Second shape parameter.

## Return Value

The value of the regularized incomplete beta function.

## Description

The regularized incomplete beta function $I_x(a, b)$ is defined

$$I_x(a, b) = B_x(a, b) / B(a, b)$$

where

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt$$

is the incomplete beta function,

$$B(a,b) = B_1(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

is the (complete) beta function, and $\Gamma(a)$ is the gamma function.

The regularized incomplete beta function `imsl_f_beta_incomplete` (`x`, `a`, `b`) is identical to the beta probability distribution function `imsl_f_beta_cdf` (`x`, `a`, `b`) which represents the probability that a beta random variable *X* with shape parameters `a` and `b` takes on a value less than or equal to `x`. The regularized incomplete beta function requires that 0 ≤ `x` ≤ 1, `a` > 0, and `b` > 0 and it underflows for sufficiently small *x* and large `a`. This underflow is not reported as an error. Instead, the value zero is returned.

# Example

Suppose *X* is a beta random variable with shape 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>
#include <stdio.h>

int main()
{
   float          p, a, b, x;

   a = 12.0;
   b = 12.0;
   x = 0.6;
   p = imsl_f_beta_incomplete(x, a, b);
   printf("The probability that X is less than %3.1f is %6.4f\n", x, p);

   x = 0.5;
   p -= imsl_f_beta_incomplete(x, a, b);
   printf("The probability that X is between %3.1f and", x);
   printf(" 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
```

# 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 $x \ll 0$ and overflows for large $x$. It also overflows for values near negative integers.

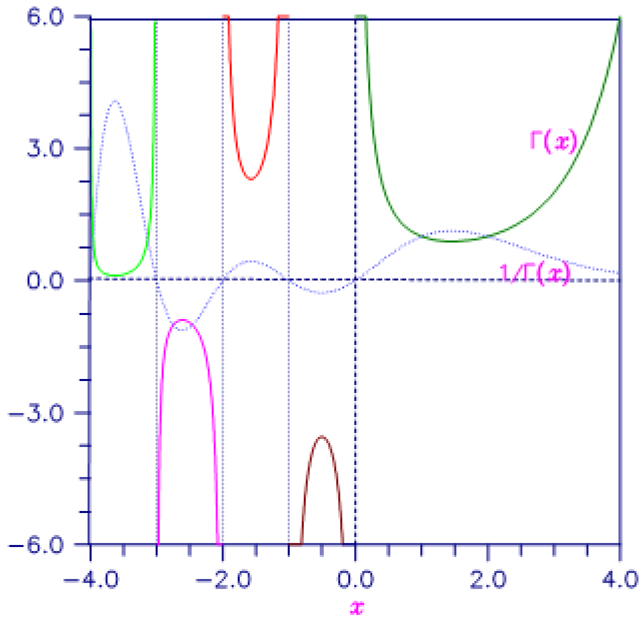



Figure 9.13 — Plot of Plot of Γ(x) and 1/Γ(x)

# Example

In this example, Γ(1.5) is computed and printed.

```
#include <stdio.h>
#include <imsl.h>

int 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.

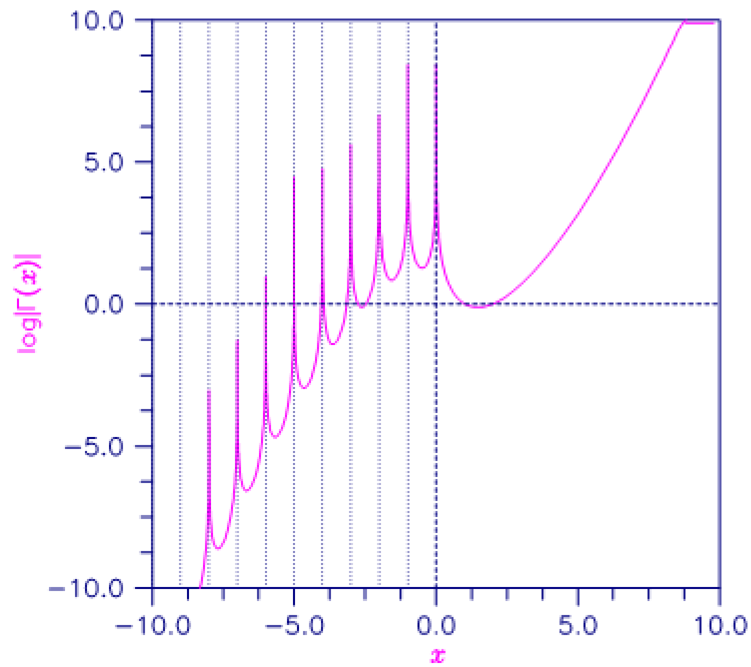



Figure 9.14 — Plot of log |Γ(x)|

## Example

In this example, log |Γ(3.5)| is computed and printed.

```
#include <stdio.h>
#include <imsl.h>

int main()
{
   float      x = 3.5;
   float      ans;

   ans = imsl_f_log_gamma(x);
   printf("log gamma(%f) = %f\n", x, ans);
}
```

### Output

```
log gamma(3.500000) = 1.200974
```

## Warning Errors

| | |
|---|---|
| `IMSL_NEAR_NEG_INT_WARN` | The result is accurate to less than one-half precision because `x` is too close to a negative integer. |

## Fatal Errors

| | |
|---|---|
| `IMSL_NEGATIVE_INTEGER` | The argument for the function cannot be a negative integer. |
| `IMSL_NEAR_NEG_INT_FATAL` | The argument for the function is too close to a negative integer. |
| `IMSL_LARGE_ABS_ARG_OVERFLOW` | $\|x\|$ must not be so large that the result overflows. |

# gamma_incomplete

Evaluates the incomplete gamma function $\gamma(a, x)$.

## Synopsis

*#include* `<imsl.h>`

*float* `imsl_f_gamma_incomplete` (*float* `a`, *float* `x`)

The type ***double*** procedure is `imsl_d_gamma_incomplete`.

## Required Arguments

*float* `a` (Input)
Parameter of the incomplete gamma function is to be evaluated. It must be positive.

*float* `x` (Input)
Point at which the incomplete gamma function is to be evaluated. It must be nonnegative.

## Return Value

The value of the incomplete gamma function $\gamma(a, x)$.

## Description

The incomplete gamma function, $\gamma(a, x)$, is defined to be

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt \text{ for } x > 0$$

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 $a$.

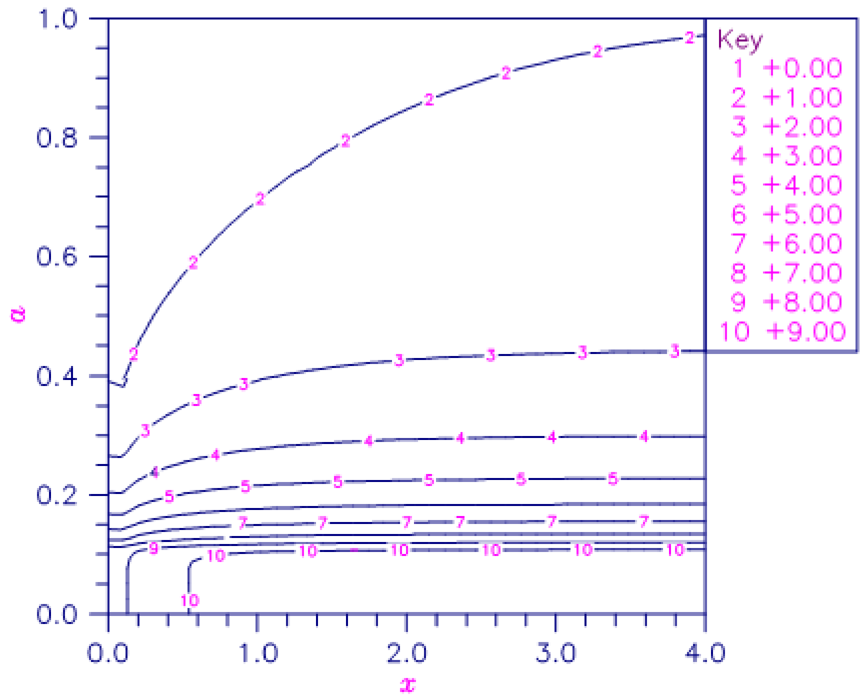



Figure 9.15 — Plot of γ(a, x)

# Example

Evaluate the incomplete gamma function at *a* = 1 and *x* = 3.

```
#include <stdio.h>
#include <imsl.h>

int 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. |

# psi

Evaluates the derivative of the log gamma function.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_psi (*float* x)

The type ***double*** function is imsl_d_psi.

## Required Arguments

*float* x (Input)
Argument at which the function is to be evaluated.

## Return Values

The value of the derivative of the log gamma function at x. NaN is returned if an error occurs.

## Description

The psi function, also called the digamma function, is defined to be

$$\psi(x) = \frac{d}{dx}\ln\Gamma(x)$$

See imsl_f_gamma for the definition of $\Gamma(x)$.

The argument x must not be exactly zero or a negative integer, or $\psi(x)$ is undefined. Also, x must not be too close to a negative integer such that the accuracy of the result is less than half precision. If no value can be computed, then NaN is returned.

## Example

In this example, $\psi(1.915)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int main(){
   float x=1.915, ans;
```

```
    ans=imsl_f_psi(x);
    printf("psi(%f) = %f\n", x, ans);
}
```

### Output

```
psi(1.915000) = 0.366452
```

## 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. |

# psi1

Evaluates the second derivative of the log gamma function.

## Synopsis

*#include* `<imsl.h>`

*float* imsl_f_psi1 (*float* x)

The type ***double*** function is `imsl_d_psi1.`

## Required Arguments

*float* `x` (Input)
Argument at which the function is to be evaluated.

## Return Value

The value of the second derivative of the log gamma function at `x`. NaN is returned if an error occurs.

## Description

The `psi1` function, also called the trigamma function, is defined to be

$$\psi_1(x) = \frac{d^2}{dx^2}\ln\Gamma(x)$$

See `imsl_f_gamma` for the definition of Γ(x).

The argument `x` must not be exactly zero or a negative integer, or $\psi_1(x)$ is undefined. Also, `x` must not be too close to a negative integer such that the accuracy of the result is less than half precision.

## Example

In this example, $\psi_1(1.915)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int main(){
   float x=1.915, ans;

   ans=imsl_f_psi1(x);
```

```
    printf("psi1(%f) = %f\n", x, ans);
}
```

### Output

```
psi1(1.915000) = 0.681164
```

## 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. |

# 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.

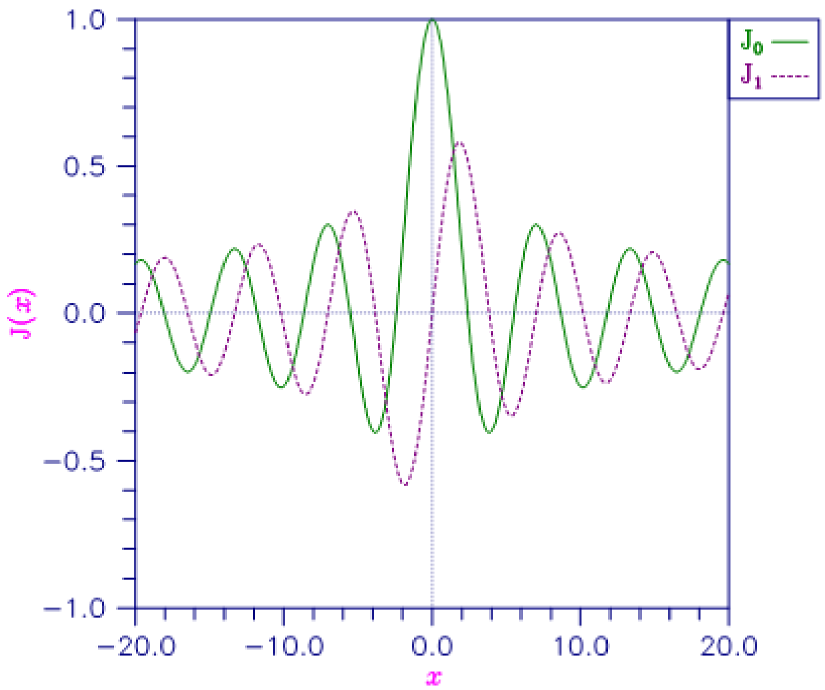



Figure 9.16 — Plot of J0 (x) and J1 (x)

# Example

The Bessel function $J_0(1.5)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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` | $\lvert x \rvert$ should be less than $1/\sqrt{\varepsilon}$ where $\varepsilon$ 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 $1/\varepsilon$ where $\varepsilon$ 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>
#include <stdio.h>

int 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 $\lvert x \rvert > 2s$ 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 $\varepsilon$ 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 $1/\varepsilon$ where $\varepsilon$ 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

*#include* <imsl.h>

*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

$$J_\nu(z) = \frac{1}{\pi}\int_0^\pi \cos\left(z\ \sin\theta - \nu\theta\right)d\theta - \frac{\sin(\nu\pi)}{\pi}\int_0^\infty e^{z\ \sinh\ t - \nu t} dt$$
$$\text{for}\quad |\arg z| < \frac{\pi}{2}$$

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 $\rho = e^{ip/2}$:

$$Y_\nu(z) = \begin{cases} \rho I_\nu(z/\rho) & \text{for} \quad -\pi/2 < \arg z \le \pi \\ \rho^3 I_\nu(\rho^3 z) & \text{for} \quad -\pi < \arg z \le \pi/2 \end{cases}$$

## Example

In this example, $J_{0.3+n-1}$ (1.2 + 0.5*i*), ν = 1, ..., 4 is computed and printed.

```
#include <imsl.h>
#include <stdio.h>

int 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)
```

## Fatal Errors

| | |
|---|---|
| `IMSL_BESSEL_CONT_FRAC` | Continued fractions have failed to converge. The double precision version of this function provides the most accurate solution. |

# 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.

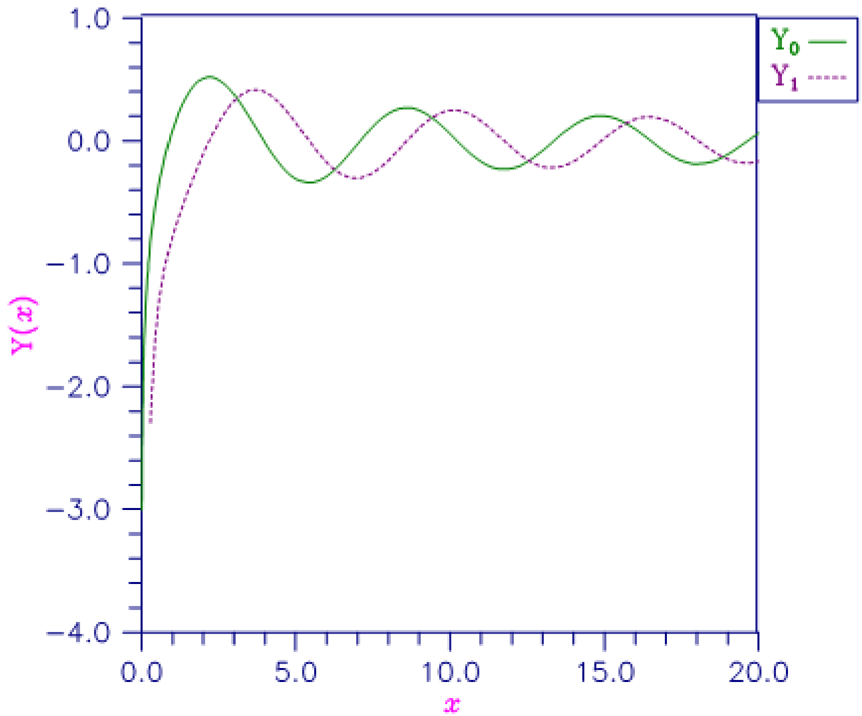



Figure 9.17 — Plot of Y0(x) and Y1(x)

## Example

The Bessel function $Y_0(1.5)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\varepsilon$ is the machine precision to prevent the answer from being less accurate than half precision. |

## Fatal Errors

| | |
|---|---|
| `IMSL_LARGE_ABS_ARG_FATAL` | $\|x\|$ should be less than $1/\varepsilon$ where $\varepsilon$ 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>
#include <stdio.h>

int 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 | $\lvert x \rvert$ should be less than $1/\sqrt{\varepsilon}$ where $\varepsilon$ 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 | $\lvert x \rvert$ should be less than $1/\varepsilon$ where $\varepsilon$ 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

*#include* <imsl.h>

*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:

$$Y_\nu\left(ze^{\pi i/2}\right) = e^{(\nu+1)\pi i/2} I_\nu(z) - \frac{2}{\pi} e^{-\nu\pi i/2} K_\nu(z) \quad \text{for } -\pi < \arg z \le \frac{\pi}{2}$$

## Example

In this example, $Y_{0.3+n-1}(1.2 + 0.5i)$, $\nu = 1, \ldots, 4$ is computed and printed.

```
#include <imsl.h>
#include <stdio.h>

int 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.

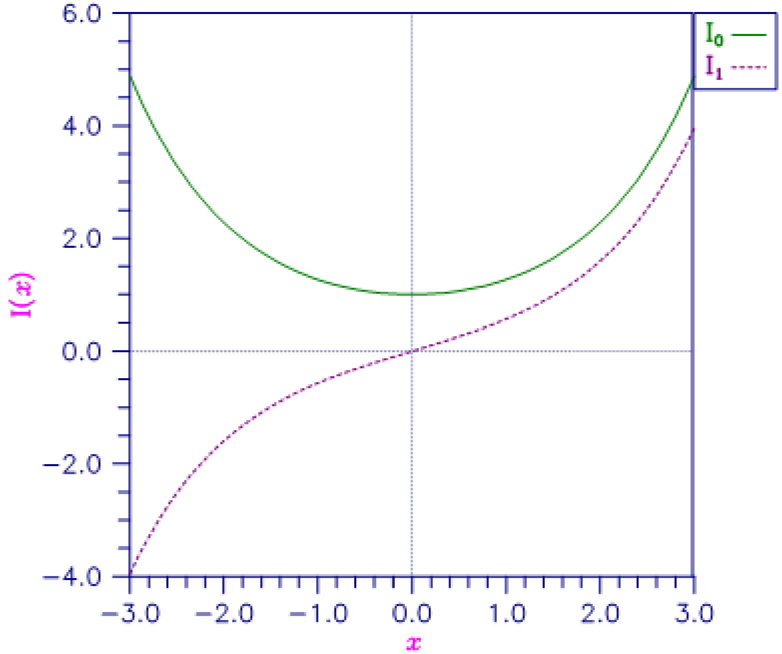



Figure 9.18 — Plot of I0(x) and I1(x)

# Example

The Bessel function $I_0(1.5)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $e^{\|x\|}$ 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 $I_0(x)$ 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 <stdio.h>
#include <math.h>

int 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>
#include <stdio.h>

int main()
{
   float       x = 1.5;
   float       ans;

   ans = imsl_f_bessel_I1(x);
   printf("I1(%f) = %f\n", x, ans);
```

```
}
```

### Output

```
I1(1.500000) = 0.981666
```

# Alert Errors

| | |
|---|---|
| IMSL_SMALL_ABS_ARG_UNDERFLOW | The argument should not be so close to zero that $I_1(x) \approx x/2$ underflows. |

# Fatal Errors

| | |
|---|---|
| IMSL_LARGE_ABS_ARG_FATAL | The absolute value of $x$ must not be so large that $e^{\lvert x \rvert}$ overflows. |

# bessel_exp_I1

Evaluates the exponentially scaled modified Bessel function of the first kind of order one.

## Synopsis

*#include* <imsl.h>

*float* `imsl_f_bessel_exp_I1` (*float* `x`)

The type *double* function is `imsl_d_bessel_exp_I1`.

## Required Arguments

*float* `x` (Input)
Point at which the Bessel function is to be evaluated.

## Return Value

The value of the scaled Bessel function $e^{-|x|} I_1(x)$. If no solution can be computed, NaN is returned.

## Description

The function `imsl_f_bessel_I1` underflows if $|x| / 2$ underflows. The Bessel function $I_1(x)$ is defined to be

$$I_1(x) = \frac{1}{\pi}\int_0^{\pi} e^{x \cos \theta} \cos\theta d\theta$$

## Example

The expression $e^{-4.5} I_0(4.5)$ is computed directly by calling `imsl_f_bessel_exp_I1` and in-directly 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 `insl_f_bessel_I1`.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

int main()
{
   float  x = 4.5;
   float  ans;
   float  error;
```

```
    ans = imsl_f_bessel_exp_I1 (x);
    printf("(e**(-4.5))I1(4.5) = %f\n\n", ans);

    error = fabs(ans - (exp(-x)*imsl_f_bessel_I1(x)));
    printf ("Error = %e\n", error);
}
```

### Output

```
(e**(-4.5))I1(4.5) = 0.170959

Error = 1.469216e-09
```

# bessel_Ix

Evaluates a sequence of modified Bessel functions of the first kind with real order and complex arguments.

## Synopsis

*#include* `<imsl.h>`

*f_complex* `*imsl_c_bessel_Ix` (*float* `xnu`, *f_complex* `z`, *int* `n`, …, 0)

The type ***d_complex*** function is `imsl_z_bessel_Ix`.

## Required Arguments

*float* `xnu` (Input)
The lowest order desired. Argument `xnu` must be greater than −1/2.

*f_complex* `z` (Input)
Argument for which the sequence of Bessel functions is to be evaluated.

*int* `n` (Input)
Number of elements in the sequence.

## Return Value

A pointer to the `n` values of the function through the series. Element *i* contains the value of the Bessel function of order `xnu` + *i* for *i* = 0, …, *n* − 1.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*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

$$I_\nu(z) = e^{-\nu\pi i/2} J_\nu\left(ze^{\pi i/2}\right) \text{ for } -\pi < \arg z \le \frac{\pi}{2}$$

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, $J_{0.3+n-1}$ (1.2 + 0.5*i*), ν = 1, …, 4 is computed and printed.

```
#include <imsl.h>
#include<stdio.h>

int 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.

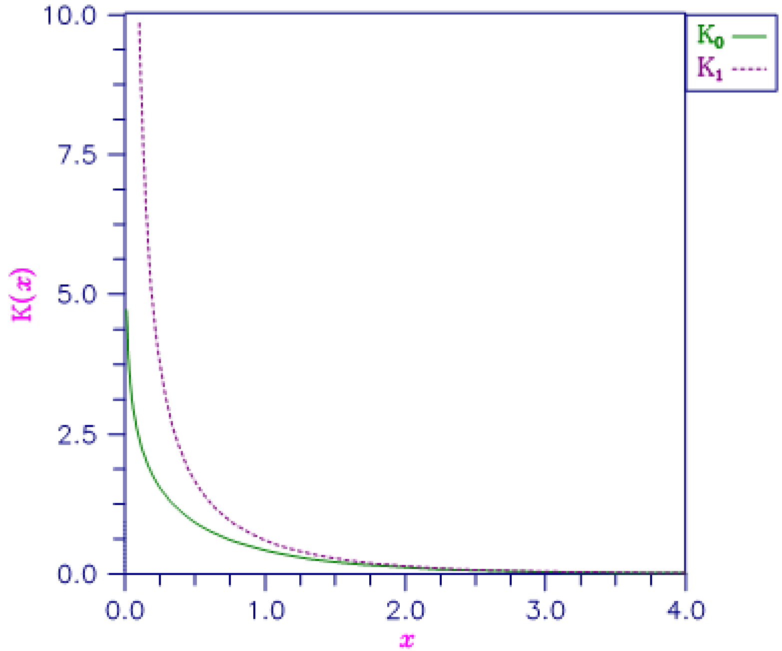



Figure 9.19 — Plot of $K_0(x)$ and $K_1(x)$

# Example

The Bessel function $K_0(1.5)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 <stdio.h>
#include <math.h>

int 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>
#include <stdio.h>

int 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 <stdio.h>
#include <math.h>

int 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, …, *n* − 1.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*f_complex* `*imsl_c_bessel_Kx` (*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 $K_\nu(z)$ is defined to be

$$K_\nu(z) = \frac{\pi}{2} e^{\nu\pi i/2}\left[ iJ_\nu\left(ze^{\pi i/2}\right) - Y_\nu\left(ze^{\pi i/2}\right)\right] \text{ for } -\pi < \arg z \le \frac{\pi}{2}$$

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, $K_{0.3+n-1}$ (1.2 + 0.5$i$), ν = 1, …, 4 is computed and printed.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int          n = 4;
   int          i;
   float        xnu = 0.3;
   static f_complex    z = {1.2, 0.5};
   f_complex *sequence;

   sequence = imsl_c_bessel_Kx(xnu, z, n, 0);

   for (i = 0; i < n; i++)
       printf("K sub %4.2f ((%4.2f,%4.2f)) = (%5.3f,%5.3f)\n",
           xnu+i, z.re, z.im, sequence[i].re, sequence[i].im);
}
```

### Output

```
K sub 0.30 ((1.20,0.50)) = (0.246,-0.200)
K sub 1.30 ((1.20,0.50)) = (0.336,-0.362)
K sub 2.30 ((1.20,0.50)) = (0.587,-1.126)
K sub 3.30 ((1.20,0.50)) = (0.719,-4.839)
```

# elliptic_integral_K

Evaluates the complete elliptic integral of the kind *K*(*x*).

## Synopsis

*#include* <imsl.h>

*float* imsl_f_elliptic_integral_K (*float* x)

The type *double* function is imsl_d_elliptic_integral_K.

## Required Arguments

*float* x (Input)
Argument for which the function value is desired.

## Return Value

The complete elliptic integral *K*(*x*).

## Description

The complete elliptic integral of the first kind is defined to be

$$K(x) = \int_0^{\pi/2} \frac{d\theta}{\left[1 - x\sin^2\theta\right]^{1/2}} \text{ for } 0 \le x < 1$$

The argument *x* must satisfy $0 \le x < 1$; otherwise, imsl_f_elliptic_integral_K returns imsl_f_machine(2), the largest representable floating-point number. For more information, see the description for machine (float).

The function *K*(*x*) is computed using the routine imsl_f_elliptic_integral_RF and the relation $K(x) = R_F(0, 1 - x, 1)$.

## Example

The integral *K*(0) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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. For more information, see the description for `imsl_f_machine`.

The function $E(x)$ is computed using the routine `imsl_f_elliptic_integral_RF` and `imsl_f_elliptic_integral_RD`. The computation is done using the relation

$$E(x) = R_F(0, 1 - x, 1) - \tfrac{x}{3} R_D(0, 1 - x, 1)$$

## Example

The integral $E(0.33)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{[(t+x)(t+y)(t+z)]^{1/2}}$$

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. For more information, see the description for imsl_f_machine.

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>
#include <stdio.h>

int 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 defined 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 $\varepsilon$ = `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 $b$. For more information, see the description for `imsl_f_machine`.

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>
#include <stdio.h>

int 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 $\rho$ 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. For more information, see the description for `imsl_f_machine`.

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>
#include <stdio.h>

int 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. For more information, see the description for imsl_f_machine.

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>
#include <stdio.h>

int 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

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt$$

## Example

The Fresnel integral *C*(1.75) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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

$$Ai(x) = \frac{1}{\pi}\int_0^{\infty}\cos\left(xt + \frac{1}{3}t^3\right)dt = \sqrt{\frac{x}{3\pi^2}}K_{1/3}\left(\frac{2}{3}x^{3/2}\right)$$

The Bessel function $K_v(x)$ is defined in bessel_exp_K0.

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 $\varepsilon$ = 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.

For more information, see the description for imsl_f_machine.

## Example

In this example, Ai(−4.9) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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

$$Bi(x) = \frac{1}{\pi}\int_0^\infty \exp\left(xt - \frac{1}{3}t^3\right)dt + \frac{1}{\pi}\int_0^\infty \sin\left(xt + \frac{1}{3}t^3\right)dt$$

It can also be expressed in terms of modified Bessel functions of the first kind, $I_\nu(x)$, and Bessel functions of the first kind $J_\nu(x)$ (see bessel_Ix and bessel_Jx):

$$Bi(x) = \sqrt{\frac{x}{3}}\left[I_{-1/3}\left(\frac{2}{3}x^{3/2}\right) + I_{1/3}\left(\frac{2}{3}x^{3/2}\right)\right] \text{ for } x > 0$$

and

$$Bi(x) = \sqrt{\frac{-x}{3}}\left[J_{-1/3}\left(\frac{2}{3}|x|^{3/2}\right) - J_{1/3}\left(\frac{2}{3}|x|^{3/2}\right)\right] \text{ for } x < 0$$

Let $\varepsilon$ = 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. For more information, see the description for imsl_f_machine.

## Example

In this example, Bi(−4.9) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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($x$). 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 $\varepsilon$ = 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. For more information, see the description for imsl_f_machine.

## Example

In this example, Ai′(−4.9) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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*). 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 $\varepsilon$ = imsl_f_machine(4) is the machine precision. In addition, *x* should not be so large that $exp[(2/3)x^{3/2}]$ overflows. For more information, see the description for imsl_f_machine.

## Example

In this example, Bi′(−4.9) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\Re J_0(xe^{3\pi/4})$. 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, $\mathrm{ber}_0$ (0.4) is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\mathrm{bei}_0(x)$ is defined to be $\Im J_0(xe^{3\pi/4})$. 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, $\mathtt{bei}_0(0.4)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\mathrm{ker}_0(x)$ is defined to be $\Re K_0(xe^{\pi/4})$. 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, $\mathrm{ker}_0(0.4)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\Im K_0(xe^{\pi/4})$. 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>
#include <stdio.h>

int main()
{
   float  x = 0.4;
   float  ans;

   x = imsl_f_kelvin_kei0 (x);
```

```
    printf ("kei0(0.4) = %f\n", x);
}
```

### Output

```
kei0(0.4) = -0.703800
```

# kelvin_ber0_derivative

Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_kelvin_ber0_derivative (*float* x)

The type ***double*** function is imsl_d_kelvin_ber0_derivative.

## Required Arguments

*float* x (Input)
Argument for which the function value is desired.

## Return Value

The derivative of the Kelvin function of the first kind, ber, of order zero evaluated at ***x***.

## Description

The function $\mathrm{ber}_0'(x)$ is defined to be

$$\frac{d}{dx}\mathrm{ber}_0(x)$$

The function imsl_f_kelvin_ber0_derivative is based on the work of Burgoyne (1963).

If $|x| > 119$, NaN is returned.

## Example

In this example, $\mathrm{ber}_0'(0.6)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\mathrm{bei}_0'(x)$ is defined to be

$$\frac{d}{dx}\mathrm{bei}_0(x)$$

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, $\mathrm{bei}_0'(0.6)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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 $\mathrm{kei}_0'(x)$ is defined to be

$$\frac{d}{dx}\mathrm{kei}_0(x)$$

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, $\mathrm{kei}_0'(0.6)$ is evaluated.

```
#include <imsl.h>
#include <stdio.h>

int 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 $\sigma^2$ 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:

$$\phi(x) = \text{erfc}\left(-x/\sqrt{2.0}\right)/2$$

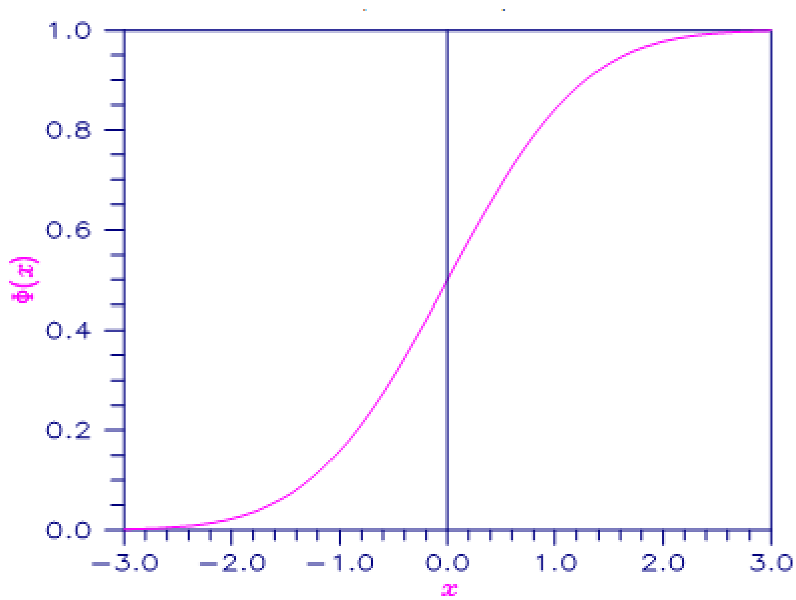



Figure 9.20 — Plot of Φ(x)

## 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>

int 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*) = $\Phi^{-1}(p)$ 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>

int 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 cumulative distribution function (CDF).

## 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. Argument `df` must be greater than 0.

## Return Value

The probability *p* that a chi-squared random variable takes a value less than or equal to `chi_squared`.

## Description

Function `imsl_f_chi_squared_cdf` evaluates the distribution function, $F(x,\nu)$, of a chi-squared random variable $x$ = `chi_squared` with $\nu$ = `df` degrees of freedom, where:

$$F(x,\nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

and $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point $x$ is the probability that the random variable takes a value less than or equal to $x$.

For $\nu > \nu_{max}$ = 1.e7, `imsl_f_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun [A&S] 1964, Equation 26.4.17) for *p* in terms of the normal CDF, which is evaluated using function `imsl_f_normal_cdf`.

For $\nu \le \nu_{max}$, `imsl_f_chi_squared_cdf` uses series expansions to evaluate *p*: for $x < \nu$, `imsl_f_chi_squared_cdf` calculates *p* using A&S series 6.5.29, and for $x \ge \nu$, `imsl_f_chi_squared_cdf` calculates *p* using the continued fraction expansion of the incomplete gamma function given in A&S equation 6.5.31.

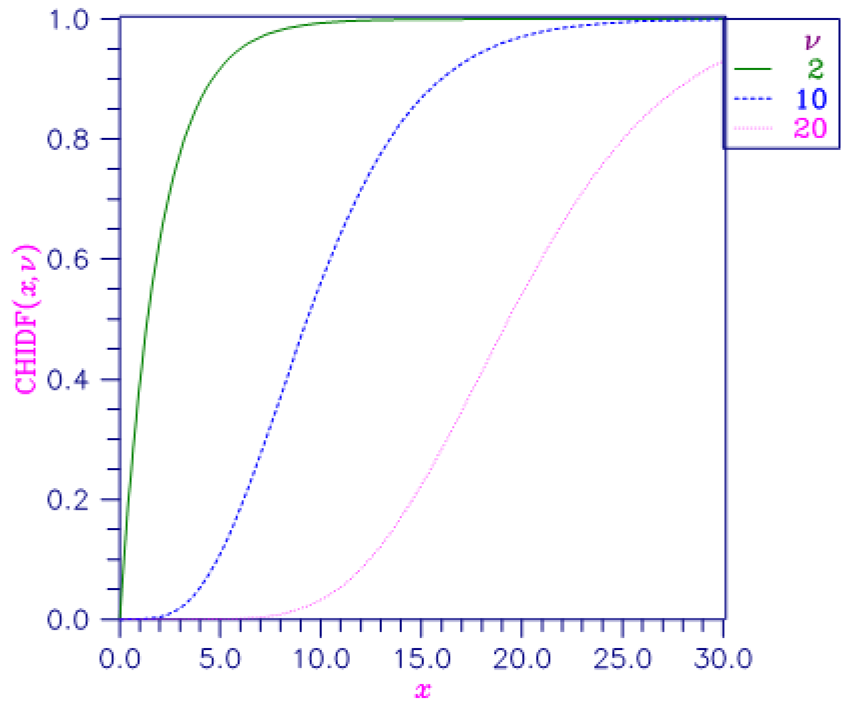



Figure 9.21 — Plot of Fx (x, df)

## Example

Suppose *X* is a chi–squared random variable with two degrees of freedom. In this example, we find the probability that *X* is less than 0.15 and the probability that *X* is greater than 3.0.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
    float  chi_squared = 0.15, df = 2.0, p;

    p    = imsl_f_chi_squared_cdf(chi_squared, df);
    printf("The probability that chi-squared"
        " with %1.0f df is less than %4.2f is %5.4f\n",
        df, chi_squared, p);

    chi_squared = 3.0;
    p    = 1.0 - imsl_f_chi_squared_cdf(chi_squared, df);
    printf("The probability that chi-squared"
        " with %1.0f df is greater than %3.1f is %5.4f\n",
        df, chi_squared, 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 | Since "chi_squared" = #is less than zero, the distribution function is zero at "chi_squared." |

## 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 0.

## 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 $\nu$ = `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 $\nu \leq 2$ 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 ≤ ν < 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 ν ≥ 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>
#include <stdio.h>

int 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_VALUE2` | Unable to bracket the value of the inverse chi-squared at "`p`" = #, with "`df`" = #. |
| `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 $\nu_1$ and $\nu_2$ 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.

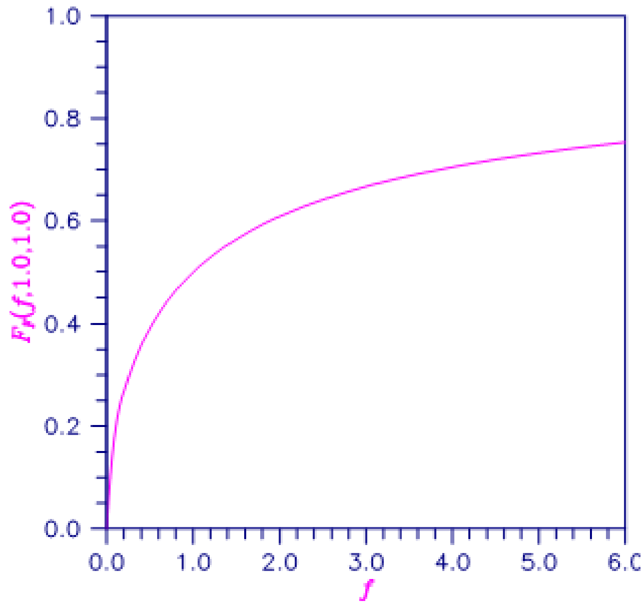



Figure 9.22 — Plot of $F_F$ (f, 1.0, 1.0)

# 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>
#include <stdio.h>

int 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 $\nu_1$ = df_numerator numerator degrees of freedom and $\nu_2$ = 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 $\nu_1$ and $\nu_2$ degrees of freedom and $Y = (\nu_1, X)/(\nu_2 + \nu_1 X)$, then $Y$ is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. If $P \leq 0.5$, imsl_f_F_inverse_cdf uses this relationship directly; otherwise, it also uses a relationship between *F* random variables that can be expressed as follows:

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

# Example

In this example, the 99-th percentage point is calculated for an *F* random variable with seven degrees of freedom. The same calculation is made for a similar variable with one degree of freedom.

```
#include <imsl.h>
#include <stdio.h>

int 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* cumulative distribution function (CDF).

## 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* cumulative 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

Function imsl_f_t_cdf evaluates the cumulative distribution function of a Student's *t* random variable with $\nu$ = df degrees of freedom. If $t^2 \geq \nu$, the following identity relating the Student's *t* cumulative distribution function $TCDF(t, \nu)$ to the incomplete beta ratio function $I_x(a, b)$ is used:

$$TCDF\left(t \leq 0, v\right) = \frac{1}{2} I_x\left(\frac{v}{2}, \frac{1}{2}\right)$$

where

$$x = \frac{v}{t^2 + v}$$

and

$$TCDF(t > 0,\nu) = 1 - TCDF(-t,\nu)$$

If $t^2 < \nu$, the solution space is partitioned into four algorithms as follows: If $\nu \geq 64$ and $t^2 / \nu \leq 0.1$, a Cornish-Fisher expansion is used to evaluate the distribution function. If $\nu < 64$ and an integer and $|t| < 2.0$, a trigonometric series is used (see Abramowitz and Stegun 1964, Equations 26.7.3 and 26.7.4 with some rearrangement). If $\nu < 64$ and an integer and $|t| > 2.0$, a series given by Hill (1970) that converges well for large values of $t$ is used. For the remaining $t^2 < \nu$ cases, *TCDF*($t$, $\nu$) is calculated using the identity:

$$TCDF(t,\nu) = I_x\left(\frac{\nu}{2},\frac{\nu}{2}\right)$$

where

$$x = \frac{t + \sqrt{t^2 + \nu}}{2\sqrt{t^2 + \nu}}$$

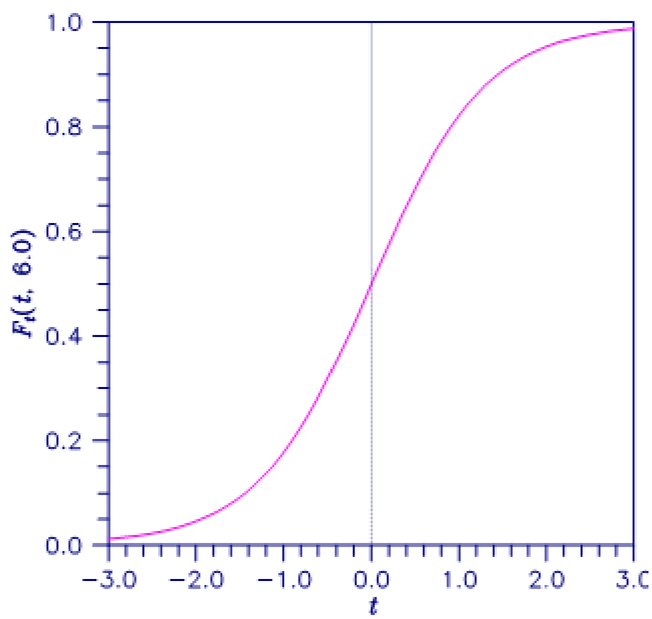



Figure 9.23 — Plot of Ft (t,6.0)

## Example

This example finds the probability that a $t$ random variable with 6 degrees of freedom is greater in absolute value than 2.447. The fact that $t$ is symmetric about 0 is used.

```
#include <imsl.h>
#include <stdio.h>
```

```
int main ()
{
    float  t = 2.447, df = 6.0, p;

    p  = 2.0*imsl_f_t_cdf(-t,df);
    printf("Pr(|t(%1.0f)| > %5.3f) = %6.4f\n", df, t, p);
}
```

## Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

# 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 $\nu$ equals 1 or 2, the inverse can be obtained in closed form. If $\nu$ 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 $\nu$ 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>

int 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 $(c, \infty)$ 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>
#include <stdio.h>

int 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*** × ***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>
#include <stdio.h>

int main()
{
   int        k = 3;
   int        n = 5;
   float      p = 0.95;
   float      pr;

   pr = imsl_f_binomial_cdf(k,n,p);
   printf("Pr(x <= 3) = %6.4f\n", pr);
}
```

### Output

```
Pr(x <= 3) = 0.0226
```

## Informational Errors

| | |
|---|---|
| IMSL_LESS_THAN_ZERO | The input argument, *k*, is less than zero. |
| IMSL_GREATER_THAN_N | The input argument, *k*, is greater than the number of Bernoulli trials, *n*. |

# hypergeometric_cdf

Evaluates the hypergeometric distribution function.

## Synopsis

*#include* <imsl.h>

*float* imsl_f_hypergeometric_cdf (*int* k, *int* n, *int* m, *int* l)

The type *double* procedure is imsl_d_hypergeometric_cdf.

## Required Arguments

*int* k (Input)
Argument for which the hypergeometric distribution function is to be evaluated.

*int* n (Input)
Sample size n must be greater than or equal to k.

*int* m (Input)
Number of defectives in the lot.

*int* l (Input)
Lot size l must be greater than or equal to n and m.

## Return Value

The probability that *k* or fewer defectives occur in a sample of size *n* drawn from a lot of size *l* that contains *m* defectives.

## Description

The function imsl_f_hypergeometric_cdf evaluates the distribution function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *x* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type. The probability function is

$$\Pr(x = j) = \frac{\binom{m}{j}\binom{l-m}{n-j}}{\binom{l}{n}} \quad \text{for } j = i, i+1, \ldots, \min(n,m)$$

where $i = max(0, n - l + m)$.

If *k* is greater than or equal to *i* and less than or equal to min (*n*, *m*), `imsl_f_hypergeometric_cdf` sums the terms in this expression for *j* going from *i* up to *k*. Otherwise, 0 or 1 is returned, as appropriate.

To avoid rounding in the accumulation, `imsl_f_hypergeometric_cdf` performs the summation differently, depending on whether *k* is greater than the mode of the distribution, which is the greatest integer in $(m + 1)(n + 1)/(l + 2)$.

# Example

Suppose *X* is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. This example evaluates the distribution function at 7.

```
#include <imsl.h>
#include <stdio.h>

int 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("Pr (x <= 7) = %6.4f\n", 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 $\theta$ = theta) is

$$f(x) = e^{-q}\,\theta^{x}/x!, \text{ for } x = 0, 1, 2, \ldots$$

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 $f(0) = e^{-q}$.

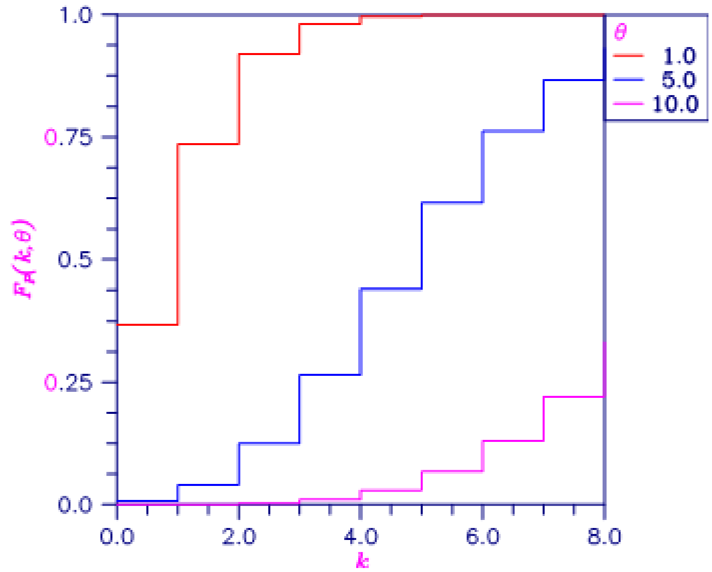



Figure 9.24 — Plot of $F_p(k, \theta)$

## Example

Suppose *X* is a Poisson random variable with θ = 10. This example evaluates the probability that *X* ≤ 7.

```
#include <imsl.h>

int 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>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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 $\rho$ =`rho`, and $|\rho| < 1$,

$$F(x,y) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp\left(-\frac{u^2 - 2\rho uv + v^2}{2\left(1-\rho^2\right)}\right) dudv$$

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

$$\sum = \begin{pmatrix} \sigma_U^2 & \sigma_{UV} \\ \sigma_{UV} & \sigma_V^2 \end{pmatrix}$$

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 $\rho = \sigma U_V/(\sigma_U \sigma_V)$.

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 $|\rho| = 1$, 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 of the IMSL C Numerical Stat Library, "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>
#include <stdio.h>

int 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 ***i***-th 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>

int 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 *i*-th 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>

int 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* | $\left(\text{cost} - \text{total depreciation from periods}\right) \times \text{rate} \times \frac{12 - \text{month}}{12}$ |
| *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>

int 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:

$$[\text{cost} - \text{salvage}(\text{total depreciation from prior periods})]\left(\frac{factor}{life}\right)$$

## 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>

int 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:

$$\left(\mathrm{cost} - \mathrm{salvage}\right) / \mathrm{life}$$

## 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>
```

```
int 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>

int 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` ≠ 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 ***i***-th 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>

int 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>

int 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

$$idollar + \frac{\left[ decimal\_dollar - idollar \right]}{10^{\left(ifrac+1\right)} / \, fraction}$$

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>

int main()
{
    int numerator, fraction = 4;
    float dollarfrc, decimal_dollar = 1.25;

    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:

$$\left(1 + \frac{nominal\_rate}{n\_periods}\right)^{(n\_periods)} - 1$$

## 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>

int 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

$$present\_value(1 + rate)^{\text{n\_periods}} +$$
$$payment[1 + rate(when)]\frac{(1 + rate)^{\text{n\_periods}} - 1}{rate} + \text{future\_value} = 0$$

# 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 <imsl.h>
#include <stdio.h>

int 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 ***i***-th 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>

int 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>

int 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

$$present\_value(1 + rate)^{\mathrm{n\_periods}} +$$
$$payment[1 + rate(when)]\frac{(1 + rate)^{\mathrm{n\_periods}} - 1}{rate} + \mathrm{future\_value} = 0$$

# 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>

int 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_return (*int* count, *float* values[],

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_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>

int 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 $i$-th payment date. $d_1$ represents the 1st payment date. $value_i$ represents the $i$-th 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>

int 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.

The modified internal rate of return also eliminates the multiple rates of return problem.

It is computed using the following:

$$\left\{\left[\frac{-(\text{pnpv})(1+\text{reinvest\_rate})^{\text{n\_periods}}}{(\text{nnpv})(1+\text{finance\_rate})}\right]^{\frac{1}{\text{n\_periods}-1}}\right\}-1$$

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>

int 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 ($500,000 per year) with an annual interest rate of 6%.

```
#include <stdio.h>
#include <imsl.h>

int 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:

$$\left[\left(1 + \text{effective\_rate}\right)^{\frac{1}{n\_periods}} - 1\right] * \text{n\_periods}$$

## 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>

int 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

$$present\_value(1 + rate)^{\text{n\_periods}} +$$
$$payment[1 + rate(when)]\frac{(1 + rate)^{\text{n\_periods}} - 1}{rate} + \text{future\_value} = 0$$

# 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>

int 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

$$present\_value(1 + rate)^{n\_periods} +$$
$$payment[1 + rate(when)]\frac{(1 + rate)^{n\_periods} - 1}{rate} + future\_value = 0$$

## 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>

int 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` ≠ 0

$$present\_value(1 + rate)^{\mathrm{n\_periods}} +$$
$$payment[1 + rate(when)]\frac{(1 + rate)^{\mathrm{n\_periods}} - 1}{rate} + \mathrm{future\_value} = 0$$

# 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>

int 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:

$$\sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{(d_i - d_1)/365}}$$

In the equation above, $d_i$ represents the $i$-th payment date, $d_1$ represents the 1st payment date, and $value_i$ represents the $i$-th 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>

int 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>

int 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>

int 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 of 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>

int 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 Notess ection 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:

$$\text{if } DSM <\ = 182$$

$$\frac{365 * discount\_rate}{360 - discount\_rate * DSM}$$

otherwise,

$$\frac{-\frac{DSM}{365}+\sqrt{\left(\frac{\text{DSM}}{365}\right)^2-\left(2*\frac{DSM}{365}-1\right)*\frac{discount\_rate*DSM}{discount\_rate*DSM-360}}}{\frac{DSM}{365}-0.5}$$

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>

int 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:

$$\frac{\frac{1}{(q * frequency)^2}\left\{\sum_{t=1}^{n} t(t+1)\left(\frac{rate}{frequency}\right)q^{-t} + n(n+1)q^{-n}\right\}}{\left(\sum_{t=1}^{n}\left(\frac{rate}{frequency}\right)q^{-t} + q^{-n}\right)}$$

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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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:

$$\frac{\dfrac{\frac{DSC}{E}*100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}}+\sum_{k=1}^{N}\left(\left(\dfrac{100*coupon\_rate}{freq*\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right)*\left(k-1+\frac{DSC}{E}\right)\right)}{\dfrac{100}{\left(1+\frac{yield}{freq}\right)^{N-1+\frac{DSC}{E}}}+\sum_{k=1}^{N}\left(\dfrac{100*coupon\_rate}{freq*\left(1+\frac{yield}{freq}\right)^{k-1+\frac{DSC}{E}}}\right)}*\frac{1}{freq}$$

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>

int 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>

int 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_modified_duration (*struct tm* settlement, *struct tm* maturity, *float* coupon_rate, *float* yield, *int* frequency, *int* basis)

The type *double* function is imsl_d_modified_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_modified_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_modified_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>

int 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_modified_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>

int 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>

int 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:

$$\left(\frac{redemption}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right)+\left[\sum_{k=1}^{N}\frac{100*\frac{rate}{freq}}{\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right]-\left(100*\frac{rate}{freq}*\frac{A}{E}\right)$$

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>

int 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 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>

int 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>

int 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>

int 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>

int 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>

int 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>

int 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:

$$\left\{\frac{\left(\frac{redemption}{100}+\frac{coupon\_rate}{freq}\right)-\left[\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{freq}\right)\right]}{\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{freq}\right)}\right\}\left(\frac{freq*E}{DSR}\right)$$

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*:*

$$price - \left( \left( \frac{redemption}{\left(1 + \frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}} \right) + \left[ \sum_{k=1}^{N} \frac{100 * \frac{rate}{freq}}{\left(1 + \frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}} \right] - \left( 100 * \frac{rate}{freq} * \frac{A}{E} \right) \right) = 0$$

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>

int 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

## Functions

### Statistics



### Random Numbers



### Low-discrepancy sequence

# 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). See the IMSL C Stat Library for a more extensive collection of statistical functions and detailed descriptions.

## Overview of Random Number Generation

"Random Numbers" describes functions for the generation of random numbers and of random samples and permutations. These functions are useful for applications in Monte Carlo or simulation studies. Before using any of the random number generators, the generator must be initialized by selecting a ***seed*** or starting value. This can be done by calling the function `imsl_random_seed_set`. If the user does not select a seed, one is generated using the system clock. A seed needs to be selected only once in a program, unless two or more separate streams of random numbers are maintained. There are other utility functions in this chapter for selecting the form of the basic generator, for restarting simulations, and for maintaining separate simulation streams.

In the following discussions, the phrases "random numbers," "random deviates," "deviates," and "variates" are used interchangeably. The phrase "pseudorandom" is sometimes used to emphasize that the numbers generated are really not "random," since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they ***simulate*** the realizations of independent and identically distributed random variables.

## The Basic Uniform Generator

The random number generators in this chaptersection use a multiplicative congruential method. The form of the generator is

$$x_i = cx_{i-1} \bmod (2^{31} - 1).$$

Each $x_i$ is then scaled into the unit interval (0,1). If the multiplier, $c$, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator will have a maximal period of $2^{31} - 2$. There are several other considerations, however. See Knuth (1981) for a good general discussion. The possible values for $c$ in the IMSL generators are 16807, 397204094, and 950706376. The selection is made by the function `imsl_random_option`. The choice of 16807 will result in the fastest execution time, but other evidence suggests that the performance of 950706376 is best among these three choices (Fishman and Moore 1982). If no selection is made explicitly, the functions use the multiplier 16807, which has been in use for some time (Lewis et al. 1969).

The generation of uniform (0,1) numbers is done by the function `imsl_f_random_uniform`. This function is ***portable*** in the sense that, given the same seed, it produces the same sequence in all computer/compiler environments.

## Shuffled Generators

The user also can select a shuffled version of these generators using imsl_random_option. The shuffled generators use a scheme due to Learmonth and Lewis (1973). In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each $x_i$ from the simple generator, the low-order bits of $x_i$ are used to select a random integer, $j$, from 1 to 128. The $j$-th entry in the table is then delivered as the random number, and $x_i$, after being scaled into the unit interval, is inserted into the $j$-th position in the table. This scheme is similar to that of Bays and Durham (1976), and their analysis is applicable to this scheme as well.

## Setting the Seed

The seed of the generator can be set in imsl_random_seed_set and can be retrieved by imsl_random_seed_get. Prior to invoking any generator in this section, the user can call imsl_random_seed_set to initialize the seed, which is an integer variable with a value between 1 and 2147483647. If it is not initialized by imsl_random_seed_set, a random seed is obtained from the system clock. Once it is initialized, the seed need not be set again.

If the user wishes to restart a simulation, by imsl_random_seed_get can be used to obtain the final seed value of one run to be used as the starting value in a subsequent run. Also, if two simultaneous random number streams are desired in one run, imsl_random_seed_set and by imsl_random_seed_get can be used before and after the invocations of the generators in each stream.

# simple_statistics

Computes basic univariate statistics.

## Synopsis

*#include* `<imsl.h>`

*float* `*imsl_f_simple_statistics` (*int* `n_observations`, *int* `_variables`, *float* `x[]`, …, 0)

The type *double* procedure is `imsl_d_simple_statistics`.

## Required Arguments

*int* `n_observations` (Input)
The number of observations.

*int* `n_variables` (Input)
The number of variables.

*float* `x[]` (Input)
Array of size `n_observations` × `n_variables` containing the data matrix.

## Return Value

A pointer to a matrix containing some simple statistics for each of the columns in `x`. If `MEDIAN` and `MEDIAN_AND_SCALE` are not used as optional arguments, the size of the matrix is 14 by `n_variables`. The columns of this matrix correspond to the columns of `x` and the rows contain the following statistics:

| Row | Statistic |
|---|---|
| 0 | the mean |
| 1 | the variance |
| 2 | the standard deviation |
| 3 | the coefficient of skewness |
| 4 | the coefficient of excess (kurtosis) |
| 5 | the minimum value |
| 6 | the maximum value |
| 7 | the range |
| 8 | the coefficient of variation (when defined)<br>If the coefficient of variation is not defined, zero is returned. |
| 9 | the number of observations (the counts) |

| Row | Statistic |
|---|---|
| 10 | a lower confidence limit for the mean (assuming normality)<br>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)<br>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

$$\overline{x} = \frac{\sum x_i}{n}$$

## Variance

$$s^2 = \frac{\sum\left(x_i - \overline{x}\right)^2}{n-1}$$

## Skewness

$$\frac{\sum\left(x_i - \overline{x}\right)^3 / n}{\left[\sum\left(x_i - \overline{x}\right)^2 / n\right]^{3/2}}$$

## Excess or Kurtosis

$$\frac{\sum\left(x_i - \overline{x}\right)^4 / n}{\left[\sum\left(x_i - \overline{x}\right)^2 / n\right]^{2}} - 3$$

## Minimum

$$x_{\min} = \min\left(x_i\right)$$

## Maximum

$$x_{\max} = \max\left(x_i\right)$$

## Range

$$x_{\max} - x_{\min}$$

## Coefficient of Variation

$$s / \overline{x} \text{ for } \overline{x} \neq 0$$

### 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

$$\text{MAD} = \text{median}\left\{ |x_i - \text{median}\{x_j\}| \right\}$$

### Simple Robust Estimate of Scale

$$\text{MAD}/\phi^{-1}(3/4)$$

where $\Phi^{-1}(3/4) \approx 0.6745$ is the inverse of the standard normal distribution function evaluated at 3/4. This standardizes MAD in order to make the scale estimate consistent at the normal distribution for estimating the standard deviation (Huber 1981, pp. 107-108).

## Example

This example uses data from Draper and Smith (1981). There are five variables and 13 observations.

```
#include <imsl.h>

#define N_VARIABLES          5
#define N_OBSERVATIONS       13

int 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* n_intervals, …, 0)

The type *double* function is imsl_d_table_oneway.

## Required Arguments

*int* n_observations (Input)
Number of observations.

*float* x[] (Input)
Array of length n_observations containing the observations.

*int* n_intervals (Input)
Number of intervals (bins).

## Return Value

Pointer to an array of length n_intervals containing the counts.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* *imsl_f_table_oneway (*int* n_observations, *float* x[], *int* n_intervals,

IMSL_DATA_BOUNDS, *float* *minimum, *float* *maximum,

IMSL_KNOWN_BOUNDS, *float* lower_bound, *float* upper_bound,

IMSL_CUTPOINTS, *float* cutpoints[],

IMSL_CLASS_MARKS, *float* class_marks[],

IMSL_RETURN_USER, *float* table_oneway[],

0)

## Optional Arguments

IMSL_DATA_BOUNDS, *float* *minimum, *float* *maximum (Output)

*or*

IMSL_KNOWN_BOUNDS, *float* lower_bound, *float* upper_bound (Input)

*or*

IMSL_CUTPOINTS, *float* cutpoints[] (Input)

*or*

IMSL_CLASS_MARKS, *float* class_marks[] (Input)

None, or exactly one, of these four optional arguments can be specified in order to define the intervals or bins for the one-way table. If none is specified, or if IMSL_DATA_BOUNDS is specified, n_intervals, intervals of equal length, are used with the initial interval starting with the minimum value in x and the last interval ending with the maximum value in x. The initial interval is closed on the left and right. The remaining intervals are open on the left and closed on the right. When IMSL_DATA_BOUNDS is explicitly specified, the minimum and maximum values in x are output in minimum and maximum. With this option, each interval is of (maximum−minimum)/n_intervals length. If IMSL_KNOWN_BOUNDS is specified, two semi-infinite intervals are used as the initial and last interval. The initial interval is closed on the right and includes lower_bound as its right endpoint. The last interval is open on the left and includes all values greater than upper_bound. The remaining n_intervals − 2 intervals are each of length

$$\frac{\text{upper\_bound} - \text{lower\_bound}}{\text{n\_intervals} - 2}$$

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>
int 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>
int 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>
int 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>
int 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 `imsl_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 imsl_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 imsl_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 imsl_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 Passing Data to User-Supplied Functions in the introduction to 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

$$\lfloor mF\left(x_{\mathrm{i}}\right)+1\rfloor \text{ where } m=\text{n\_categories}$$

and $\lfloor \cdot \rfloor$ is the function that takes the greatest integer that is no larger than the argument of the function. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred to reduce the total computing time.

If the expected count in any cell is less than 1, then a rule of thumb is that the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.

On some platforms, `imsl_f_chi_squared_test` can evaluate the user-supplied function `user_proc_cdf` in parallel. This is done only if the function `imsl_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables

## 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 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` (see Special Functions) as the hypothesized distribution function. In this example, the null hypothesis is not rejected.

```
#include <imsl.h>
#include <stdio.h>

#define SEED                    123457
#define N_CATEGORIES                10
#define N_OBSERVATIONS            1000

int main()
{
   float      *x, p_value;

   imsl_omp_options(
       IMSL_SET_FUNCTIONS_THREAD_SAFE, 1,
       0);
   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

int 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_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    imsl_random_seed_set(SEED);
                                /* Generate Normal deviates */
    x = imsl_f_random_normal (N_OBSERVATIONS, 0);
                                /* Perform chi squared test */
    chi_squared_statistics[2] =
        imsl_f_chi_squared_test (imsl_f_normal_cdf,
                N_OBSERVATIONS, N_CATEGORIES, x,
                IMSL_CUTPOINTS, &cutpoints,
                IMSL_CELL_COUNTS, &cell_counts,
                IMSL_CELL_CHI_SQUARED, &cell_chi_squared,
                IMSL_CHI_SQUARED, &chi_squared_statistics[0],
                IMSL_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
                0);
                                /* Print results */
    imsl_f_write_matrix ("\nChi Squared Statistics\n", 3, 1,
                         chi_squared_statistics,
                         IMSL_ROW_LABELS, stat_row_labels,
                         0);
    imsl_f_write_matrix ("Cut Points", 1, N_CATEGORIES-1, cutpoints, 0);
```

```
    imsl_f_write_matrix ("Cell Counts", 1, N_CATEGORIES, cell_counts,
                   0);
    imsl_f_write_matrix ("Cell Contributions to Chi-Squared", 1,
                   N_CATEGORIES, cell_chi_squared,
                   0);
}
```

### Output

```
   Chi Squared Statistics

chi-squared               13.18
degrees of freedom         9.00
p-value                    0.15

                             Cut Points
       1          2          3          4          5          6
  -1.282     -0.842     -0.524     -0.253     -0.000      0.253

       7          8          9
   0.524      0.842      1.282


                             Cell Counts
       1          2          3          4          5          6
     106        109         89         92         83         87

       7          8          9         10
     110        104        121         99

                  Cell Contributions to Chi-Squared
       1          2          3          4          5          6
    0.36       0.81       1.21       0.64       2.89       1.69

       7          8          9         10
    1.00       0.16       4.41       0.01
```

## Example 3

In this example, a discrete Poisson random sample of size 1000 with parameter $\theta$ = 5.0 is generated via function imsl_f_random_poisson. In the call to imsl_f_chi_squared_test, function imsl_f_poisson_cdf 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);

int 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_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

   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. |
| `IMSL_STOP_USER_FCN` | Request from user supplied function to stop algorithm.<br>User flag = "#". |

# covariances

more...

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 imsl_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:

$$\begin{aligned} \bar{x}_{k0} &= 0.0 \quad k = 1, \dots, p \\ c_{jk0} &= 0.0 \quad j,k = 1, \dots, p \end{aligned}$$

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*:

$$s_{jk} = \frac{\sum_{i=1}^{n}\left(x_{ji} - \overline{x}_j\right)\left(x_{ki} - \overline{x}_k\right)}{n-1}$$

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

int 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

int 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.

IMSL_RANK, *int* *rank (Output)
The rank of the fitted model is returned in *rank.

`IMSL_COEF_COVARIANCES`, *float* `**p_coef_covariances` (Output)

The address of a pointer to the $m \times m$ array containing the estimated variances and covariances of the estimated regression coefficients. Here, $m$ is the number of regression coefficients in the model. If `IMSL_NO_INTERCEPT` is specified, $m$ = `n_independent`; otherwise, $m$ = `n_independent` + 1. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*p_coef_covariances` is declared; `&p_coef_covariances` is used as an argument to this function; and `imsl_free(p_coef_covariances)` is used to free this array.

`IMSL_COEF_COVARIANCES_USER`, *float* `coef_covariances[]` (Output)

If specified, `coef_covariances` is an array of length $m \times m$ containing the estimated variances and covariances of the estimated coefficients where $m$ is the number of regression coefficients in the model.

`IMSL_COV_COL_DIM`, *int* `cov_col_dim` (Input)

The column dimension of array `coef_covariance`.

Default: `cov_col_dim` = $m$ where $m$ is the number of regression coefficients in the model.

`IMSL_X_MEAN`, *float* `**p_x_mean` (Output)

The address of a pointer to the array containing the estimated means of the independent variables. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*p_x_mean` is declared; `&p_x_mean` is used as an argument to this function; and `imsl_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 `imsl_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 `imsl_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

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \quad i = 1, 2, \ldots, n$$

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; $\beta_0$, $\beta_1$, …, $\beta_k$ are the regression coefficients whose estimated values are to be output by `imsl_f_regression`; and the $\varepsilon_i$'s are independently distributed normal errors each with mean zero and variance $\sigma^2$. Here, *n* is the number of rows in the augmented matrix (x,y), i.e., *n* equals `n_observations`. Note that by default, $\beta_0$ 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

$$SSE = \sum_{i=1}^{n} \left( y_i - \hat{y}_i \right)^2$$

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

$$SST = \sum_{i=1}^{n} \left( y_i - \overline{y} \right)^2$$

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

$$SST = \sum_{i=1}^{n} y_i^2$$

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 $\overline{x}_i$ represent the mean vector for the independent variables given the data for rows 1, 2, …, $i$. The current mean vector is defined to be

$$\bar{x}_i = \frac{\sum_{j=1}^{i} x_j}{i}$$

The *i*-th row of data has $\bar{x}_i$ subtracted from it and is then weighted by *i*/(*i* − 1). Although a crossproduct matrix is not computed, the validity of this centering operation can be seen from the following formula for the sum of squares and crossproducts matrix:

$$\sum_{i=1}^{n} (x_i - \bar{x}_n)(x_i - \bar{x}_n)^T = \sum_{i=2}^{n} \frac{i}{i-1} (x_i - \bar{x}_i)(x_i - \bar{x}_i)^T$$

An orthogonal reduction on the centered matrix is computed. When the final computations are performed, the intercept estimate and the first row and column of the estimated covariance matrix of the estimated coefficients are updated (if `IMSL_COEF_COVARIANCES` or `IMSL_COEF_COVARIANCES_USER` is specified) to reflect the statistics for the original (uncentered) data. This means that the estimate of the intercept is for the uncentered data.

As part of the final computations, `imsl_regression` checks for linearly dependent regressors. In particular, linear dependence of the regressors is declared if any of the following three conditions are satisfied:

- A regressor equals zero.
- Two or more regressors are constant.
- $\sqrt{1 - R^2_{i \cdot 1,\, 2,\, \ldots,\, i-1}}$ is less than or equal to `tolerance`. Here, $R_{i.1,2,\ldots,i-1}$ 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

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i \quad i = 1,2, \ldots ,9$$

is fitted to data taken from Maindonald (1984, pp. 203-204).

```
#include <imsl.h>

#define INTERCEPT      1
#define N_INDEPENDENT  3
#define N_COEFFICIENTS (INTERCEPT + N_INDEPENDENT)
```

```
#define N_OBSERVATIONS 9

int 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

$$y_i = \beta_0 x_{i0} + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i \quad i = 1, 2, \ldots, 4$$

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

$$SSE = \sum_{i=1}^{4} w_i \left( y_i - \hat{y}_i \right)^2$$

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

$$SST = \sum_{i=1}^{4} w_i y_i^2$$

```
#include <imsl.h>
#include <math.h>

#define N_INDEPENDENT  3
#define N_COEFFICIENTS N_INDEPENDENT
#define N_OBSERVATIONS 4

int 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 imsl_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 imsl_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* |
| 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 imsl_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 |

| Element | Analysis of Variance Statistic |
|---|---|
| 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 imsl_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,

$$R^2 = \frac{\sum\left(\hat{y}_{\mathrm{i}} - \overline{y}\right)^2}{\sum\left(y_1 - \overline{y}\right)^2} 100\%$$

where $\hat{y}_i$ is the fitted $y$ value at $x_i$ and $\overline{y}$ 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

int 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
```

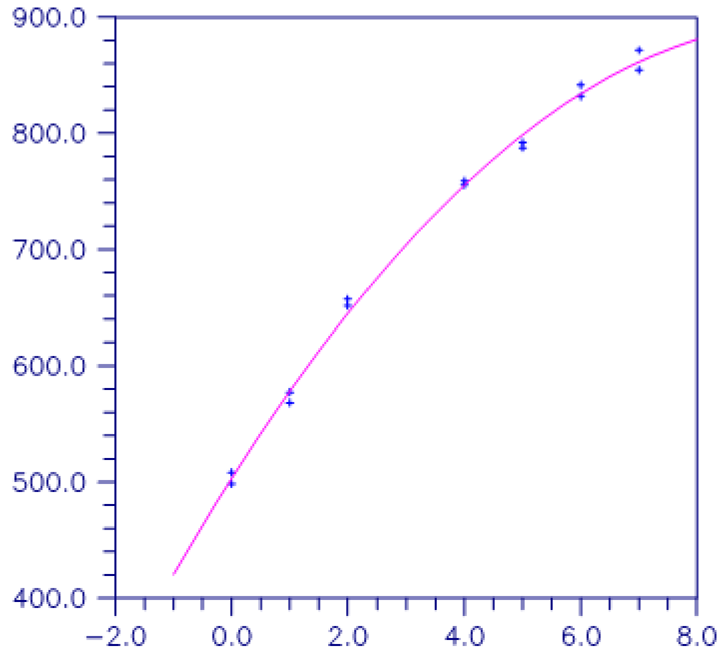



Figure 10.25 — Figure 10-1 A Polynomial Fit

### 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

int 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** | **Result** |
|---|---|
| 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<br>or, randomly,<br>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.

## 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

int 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

int 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

int 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

$$x_i \equiv c x_{i-1} \bmod (2^{31} - 1)$$

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`.

# 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

int 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 imsl_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` (See Chapter 11 of the IMSL C Stat Library user guide.)

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

int 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

more...

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 imsl_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 $\theta$ = `theta`) is

$$f(x) = (e^{-q}\theta^{x})/x!, \text{ for } x = 0, 1,2, \ldots$$

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

int 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 imsl_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

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \text{ for } x \geq 0$$

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>

int 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 imsl_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

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1}(1-x)^{q-1} \text{ for } 0 \le x \le 1$$

where $\Gamma(\cdot)$ is the gamma function.

The algorithm used depends on the values of $p$ and $q$. Except for the trivial cases of $p$ = 1 or $q$ = 1, in which the inverse CDF method is used, all of the methods use acceptance/rejection. If $p$ and $q$ are both less than 1, the method of Jöhnk (1964) is used. If either $p$ or $q$ is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both $p$ and $q$ are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used if `n_random` is less than 4; and algorithm B4PE of Schmeiser and Babu (1980) is used if `n_random` is greater than or equal to 4. Note that for $p$ and $q$ both greater than 1, calling `imsl_f_random_beta` in a loop getting less than 4 variates on each call will not yield the same set of deviates as calling `imsl_f_random_beta` once and getting all the deviates at once.

The values returned in `r` are less than 1.0 and greater than $\varepsilon$ where $\varepsilon$ is the smallest positive number such that $1.0 - \varepsilon$ is less than 1.0.

The function `imsl_random_seed_set` can be used to initialize the seed of the random number generator. The function `imsl_random_option` can be used to select the form of the generator.

## Example

In this example, `imsl_f_random_beta` is used to generate five pseudorandom beta (3, 2) variates.

```
#include <imsl.h>

int 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

int 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 imsl_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 $x_1, \ldots, x_n \in [0, 1]^d, d \geq 1$, 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

$$E = \left[0, t_1\right) \times \ \ldots \ \times \left[0, t_d\right), 0 \le t_j \le 1, 1 \le j \le d$$

λ is the Lebesque measure, and $A(E; n)$ is the number of the $x_j$ contained in *E*.

The sequence $x_1$, $x_2$, ...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*≥*d*. The lowest bound for the discrepancy is obtained for the smallest prime *b*≥*d*, so the optional argument `IMSL_BASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1$, $x_2$, ..., 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

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n) b^{-k-1}, 1 \le j \le d$$

The generator matrix for the series, $c_{kd}^{(j)}$, is defined to be

$$c_{kd}^{(j)} = j^{d-k} c_{kd}$$

and $c_{kd}$ is an element of the Pascal matrix,

$$c_{kd} = \begin{cases} \frac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

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 *x*(*G*(*n*)), where *x* is the generalized Faure sequence.

## Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that `imsl_faure_sequence_init` (see Synopsis) 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_fauer_sequence_free` (see Synopsis) frees data items, stored in the structure, that were allocated by `imsl_faure_sequence_init`.

```
#include <stdio.h>
#include <imsl.h>


int 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]);
            imsl_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

## Functions

# write_matrix

Prints a rectangular matrix (or vector) stored in contiguous memory locations.

## Synopsis

*#include* `<imsl.h>`

*void imsl*`_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`, …, 0)

For *int* `a[]`, use `imsl_i_write_matrix`.

For *double* `a[]`, use `imsl_d_write_matrix`.

For *f_complex* `a[]`, use `imsl_c_write_matrix`.

For *d_complex* `a[]`, use `imsl_z_write_matrix`.

## Required Arguments

*char* `*title` (Input)
The matrix title. Use `\n` within a title to create a new line. Long titles are automatically wrapped.

*int* `nra` (Input)
The number of rows in the matrix.

*int* `nca` (Input)
The number of columns in the matrix.

*float* `a[]` (Input)
Array of size `nra` × `nca` containing the matrix to be printed.

## Synopsis with Optional Arguments

*#include* `<imsl.h>`

*void* `imsl_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`,

`IMSL_TRANSPOSE`,

`IMSL_A_COL_DIM`, *int* `a_col_dim`,

`IMSL_PRINT_ALL`,

`IMSL_PRINT_LOWER`,

`IMSL_PRINT_UPPER`,

IMSL_PRINT_LOWER_NO_DIAG,

IMSL_PRINT_UPPER_NO_DIAG,

IMSL_WRITE_FORMAT, *char* *fmt,

IMSL_ROW_LABELS, *char* *rlabel[],

IMSL_NO_ROW_LABELS,

IMSL_ROW_NUMBER,

IMSL_ROW_NUMBER_ZERO,

IMSL_COL_LABELS, *char* *clabel[],

IMSL_NO_COL_LABELS,

IMSL_COL_NUMBER,

IMSL_COL_NUMBER_ZERO,

IMSL_RETURN_STRING, *char* **string,

IMSL_WRITE_TO_CONSOLE,

0)

## Optional Arguments

IMSL_TRANSPOSE
Print $a^T$.

IMSL_A_COL_DIM, *int* a_col_dim (Input)
The column dimension of a.
Default: a_col_dim = nca

IMSL_PRINT_ALL, *or*

IMSL_PRINT_LOWER, *or*

IMSL_PRINT_UPPER, *or*

IMSL_PRINT_LOWER_NO_DIAG, *or*

`IMSL_PRINT_UPPER_NO_DIAG`

Exactly one of these optional arguments can be specified in order to indicate that either a triangular part of the matrix or the entire matrix is to be printed. If omitted, the entire matrix is printed.

| **Keyword** | **Action** |
|---|---|
| `IMSL_PRINT_ALL` | The entire matrix is printed (the default). |
| `IMSL_PRINT_LOWER` | The lower triangle of the matrix is printed, including the diagonal. |
| `IMSL_PRINT_UPPER` | The upper triangle of the matrix is printed, including the diagonal. |
| `IMSL_PRINT_LOWER_NO_DIAG` | The lower triangle of the matrix is printed, without the diagonal. |
| `IMSL_PRINT_UPPER_NO_DIAG` | The upper triangle of the matrix is printed, without the diagonal. |

`IMSL_WRITE_FORMAT`, *char* `*fmt` (Input)

Character string containing a list of C conversion specifications (formats) to be used when printing the matrix. Any list of C conversion specifications suitable for the data type may be given. For example, `fmt` = "`%10.3f`" specifies the conversion character `f` for the entire matrix. (For the conversion character `f`, the matrix must be of type ***float***, ***double***, ***f_complex***, or ***d_complex***). Alternatively, `fmt` = "`%10.3e%10.3e%10.3f%10.3f%10.3f`" specifies the conversion character `e` for columns 1 and 2 and the conversion character `f` for columns 3, 4, and 5. (For ***complex*** matrices, two conversion specifications are required for each column of the matrix so the conversion character `e` is used in column 1. The conversion character `f` is used in column 2 and the real part of column 3.) If the end of `fmt` is encountered and if some columns of the matrix remain, format control continues with the first conversion specification in `fmt`.

Aside from restarting the format from the beginning, other exceptions to the usual C formatting rules are as follows:

— 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.

— 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.

— 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`. 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`.

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`.

## 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

int 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

int 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

int 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

int 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 in 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. For more information, see the description for imsl_f_machine.

The values that may be used for the attributes are as follows:

| Option | Value | Meaning |
|---|---|---|
| CENTERING | 0<br>1 | Matrix is left justified.<br>Matrix is centered. |
| ROW_WRAP | 0<br>m | A complete row is printed before the next row is printed. Wrapping is used if necessary.<br>Here $m$ is a positive integer. Let $n_1$ be the maximum number of columns that fit across the page, as determined by the widths in the conversion specifications starting with column 1. First, columns 1 through $n_1$ are printed for rows 1 through $m$. Let $n_2$ be the maximum number of columns that fit across the page, starting with column $n_1 + 1$. Second, columns $n_1$+1 through $n_1 + n_2$ are printed for rows 1 through $m$. This continues until the last columns are printed for rows 1 through $m$. Printing continues in this fashion for the next $m$ rows, etc. |
| PAGING | −2<br>−1<br>0<br>$k$ | No paging occurs.<br>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.<br>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.<br>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<br>1 | . . . . . . . . . . is printed for NaN.<br>A blank field is printed for NaN. |
| TITLE_PAGE | 0<br>1 | Title appears only on first page.<br>Title appears on the first page and all continuation pages. |
| FORMAT | 0<br>1<br>2 | Format is "%10.4x".<br>Format is "%12.6w".<br>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 | Description |
|---|---|---|
| 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

int 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

## Functions

# 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 in Chapter 15 of the IMSL Stat Numerical Libraries 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>
extern FILE* imsl_fopen(char* filename, char* mode);
extern int imsl_fclose(FILE* file);

int main()
{
   FILE *ofile;
   float x[] = {3.0, 2.0, 1.0};

   imsl_f_write_matrix ("x (default file)", 1, 3, x, 0);

   ofile = imsl_fopen("myfile", "w");
   imsl_output_file(
       IMSL_SET_OUTPUT_FILE, ofile,
       0);

   imsl_f_write_matrix ("x (myfile)", 1, 3, x, 0);

   imsl_fclose(ofile);
}
```

### Output

```
x (default file)
1          2          3
3          2          1
```

### File myfile

```
x (myfile)
1          2          3
3          2          1
```

## Example 2

This 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 <stdlib.h>
#include <imsl.h>

void *ex1(void* arg);
void *ex2(void* arg);
extern FILE* imsl_fopen(char* filename, char* mode);
extern int imsl_fclose(FILE* file);

int 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 = imsl_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) imsl_free(rand_nums);
    imsl_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 = imsl_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) imsl_free(x);
    imsl_fclose(file_ptr);
}
```

### Output

The content of the file `ex1.out` is shown below.

```
Random Numbers
1      0.0966
2      0.8340
3      0.9477
4      0.0359
5      0.0115
```

### Output

The content of the file `ex2.out` is shown below.

```
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 imsl_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>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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>
```

```
#include <stdio.h>

int 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`.

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` to get the "day number" for December 31, 1985.

```
#include <imsl.h>
#include <stdio.h>

int main()
{
   int        day0, day, month, year;
```

```
   day0 = imsl_date_to_days(31, 12, 1985);
   imsl_days_to_date(day0+100, &day, &month, &year);
   printf("Day 100 of 1986 is (day-month-year) %d-%d-%d\n",
       day, month, year);
}
```

## Output

```
Day 100 of 1986 is (day-month-year) 10-4-1986
```

# error_options

Sets various error handling options.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_error_options (

IMSL_SET_PRINT, *Imsl_error* type, *int* setting,

IMSL_SET_STOP, *Imsl_error* type, *int* setting,

IMSL_SET_TRACEBACK, *Imsl_error* type, *int* setting,

IMSL_FULL_TRACEBACK, *int* setting,

IMSL_GET_PRINT, *Imsl_error* type, *int* *psetting,

IMSL_GET_STOP, *Imsl_error* type, *int* *psetting,

IMSL_GET_TRACEBACK, *Imsl_error* type, *int* *psetting,

IMSL_SET_ERROR_FILE, *FILE* *file,

IMSL_GET_ERROR_FILE, *FILE* **pfile,

IMSL_ERROR_MSG_PATH, *char* *path,

IMSL_ERROR_MSG_NAME, *char* *name,

IMSL_ERROR_PRINT_PROC, *Imsl_error_print_proc* print_proc,

0)

## Optional Arguments

IMSL_SET_PRINT, *Imsl_error* type, *int* setting (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.

## 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. 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>

int 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*);

int 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. 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 <imsl.h>
#include <stdlib.h>
#include <pthread.h>
```

```
void *ex1(void* arg);
void *ex2(void* arg);

int main()
{
   pthread_t      thread1;
   pthread_t      thread2;

   /* 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 the 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 <imsl.h>
#include <stdio.h>
#include <windows.h>

DWORD WINAPI ex1(void *arg);
DWORD WINAPI ex2(void *arg);

int main(int argc, char* argv[])
{
   HANDLE thread[2];

   thread[0] = CreateThread(NULL, 0, ex1, NULL, 0, NULL);
   thread[1] = CreateThread(NULL, 0, ex2, NULL, 0, NULL);

   WaitForMultipleObjects(2, thread, TRUE, INFINITE);
   system("pause");

}

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);
```

```
}

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);
}
```

### 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_type

Gets the type corresponding to the error message from the last function called.

## Synopsis

*#include* `<imsl.h>`

*Imsl_error* imsl_error_type ()

## Return Value

An *Imsl_error* `enum` value is returned.

## Description

The *Imsl_error* `enum` type has seven values: `IMSL_NOTE`, `IMSL_ALERT`, `IMSL_WARNING`, `IMSL_FATAL`, `IMSL_TERMINAL`, `IMSL_WARNING_IMMEDIATE` and `IMSL_FATAL_IMMEDIATE`. See Kinds of Errors and Default Actions for more details.

## Example

See `error_message` for an example.

# error_message

Gets the text of the error message from the last function called.

## Synopsis

*#include* <imsl.h>

*char* *imsl_error_message()

## Return Value

Returns the current error message.

## Description

If the current error type is positive then the last error message set is returned. It does not matter if the error message was printed or not. The current error type is the number returned by `imsl_error_type`. If the current error type is zero then `NULL` is returned.

The returned string can be freed using `imsl_free`.

## Example

This example retrieves the error message from a call to `imsl_f_gamma` with an illegal argument. Error stopping is turned off so that the example continues beyond the terminal error.

```
#include <imsl.h>
#include <stdio.h>

int main(void)
{
   char  *msg;

   imsl_error_options(
       IMSL_SET_STOP, IMSL_TERMINAL, 0,
       0);

   imsl_f_gamma(0.0);
   msg = imsl_error_message();
   printf("type = %d\ncode = %d\n", imsl_error_type(), imsl_error_code());
   printf("msg = %s\n", msg);
}
```

## Output

```
*** TERMINAL Error from imsl_f_gamma. The argument for the function can not
***          be zero.

type = 5
code = 9024
msg = The argument for the function can not be 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>

int 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
```

# initialize_error_handler

Initializes the IMSL C Math Library error handling system.

## Synopsis

*#include* <imsl.h>

*int* imsl_initialize_error_handler ()

## Return Value

If the initialization succeeds, zero is returned. If there is an error, a nonzero value is returned.

## Description

This function is used to initialize the IMSL C Math Library error handling system for the current thread. It is not required, but is always allowed.

Use of this function is advised if the possibility of low heap memory exists when calling the IMSL C Math Library for the first time in the current thread. A successful return from imsl_initialize_error_handler confirms that IMSL C Math Library error handling system has been initialized and is operational. The effects of calling imsl_initialize_error_handler are limited to the calling thread only.

If imsl_initialize_error_handler is not called and initialization of the error handling system fails, an error message is printed to stderr, and execution is stopped.

## Example

In this example, the IMSL C Math Library error handler is initialized prior to calling multiple other IMSL C Math Library functions. Often this is not required, but is advised if the possibility of low heap memory exists. Even if not required, the initialization call is always allowed.

The computations performed in this example are based on Example 1 for imsl_f_spline_least_squares.

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define NDATA   90
/* Define function */
#define F(x)    (float)(1.+ sin(x)+7.*sin(3.0*x))

int main()
{
```

```
  int                  status;
  int                   i, spline_space_dim = 12;
  float                 fdata[NDATA], xdata[NDATA], *random;
  Imsl_f_spline         *sp;

  /* Initialize the IMSL C Math Library error handler. */
  status = imsl_initialize_error_handler();

  /*
   * Verify successful error handler initialization before
   * continuing.
   */
  if (status == 0) {
    /* Generate random numbers */
    imsl_random_seed_set(123457);
    random = imsl_f_random_uniform(NDATA, 0);
    /* Set up data */
    for (i = 0;  i < NDATA;  i++) {
      xdata[i] = 6.*(float)i /((float)(NDATA-1));
      fdata[i] = F(xdata[i]) + 2.*(random[i]-.5);
    }
    sp = imsl_f_spline_least_squares(NDATA, xdata, fdata,
      spline_space_dim, 0);
    printf("       x         error  \n");
    for(i = 0;  i < 10;  i++) {
      float x, error;
      x = 6.*i/9.;
      error = F(x) - imsl_f_spline_value(x, sp, 0);
      printf("%10.3f  %10.3f\n", x, error);
    }
  } else {
    printf("Unable to initialize IMSL C Math Library error handler.\n");
  }
}
```

# set_user_fcn_return_flag

Indicates a condition has occurred in a user-supplied function necessitating a return to the calling function.

## Synopsis

*#include* <imsl.h>

*void* imsl_set_user_fcn_return_flag (*int* code)

## Required Arguments

*int* code (Input)
A user-defined number that indicates the reason for the return from the user-supplied function.

## Description

Given a certain condition in a user-supplied function, imsl_set_user_fcn_return_flag stops executing any IMSL algorithm that has called the function and then returns to the calling function or main program. Upon invocation of imsl_set_user_fcn_return_flag, a flag is set in the IMSL error handler. Upon returning from the user-supplied function, the error IMSL_STOP_USER_FCN is issued with severity IMSL_FATAL. Typically, if you use this function, you would disable stopping on IMSL C MATH errors, thus gaining greater control in situations where you need to prematurely return from an algorithm. (See Programming Notes.)

## Programming Notes

- Since the default behavior of IMSL error handling is to stop execution on IMSL_TERMINAL and IMSL_FATAL errors, execution of the main program stops when the IMSL_STOP_USER_FCN error message is issued unless you alter this behavior by turning stopping off using imsl_error_options.
- In a user-supplied function, the user is responsible for checking error conditions such as memory allocation, return status for any function calls, valid return values, etc.
- Use of this function is valid only if called from within a user-supplied function.

## Examples

### Example 1

This example is based on imsl_f_int_fcn. In this example, the user, for any hypothetical reason, wants to stop the evaluation of the user-supplied function, fcn, when x is less than 0.5.

```
#include <math.h>
```

```
#include <imsl.h>
#include <stdio.h>

float fcn(float x);
float q;
float exact;

int main()
{
   /* Turn off stopping on IMSL_FATAL errors. */
   imsl_error_options(IMSL_SET_STOP, IMSL_FATAL, 0, 0);

   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

   /* evaluate the integral */
   q = imsl_f_int_fcn (fcn, 0.0, 2.0, 0);

   /* The following lines will be executed because
      stopping is turned off. */
   if (q != q) {
       printf("integral = NaN\n");
   } else {
       exact = exp(2.0) + 1.0;
       printf("integral = %10.3f\nexact = %10.3f\n", q, exact);
   }
}

float fcn(float x)
{
   float y;

   /* For a hypothetical reason, stop execution when x < 0.5. */
   if (x < 0.5) {
       imsl_set_user_fcn_return_flag(1);
       return 0;
   }

   y = x * (exp(x));
   return y;
}
```

### Output

```
*** FATAL   Error IMSL_STOP_USER_FCN from imsl_f_int_fcn. Request
***         from user supplied function to stop algorithm. User flag = "1".

integral = NaN
```

## Example 2

This example is based on `imsl_f_chi_squared_test`, Example 3. This example demonstrates how to handle the error condition if the user-supplied function calls a C Math Library function. In this example, `THETA` is set to 0 to force an error condition in calling the `imsl_f_poisson_cdf` function in the user-supplied function.

```
#include <imsl.h>
#include <stdio.h>

#define SEED                    123457
#define N_CATEGORIES            10
#define N_PARAMETERS_ESTIMATED 0
#define N_NUMBERS               1000
#define THETA                   0.0

float          user_proc_cdf(float);

int 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"};

   /* Turn off stopping on IMSL_FATAL errors. */
   imsl_error_options(IMSL_SET_STOP, IMSL_FATAL, 0, 0);

   imsl_random_seed_set(SEED);

   /* Generate the data */
   poisson = imsl_random_poisson(N_NUMBERS, 5.0, 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);

    /* The following lines will be executed because
       stopping is turned off. */
    if (chi_squared_statistics[2] != chi_squared_statistics[2]) {
        printf("p-value = NaN\n");
    } else {
        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,
            IMSL_WRITE_FORMAT,  "%9.1f",
            0);
    }
}

float user_proc_cdf(float k)
{
    float          cdf_v;
    int            setting;

    /* The user is responsible for checking error conditions in the
       user-supplied function, even if the user-supplied function
       is calling an IMSL function.

       For theta = 0.0 (an invalid input), imsl_f_poisson_cdf issues
       an IMSL_TERMINAL error. Thus, stopping is turned off on
       IMSL_TERMINAL errors. */

    /* Get the current terminal error stopping setting which will be
       used for restoring the setting later. */
    imsl_error_options(IMSL_GET_STOP, IMSL_TERMINAL, &setting, 0);

    /* Disable stopping on terminal errors. */
    imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, 0, 0);

    cdf_v = imsl_f_poisson_cdf ((int) k, THETA);

    /* If there is a terminal error, stop and return to main. */
    if (imsl_error_type() == IMSL_TERMINAL) {
        imsl_set_user_fcn_return_flag(1);
        return 0;
    }
```

```
   /* Restore stopping setting */
   imsl_error_options(IMSL_SET_STOP, IMSL_TERMINAL, setting, 0);

   return cdf_v;
}
```

## Output

```
*** TERMINAL Error from imsl_f_poisson_cdf. The mean of the Poisson
***          distribution, "theta" = 0.000000e+000, must be positive.


*** FATAL    Error IMSL_STOP_USER_FCN from imsl_f_chi_squared_test.
***          Request from user supplied function to stop algorithm. User
***          flag = "1".

p-value = NaN
```

# initialize

**Note**: This function is deprecated and its use is not supported. To view the deprecated documentation, see imsl_initialize.pdf on the Rogue Wave website. You can also access a local copy in your IMSL installation directory at
`pdf\deprecated_routines\math\imsl_initialize.pdf`.

# free

Frees memory returned from an IMSL C Math Library function.

## Synopsis

*#include* <imsl.h>

*void* imsl_free (*void* *data)

## Required Arguments

*void* *data (Input)
A pointer to data returned from an IMSL C Math Library function.

## Description

The function imsl_free frees memory using the C runtime library used by the IMSL C Math Library for allocation. It is a wrapper around the standard C runtime function free.

Function imsl_free can always be used to free memory allocated by the IMSL C Math Library, but is required if an application has linked to multiple copies of the C runtime library, with each copy having its own set of heap allocation structures. In this situation, using the C runtime function free can result in memory being allocated with one copy of the C runtime library and freed with a different copy, which may cause abnormal termination. Using imsl_free ensures that the same C runtime library is used for both allocation and freeing.

**Note** that imsl_free should be used only to free memory that was allocated by IMSL C Math Library.

### Example

This example computes a set of random numbers, prints them, and then frees the array returned from the random number generation function.

```
#include <imsl.h>
#include <stdio.h>
int main()
{
   int seed = 123457;
   int n_random = 5;
   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]);

   imsl_free(r);
}
```

## Output

```
Standard normal random deviates:  1.8279 -0.6412 0.7266 0.1747 1.0145
```

# fopen

Opens a file using the C runtime library used by the IMSL C Math Library.

## Synopsis

*#include* <imsl.h>

*#include* <stdio.h>

*FILE* `*imsl_fopen` (*char* `*filename`, *char* `*mode`)

## Required Arguments

*char* `*filename` (Input)
The name of the file to be opened.

*char* `*mode` (Input)
The type of access to be permitted to the file. This string is passed to the C runtime function `fopen`, which determines the valid `mode` values.

## Return Value

A pointer for the file structure, `FILE`, defined in `stdio.h`. To close the file, use `imsl_fclose`. If there is a fatal error, then `NULL` is returned.

## Description

The function `imsl_fopen` opens a file using the C runtime library used by the IMSL C Math Library. It is a wrapper around the standard C runtime function `fopen`.

Function `imsl_fopen` can always be used to open a file which will be used by the IMSL C Math Library, but is required if an application has linked to multiple copies of the C runtime library, with each copy having its own set of file instructions. In this situation, using the C runtime function `fopen` can result in a file being opened with one copy of the C runtime library and reading or writing to it with a different copy, which may cause abnormal behavior or termination. Using `imsl_fopen` ensures that the same C runtime library is used for both the open operation and reading and writing within an IMSL C Math Library function to which the file pointer has been passed as an input argument.

> **Note** that `imsl_fopen` should only be used to open a file whose file pointer will be input to an IMSL C Math Library function. Use `imsl_fclose` to close files opened with `imsl_fopen`.

**Note:** This function is not prototyped in `imsl.h`. This is to avoid including `stdio.h` within `imsl.h`. An extern declaration should be explicitly used to assure compatibility with linkers.

# Example

This example writes a matrix to the file `matrix.txt`. The function `imsl_fopen` is used to open a file. This function returns a file pointer, which is passed to `imsl_output_file`. The matrix is written by `imsl_f_write_matrix`, which uses the file pointer from `imsl_output_file`. The function `imsl_fclose` is then used to close the file.

```
#include <imsl.h>
#include <stdio.h>

extern FILE* imsl_fopen(char* filename, char* mode);
extern int imsl_fclose(FILE* file);

int main()
{
   FILE       *ofile;
   float      x[] = {3.0, 2.0, 1.0};

   imsl_f_write_matrix ("x (default file)", 1, 3, x, 0);

   ofile = imsl_fopen("myfile", "w");
   imsl_output_file(
       IMSL_SET_OUTPUT_FILE, ofile,
       0);
   imsl_f_write_matrix ("x (myfile)", 1, 3, x,
       0);

   imsl_fclose(ofile);
}
```

## Output

```
The content below is stored in the matrix.txt file.
     Matrix written to file matrix.txt
            1           2           3
1         1.1         2.4         3.6
2         4.3         5.1         6.7
3         7.2         8.9         9.3
```

# fclose

Closes a file opened by imsl_fopen.

## Synopsis

*#include* <imsl.h>

*#include* <stdio.h>

*int* imsl_fclose (*FILE* *file)

## Required Arguments

*FILE* *file (Input/Output)
A file pointer returned from imsl_fopen.

## Return Value

The return value is zero if the file is successfully closed. If there is an error, EOF is returned. EOF is defined in stdio.h.

## Description

The function imsl_fclose is a wrapper around the standard C runtime function fclose. It is used to close files opened with imsl_fopen.

**Note** that imsl_fopen should only be used to open a file whose file pointer will be input to an IMSL C Math Library function. Use imsl_fclose to close files opened with imsl_fopen.

**Note:** This function is not prototyped in imsl.h. This is to avoid including stdio.h within imsl.h. An extern declaration should be explicitly used to assure compatibility with linkers.

## Example

See imsl_fopen for an example of its use.

# omp_options

Sets various OpenMP options.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* imsl_omp_options (

IMSL_SET_FUNCTIONS_THREAD_SAFE, *int* setting,

IMSL_GET_FUNCTIONS_THREAD_SAFE, *int* *psetting,

0)

## Return Value

The return value for this function is void.

## Optional Arguments

IMSL_SET_FUNCTIONS_THREAD_SAFE, *int* setting (Input)
If nonzero, user supplied functions are assumed to be thread-safe. This allows user functions to be evaluated in parallel with different arguments.
Default: User supplied functions are not assumed to be thread-safe and are not evaluated in parallel by IMSL C Math Library functions.

IMSL_GET_FUNCTIONS_THREAD_SAFE, *int* *psetting (Output)
Sets the integer pointed to by psetting to zero if user functions are not assumed to be thread-safe and to one if they are assumed to be thread-safe.

## Description

The performance of some IMSL C Math Library functions can be improved if they evaluate user supplied functions in parallel. Unfortunately, incorrect results can occur if the user supplied functions are not thread-safe. By default, the IMSL C Math Library assumes user supplied functions are not thread-safe and thus will not evaluate them in parallel. To change this assumption, use the optional argument IMSL_SET_FUNCTIONS_THREAD_SAFE with its argument equal to one.

This function can be used multiple times in an application to change the thread-safe assumption.

# Example

This example computes the integral $\int_0^2 xe^x dx$. A call to the function `imsl_omp_options` is used to indicate that function `fcn` is thread-safe and so can be safely evaluated by multiple, simultaneous threads.

```
#include <stdio.h>
#include <math.h>
#include <imsl.h>

float fcn(float x);

int main()
{
   float q;
   float exact;

   imsl_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

   /* Evaluate the integral and print result */
   q = imsl_f_int_fcn (fcn, 0.0, 2.0, 0);
   exact = exp(2.0) + 1.0;
   printf("integral = %10.3f\nexact    = %10.3f\n", q, exact);
}

float fcn(float x)
{
   return x * (exp(x));
}
```

### Output

```
integral =      8.389
exact    =     8.389
```

# 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 | $1.01325 \times 10^{5}$ N/m$^2$ ‡ | 2 |

| Name | Description | Value | Reference |
|---|---|---|---|
| AU | Astronomical unit | $1.496 \times 10^{11}$ m | |
| Avogadro | Avogadro's number, $N$ | $6.022045 \times 10^{23}$ 1/mole | 1 |
| Boltzman | Boltzman's constant, $k$ | $1.380662 \times 10^{-23}$ 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, $m_e$ | $9.109534 \times 10^{-31}$ kg | 1 |
| ElectronVolt<br>Euler | ElectronVolt, $ev$<br>Euler's constant, $\gamma$ | $1.6021892 \times 10^{-19}$J<br>0.577…‡ | 1<br>3 |
| Faraday | Faraday constant, $F$ | $9.648456 \times 10^{4}$ C/mole | 1 |
| FineStructure | Fine structure, $\alpha$ | $7.2973506 \times 10^{-3}$ | 1 |
| Gamma | Euler's constant, $\gamma$ | 0.577…‡ | 3 |
| Gas | Gas constant, $R_0$ | 8.31441 J/mole/K | 1 |
| Gravity | Gravitational constant, $G$ | $6.6720 \times 10^{-11}$ N $m^2/kg^2$ | 1 |
| Hbar | Planck's constant/2$\pi$ | $1.0545887 \times 10^{-34}$ J s | 1 |
| PerfectGasVolume | Std vol ideal gas | $2.241383 \times 10^{-2}$ $m^3$/mole | 1 |
| Pi | Pi, $\pi$ | 3.141…‡ | 3 |
| Planck | Planck's constant, $h$ | $6.626176 \times 10^{-34}$ J s | 1 |
| ProtonMass | Proton mass, $M_p$ | $1.6726485 \times 10^{-27}$ kg | 1 |
| Rydberg | Rydberg's constant, $R_\mu$ | $1.097373177 \times 10^{7}$/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 | $1.01325 \times 10^{5}$ $N/m^2$ ‡ | 2 |
| StefanBoltzman | Stefan-Boltzman, $\sigma$ | $5.67032 \times 10^{-8} W/K^4/m^2$ | 1 |
| WaterTriple | Triple point of water | $2.7316 \times 10^{2}$ 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 |

| Unit | Description |
|---|---|
| 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 | l, 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 $\gamma$ 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>
```

```
int 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>

int 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>`

*long* `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.

$$\texttt{imsl\_i\_machine(0)} = \text{Number of bits per byte}$$

Assume that integers are represented in *M*-digit, base-*A* form as

$$\sigma \sum_{k=0}^{M} x_k A^k$$

where **σ** is the sign and $0 \le x_k < A$ for $k = 0, \ldots, M$. Then,

| n | **Definition** |
|---|---|
| 0 | *C*, bits per character |
| 1 | *A*, the base |
| 2 | $M_S$, the number of base-A digits in a *short int* |
| 3 | $A^{M_s} - 1$, the largest *short int* |
| 4 | $M_l$, the number of base-A digits in a *long int* |
| 5 | $A^{M_1} - 1$, the largest *long int* |

Assume that floating-point numbers are represented in *N*-digit, base *B* form as

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k}$$

where $\sigma$ is the sign and $0 \le x_k < B$ for $k = 1, \ldots, N$ for and $E_{\$} \le E \le E_{"}$.

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 32-bit machine with IEEE (Institute for Electrical and Electronics Engineer) arithmetic.

```
#include <imsl.h>
#include <stdio.h>

int main() {
   int n;
   long ans;

   for (n = 0; n <= 12; n++) {
       ans = imsl_i_machine(n);
       printf("imsl_i_machine(%d) = %ld\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

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k}$$

where $\sigma$ 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),

$$E_{\min_f} = \mathrm{imsl\_i\_machine(8)}$$

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 | $B^{E\max_f}\left(1 - B^{-N_f}\right)$, 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>
#include <stdio.h>

int 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 $\texttt{perm}_i = i$, the elements of `perm` are moved in the same manner as are the elements of `x`.

## Examples

### Example 1

In this example, an input vector is sorted algebraically.

```
#include <stdio.h>
#include <imsl.h>

int 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>

int 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, $x_j \leq x_i$ 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, $|y_j| \leq |y_i|$ for $j < i$.

If the option `IMSL_PERMUTATION` is chosen, a record of the permutations to the array `x` is returned. That is, after the initialization of $\texttt{perm}_i = i$, the elements of `perm` are moved in the same manner as are the elements of `x`.

# Examples

## Example 1

In this example, an input vector is sorted algebraically.

```
#include <stdio.h>
#include <imsl.h>

int 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>

int 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. By default, the two norm of *x*, $\|x\|_2$ , is computed.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_f_vector_norm (*int* n, *float* *x,

IMSL_ONE_NORM,

IMSL_INF_NORM, *int* *index,

IMSL_SECOND_VECTOR, *float* *y,0)

## Optional Arguments

`IMSL_ONE_NORM`
Compute the one norm,

$$\|x\|_1 = \sum_{i=0}^{n-1} |x_i|$$

`IMSL_INF_NORM`, *int* `*index` (Output)
Compute the infinity norm,

$$\|x\|_\infty = \max_{0 \le i < n} |x_i|$$

`IMSL_SECOND_VECTOR`, *float* `*y` (Input)
Compute the norm of *x* minus *y*,

$$\|x - y\|, \text{ instead of } \|x\|$$

## Description

By default, `imsl_f_vector_norm` computes the Euclidean norm

$$\left( \sum_{i=0}^{n-1} x_i^2 \right)^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\sum_{i=0}^{n-1} | x_i |$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\max |x_i|$$

is returned. In the case of the infinity norm, the program also returns the index of the element with maximum modulus. If `IMSL_SECOND_VECTOR` is selected, then the norm of $x - y$ is computed.

## Examples

### Example 1

In this example, the Euclidean norm of an input vector is computed.

```
#include <stdio.h>
```

```
#include <imsl.h>

int 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>

int 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
```

# vector_norm (complex)

Computes various norms of a vector or the difference of two vectors.

## Synopsis

*#include* <imsl.h>

*float* imsl_c_vector_norm (*int* n, *f_complex* x [ ],…, 0)

The type *d_complex* function is imsl_z_vector_norm.

## Required Arguments

*int* n (Input)

The length of the input vector(s).

*f_complex* 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. By default, the two norm of *x*, $\|x\|_2$ , is computed.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*float* imsl_c_vector_norm (*int* n, *f_complex* x [ ],

IMSL_ONE_NORM,

IMSL_INF_NORM, *int* *index,

IMSL_SECOND_VECTOR, *f_complex* y [ ],

0)

## Optional Arguments

IMSL_ONE_NORM

Compute the one norm,

$$\|x\|_1 = \sum_{i=0}^{n-1} |x_i|$$

IMSL_INF_NORM, *int* `*index` (Output)

Compute the infinity norm,

$$\|x\|_\infty = \max_{0 \le i < n} |x_i|$$

The index at which the vector has its maximum absolute value is also returned.

IMSL_SECOND_VECTOR, *f_complex* `y[]` (Input)

Compute the norm of *x* minus *y*,

$$\|x - y\|, \text{ instead of } \|x\|$$

## Description

By default, `imsl_c_vector_norm` computes the Euclidean norm

$$\left(\sum_{i=0}^{n-1} x_i^2\right)^{\frac{1}{2}}$$

If the option IMSL_ONE_NORM is selected, the 1-norm

$$\sum_{i=0}^{n-1} |x_i|$$

is returned. If the option IMSL_INF_NORM is selected, the infinity norm

$$\max |x_i|$$

is returned. In the case of the infinity norm, the program also returns the index of the element with maximum modulus. If IMSL_SECOND_VECTOR is selected, then the norm of $x - y$ is computed.

## Examples

### Example 1

In this example, the Euclidean norm of an input vector is computed.

```
#include <stdio.h>
#include <imsl.h>
```

```
int main()
{
    f_complex x[4] = {
        {1.0, 2.0},
        {3.0, 4.0},
        {-2.0, -1.0},
        {4.0, 5.0}
    };
    float norm;

    norm = imsl_c_vector_norm (4, x, 0);

    printf("Euclidean norm of x = %f\n", norm);
}
```

### Output

```
Euclidean norm of x = 8.717798
```

## Example 2

This example computes max $|x_i - y_i|$ and prints the norm and index.

```
#include <stdio.h>
#include <imsl.h>

int main()
{     f_complex x[4] = {
        {1.0, 2.0},
        {3.0, 4.0},
        {-2.0, -1.0},
        {4.0, 5.0}
    };
    f_complex y[4] = {
        {4.0, 3.0},
        {2.0, 1.0},
        {-1.0, -2.0},
        {-5.0, -4.0}
    };
    float norm;
    int index;

    norm = imsl_c_vector_norm (4, 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 = 12.727922 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 imsl_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* × 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>

int 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 imsl_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

$$A = \begin{bmatrix} 1+4i & 2+3i & 9+6i \\ 5+2i & 4-3i & 7+i \end{bmatrix} \quad B = \begin{bmatrix} 3-6i & 2+4i \\ 7+3i & 4-5i \\ 9+2i & 1+3i \end{bmatrix}$$

$$x = \begin{bmatrix} 7+4i \\ 2+2i \\ 1-5i \end{bmatrix} \quad y = \begin{bmatrix} 3+4i \\ 4+2i \\ 2-3i \end{bmatrix}$$

The arrays $A^H$, $Ax$, $x^TA^T$, $AB$, $B^HA^T$, $x^Ty$, and $xy^H$ are computed and printed.

```
#include <imsl.h>

int 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 imsl_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

$$A \in \mathfrak{R}^{\text{nrowa}\times\text{ncola}}$$

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nlcb, *int* nucb, *float* *b (Input)

The sparse matrix

$$B \in \mathfrak{R}^{\text{nrowb}\times\text{ncolb}}$$

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>
int 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`, described in the chapter Eigensystem Analysis. 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 <stdio.h>
#include <stdlib.h>
#include <math.h>

int 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 imsl_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

$$A \in \mathbb{C}^{\text{nrowa}\times\text{ncola}}$$

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nlcb, *int* nucb, *f_complex* *b (Input)
The sparse matrix

$$B \in \mathbb{C}^{\text{nrowb}\times\text{ncolb}}$$

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>

int 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);
}
```

### 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>

int 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);
    imsl_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);
    imsl_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);
    imsl_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 returned value is 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 imsl_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

$$B \in \Re^{\text{nrowb} \times \text{ncolb}}$$

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>
int 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>

int 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 product, 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

$$A \in \mathbb{C}^{\text{nrowa} \times \text{ncola}}$$

with nza nonzero elements.

IMSL_B_MATRIX, *int* nrowb, *int* ncolb, *int* nzb, *Imsl_c_sparse_elem* *b (Input)
The sparse matrix

$$B \in \mathbb{C}^{\text{nrowb} \times \text{ncolb}}$$

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 non-zero 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 imsl_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>

int main()
{
       Imsl_c_sparse_elem a[] = {0, 0, {10.0, 7.0},
                                1, 1, {3.0, 2.0},
                                1, 2, {-3.0, 0.0},
                                1, 3, {-1.0, 2.0},
                                2, 2, {4.0, 2.0},
                                3, 0, {-2.0, -4.0},
                                3, 3, {1.0, 6.0},
                                3, 4, {-1.0, 3.0},
                                4, 0, {-5.0, 4.0},
                                4, 3, {-5.0, 0.0},
                                4, 4, {12.0, 2.0},
                                4, 5, {-7.0, 7.0},
                                5, 0, {-1.0, 12.0},
                                5, 1, {-2.0, 8.0},
                                5, 5, {3.0, 7.0}};
       f_complex    b[] = {{1.0, 1.0}, {2.0, 2.0}, {3.0, 3.0},
                             {4.0, 4.0}, {5.0, 5.0}, {6.0, 6.0}};

       int          n = 6;
       int          nz = 15;
       f_complex   *x;

                       /* Set x = A*b */

       x = imsl_c_mat_mul_rect_coordinate ("A*x",
```

```
               IMSL_A_MATRIX, n, n, nz, a,
               IMSL_X_VECTOR, n, b,
               0);

       imsl_c_write_matrix ("Product Ab", 1, n, x, 0);
}
```

### Output

```
                             Product Ab
                   1                      2                      3
(         3,         17)  (      -19,          5)  (         6,         18)

                   4                      5                      6
(       -38,         32)  (      -63,         49)  (       -57,         83)
```

## Example 2

Using the same matrix *A* and vector *x* given in the last example, the products $Ax$, $A^Tx$, $A^Hx$ and $AA^H$ are computed.

```
#include <imsl.h>
#include <stdio.h>

int 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, nz = 15, nz_z, i;
   f_complex   *b;
```

```
    /* Set b = A*x */
    b = imsl_c_mat_mul_rect_coordinate ("A*x",
        IMSL_A_MATRIX, n, n, nz, a,
        IMSL_X_VECTOR, n, x,
        0);
    imsl_c_write_matrix ("Ax", 1, n, b,
        0);
    imsl_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);
    imsl_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);
    imsl_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 $\alpha$ and $\beta$, 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>

int 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);
       imsl_free(c);
}
```

### Output

```
                   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>

int 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);
       imsl_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 $\alpha$ and $\beta$, 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>

int 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);
       imsl_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

$$(3 + 2i)A^{\mathrm{H}} + (4 + i)\, B^{\mathrm{H}}$$

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>

int 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);
        imsl_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 $\alpha$ and $\beta$, 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 $\alpha$. 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 $\beta$ 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>
#include <stdio.h>

int 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);

    imsl_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>
```

```
#include <stdio.h>

int 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);
   imsl_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 $\alpha$ and $\beta$ 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 $\alpha$. 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 $\beta$ 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 ele-ments. Matrix *B* has seven nonzero elements.

```
#include <imsl.h>
#include <stdio.h>

int 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, 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);
}
```

### 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+3*i***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>
#include <stdio.h>

int 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, 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);
}
```

### 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)

## Optional Arguments

IMSL_ONE_NORM,
Compute the 1-norm of matrix *A*,

`IMSL_INF_NORM`,

Compute the infinity norm of matrix *A*,

## Description

By default, `imsl_f_matrix_norm` computes the Frobenius norm

$$\|A\|_2 = \left[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{ij}^2\right]^{\frac{1}{2}}$$

If the option `IMSL_ONE_NORM` is selected, the 1-norm

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} |A_{ij}|$$

is returned. If the option `IMSL_INF_NORM` is selected, the infinity norm

$$\|A\|_\infty = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} |A_{ij}|$$

is returned.

## Example

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*.

```
#include <imsl.h>

int 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 com-puted, 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>

int 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>

int 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>

int 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>

int 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 imsl_free. If no test was generated, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* *imsl_f_generate_test_band (*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:

| | |
|---|---|
| $a_{i,i} = 4$ | $0 \le i < n$ |
| $a_{i,i+1} = -1$ | $0 \le i < n - 1$ |
| $a_{i+1,i} = -1$ | $0 \le i < n - 1$ |
| $a_{i,i+c} = -1$ | $0 \le i < n - c$ |
| $a_{i+c,i} = -1$ | $0 \le i < n - c$ |

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.

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>

int 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 imsl_free. If no test was generated, then NULL is returned.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*void* *imsl_c_generate_test_band (*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:

| | |
|---|---|
| $a_{i,i} = 6$ | $0 \le i < n$ |
| $a_{i,i+1} = -1 - i$ | $0 \le i < n - 1$ |
| $a_{i+1,i} = -1 - i$ | $0 \le i < n - 1$ |
| $a_{i,i+c} = -1 + i$ | $0 \le i < n - c$ |
| $a_{i+c,i} = -1 + i$ | $0 \le i < n - c$ |

for any $n \ge 3$ and $2 \le c \le n - 1$.

$E$-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>

int 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 `imsl_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

| | |
|---|---|
| $a_{i,i} = 4$ | $0 \le i < n$ |
| $a_{i,i+1} = -1$ | $0 \le i < n - 1$ |
| $a_{i+1,i} = -1$ | $0 \le i < n - 1$ |
| $a_{i,i+c} = -1$ | $0 \le i < n - c$ |
| $a_{i+c,i} = -1$ | $0 \le i < n - c$ |

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:

| | |
|---|---|
| $a_{i,i} = 1$ | $0 \le i < n$ |
| $a_{i,i+c} = i + 2$ | $0 \le i < n - c$ |
| $a_{i,i-n+c} = i + 2$ | $n - c \le i < n$ |
| $a_{i,i+c+1} = -(i + 1)$ | $0 \le i < n - c - 1$ |
| $a_{i,i-n+c+1} = -(i + 1)$ | $n - c - 1 \le i < n$ |
| $a_{i,i+c+2} = 16$ | $0 \le i < n - c - 2$ |
| $a_{i,i-n+c+2} = 16$ | $n - c - 2 \le i < n$ |
| $a_{i,n-11+i+j} = 100j$ | $1 \le i < 11 - j,\ 0 \le j < 10$ |

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>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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 imsl_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:

| | |
|---|---|
| $a_{i,i} = 6$ | $0 \le i < n$ |
| $a_{i,i+1} = -1 - i$ | $0 \le i < n - 1$ |
| $a_{i+1,i} = -1 - i$ | $0 \le i < n - 1$ |
| $a_{i,i+c} = -1 + i$ | $0 \le i < n - c$ |
| $a_{i+c,i} = -1 + i$ | $0 \le i < n - c$ |

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:

| | |
|---|---|
| $a_{i,i} = 1$ | $0 \le i < n$ |
| $a_{i,i+c} = i + 2$ | $0 \le i < n - c$ |
| $a_{i,i-n+c} = i + 2$ | $n - c \le i < n$ |
| $a_{i,i+c+1} = -(i + 1)$ | $0 \le i < n - c - 1$ |
| $a_{i,i+c+1} = -(i + 1)$ | $n - c - 1 \le i < n$ |
| $a_{i,i+c+2} = 16$ | $0 \le i < n - c - 2$ |
| $a_{i,i-n+c+2} = 16$ | $n - c - 2 \le i < n$ |
| $a_{i,n-11+i+j} = 100j$ | $1 \le i < 11 - j,\ 0 \le j < 10$ |

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>
#include <stdio.h>

int main()
{
   int i, n = 5, c = 3, 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>
#include <stdio.h>

int main()
{
   int i, n = 5, c = 3, 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)
```

# Programming Notes for Using NVIDIA® CUDA™ Toolkit

This reference material is intended for users who want to use the computational resources of their NVIDIA GPU board for numeric processing when using the IMSL C Numerical Library. Users who do not have the NVIDIA GPU board can ignore this section.

## Rationale and General Algorithm

NVIDIA® CUDA™ technology leverages the massively parallel processing power of NVIDIA GPUs. The ***NVIDIA CUDA Toolkit*** provides functions which can be used as building blocks for an application taking advantage of this technology. IMSL C Numerical Library has incorporated the use of some of these functions to improve the overall performance of the library.

No direct use or knowledge of the ***NVIDIA CUDA Toolkit*** is required to take advantage of these functions. The program or application is simply rebuilt using environment variables which link with the ***NVIDIA CUDA Toolkit*** libraries.

The strategy for using the NVIDIA GPU is given by the following algorithm:

- ***If*** an NVIDIA-enabled version of an IMSL function is called and the maximum of vector or matrix dimensions are ***greater than or equal to*** a threshold value, ***then***
- Copy the required vector and matrix data from the CPU to the GPU
- Compute the result on the GPU
- Copy the result from the GPU to the CPU
- ***Else***, use the IMSL equivalent version of the function that does not use the GPU.

Normally a code that calls an IMSL/NVIDIA code does not have to be aware of the copy steps or the threshold size. These are hidden from the user code. Users have the option of changing the threshold size. This is important because using the GPU may be slower than using a CPU version of the code until array sizes become sufficiently large. Thereafter the GPU version is typically faster and increasingly much faster as the problem size increases. The default threshold value is 32 but it may not be optimal. This default allows the functions to perform correctly without initial attention to this value.

The user can change the threshold value for all or specific IMSL/NVIDIA functions by using the IMSL function `imsl_cuda_set`. The threshold values can be obtained using the IMSL function `imsl_cuda_get`.

The floating point results obtained using the CPU vs. the GPU will likely differ in units of the low order bits in each component. These differences come from non-equivalent strategies of floating point arithmetic and rounding modes that are implemented in the NVIDIA board. This can be an important detail when comparing results for purposes of benchmarking or code regression. Generally either result should be acceptable for numerical work.

# Implementation

## Basic Linear Algebra Subprograms

IMSL C Numerical Library incorporates the use of many Basic Linear Algebra Subprograms (BLAS) throughout the product. These functions are named using IMSL conventions and used internally. They are not accessible directly by the user.

NVIDIA Corp. implemented certain Level 1, 2 and 3 BLAS in the ***NVIDIA CUDA Toolkit.*** The NVIDIA external names and argument protocols are different from those used by the IMSL C Numerical Library. Wrappers have been written to allow for the IMSL C Numerical Library to access selected routines in the ***NVIDIA CUDA Toolkit***.

In Table 12.9, we document an ***enumeration*** that includes those BLAS for which a CUDA Toolkit implementation is provided in the IMSL C Numerical Library. The naming convention used is the name of the BLAS function prefaced by '`IMSL_CUDA_`'.

## Transforms

***NVIDIA CUDA Toolkit*** implementations of complex two-dimensional FFT (Fast Fourier Transform) functions can be accessed when using functions `imsl_c_fft_2d_complex` and `imsl_z_fft_2d_complex`. The enumerations defined to enable the user to manipulate the parameters used by these function are documented in Table 12.9.

## Utility Functions

There are three utility functions provided in the IMSL C Math Library that can be used to help manage the use of ***NVIDIA CUDA Toolkit***. These utilities appear in Table 12.10 and are described in more detail in their corresponding function descriptions.

Note: Some NVIDIA hardware does not provide double precision arithmetic. Since the double precision functions are included in the ***NVIDIA CUDA Toolkit*** library, those functions will appear to execute correctly even though they do not return correct results. When the IMSL software detects that the correct results are not returned, a warning error message will be printed and the IMSL equivalent of the function which does not use the GPU will be used. The user can eliminate this error by using function `imsl_cuda_set` to set the threshold value to zero.

Table 12.9 — Enumerations of NVIDIA Toolkit-Enabled Functions

| | | |
|---|---|---|
| IMSL_CUDA_SGEMV | IMSL_CUDA_DGER | IMSL_CUDA_STRSM |
| IMSL_CUDA_SGER | IMSL_CUDA_DSYR | IMSL_CUDA_DTRSM |
| IMSL_CUDA_SSYR | IMSL_CUDA_DGEMM | IMSL_CUDA_C_FFT_2D_COMPLEX |
| IMSL_CUDA_SGEMM | IMSL_CUDA_SGBMV | IMSL_CUDA_Z_FFT_2D_COMPLEX |
| IMSL_CUDA_DGEMV | IMSL_CUDA_DGBMV | |

Table 12.10 — NVIDIA CUDA Toolkit Utilities

| |
|---|
| imsl_cuda_get |
| imsl_cuda_set |
| imsl_cuda_free |

## Required NVIDIA Copyright Notice:

# cuda_get

Gets the threshold used by the specified function to determine if the ***NVIDIA CUDA Toolkit*** algorithm will be used.

## Synopsis

*#include* <imsl.h>

*int* imsl_cuda_get (*Imsl_cuda* cuda_name)

## Required Arguments

*Imsl_cuda* cuda_name (Input)
An enumerator which specifies the IMSL function for which the threshold will be retrieved. cuda_name must be one of the values defined in Table 12.9.

## Return Value

Returns the threshold value used to determine when the ***NVIDIA CUDA Toolkit*** version of the function specified by cuda_name will be used. A return value of zero indicates that the IMSL version of the specified function will always be used. A return value greater than zero is the threshold value being used by the function specified by cuda_name. If the problem size is greater than or equal to threshold, the NVIDIA Toolkit algorithm will be used.

See Programming Notes for Using NVIDIA® CUDA™ Toolkit for more information on NVIDIA's CUDA Toolkit integration into IMSL C Numerical Library.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*int* imsl_cuda_get (*Imsl_cuda* cuda_name,
IMSL_GET_DEVICE, *int* *idevice,
0)

## Optional Arguments

IMSL_GET_DEVICE, *int* *idevice (Output)
Returns a value specifying whether the NVIDIA CUDA Toolkit algorithm was used for the last call to the IMSL function specified by cuda_name. Returns 1 if the NVIDIA Toolkit algorithm was used and 0 if the IMSL version of the algorithm was used. A value of −1 indicates that the IMSL function specified by cuda_name has not been used.

## Description

This function returns the threshold value for a specified function. It can optionally be used to return information on the last invocation of the specified function.

## Example

Function imsl_f_lin_sol_gen uses Basic Linear Algebra Subprogram SGER to improve performance. In this example the threshold value of the function SGER is manipulated to force the use of both the IMSL and NVIDIA CUDA Toolkit algorithms.

```
#include <imsl.h>
#include <stdio.h>

int main() {
    int    n = 3, iswitch, idevice;
    float  *x;

    float   lv_a[] =
    {
        1.0, 3.0, 3.0,
        1.0, 3.0, 4.0,
        1.0, 4.0, 3.0
    };

    float   lv_b[] =
    {
        1.0, 4.0, -1.0
    };

    /* Get the current threshold value for SGER */
    iswitch = imsl_cuda_get(IMSL_CUDA_SGER,0);

    /* Set the threshold value of SGER to 0
    so that use of the IMSL version is ensured */
    imsl_cuda_set(IMSL_CUDA_SGER, 0, 0);

    /* Call routine which uses SGER */
    x = imsl_f_lin_sol_gen (n, lv_a, lv_b, 0);
    imsl_free(x);

    /* Check to see what version of SGER was used */
    imsl_cuda_get(IMSL_CUDA_SGER,
        IMSL_GET_DEVICE, &idevice,
        0);

    if (!idevice)
        printf("The IMSL version of SGER was used.\n");
    else
```

```
        printf("Error: The CUDA version of SGER was used.\n");

    /* Set the threshold value to be very small to ensure the
    CUDA version of SGER will be used. */
    imsl_cuda_set(IMSL_CUDA_SGER, 1, 0);

    /* Call routine which uses SGER */
    x = imsl_f_lin_sol_gen(n, lv_a, lv_b, 0);
    imsl_free(x);

    /* Check to see what version of SGER */
    imsl_cuda_get(IMSL_CUDA_SGER,
        IMSL_GET_DEVICE, &idevice,
        0);

    if (!idevice)
        printf("Error:  The IMSL version of SGER was used.\n");
    else
        printf("The CUDA version of SGER was used.\n");

    /* Set the threshold value to the original setting */
    imsl_cuda_set(IMSL_CUDA_SGER, iswitch, 0);

    /* Release GPU memory */
    imsl_cuda_free();
}
```

## Output

```
The IMSL version of SGEMM was used.
The CUDA version of SGEMM was used.
```

## Warning Errors

| | |
|---|---|
| IMSL_CUDA_ENUM_NAME | The argument specified for "cuda_name" = # is not valid. |
| IMSL_CUDA_NOT_IMPLEMENTED | The specified function name does not have a CUDA implementation. |
| IMSL_FCN_NOT_USED | The specified function name has not yet been used. |
| IMSL_CUDA_NOT_AVAIL | The CUDA Toolkit algorithms are not implemented using this version of the library. Use the CUDA link environment variables to leverage the CUDA Toolkit algorithms. |

# cuda_set

Set the threshold used by the specified function to determine if the ***NVIDIA CUDA Toolkit*** algorithm will be used.

## Synopsis

*#include* <imsl.h>

*int* imsl_cuda_set (*Imsl_cuda* cuda_name, *int* threshold)

## Required Arguments

*Imsl_cuda* cuda_name (Input)
An enumerator which specifies the IMSL function for which the threshold will be set. cuda_name must be one of the values defined in Table 12.9.

*int* threshold (Input)
The threshold value which determines if the ***NVIDIA CUDA Toolkit*** algorithm will be used. A value of zero ensures that the IMSL version of the function will always be used. If the problem size is greater than or equal to threshold, the NVIDIA Toolkit algorithm will be used.

## Return Value

A return value of 0 indicates the threshold was set successfully. A return value of 1 indicates the threshold was not set successfully.

## Synopsis with Optional Arguments

*#include* <imsl.h>

*int* imsl_cuda_set (*Imsl_cuda* cuda_name, *int* threshold,
IMSL_SET_ALL,
0)

## Optional Arguments

IMSL_SET_ALL (Input)
Sets the threshold of all IMSL/NVIDIA implemented functions.

## Description

This routine sets the threshold value for a specified function.

# Example

Refer to `imsl_cuda_get` for the document example.

# Warning Errors

| | |
|---|---|
| `IMSL_CUDA_ENUM_NAME` | The argument specified for "`cuda_name`" = # is not valid. |
| `IMSL_CUDA_NOT_IMPLEMENTED` | The specified function name does not have a CUDA implementation. |
| `IMSL_CUDA_SET_ERROR` | Invalid argument. This argument must be greater than or equal to 0. |
| `IMSL_CUDA_NOT_AVAIL` | The CUDA Toolkit algorithms are not implemented using this version of the library. Use the CUDA link environment variables to leverage the CUDA Toolkit algorithms. |

# cuda_free

Releases NVIDIA memory allocated by the IMSL C Numerical Library.

## Synopsis

*#include* `<imsl.h>`

*int* `imsl_cuda_free` ()

## Return Value

A return value of zero indicates that no error was encountered when releasing the memory. A return value of one indicates that an error was generated when releasing the memory.

## Description

Allocation of NVIDIA memory by the IMSL C Numerical Library is done as infrequently as possible by reusing allocated memory.

When allocating memory either through the IMSL C Numerical Library or through your own application, an error message indicating not enough memory exists on the GPU may possibly be generated. Sufficient memory may be available but may be reserved by the IMSL C Numerical Library. This function is provided to force the reserved NVIDIA memory to be released.

## Example

Refer to `imsl_cuda_get` for the document example.

## Warning Errors

| | |
|---|---|
| `IMSL_CUDA_FREE` | An error was encountered freeing memory from the GPU. |
| `IMSL_CUDA_NOT_AVAIL` | The CUDA Toolkit algorithms are not implemented using this version of the library. Use the CUDA link environment variables to leverage the CUDA Toolkit algorithms. |

# Reference Material

## Contents

# 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 an IMSL function, exactly one error state exists. (A code 0 "error" is no error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except `IMSL_TERMINAL` may be informational errors. The include file, `imsl.h`, defines `IMSL_NOTE`, `IMSL_ALERT`, `IMSL_WARNING`, `IMSL_FATAL`, `IMSL_TERMINAL`, `IMSL_WARNING_IMMEDIATE`, and `IMSL_FATAL_IMMEDIATE` as an enumerated data type `Imsl_error`.

`IMSL_NOTE`. A ***note*** is issued to indicate the possibility of a trivial error or simply to provide information about the computations.

Default attributes: PRINT=NO, STOP=NO.

`IMSL_ALERT`. An ***alert*** indicates that a function value has been set to 0 due to underflow.

Default attributes: PRINT=NO, STOP=NO.

`IMSL_WARNING`. A ***warning*** indicates the existence of a condition that may require corrective action by the user or calling routine. A warning error may be issued because the results are accurate to only a few decimal places, because some of the output may be erroneous, but most of the output is correct, or because some assumptions underlying the analysis technique are violated. Usually no corrective action is necessary, and the condition can be ignored.

Default attributes: PRINT=YES, STOP=NO.

`IMSL_FATAL`. A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling routine must take corrective action to recover.

Default attributes: PRINT=YES, STOP=YES.

`IMSL_TERMINAL`. A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors may also be caused by various programming errors impossible to diagnose correctly in C. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the function with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error, because corrective action within the program is generally not reasonable. In normal usage, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur.

Default attributes: PRINT=YES, STOP=YES.

`IMSL_WARNING_IMMEDIATE`. An *immediate warning* error is identical to a warning error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=NO.

`IMSL_FATAL_IMMEDIATE`. An *immediate fatal* error is identical to a fatal error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=YES.

The user can set PRINT and STOP attributes by calling `imsl_error_options` as described in Chapter 12, “Utilities”.

# Errors in Lower-Level Functions

It is possible that a user’s program may call an IMSL C Math Library function that in turn calls a nested sequence of lower-level functions. If an error occurs at a lower level in such a nest of functions, and if the lower-level function cannot pass the information up to the original user-called function, then a traceback of the functions is produced. The only common situation in which this can occur is when an IMSL C Math Library function calls a user-supplied routine that in turn calls another IMSL C Math Library function.

# Functions for Error Handling

The user may interact in three ways with the IMSL error-handling system:

1. Change the default actions.
2. Determine the code of an informational error so as to take corrective action.
3. Initialize the error handling systems.

The functions that support these actions are:

- imsl_error_options

  Sets the actions to be taken when errors occur.

- imsl_error_type

  Retrieves the *Imsl_error* enum error type value.

- imsl_error_code

  Retrieves the integer code for an informational error.

- imsl_error_message

  Retrieves the error message string.

- imsl_initialize_error_handler

  Initializes the IMSL C Math Library error handling system for the current thread. This function is not required but is always allowed. Use of this function is advised if the possibility of low heap memory exists when calling the IMSL C Math Library for the first time in the current thread.

These functions are documented in Chapter 15, ***Utilities***.

## 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 for each thread by calling imsl_error_options from within each thread.

See Example 3 and Example 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:

1. f_complex z for single-precision complex values
2. d_complex w for double-precision complex values

Each complex value is a C language ***structure*** that consists of a pair of real values, the ***real*** and ***imaginary*** part of the complex number. To access the real part of a single-precision complex number ***z***, use the subexpression `z.re`. For the imaginary part, use the subexpression `z.im`. Use subexpressions `w.re` and `w.im` for the real and imaginary parts of a double-precision complex number ***w***. The structure is declared within `imsl.h` as follows:

```
typedef struct{
    float re;
    float im;
} f_complex;
```

Several standard operations and functions are available for users to perform calculations with complex numbers within their programs. The operations are provided for both single and double precision data types. Notice that even the ordinary arithmetic operations of "+", "-", "*", and "/" must be performed using the appropriate functions.

A uniform prefix name is used as part of the names for the operations and functions. The prefix `imsl_c_` is used for `f_complex` data. The prefix `imsl_z_` is used with `d_complex` data.

# Single-Precision Complex Operations and Functions

| Operation | Function Name | Function Result | Function Argument(s) |
|---|---|---|---|
| $z = -x$ | `z = imsl_c_neg(x)` | `f_complex` | `f_complex` |
| $z = x + y$ | `z = imsl_c_add(x,y)` | `f_complex` | `f_complex` (both) |
| $z = x - y$ | `z = imsl_c_sub(x,y)` | `f_complex` | `f_complex` (both) |
| $z = x * y$ | `z = imsl_c_mul(x,y)` | `f_complex` | `f_complex` (both) |
| $z = x / y$ | `z = imsl_c_div(x,y)` | `f_complex` | `f_complex` (both) |
| $x = = y$[a] | `z = imsl_c_eq(x,y)` | `Int` | `f_complex` (both) |
| $z = x$ *Drop Precision* | `z = imsl_cz_convert(x)` | `f_complex` | `d_complex` |

[a] Result has the value 1 if *x* and *y* are valid numbers with real and imaginary parts identical; otherwise, result has the value 0.

| Operation | Function Name | Function Result | Function Argument(s) |
|---|---|---|---|
| $z = a + ib$ *Ascend Data* | `z = imsl_cf_convert(a,b)` | `f_complex` | `float` (both) |
| $z = \bar{x}$ | `z = imsl_c_conjg(x)` | `f_complex` | `f_complex` |
| $a = \lvert z \rvert$ | `a = imsl_c_abs(z)` | `float` | `f_complex` |
| $a = \arg(z)$ $-\pi < a \le \pi$ | `a = imsl_c_arg(z)` | `float` | `f_complex` |
| $z = \sqrt{z}$ | `z = imsl_c_sqrt(z)` | `f_complex` | `f_complex` |
| $z = \cos(z)$ | `z = imsl_c_cos(z)` | `f_complex` | `f_complex` |
| $z = \sin(z)$ | `z = imsl_c_sin(z)` | `f_complex` | `f_complex` |
| $z = \exp(z)$ | `z = imsl_c_exp(z)` | `f_complex` | `f_complex` |
| $z = \log(z)$ | `z = imsl_c_log(z)` | `f_complex` | `f_complex` |
| $z = x^a$ | `z = imsl_cf_power(x,a)` | `f_complex` | `f_complex, float` |
| $z = x^y$ | `z = imsl_cc_power(x,y)` | `f_complex` | `f_complex` (both) |
| $c = a^k$ | `c = imsl_fi_power(a,k)` | `float` | `float, int` |
| $c = a^b$ | `c = imsl_ff_power(a,b)` | `float` | `float` (both) |
| $m = j^k$ | `m = imsl_ii_power(j,k)` | `Int` | `int` (both) |

# Double-Precision Complex Operations and Functions

| Operation | Function Name | Function Result | Function Argument(s) |
|---|---|---|---|
| $z = -x$ | `z = imsl_z_neg(x)` | `d_complex` | `d_complex` |
| $z = x + y$ | `z = imsl_z_add(x,y)` | `d_complex` | `d_complex` (both) |
| $z = x - y$ | `z = imsl_z_sub(x,y)` | `d_complex` | `d_complex` (both) |
| $z = x * y$ | `z = imsl_z_mul(x,y)` | `d_complex` | `d_complex` (both) |
| $z = x / y$ | `z = imsl_z_div(x,y)` | `d_complex` | `d_complex` (both) |
| $x==y$[b] | `z = imsl_z_eq(x,y)` | `Int` | `d_complex` (both) |
| $z = x$ *Drop Precision* | `z = imsl_zc_convert(x)` | `d_complex` | `f_complex` |
| $z = a + ib$ *Ascend Data* | `z = imsl_zd_convert(a,b)` | `d_complex` | `double` (both) |

[b] Result has the value 1 if *x* and *y* are valid numbers with real and imaginary parts identical; otherwise, result has the value 0.

| Operation | Function Name | Function Result | Function Argument(s) |
|---|---|---|---|
| $z = x$ | `z = imsl_z_conjg(x)` | `d_complex` | `d_complex` |
| $a = \lvert z \rvert$ | `a = imsl_z_abs(z)` | `Double` | `d_complex` |
| $a = \arg(z)$ $-\pi < a \le \pi$ | `a = imsl_z_arg(z)` | `Double` | `d_complex` |
| $z = \sqrt{z}$ | `z = imsl_z_sqrt(z)` | `d_complex` | `d_complex` |
| $z = \cos(z)$ | `z = imsl_z_cos(z)` | `d_complex` | `d_complex` |
| $z = \sin(z)$ | `z = imsl_z_sin(z)` | `d_complex` | `d_complex` |
| $z = \exp(z)$ | `z = imsl_z_exp(z)` | `d_complex` | `d_complex` |
| $z = \log(z)$ | `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) |

## Example

The following sample code computes and prints several quantities associated with complex numbers. Note that the quantity

$$w = \sqrt{3 + 4i}$$

has a rounding error associated with it. Also the quotient $z = (1 + 2i) / (3 + 4i)$ has a rounding error. The result is acceptable in both cases because the relative errors $|w - (2 + 2i)| / |w|$ and $|z * (3 + 4i) - (1 + 2i)| / |(1 + 2i)|$ are approximately the size of machine precision.

```
#include <imsl.h>

main()
{
    f_complex           x = {1,2};
    f_complex           y = {3,4};
    f_complex           z;
    f_complex           w;
    int                 isame;
    float               eps = imsl_f_machine(4);
                                /* Echo inputs x and y */
    printf("Data:  x = (%g, %g)\n       y = (%g, %g)\n\n",
           x.re, x.im, y.re, y.im);
                                /* Add inputs */
    z = imsl_c_add(x,y);
    printf("Sum:  z = x + y = (%g, %g)\n\n", z.re, z.im);
                                /* Compute square root of y */
    w = imsl_c_sqrt(y);
    printf("Square Root: w = sqrt(y) = (%g, %g)\n", w.re, w.im);
                                /* Check results */
    z = imsl_c_mul(w,w);
    printf("Check:       w*w = (%g, %g)\n", z.re, z.im);
    isame = imsl_c_eq(y,z);
    printf("             y == w*w = %d\n", isame);
    z = imsl_c_sub(z,y);
    printf("Difference:  w*w - y = (%g, %g) = (%g, %g) * eps\n\n",
           z.re, z.im, z.re/eps, z.im/eps);
                                /* Divide inputs */
    z = imsl_c_div(x,y);
    printf("Quotient:    z = x/y = (%g, %g)\n", z.re, z.im);
                                /* Check results */
    w = imsl_c_sub(x, imsl_c_mul(z, y));
    printf("Check:       w = x - z*y = (%g, %g) = (%g, %g) * eps\n",
           w.re, w.im, w.re/eps, w.im/eps);
}
```

## Output

```
Data:   x = (1, 2)
        y = (3, 4)

Sum:    z = x + y = (4, 6)

Square Root: w = sqrt(y) = (2, 1)
Check:       w*w = (3, 4)
             y == w*w = 0
Difference:  w*w - y = (-2.38419e-07, 4.76837e-07) = (-2, 4) * eps

Quotient:    z = x/y = (0.44, 0.08)
Check:       w = x - z*y = (5.96046e-08, 0) = (0.5, 0) * eps
```

# Appendix A: References


## Abramowitz and Stegun

Abramowitz, Milton, and Irene A. Stegun (editors) (1964), ***Handbook of Mathematical Functions with Formulas***, ***Graphs***, ***and Mathematical Tables***, National Bureau of Standards, Washington.

## Ahrens and Dieter

Ahrens, J.H., and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, ***Computing,*** **12**, 223–246.

## Akima

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, ***Journal of the ACM***, **17**, 589–602.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, ***ACM Transactions on Mathematical Software***, **4**, 148–159.

## Altman and Gondzio

Altman, Anna, and Jacek Gondzio (1998), ***Regularized Symmetric Indefinite Systems in Interior Point Methods for Linear and Quadratic Optimization***, Logilab Technical Report 1998.6, Logilab, HEC Geneva, Section of Management Studies, Geneva.

## 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.

## Beckers

Beckers, Stan (1980), The Constant Elasticity of Variance Model and Its Implications For Option Pricing , The Journal of Finance, Vol. 35, No. 3, pp. 661-673.

## Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

## Blom and Zegeling

Blom, JG, and Zegeling, PA (1994), *A Moving-grid Interface for Systems of One-dimensional Time-dependent Partial Differential Equations*, ACM Transactions on Mathematical Software, Vol 20, No.2, 194-214.

## 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.

### Brenan, Campbell, and Petzold

Brenan, K.E., S.L. Campbell, L.R. Petzold (1989), *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elseview Science Publ. Co.

### Brent

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Brent, R.P. (1971), *An Algorithm With Guaranteed Convergence for Finding a Zero of a Function*, *The Computer Journal*, **14**, 422−425.

### 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.

## Demmel et al.

Demmel, J.W., J.R. Gilbert, and X.S. Li, (1999), ***SuperLU Users' Guide***, Tech. Rep. LBNL–44289, Lawrence Berkeley National Laboratory.

Demmel, J.W., S.C. Eisenstat, J.R. Gilbert, X.S. Li, and J.W.H. Liu, (1999), ***A Supernodal Approach To Sparse Partial Pivoting***, SIAM Journal on Matrix Analysis and its Applications, **20**, 720-755.

Demmel, J.W., J. R. Gilbert, and X. S. Li (1999c) , An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, ***SIAM Journal on Matrix Analysis and its Applications***, **20(4)**, 915- 952.

## 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.

## Doornik

Doornik, J. A., An Improved Ziggurat Method to Generate Normal Random Samples, `http://www.doornik.com/research/ziggurat.pdf`.

## 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 et al.

Duff, Ian S., R. G. Grimes, and J. G. Lewis (1992) first ed, ***Users' Guide for the Harwell-Boeing Sparse Matrix Collection***, CERFACS, Toulouse Cedex, France.

## 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 231 – 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**, 187198.

## 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.

Gill, P.E., W. Murray, and M.H. Wright (1981), ***Practical Optimization***, Academic Press Inc. Limited, London.

## 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.

## Gondzio (1994)

Gondzio, Jacek (1994), ***Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming***, Logilab Technical Report 1994.20, Logilab, HEC Geneva, Section of Management Studies, Geneva.

## Gondzio (1995)

Gondzio, Jacek (1995), HOPDM - ***Modular Solver for LP Problems***, User's Guide to version 2.12, WP-95-50, International Institute for Applied Systems Analysis, Laxenburg, Austria.

## 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.

## Hanson

Hanson, R. J. (2008), ***Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements***, White Paper.

## Hanson and Krogh

Hanson, R. J., and Krogh, F. T., (2008), ***Solving Constrained Differential-Algebraic Systems Using Projections***, White Paper.

## 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 nonstiff 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.

## Kochanek and Bartels

Kochanek, Doris H. U., and Bartels, Richard H (1984), ***Interpolating Splines with Local Tension, Continuity, and Bias Control,*** ACM SIGGRAPH, vol. 18, no. 3, pp. 33–41.

## Krogh

Krogh, Fred, T. (2005), ***An Algorithm for Linear Programming,*** `http://mathalacarte.com/fkrogh/pub/lp.pdf`, Tujunga, CA.

Krogh, Fred, T. (1974), "Changing Stepsize in the Integration of Differential Equations Using Modified Divided Differences" in ***Proceedings of the Conference on the Numerical Solution of Ordinary Differential Equations***,Springer Verlag, Berlin, no. 362, pp. 22-71.

## 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.

## Marsaglia and Tsang

Marsaglia, G. and Tsang, W. W (2000), The Ziggurat Method for Generating Random Variables, *Journal of Statistical Software*, Volume 5-8, pages 1-7.

## 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 ***Lp*** 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.

## Nelder and Mead

Nelder, J.A., and Mead, R. (1965), A simplex method for function minimization, ***The Computer Journal***, **7(4)**, 308-313.

## 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.

## NVIDIA

NVIDIA Corporation (©2005-2011), © All rights reserved. Portions of the NVIDIA **SGEMM** and **DGEMM** library routines were written by Vasily Volkov and are subject to the Modified Berkeley Software Distribution License. (©) 2007-09, Regents of the University of California.

## 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.

## Pennington and Berzins

Pennington, S. V., Berzins, M., (1994), Software for first-order partial differential equations. 63−99.

## Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, ***Communications of the ACM***, **13**, 624.

## Petzold

Petzold, L.R. (1982), A description of DASSL: A differential/ algebraic system solver, ***Proceedings of the IMACS World Congress***, Montreal, Canada.

## Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. 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.

## Ralston

Ralston, Anthony (1965), ***A First Course in Numerical Analysis***, McGraw-Hill, NY.

## Rauber et. al.

Rauber, T., G. Rünger, and C. Scholtes (1999), Scalability of Sparse Cholesky Factorization, ***International Journal of High Speed Computing***, **10**, No. 1, 19 - 52.

## 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.

## Salane

Salane, D.E. (1986), Adaptive Routines for Forming Jacobians Numerically, SAND86-1319, Sandia National Laboratories.

## 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

## Verwer et al

Verwer, J. G., Blom, J. G., Furzeland, R. M., and Zegeling, P. A. (1989), A moving-grid method for one-dimensional PDEs Based on the Method of Lines, ***Adaptive Methods for Partial Differential Equations***, Eds., J. E. Flaherty, P. J. Paslow, M. S. Shephard, and J. D. Vasiilakis, SIAM Publications, Philadelphia, PA (USA) pp. 160-175.

## 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.

## Wilmott et al

Wilmott, P., Howison, and S., Dewynne, J., (1996), ***The Mathematics of Financial Derivatives (A Student Introduction)***, Cambridge Univ. Press, New York, NY. 317 pages.

Appendix B: **Alphabetical**

# Summary of Functions

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [Y] [Z]

# A

| Function | Purpose Statement |
|---|---|
| accr_interest_maturity | Evaluates the accrued interest for a security that pays at maturity. |
| accr_interest_periodic | Evaluates the accrued interest for a security that pays periodic interest. |
| airy_Ai | Evaluates the Airy function. |
| airy_Ai_derivative | Evaluates the derivative of the Airy function. |
| airy_Bi | Evaluates the Airy function of the second kind. |
| airy_Bi_derivative | Evaluates the derivative of the Airy function of the second kind. |

# B

| Function | Purpose Statement |
|---|---|
| bessel_exp_I0 | Evaluates the exponentially scale modified Bessel function of the first kind of order zero. |
| bessel_exp_I1 | Evaluates the exponentially scaled modified Bessel function of the first kind of order one. |
| bessel_exp_K0 | Evaluates the exponentially scaled modified Bessel function of the second kind of order zero. |
| bessel_exp_K1 | Evaluates the exponentially scaled modified Bessel function of the second kind of order one. |
| bessel_I0 | Evaluates the real modified Bessel function of the first kind of order zero $I_0(x)$. |
| bessel_I1 | Evaluates the real modified Bessel function of the first kind of order one $I_1(x)$. |
| bessel_Ix | Evaluates a sequence of modified Bessel functions of the first kind with real order and complex arguments. |
| bessel_J0 | Evaluates the real Bessel function of the first kind of order zero $J_0(x)$. |
| bessel_J1 | Evaluates the real Bessel function of the first kind of order one $J_1(x)$. |
| bessel_Jx | Evaluates a sequence of Bessel functions of the first kind with real order and complex arguments. |
| bessel_K0 | Evaluates the real modified Bessel function of the second kind of order zero $K_0(x)$. |
| bessel_K1 | Evaluates the real modified Bessel function of the second kind of order one $K_1(x)$. |
| bessel_Kx | Evaluates a sequence of modified Bessel functions of the second kind with real order and complex arguments. |
| bessel_Y0 | Evaluates the real Bessel function of the second kind of order zero $Y_0(x)$. |
| bessel_Y1 | Evaluates the real Bessel function of the second kind of order one $Y_1(x)$. |
| bessel_Yx | Evaluates a sequence of Bessel functions of the second kind with real order and complex arguments. |
| beta | Evaluates the real beta function $\beta(x, y)$. |
| beta_cdf | Evaluates the beta probability distribution function. |
| beta_incomplete | Evaluates the real incomplete beta function $I_x = \beta x(a, \mathrm{b})/\beta(a, b)$. |

| | |
|---|---|
| beta_inverse_cdf | Evaluates the inverse of the beta distribution function. |
| binomial_cdf | Evaluates the binomial distribution function. |
| bivariate_normal_cdf | Evaluates the bivariate normal distribution function. |
| bond_equivalent_yield | Evaluates the bond-equivalent for a Treasury yield. |
| bounded_least_squares | Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm. |
| 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. |

# C

| Function | Purpose Statement |
|---|---|
| chi_squared_cdf | Evaluates the chi-squared distribution function. |
| chi_squared_inverse_cdf | Evaluates the inverse of the chi-squared distribution function. |
| chi_squared_test | Performs a chi-squared goodness-of-fit test. |
| constant | Returns the value of various mathematical and physical constants. |
| constrained_nlp | Solves a general nonlinear programming problem using a sequential equality constrained quadratic programming method. |
| convexity | Evaluates the convexity for a security. |
| convolution | Computes the convolution, and optionally, the correlation of two real vectors. |
| convolution (complex) | Computes the convolution, and optionally, the correlation of two complex vectors. |
| coupon_days | Evaluates the number of days in the coupon period that contains the settlement date. |
| coupon_number | Evaluates the number of coupons payable between the settlement date and maturity date. |
| covariances | Computes the sample variance-covariance or correlation matrix. |
| ctime | Returns the number of CPU seconds used. |
| cub_spline_integral | Computes the integral of a cubic spline. |
| cub_spline_interp_e_cnd | Computes a cubic spline interpolant, specifying various endpoint conditions. |
| cub_spline_interp_shape | Computes a shape-preserving cubic spline. |
| cub_spline_smooth | Computes a smooth cubic spline approximation to noisy data by using cross-valida-tion to estimate the smoothing parameter or by directly choosing the smoothing parameter. |
| cub_spline_tcb | Computes a tension-continuity-bias (TCB) cubic spline interpolant. This is also called a Kochanek-Bartels spline and is a generalization of the Catmull–Rom spline. |
| cub_spline_value | Computes the value of a cubic spline or the value of one of its derivatives. |
| cuda_free | Releases NVIDIA memory allocated by the IMSL C Numerical Library. |
| cuda_get | Gets parameters used by the specified function to determine if the *NVIDIA CUDA Toolkit* algorithm will be used. |

| | |
|---|---|
| cuda_set | Set parameters used by the specified function to determine if the *NVIDIA CUDA Toolkit* algorithm will be used. |
| cumulative_interest | Evaluates the cumulative interest paid between two periods. |
| cumulative_principal | Evaluates the cumulative principal paid between two periods. |

# D

| Function | Purpose Statement |
|---|---|
| date_to_days | Evaluates the number of days from January 1, 1900, to the given date. |
| days_before_settlement | Evaluates the number of days from the beginning of the coupon period to the settlement date. |
| days_to_date | Gives the date corresponding to the number of days since January 1, 1900. |
| days_to_next_coupon | Evaluates the number of days from settlement date to the next coupon date. |
| dea_petzold_gear | Solves a first order differential-algebraic system of equations, $g(t, y, y') = 0$, using the Petzold-Gear BDF method. |
| depreciation_amordegrc | Evaluates the depreciation for each accounting period. Similar to depreciation_amorlinc. |
| depreciation_amorlinc | Evaluates the depreciation for each accounting period. Similar to depreciation_amordegrc. |
| depreciation_db | Evaluates the depreciation of an asset for a specified period using the fixed-declining balance method. |
| depreciation_ddb | Evaluates the depreciation of an asset for a specified period using the double-declining method. |
| depreciation_sln | Evaluates the straight line depreciation of an asset for one period. |
| depreciation_syd | Evaluates the sum-of-years digits depreciation of an asset for a specified period. |
| depreciation_vdb | Evaluates the depreciation of an asset for any given period, including partial periods, using the double-declining balance method. |
| differential_algebraic_eqs | Solves a first order differential-algebraic system of equations, g(t, y, y′) = 0, with optional additional constraints and user-defined linear system solver. |
| discount_price | Evaluates the price per $100 face value of a discounted security. |
| discount_rate | Evaluates the discount rate for a security. |
| discount_yield | Evaluates the annual yield for a discounted security. |
| dollar_decimal | Converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number. |
| dollar_fraction | Converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fraction. |
| duration | Evaluates the annual duration of a security with periodic interest payment. |

# E

| Function | Purpose Statement |
|---|---|
| effective_rate | Evaluates the effective annual interest rate. |
| eig_gen | Computes the eigenexpansion of a real matrix *A*. |
| eig_gen (complex) | Computes the eigenexpansion of a complex matrix *A*. |
| eig_herm (complex) | Computes the eigenexpansion of a complex Hermitian matrix *A*. |
| eig_sym | Computes the eigenexpansion of a real symmetric matrix *A*. |
| eig_symgen | Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. *A* and *B* are real and symmetric. *B* is positive definite. |
| elliptic_integral_E | Evaluates the complete elliptic integral of the second kind $E(x)$. |
| elliptic_integral_K | Evaluates the complete elliptic integral of the kind $K(x)$. |
| elliptic_integral_RC | Evaluates an elementary integral from which inverse circular functions, logarithms, and inverse hyperbolic functions can be computed. |
| elliptic_integral_RD | Evaluates Carlson's elliptic integral of the second kind $\mathrm{RD}(x, y, z)$. |
| elliptic_integral_RF | Evaluates Carlson's elliptic integral of the first kind $\mathrm{RF}(x, y, z)$. |
| elliptic_integral_RJ | Evaluates Carlson's elliptic integral of the third kind $\mathrm{RJ}(x, y, z, \rho)$. |
| erf | Evaluates the real error function $\mathrm{erf}(x)$. |
| erf_inverse | Evaluates the real inverse error function $\mathrm{erf}^{-1}(x)$. |
| erfc | Evaluates the real complementary error function $\mathrm{erfc}(x)$. |
| erfc_inverse | Evaluates the real inverse complementary error function $\mathrm{erfc}^{-1}(x)$. |
| erfce | Evaluates the exponentially scaled complementary error function. |
| erfe | Evaluates a scaled function related to $\mathrm{erfc}(z)$. |
| error_code | Gets the code corresponding to the error message from the last function called. |
| error_message | Gets the text of the error message from the last function called. |

| | |
|---|---|
| error_options | Sets various error handling options. |
| error_type | Gets the type corresponding to the error message from the last function called. |

# F

| Function | Purpose Statement |
|---|---|
| F_cdf | Evaluates the *F* distribution function. |
| F_inverse_cdf | Evaluates the inverse of the *F* distribution function. |
| 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. |
| faure_next_point | Evaluates a shuffled Faure sequence. |
| fclose | Closes a file opened by `imsl_fopen`. |
| fcn_derivative | Computes the first, second or third derivative of a user-supplied function. |
| feynman_kac | Solves a generalized Feynman-Kac equation on a finite interval using Hermite quintic splines. |
| feynman_kac_evaluate | Computes the value of a Hermite quintic spline or the value of one of its derivatives. |
| fft_2d_complex | Computes the complex discrete two-dimensional Fourier transform of a complex two-dimensional array. |
| fft_complex | Computes the complex discrete Fourier transform of a complex sequence. |
| fft_complex_init | Computes the parameters for imsl_c_fft_complex. |
| fft_cosine | Computes the discrete Fourier cosine transformation of an even sequence. |
| fft_cosine_init | Computes the parameters needed for imsl_f_fft_cosine. |
| fft_real | Computes the real discrete Fourier transform of a real sequence. |
| fft_real_init | Computes the parameters for imsl_f_fft_real. |
| fft_sine | Computes the discrete Fourier sine transformation of an odd sequence. |
| fft_sine_init | Computes the parameters needed for imsl_f_fft_sine. |
| fopen | Opens a file using the C runtime library used by the IMSL C Math Library. |
| free | Frees memory returned from an IMSL C Math Library function. |
| fresnel_integral_C | Evaluates the cosine Fresnel integral. |
| fresnel_integral_S | Evaluates the sine Fresnel integral. |
| future_value | Evaluates the future value of an investment. |
| future_value_schedule | Evaluates the future value of an initial principal after applying a series of compound interest rates. |

# G

| Function | Purpose Statement |
|---|---|
| gamma | Evaluates the real gamma function $\Gamma(x)$. |
| gamma_cdf | Evaluates the gamma distribution function. |
| gamma_incomplete | Evaluates the incomplete gamma function $\gamma(a, x)$. |
| gauss_quad_rule | Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions. |
| geneig | Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with $A$ and $B$ real. |
| geneig (complex) | Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$, with $A$ and $B$ complex. |
| generate_test_band | Generates test matrices of class $E(n, c)$. |
| generate_test_band (complex) | Generates test matrices of class $Ec(n, c)$. |
| generate_test_coordinate | Generates test matrices of class $D(n, c)$ and $E(n, c)$. |
| generate_test_coordinate (complex) | Generates test matrices of class $D(n, c)$ and $E(n, c)$. |

# H

| Function | Purpose Statement |
|---|---|
| hypergeometric_cdf | Evaluates the hypergeometric distribution function. |

| Function | Purpose Statement |
|---|---|
| initialize | Deprecated.<br>Initializes the IMSL C Math Library. |
| initialize_error_handler | Initializes the IMSL C Math Library error handling system. |
| int_fcn | Integrates a function using a globally adaptive scheme based on Gauss-Kronrod rules. |
| int_fcn_2d | Computes a two-dimensional iterated integral. |
| int_fcn_alg_log | Integrates a function with algebraic-logarithmic singularities. |
| int_fcn_cauchy | Computes integrals of the form<br>$\int_a^b \frac{f(x)}{x-c} dx$<br>in the Cauchy principal value sense. |
| int_fcn_fourier | Computes a Fourier sine or cosine transform. |
| int_fcn_hyper_rect | Integrates a function on a hyper-rectangle. |
| int_fcn_inf | Integrates a function over an infinite or semi-infinite interval. |
| int_fcn_qmc | Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method. |
| int_fcn_sing | Integrates a function, which may have endpoint singularities, using a globally adaptive scheme based on Gauss-Kronrod rules. |
| int_fcn_sing_1d | Integrates a function with a possible internal or endpoint singularity. |
| int_fcn_sing_2d | Integrates a function of two variables with a possible internal or endpoint singularity. |
| int_fcn_sing_3d | Integrates a function of three variables with a possible internal or endpoint singularity. |
| int_fcn_sing_pts | Integrates a function with singularity points given. |
| int_fcn_smooth | Integrates a smooth function using a nonadaptive rule. |
| int_fcn_trig | Integrates a function containing a sine or a cosine factor. |
| interest_payment | Evaluates the interest payment for a given period for an investment. |
| interest_rate_annuity | Evaluates the interest rate per period for an annuity. |
| interest_rate_security | Evaluates the interest rate for a fully invested security. |
| internal_rate_of_return | Evaluates the internal rate of return for a schedule of cash flows. |

| | |
|---|---|
| `internal_rate_schedule` | Evaluates the internal rate of return for a schedule of cash flows that is not necessarily periodic. |
| `inverse_laplace` | Computes the inverse Laplace transform of a complex function. |

# J

| Function | Purpose Statement |
|---|---|
| jacobian | Approximates the Jacobian of m functions in n unknowns using divided differences. |

# K

| Function | Purpose Statement |
|---|---|
| kelvin_bei0 | Evaluates the Kelvin function of the first kind, bei, of order zero. |
| kelvin_bei0_derivative | Evaluates the derivative of the Kelvin function of the first kind, bei, of order zero. |
| kelvin_ber0 | Evaluates the Kelvin function of the first kind, ber, of order zero. |
| kelvin_ber0_derivative | Evaluates the derivative of the Kelvin function of the first kind, ber, of order zero. |
| kelvin_kei0 | Evaluates the Kelvin function of the second kind, kei, of order zero. |
| kelvin_kei0_derivative | Evaluates the derivative of the Kelvin function of the second kind, kei, of order zero. |
| kelvin_ker0 | Evaluates the Kelvin function of the second kind, der, of order zero. |
| kelvin_ker0_derivative | Evaluates the derivative of the Kelvin function of the second kind, ker, of order zero. |

| Function | Purpose Statement |
|---|---|
| lin_least_squares_gen | Solves a linear least-squares problem $Ax = b$. |
| lin_lsq_lin_constraints | Solves a linear least squares problem with linear constraints. |
| lin_prog | Solves a linear programming problem using the revised simplex algorithm. |
| lin_sol_def_cg | Solves a real symmetric definite linear system using a conjugate gradient method. |
| lin_sol_gen | Solves a real general system of linear equations $Ax = b$. |
| lin_sol_gen (complex) | Solves a complex general system of linear equations $Ax = b$. |
| lin_sol_gen_band | Solves a real general band system of linear equations $Ax = b$. |
| lin_sol_gen_band (complex) | Solves a complex general system of linear equations $Ax = b$. |
| lin_sol_gen_coordinate | Solves a sparse system of linear equations $Ax = b$. |
| lin_sol_gen_coordinate (complex) | Solves a system of linear equations $Ax = b$, with sparse complex coefficient matrix $A$. |
| lin_sol_gen_min_residual | Solves a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method. |
| lin_sol_nonnegdef | Solves a real symmetric nonnegative definite system of linear equations $Ax = b$. |
| lin_sol_posdef | Solves a real symmetric positive definite system of linear equations $Ax = b$. |
| lin_sol_posdef (complex) | Solves a complex Hermitian positive definite system of linear equations $Ax = b$. |
| lin_sol_posdef_band | Solves a real symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode. |
| lin_sol_posdef_band (complex) | Solves a complex Hermitian positive definite system of linear equations $Ax = b$ in band symmetric storage mode. |
| lin_sol_posdef_coordinate | Solves a sparse real symmetric positive definite system of linear equations $Ax = b$. |
| lin_sol_posdef_coordinate (complex) | Solves a sparse Hermitian positive definite system of linear equations $Ax = b$. |
| lin_svd_gen | Computes the SVD, A = $USV^T$, of a real rectangular matrix A. |

| | |
|---|---|
| lin_svd_gen (complex) | Computes the SVD, $A = USV^H$, of a complex rectangular matrix $A$. |
| linear_programming | Solves a linear programming problem. |
| log_beta | Evaluates the logarithm of the real beta function $\ln \beta(x, y)$. |
| log_gamma | Evaluates the logarithm of the absolute value of the gamma function $\log \lvert\Gamma(x)\rvert$. |

# M

| Function | Purpose Statement |
|---|---|
| machine (float) | Returns information describing the computer's floating-point arithmetic. |
| machine (integer) | Returns integer information describing the computer's arithmetic. |
| mat_add_band | Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$. |
| mat_add_band (complex) | Adds two band matrices, both in band storage mode, $C \leftarrow \alpha A + \beta B$. |
| mat_add_coordinate | Performs element-wise addition of two real matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$. |
| mat_add_coordinate (complex) | Performs element-wise addition on two complex matrices stored in coordinate format, $C \leftarrow \alpha A + \beta B$. |
| mat_mul_rect | Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, the bilinear form, or any triple product. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| matrix_norm | Computes various norms of a rectangular matrix. |
| matrix_norm_band | Computes various norms of a matrix stored in band storage mode. |
| matrix_norm_coordinate | Computes various norms of a matrix stored in coordinate format. |
| min_con_gen_lin | Minimizes a general objective function subject to linear equality/inequality constraints. |
| min_uncon | Finds the minimum point of a smooth function $f(x)$ of a single variable using only function evaluations. |

| | |
|---|---|
| min_uncon_deriv | Finds the minimum point of a smooth function $f(x)$ of a single variable using both function and first derivative evaluations. |
| min_uncon_golden | Finds the minimum point of a nonsmooth function of a single variable. |
| min_uncon_multivar | Minimizes a function $f(x)$ of $n$ variables using a quasi-Newton method. |
| min_uncon_polytope | Minimizes a function of $n$ variables using a direct search polytope algorithm. |
| modified_duration | Evaluates the modified Macauley duration of a security. |
| modified_internal_rate | Evaluates the modified internal rate of return for a series of periodic cash flows. |
| modified_method_of_lines | Solves a system of partial differential equations of the form $u_t + f(x, t, u, u_x, u_{xx})$ using the method of lines. |

# N

| Function | Purpose Statement |
|---|---|
| net_present_value | Evaluates the net present value of an investment based on a series of periodic. |
| next_coupon_date | Evaluates the next coupon date after the settlement date. |
| nominal_rate | Evaluates the nominal annual interest rate. |
| nonlin_least_squares | Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm. |
| nonneg_least_squares | Computes the non-negative least squares (nnls) solution. |
| nonneg_matrix_factorization | Given an $m \times n$ real matrix $A \geq 0$ and an integer $k \leq \min(m, n)$, compute a factorization $A \cong FG$. |
| normal_cdf | Evaluates the standard normal (Gaussian) distribution function. |
| normal_inverse_cdf | Evaluates the inverse of the standard normal (Gaussian) distribution function. |
| number_of_periods | Evaluates the number of periods for an investment based on periodic and constant payment and a constant interest rate. |

# O

| Function | Purpose Statement |
|---|---|
| ode_adams_2nd_order | Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method. |
| ode_adams_gear | Solves a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods. |
| Ode_adams_krogh | Solves an initial-value problem for a system of ordinary differential equations of order one or two using a variable order Adams method |
| ode_runge_kutta | Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method. |
| omp_options | Sets various OpenMP options. |
| output_file | Sets the output file or the error message output file. |

# P

| Function | Purpose Statement |
| --- | --- |
| page | Sets or retrieve the page width or length. |
| payment | Evaluates the periodic payment for an investment. |
| pde_1d_mg | Solves a system of one-dimensional time-dependent partial differential equations using a moving-grid interface. |
| poisson_cdf | Evaluates the Poisson distribution function. |
| poly_regression | Performs a polynomial least-squares regression. |
| present_value | Evaluates the present value of an investment. |
| present_value_schedule | Evaluates the present value for a schedule of cash flows that is not necessarily periodic. |
| previous_coupon_date | Evaluates the previous coupon date before the settlement date. |
| price | Evaluates the price per $100 face value of a security that pays periodic interest. |
| price_maturity | Evaluates the price per $100 face value of a security that pays interest at maturity. |
| principal_payment | Evaluates the payment on the principal for a given period. |
| psi | Evaluates the derivative of the log gamma function. |
| psi1 | Evaluates the second derivative of the log gamma function. |

# Q

| Function | Purpose Statement |
|---|---|
| quadratic_prog | Solves a quadratic programming problem subject to linear equality or inequality constraints. |

# R

| Function | Purpose Statement |
|---|---|
| radial_evaluate | Evaluates a radial basis fit. |
| radial_scattered_fit | Computes an approximation to scattered data in $\mathbf{R}^n$ for $n \geq 2$ using radial basis functions. |
| random_beta | Generates pseudorandom numbers from a beta distribution. |
| random_exponential | Generates pseudorandom numbers from a standard exponential distribution. |
| random_gamma | Generates pseudorandom numbers from a standard gamma distribution. |
| random_normal | Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method. |
| random_option | Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator. |
| random_poisson | Generates pseudorandom numbers from a Poisson distribution. |
| random_seed_get | Retrieves the current value of the seed used in the IMSL random number generators. |
| random_seed_set | Initializes a random seed for use in the IMSL random number generators. |
| random_uniform | Generates pseudorandom numbers from a uniform (0, 1) distribution. |
| ranks | Computes the ranks, normal scores, or exponential scores for a vector of observations. |
| read_mps | Reads an MPS file containing a linear programming problem or a quadratic programming problem. |
| received_maturity | Evaluates the amount received for a fully invested security. |
| regression | Fits a multiple linear regression model using least squares. |

# S

| Function | Purpose Statement |
|---|---|
| scattered_2d_interp | Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables. |
| set_user_fcn_return_flag | Indicates a condition has occurred in a user-supplied function necessitating a return to the calling function. |
| simple_statistics | Computes basic univariate statistics. |
| smooth_1d_data | Smooth one-dimensional data by error detection. |
| sort | Sorts a vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort per-muta-tion can be returned. |
| sort (integer) | Sorts an integer vector by algebraic value. Optionally, a vector can be sorted by absolute value, and a sort per-mutation can be returned. |
| sparse_cholesky_smp | Computes the Cholesky factorization of a sparse real symmetric positive definite matrix A by an OpenMP par-allelized supernodal algorithm and solves the sparse real positive definite system of linear equations $Ax = b$. |
| sparse_cholesky_smp (complex) | Computes the Cholesky factorization of a sparse (com-plex) Hermitian positive definite matrix A by an OpenMP parallelized supernodal algorithm and solves the sparse Hermitian positive definite system of linear equations $Ax = b$. |
| sparse_lin_prog | Solves a sparse linear programming problem by an infeasible primal-dual interior-point method. |
| sparse_quadratic_prog | Solves a sparse convex quadratic programming prob-lem by an infeasible primal-dual interior-point method. |
| spline_2d_integral | Evaluates the integral of a tensor-product spline on a rectangular domain. |
| spline_2d_interp | Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data. |
| spline_2d_least_squares | Computes a two-dimensional, tensor-product spline approximant using least squares. |
| spline_2d_value | Computes the value of a tensor-product spline or the value of one of its partial deriva-tives. |
| spline_integral | Computes the integral of a spline. |
| spline_interp | Computes a spline interpolant. |
| spline_knots | Computes the knots for a spline interpolant. |
| spline_least_squares | Computes a least-squares spline approximation. |
| spline_lsq_constrained | Computes a least-squares constrained spline approximation. |

| | |
|---|---|
| spline_nd_interp | Performs a multidimensional interpolation and differentiation for up to 7 dimensions. |
| spline_value | Computes the value of a spline or the value of one of its derivatives. |
| superlu | Computes the LU factorization of a general sparse matrix by a column method and solves the real sparse linear system of equations $Ax = b$. |
| superlu (complex) | Computes the $LU$ factorization of a general complex sparse matrix by a column method and solves the complex sparse linear system of equations $Ax = b$. |
| superlu_smp | Computes the $LU$ factorization of a general sparse matrix by a left-looking column method using OpenMP parallelism, and solves the real sparse linear system of equations $Ax = b$ |
| superlu_smp (complex) | Computes the $LU$ factorization of a general complex sparse matrix by a left-looking column method using OpenMP parallelism and solves the complex sparse linear system of equations $Ax = b$. |

# T

| Function | Purpose Statement |
|---|---|
| `t_cdf` | Evaluates the Student's *t* distribution function. |
| `t_inverse_cdf` | Evaluates the inverse of the Student's *t* distribution function. |
| `table_oneway` | Tallies observations into a one-way frequency table. |
| `treasury_bill_price` | Computes the price per $100 face value for a Treasury bill. |
| `treasury_bill_yield` | Computes the yield for a Treasury bill. |

# U

| Function | Purpose Statement |
|---|---|
| user_fcn_least_squares | Computes a least-squares fit using user-supplied functions. |

# V

| Function | Purpose Statement |
|---|---|
| vector_norm | Computes various norms of a vector or the difference of two vectors. |
| vector_norm (complex) | Computes various norms of a vector or the difference of two vectors. |
| version | Returns integer information describing the version of the library, license number, operating system, and compiler. |

# W

| Function | Purpose Statement |
|---|---|
| write_matrix | Prints a rectangular matrix (or vector) stored in contiguous memory locations. |
| write_options | Sets or retrieve an option for printing a matrix. |

# Y

| Function | Purpose Statement |
|---|---|
| year_fraction | Evaluates the year fraction that represents the number of whole days between two dates. |
| yield_maturity | Evaluates the annual yield of a security that pays interest at maturity. |
| yield_periodic | Evaluates the yield of a security that pays periodic interest. |

# Z

| Function | Purpose Statement |
| --- | --- |
| zeros_function | Finds the real zeros of a real, continuous, univariate function. |
| zeros_poly | Finds the zeros of a polynomial with real coefficients using the Jenkins-Traub three-stage algorithm. |
| zeros_poly (complex) | Finds the zeros of a polynomial with complex coefficients using the Jenkins-Traub three-stage algorithm. |
| zeros_sys_eqn | Solves a system of $n$ nonlinear equations $f(x) = 0$ using a modified Powell hybrid algorithm. |
| zero_univariate | Finds a zero of a real univariate function. |

# Product Support

## Contacting IMSL Support

Users within support warranty may contact Rogue Wave Software regarding the use of the IMSL C Numerical Library. IMSL Support can consult on the following topics:

- Clarity of documentation
- Possible IMSL-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem

Not included in these topics are mathematical/statistical consulting and debugging of your program.

**Refer to the following for IMSL Product Support contact information:**

`http://www.roguewave.com/help-support/customer-support`

The following describes the procedure for consultation with Rogue Wave:

1. Include your IMSL license number
2. Include the product name and version number: IMSL C Numerical Library Version 8.5.0
3. Include compiler and operating system version numbers
4. Include the name of the routine for which assistance is needed and a description of the problem

# Index