Visual Numerics®

VERSION 5.5

IMSL

C Numerical Library™

User's Guide

VOLUME 3 of 4: C Stat Library™ [CHAPTERS 1-7]

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

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

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

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

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

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

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

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
Germany

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

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

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

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

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

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

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

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

IMSL Fortran and C and Java
Application Development Tools

CStat Library /V1 Table of Contents

Introduction

IMSL C/Stat/Library

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

Getting Started

To use any of the C/Stat/Library functions, you must first write a program in C to call the function. Each function conforms to established conventions in programming and documentation. First priority in development is given 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 C/Stat/Library function to all other C functions that you use.

ANSI C vs. Non-ANSI C

All of the examples in this documentation conform to ANSI C. If you are not using ANSI C, you will need to modify your examples in functions that are declared or in those arrays that are initialized as type *float*.

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

```
#include <imsls.h>

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

main()
{
    int         n_observations = 3,
                n_parameters = 1,
                n_independent = 1;
    float       *theta_hat;
11    float       x[3] = {1.0, 2.0, 3.0};
12    float       y[3] = {2.0, 4.0, 3.0};
13                     /* Evaluate the integral */
14    theta_hat = imsls_f_nonlinear_regression(fcn, n_parameters,
15                n_observations, n_independent, x, y, 0);
16                     /* Print the result and the exact answer */
17    imsls_f_write_matrix("estimated coefficient", 1, 1, theta_hat, 0);
18 }
19 float fcn(int n_independent, float x[], int n_parameters,
20           float theta[])
21 {
22    return exp(theta[0]*x[0]);
23 }
```

If using non-ANSI C, you will need to modify lines 3, 11, 12, 19, and 20 as follows:

```
3  float          fcn(); /* Function is not prototyped */
    .
    .
    .
11    static float       x[3] = {1.0, 2.0, 3.0};
12    static float       y[3] = {2.0, 4.0, 3.0};
    .
    .
    .
19  float fcn(n_independent, x, n_parameters,
20            theta)     /*Declaration of variable names*/
20a int n_independent;
20b float x[];
20c int n_parameters;
20d float theta[];       /*Type definitions of variables*/
```

The imsls.h File

The include file `<imsls.h>` is used in all the examples in this manual. This file contains prototypes for all IMSL-defined functions; the structures, *Imsls_f_regression*, *Imsls_d_regression*, *Imsls_f_poly_regression*, *Imsls_d_poly_regression*, *Imsls_f_arma*, and *Imsls_d_arma*; and the enumerated data types, *Imsls_arma_method*, *Imsls_permute*, *Imsls_dummy_method*, *Imsls_write_options*, *Imsls_page_options*, and *Imsls_error*.

Thread Safe Usage

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

Signal Handling

When calling C/Stat/Library from a multithreaded application it is necessary to turn C/Stat/Library 's signal-handling capability off. This is accomplished by making a single call to `imsls_error_options` *before* any calls are made to C/Stat/Library. For an example of turning off C/Stat/Library's internal-signal handling , see Chapter 14, “Utilities”, Example 3 of `imsls_error_options`.

C/Stat/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/Stat/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/Stat/Library error handler using `imsls_error_options`. For an example of setting error handler options for separate threads, see Chapter 14, “Utilities”, Example 3 of `imsls_error_options`.

Routines that Produce Output

A number of routines in C/Stat/Library can be used to produce output. The function `imsls_output_file` can be used to control which file the output is directed. In an application with a single thread of execution, a single call to `imsls_output_file` can be used to set the file to which the output will be directed. In a multithreaded application each thread must call `imsls_output_file` to change the default setting of where output will be directed. See Chapter 14, “Utilities”, Example 2 of `imsls_output_ file` for more details.

Input Arguments

In a multithreaded application attention must be given to the data sent to C/Stat/Library. Some arguments that may appear to be input-only are temporarily modified during the call and restored before returning to the caller. Care must be used to avoid usage of the same data space in separate threads calling functions in C/Stat/Library.

Matrix Storage Modes

In this section, the word *matrix* is used to refer to a mathematical object and the word *array* is used to refer to its representation as a C data structure. In the following list of array types, the C/Stat/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 regression, the pointer points to the estimated coefficients. Normally, the input array values are not changed by the functions.

In the C/Stat/Library, an array is a pointer to a contiguous block of data. An array is *not* a pointer to a pointer to the rows of the matrix. Typical declarations are as follows:

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

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

General Mode

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

Rectangular Mode

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

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 *int*, *float*, or *double*.

Memory Allocation for Output Arrays

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

`IMSLS_RETURN_USER`, *float* `a[]`

then the computed answers are stored in the user-provided array `a`, and the pointer returned by the function is set to point to the user-provided array `a`. If an invocation does not use `IMSLS_RETURN_USER`, then a pointer to the function is internally initialized (through a memory allocation request to `malloc`) and stores the answers there. (To release this space, `free` can be used. Both `malloc` and `free` are standard C library functions declared in the header.) 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, the optional arguments

`IMSLS_ANOVA_TABLE`, *float* `**anova_table` (Output)
`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)

specify two mutually exclusive optional arguments. If the first option is chosen, *float* `**anova_table` refers to the address of a pointer to an internally allocated array containing the analysis of variance statistics. On return, the pointer is initialized (through a memory allocation request to `malloc`), and the array is stored there. Typically, *float* `*anova_table` is declared, `&anova_table` is used as an argument to this function, and `free(anova_table)` is used to release the space. In the second option, the analysis of variance statistics are stored in the user-provided array `anova_table`.

Finding the Right Function

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

Often, the quickest way to use the C/Stat/Library is to find an example similar to your problem, then mimic the example. Each function documented has at least one example demonstrating its application.

Organization of the Documentation

This manual contains a concise description of each function with at least one 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 an introduction followed by a table of contents listing the functions included in the chapter. Documentation of the functions consists of the following information:

- **Section Name:** Usually, the common root for the type *float* and type *double* versions of the function.
- **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.

  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.

  References: References are listed alphabetically by author.

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 are also available in type *int*. The following list is of each type and the corresponding prefix of the function name in which multiple type versions exist:

Type	Prefix
float	`imsls_f_`
double	`imsls_d_`
int	`imsls_i_`

The section names for the functions contain only the common root to make finding the functions easier. For example, the functions `imsls_f_simple_statistics` and `imsls_d_simple_statistics` can be found in Chapter 1, Basic Statistics, in the "`simple_statistics`" section.

Where appropriate, the same variable name is used consistently throughout the C/Stat/Library. For example, `anova_table` denotes the array containing the analysis of variance statistics and `y` denotes a vector of responses for a dependent variable.

When writing programs accessing the C/Stat/Library, 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 "`imsls_`" in any combination of uppercase or lowercase characters.

Error Handling, Underflow, and Overflow

The functions in the C/Stat/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 `IMSLS_FATAL`, `IMSLS_WARNING`, etc. See the section "User Errors" in the Reference Material for further details.

In general, the C/Stat/Library codes are written so that computations are not affected by underflow, provided the system (hardware or software) replaces an underflow with the value 0. Normally, system error messages indicating underflow can be ignored.

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

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

Printing Results

Most functions in the C/Stat/Library do not print any of the results; the output is returned in C variables. The C/Stat/Library does contain some special functions just for printing arrays. For example, IMSL function `imsls_f_write_matrix` is convenient for printing matrices of type *float*. See Chapter 13, "Printing Functions," for detailed descriptions of these functions.

Missing Values

Some of the functions in the C/Stat/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 function `imsls_f_machine`, described in Chapter 14, "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/Stat/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/Stat/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/Stat/Library function `imsls_f_kolmogorov_one` and optional argument `IMSLS_FCN_W_DATA`.

```
#include <imsls.h>
#include <stdio.h>
float cdf_w_data(float, void *data_ptr);
float cdf(float);
void main()
{
  float *statistics=NULL, *diffs = NULL, *x=NULL;
  int nobs = 100, nmiss;
  float usr_data[] = {0.5, .2886751};

  imsls_random_seed_set(123457);
  x = imsls_f_random_uniform(nobs, 0);

  statistics = imsls_f_kolmogorov_one(cdf, nobs, x,
                               IMSLS_N_MISSING, &nmiss,
                               IMSLS_DIFFERENCES, &diffs,
                               IMSLS_FCN_W_DATA, cdf_w_data, usr_data,
                               0);
  printf("D = %8.4f\n", diffs[0]);
  printf("D+ = %8.4f\n", diffs[1]);
  printf("D- = %8.4f\n", diffs[2]);
  printf("Z = %8.4f\n", statistics[0]);
  printf("Prob greater D one sided = %8.4f\n", statistics[1]);
  printf("Prob greater D two sided = %8.4f\n", statistics[2]);
  printf("N missing = %d\n", nmiss);
}
/*
 * User function that accepts additional data in a (void*) pointer.
 * This (void*) pointer can be cast to any type and dereferenced to
 * get at any sort of data-type or structure that is needed.
 * For example, to get at the data in this example
 *  *((float*)data_ptr)   contains the value  0.5
 *  *((float*)data_ptr+1) contains the value  0.2886751.
 */
float cdf_w_data(float x, void *data_ptr)
{
  float mean, std, z;
  mean = *((float*)data_ptr);
  std =  *((float*)data_ptr+1);

  z = (x-mean)/std;
  return(imsls_f_normal_cdf(z));
}
/*  Dummy function to satisfy C prototypes. */
float cdf(float x)
{
  return;
}
```

Chapter 1: Basic Statistics

Routines

Usage Notes

The functions for computations of basic statistics generally have relatively simple arguments. In most cases, the first required argument is the number of observations. The data are input in either a one- or two-dimensional array. As usual, when a two-dimensional array is used, the rows contain observations and the columns represent variables. Most of the functions in this chapter allow for missing values. Missing value codes can be set by using function `imsls_f_machine`, described in Chapter 14, Utilities.

Several functions in this chapter perform statistical tests. These functions generally return a "*p*-value" for the test, often as the return value for the C function. The *p*-value is between 0 and 1 and is the probability of observing data that would yield a test statistic as extreme or more extreme under the assumption of the null hypothesis. Hence, a small *p*-value is evidence for the rejection of the null hypothesis.

simple_statistics

Computes basic univariate statistics.

Synopsis

#include `<imsls.h>`

float `*imsls_f_simple_statistics` (*int* `n_observations`,
    int `n_variables`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_simple_statistics`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `n_variables` (Input)
Number of variables.

float `x[]` (Input)
Array of size `n_observations` × `n_variables` containing the data matrix.

Return Value

A pointer to an array containing some simple statistics for each of the columns in `x`. If `IMSLS_MEDIAN` and `IMSLS_MEDIAN_AND_SCALE` are not used as optional arguments, the size of the matrix is 14 × `n_variables`. The columns of this matrix correspond to the columns of `x`, and the rows contain the following statistics:

Row	Statistic
0	mean
1	variance
2	standard deviation
3	coefficient of skewness
4	coefficient of excess (kurtosis)
5	minimum value
6	maximum value
7	range
8	coefficient of variation (when defined) If the coefficient of variation is not defined, 0 is returned.
9	number of observations (the counts)

Row	Statistic
10	lower confidence limit for the mean (assuming normality) The default is a 95-percent confidence interval.
11	upper confidence limit for the mean (assuming normality)
12	lower confidence limit for the variance (assuming normality) The default is a 95-percent confidence interval.
13	upper confidence limit for the variance (assuming normality))

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_simple_statistics (int n_observations,
        int n_variables, float x[],
        IMSLS_CONFIDENCE_MEANS, float confidence_means,
        IMSLS_CONFIDENCE_VARIANCES, float confidence_variances,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_STAT_COL_DIM, int stat_col_dim,
        IMSLS_MEDIAN, or
        IMSLS_MEDIAN_AND_SCALE,
        IMSLS_MISSING_LISTWISE, or
        IMSLS_MISSING_ELEMENTWISE,
        IMSLS_FREQUENCIES, float frequencies[],
        IMSLS_WEIGHTS, float weights[],
        IMSLS_RETURN_USER, float simple_statistics[],
        0)
```

Optional Arguments

`IMSLS_CONFIDENCE_MEANS`, *float* `confidence_means` (Input)
: 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 `IMSLS_CONFIDENCE_MEANS` is not specified, a 95-percent confidence interval is computed.

`IMSLS_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 `IMSLS_CONFIDENCE_VARIANCES` is not specified, a 95-percent confidence interval is computed.

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of array `x`.
Default: `x_col_dim` = `n_variables`

`IMSLS_STAT_COL_DIM`, *int* `stat_col_dim` (Input)
Column dimension of the returned value array, or if `IMSLS_RETURN_USER` is specified, the column dimension of array `simple_statistics`.
Default: `stat_col_dim` = `n_variables`

`IMSLS_MEDIAN`, *or*
`IMSLS_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 `IMSLS_MEDIAN` is specified, the medians are computed and stored in one additional row (row number 14) in the returned matrix of simple statistics. If `IMSLS_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.

`IMSLS_MISSING_LISTWISE`, *or*
`IMSLS_MISSING_ELEMENTWISE`
If `IMSLS_MISSING_ELEMENTWISE` is specified, all non missing data for any variable is used in computing the statistics for that variable. If `IMSLS_MISSING_LISTWISE` is specified and if an observation (row of `x`) contains a missing value, the observation is excluded from computations for all variables. The default is `IMSLS_MISSING_LISTWISE`. In either case, if weights and/or frequencies are specified and the value of the weight and/or frequency is missing, the observation is excluded from computations for all variables.

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of length `n_observations` containing the frequency for each observation.
Default: Each observation has a frequency of 1

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_observations` containing the weight for each observation.
Default: Each observation has a weight of 1

`IMSLS_RETURN_USER`, *float* `simple_statistics[]` (Output)
User-supplied array containing the matrix of statistics. If neither `IMSLS_MEDIAN` nor `IMSLS_MEDIAN_AND_SCALE` is specified, the matrix is 14 × `n_variables`. If `IMSLS_MEDIAN` is specified, the matrix is 15 × `n_variables`. If `IMSLS_MEDIAN_AND_SCALE` is specified, the matrix is 17 × `n_variables`.

Description

For the data in each column of `x`, `imsls_f_simple_statistics` computes the sample mean, variance, minimum, maximum, and other basic statistics. This function also computes confidence intervals for the mean and variance (under the hypothesis that the sample is from a normal population).

Frequencies are interpreted as multiple occurrences of the other values in the observations. In other words, a row of `x` with a frequency variable having a value of 2 has the same effect as two rows with frequencies of 1. The total of the frequencies is used in computing all the statistics based on moments (mean, variance, skewness, and kurtosis). Weights are not viewed as replication factors. The sum of the weights is used only in computing the mean (the weighted mean is used in computing the central moments). Both weights and frequencies can be 0, but neither can be negative. In general, a 0 frequency means that the row is to be eliminated from the analysis; no further processing or error checking is done on the row. A weight of 0 results in the row being counted, and updates are made of the statistics.

The definitions of some of the statistics are given below in terms of a single variable x of which the i-th datum is x_i.

Mean

$$\bar{x}_w = \frac{\sum f_i w_i x_i}{\sum f_i w_i}$$

Variance

$$s_w^2 = \frac{\sum f_i w_i (x_i - \bar{x}_w)^2}{n-1}$$

Skewness

$$\frac{\sum f_i w_i (x_i - \bar{x}_w)^3 / n}{\left[\sum f_i w_i (x_i - \bar{x}_w)^2 / n\right]^{3/2}}$$

Excess or Kurtosis

$$\frac{\sum f_i w_i (x_i - \bar{x}_w)^4 / n}{\left[\sum f_i w_i (x_i - \bar{x}_w)^2 / n\right]^{2}} - 3$$

Minimum

$$x_{\min} = \min(x_i)$$

Maximum

$$x_{\max} = \max(x_i)$$

Range

$$x_{\max} - x_{\min}$$

Coefficient of Variation

$$\frac{s_w}{\bar{x}_w} \qquad \text{for } \bar{x}_w \neq 0$$

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

$$\text{MAD} = \text{median } \{|x_i - \text{median } \{x_j\}|\}$$

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

Data from Draper and Smith (1981) are used in this example, which includes 5 variables and 13 observations.

```
#include <imsls.h>

#define N_VARIABLES              5
#define N_OBSERVATIONS          13

main()
{
    float       *simple_statistics;
    float       x[] = {
         7., 26.,  6., 60.,  78.5,
         1., 29., 15., 52.,  74.3,
        11., 56.,  8., 20., 104.3,
        11., 31.,  8., 47.,  87.6,
         7., 52.,  6., 33.,  95.9,
        11., 55.,  9., 22., 109.2,
         3., 71., 17.,  6., 102.7,
         1., 31., 22., 44.,  72.5,
         2., 54., 18., 22.,  93.1,
        21., 47.,  4., 26., 115.9,
         1., 40., 23., 34.,  83.8,
        11., 66.,  9., 12., 113.3,
        10., 68.,  8., 12., 109.4};
    char        *row_labels[] = {
        "means", "variances", "std. dev", "skewness", "kurtosis",
        "minima", "maxima", "ranges", "C.V.", "counts", "lower mean",
        "upper mean", "lower var", "upper var"};
    simple_statistics = imsls_f_simple_statistics(N_OBSERVATIONS,
        N_VARIABLES, x, 0);

    imsls_f_write_matrix("* * * Statistics * * *\n", 14, N_VARIABLES,
        simple_statistics,
        IMSLS_ROW_LABELS,  row_labels,
        IMSLS_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
```

normal_one_sample

Computes statistics for mean and variance inferences using a sample from a normal population.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_one_sample` (*int* `n_observations`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_normal_one_sample`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `x[]` (Input)
Array of length `n_observations`.

Return Value

The mean of the sample.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float imsls_f_normal_one_sample (int n_observations, float x[],
        IMSLS_CONFIDENCE_MEAN, float confidence_mean,
        IMSLS_CI_MEAN, float *lower_limit, float *upper_limit,
        IMSLS_STD_DEV, float *std_dev,
        IMSLS_T_TEST, int *df, float *t, float *p_value,
        IMSLS_T_TEST_NULL, float mean_hypothesis_value,
        IMSLS_CONFIDENCE_VARIANCE, float confidence_variance,
        IMSLS_CI_VARIANCE, float *lower_limit,
                float *upper_limit,
        IMSLS_CHI_SQUARED_TEST, int *df, float *chi_squared,
                float *p_value,
        IMSLS_CHI_SQUARED_TEST_NULL,
                float variance_hypothesis_value,
        0)
```

Optional Arguments

`IMSLS_CONFIDENCE_MEAN`, *float* `confidence_mean` (Input)
Confidence level (in percent) for two-sided interval estimate of the mean. Argument `confidence_mean` 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 (at least 50 percent), set `confidence_mean` $= 100.0 - 2.0 \times (100.0 - c)$. If `IMSLS_CONFIDENCE_MEAN` is not specified, a 95-percent confidence interval is computed.

`IMSLS_CI_MEAN`, *float* `*lower_limit`, *float* `*upper_limit` (Output)
Argument `lower_limit` contains the lower confidence limit for the mean, and argument `upper_limit` contains the upper confidence limit for the mean.

`IMSLS_STD_DEV`, *float* `*std_dev` (Output)
Standard deviation of the sample.

`IMSLS_T_TEST`, *int* `*df`, *float* `*t`, *float* `*p_value` (Output)
Argument `df` is the degrees of freedom associated with the t test for the mean, `t` is the test statistic, and `p_value` is the probability of a larger t in absolute value. The t test is a test, against a two-sided alternative, of the hypothesis $\mu = \mu_0$, where μ_0 is the null hypothesis value as described in `IMSLS_T_TEST_NULL`.

`IMSLS_T_TEST_NULL`, *float* `mean_hypothesis_value` (Input)
Null hypothesis value for t test for the mean.
Default: `mean_hypothesis_value` = 0.0

`IMSLS_CONFIDENCE_VARIANCE`, *float* `confidence_variance` (Input)

Confidence level (in percent) for two-sided interval estimate of the variances. Argument `confidence_variance` must be between 0.0 and 100.0 and is often 90.0, 95.0, 99.0. For a one-sided confidence interval with confidence level *c* (at least 50 percent), set `confidence_variance` = 100.0 – 2.0 × (100.0 – *c*). If this option is not used, a 95-percent confidence interval is computed.

`IMSLS_CI_VARIANCE`, *float* `*lower_limit`, *float* `*upper_limit` (Output)

Contains the lower and upper confidence limits for the variance.

`IMSLS_CHI_SQUARED_TEST`, *int* `*df`, *float* `*chi_squared`, *float* `*p_value` (Output)

Argument `df` is the degrees of freedom associated with the chi-squared test for variances, `chi_squared` is the test statistic, and `p_value` is the probability of a larger chi-squared. The chi-squared test is a test of the hypothesis $\sigma^2 = \sigma_0^2$ where σ_0^2 is the null hypothesis value as described in `IMSLS_CHI_SQUARED_TEST_NULL`.

`IMSLS_CHI_SQUARED_TEST_NULL`, *float* `variance_hypothesis_value` (Input)

Null hypothesis value for the chi-squared test.

Default: `variance_hypothesis_value` = 1.0

Description

Statistics for mean and variance inferences using a sample from a normal population are computed, including confidence intervals and tests for both mean and variance. The definitions of mean and variance are given below. The summation in each case is over the set of valid observations, based on the presence of missing values in the data.

Mean, return value

$$\bar{x} = \frac{\sum x_i}{n}$$

Standard deviation, std_dev

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

The *t* statistic for the two-sided test concerning the population mean is given by

$$t = \frac{\bar{x} - \mu_0}{s / \sqrt{n}}$$

where *s* and $\bar{x}$ are given above. This quantity has a *T* distribution with *n* – 1 degrees of freedom.

The chi-squared statistic for the two-sided test concerning the population variance is given by

$$\chi^2 = \frac{(n-1)s^2}{\sigma_0^2}$$

where s is given above. This quantity has a χ^2 distribution with $n - 1$ degrees of freedom.

Examples

Example 1

This example uses data from Devore (1982, p. 335), which is based on data published in the *Journal of Materials*. There are 15 observations; the mean is the only output.

```
#include <imsls.h>

main()
{
#define N_OBSERVATIONS 15

    float  mean;
    float x[N_OBSERVATIONS] = {
        26.7, 25.8, 24.0, 24.9, 26.4,
        25.9, 24.4, 21.7, 24.1, 25.9,
        27.3, 26.9, 27.3, 24.8, 23.6};

                    /* Perform analysis */
    mean = imsls_f_normal_one_sample(N_OBSERVATIONS, x, 0);

                    /* Print results */
    printf("Sample Mean = %5.2f", mean);
}
```

Output

```
Sample Mean = 25.3
```

Example 2

This example uses the same data as the initial example. The hypothesis H_0: $\mu = 20.0$ is tested. The extremely large t value and the correspondingly small p-value provide strong evidence to reject the null hypothesis.

```
#include <imsls.h>

main()
{
#define N_OBSERVATIONS 15

    int     df;
    float  mean, s, lower_limit, upper_limit, t, p_value;
    static float x[N_OBSERVATIONS] = {
        26.7, 25.8, 24.0, 24.9, 26.4,
        25.9, 24.4, 21.7, 24.1, 25.9,
        27.3, 26.9, 27.3, 24.8, 23.6};

                     /* Perform analysis +*/
    mean = imsls_f_normal_one_sample(N_OBSERVATIONS, x,
        IMSLS_STD_DEV, &s,
        IMSLS_CI_MEAN, &lower_limit, &upper_limit,
        IMSLS_T_TEST_NULL, 20.0,
        IMSLS_T_TEST, &df, &t, &p_value,
        0);

                     /* Print results */
    printf("Sample Mean                 = %5.2f\n", mean);
    printf("Sample Standard Deviation = %5.2f\n", s);
    printf("95%% CI for the mean is (%5.2f,%5.2f)\n", lower_limit,
        upper_limit);
    printf("df = %3d\n", df);
    printf("t = %5.2f\n", t);
    printf("p-value = %8.5f\n", p_value);
}
```

Output

```
Sample Mean               = 25.31
Sample Standard Deviation =  1.58
95% CI for the mean is (24.44,26.19)
df =  14
t = 13.03
p-value =  0.00000
```

normal_two_sample

Computes statistics for mean and variance inferences using samples from two normal populations.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_two_sample` (*int* `n1_observations`, *float* `x1[]`,
int `n2_observations`, *float* `x2[]`, ..., 0)

The type *double* function is `imsls_d_normal_two_sample`.

Required Arguments

int `n1_observations` (Input)
Number of observations in the first sample, `x1`.

float `x1[]` (Input)
Array of length `n1_observations` containing the first sample.

int `n2_observations` (Input)
Number of observations in the second sample, `x2`.

float `x2[]` (Input)
: Array of length `n2_observations` containing the second sample.

Return Value

Difference in means, `x1_mean` – `x2_mean`.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float imsls_f_normal_two_sample (int n1_observations, float x1[],
        int n2_observations, float x2[],
        IMSLS_MEANS, float *x1_mean, float *x2_mean,
        IMSLS_CONFIDENCE_MEAN, float confidence_mean,
        IMSLS_CI_DIFF_FOR_EQUAL_VARS, float *lower_limit,
                float *upper_limit,
        IMSLS_CI_DIFF_FOR_UNEQUAL_VARS, float *lower_limit,
                float *upper_limit
        IMSLS_T_TEST_FOR_EQUAL_VARS, int *df, float *t,
                float *p_value,
        IMSLS_T_TEST_FOR_UNEQUAL_VARS, float *df, float *t,
                float *p_value,
        IMSLS_T_TEST_NULL, float mean_hypothesis_value,
        IMSLS_POOLED_VARIANCE, float *pooled_variance,
        IMSLS_CONFIDENCE_VARIANCE, float confidence_variance,
        IMSLS_CI_COMMON_VARIANCE, float *lower_limit,
                float *upper_limit,
        IMSLS_CHI_SQUARED_TEST, int *df, float *chi_squared,
                float *p_value,
        IMSLS_CHI_SQUARED_TEST_NULL,
                float variance_hypothesis_value,
        IMSLS_STD_DEVS, float *x1_std_dev, float *x2_std_dev,
        IMSLS_CI_RATIO_VARIANCES, float *lower_limit,
                float *upper_limit,
        IMSLS_F_TEST, int *df_numerator, int *df_denominator,
                float *F, float *p_value,
        0)
```

Optional Arguments

`IMSLS_MEANS`, *float* `*x1_mean`, *float* `*x2_mean` (Output)
: Means of the first and second samples.

`IMSLS_CONFIDENCE_MEAN`, *float* `confidence_mean` (Input)
: Confidence level for two-sided interval estimate of the mean of `x1` minus the mean of `x2`, in percent. Argument `confidence_mean` 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* (at least 50 percent),

set `confidence_mean` = 100.0 – 2.0 × (100.0 – *c*).
Default: `confidence_mean` = 95.0

`IMSLS_CI_DIFF_FOR_EQUAL_VARS`, *float* `*lower_limit`, *float* `*upper_limit` (Output)
Argument `lower_limit` contains the lower confidence limit, and `upper_limit` contains the upper limit for the mean of the first population minus the mean of the second, assuming equal variances.

`IMSLS_CI_DIFF_FOR_UNEQUAL_VARS`, *float* `*lower_limit`, *float* `*upper_limit` (Output)
Argument `lower_limit` contains the approximate lower confidence limit, and `upper_limit` contains the approximate upper limit for the mean of the first population minus the mean of the second, assuming unequal variances.

`IMSLS_T_TEST_FOR_EQUAL_VARS`, *int* `*df`, *float* `*t`, *float* `*p_value` (Output)
A *t* test for $\mu_1 - \mu_2 = c$, where *c* is the null hypothesis value. (See the description of `IMSLS_T_TEST_NULL`.) Argument `df` contains the degrees of freedom, argument `t` contains the *t* value, and argument `p_value` contains the probability of a larger *t* in absolute value, assuming equal means. This test assumes equal variances.

`IMSLS_T_TEST_FOR_UNEQUAL_VARS`, *float* `*df`, *float* `*t`, *float* `*p_value` (Output)
A *t* test for $\mu_1 - \mu_2 = c$, where *c* is the null hypothesis value. (See the description of `IMSLS_T_TEST_NULL`.) Argument `df` contains the degrees of freedom for Satterthwaite's approximation, argument `t` contains the *t* value, and argument `p_value` contains the approximate probability of a larger *t* in absolute value, assuming equal means. This test does not assume unequal variances.

`IMSLS_T_TEST_NULL`, *float* `mean_hypothesis_value` (Input)
Null hypothesis value for the *t* test.
Default: `mean_hypothesis_value` = 0.0

`IMSLS_POOLED_VARIANCE`, *float* `*pooled_variance` (Output)
Pooled variance for the two samples.

`IMSLS_CONFIDENCE_VARIANCE`, *float* `confidence_variance` (Input)
Confidence level for inference on variances. Under the assumption of equal variances, the pooled variance is used to obtain a two-sided `confidence_variance` percent confidence interval for the common variance if `IMSLS_CI_COMMON_VARIANCE` is specified. Without making the assumption of equal variances, the ratio of the variances is of interest. A two-sided `confidence_variance` percent confidence

interval for the ratio of the variance of the first sample to that of the second sample is computed and is returned if `IMSLS_CI_RATIO_VARIANCES` is specified. The confidence intervals are symmetric in probability.
Default: `confidence_variance` = 95.0

`IMSLS_CI_COMMON_VARIANCE`, *float* `*lower_limit`, *float* `*upper_limit` (Output)
Argument `lower_limit` contains the lower confidence limit, and `upper_limit` contains the upper limit for the common, or pooled, variance.

`IMSLS_CHI_SQUARED_TEST`, *int* `*df`, *float* `*chi_squared`, *float* `*p_value` (Output)
The chi-squared test for $\sigma^2 = \sigma_0^2$ where σ^2 is the common, or pooled, variance, and σ_0^2 is the null hypothesis value. (See description of `IMSLS_CHI_SQUARED_TEST_NULL`.) Argument `df` contains the degrees of freedom, argument `chi_squared` contains the chi-squared value, and argument `p_value` contains the probability of a larger chi-squared in absolute value, assuming equal means.

`IMSLS_CHI_SQUARED_TEST_NULL`, *float* `variance_hypothesis_value` (Input)
Null hypothesis value for the chi-squared test.
Default: `variance_hypothesis_value` = 1.0

`IMSLS_STD_DEVS`, *float* `*x1_std_dev`, *float* `*x2_std_dev` (Output)
Standard deviations of the first and second samples.

`IMSLS_CI_RATIO_VARIANCES`, *float* `*lower_limit`, *float* `*upper_limit` (Output)
Argument `lower_limit` contains the approximate lower confidence limit, and `upper_limit` contains the approximate upper limit for the ratio of the variance of the first population to the second.

`IMSLS_F_TEST`, *int* `*df_numerator`, *int* `*df_denominator`, *float* `*F`, *float* `*p_value` (Output)
The *F* test for equality of variances. Argument `df_numerator` and `df_denominator` contain the numerator degrees of freedom, argument `F` contains the *F* test value, and argument `p_value` contains the probability of a larger *F* in absolute value, assuming equal variances.

Description

Function `imsls_f_normal_two_sample` computes statistics for making inferences about the means and variances of two normal populations, using independent samples in `x1` and `x2`. For inferences concerning parameters of a single normal population, see function `imsls_normal_one_sample` on page 7.

Let μ_1 and σ_1^2 be the mean and variance of the first population, and let μ_2 and σ_2^2 be the corresponding quantities of the second population. The function contains test confidence intervals for difference in means, equality of variances, and the pooled variance.

The means and variances for the two samples are as follows:

$$\bar{x}_1 = (\sum x_{1i} / n_1), \qquad \bar{x}_2 = (\sum x_{2i}) / n_2$$

and

$$s_1^2 = \sum (x_{1i} - \bar{x}_1)^2 / (n_1 - 1), \qquad s_2^2 = \sum (x_{2i} - \bar{x}_2)^2 / (n_2 - 1)$$

Inferences about the Means

The test that the difference in means equals a certain value, for example, μ_0, depends on whether or not the variances of the two populations can be considered equal. If the variances are equal and `mean_hypothesis_value` equals 0, the test is the two-sample *t* test, which is equivalent to an analysis-of-variance test. The pooled variance for the difference-in-means test is as follows:

$$s^2 = \frac{(n_1 - 1)s_1 + (n_2 - 1)s_2}{n_1 + n_2 - 2}$$

The *t* statistic is as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2 - \mu_0}{s\sqrt{(1/n_1) + (1/n_2)}}$$

Also, the confidence interval for the difference in means can be obtained by specifying `IMSLS_CI_DIFF_FOR_EQUAL_VARS`.

If the population variances are not equal, the ordinary *t* statistic does not have a *t* distribution and several approximate tests for the equality of means have been proposed. (See, for example, Anderson and Bancroft 1952, and Kendall and Stuart 1979.) One of the earliest tests devised for this situation is the Fisher-Behrens test, based on Fisher's concept of fiducial probability. A procedure used if `IMSLS_T_TEST_FOR_UNEQUAL_VARS` and/or `IMSLS_CI_DIFF_FOR_UNEQUAL_VARS` are specified is the Satterthwaite's procedure, as suggested by H.F. Smith and modified by F.E. Satterthwaite (Anderson and Bancroft 1952, p. 83).

The test statistic is

$$t' = (\bar{x}_1 - \bar{x}_2 - \mu_0) / s_d$$

where

$$s_d = \sqrt{\left(s_1^2 / n_1\right) + \left(s_2^2 / n_2\right)}$$

Under the null hypothesis of $\mu_1 - \mu_2 = c$, this quantity has an approximate t distribution with degrees of freedom df (in IMSLS_T_TEST_FOR_UNEQUAL_VARS), given by the following equation:

$$\mathrm{df} = \frac{s_d^4}{\frac{\left(s_1^2 / n_1\right)^2}{n_1 - 1} + \frac{\left(s_2^2 / n_2\right)^2}{n_2 - 1}}$$

Inferences about Variances

The F statistic for testing the equality of variances is given by $F = s_{\max}^2 / s_{\min}^2$, where $s_{\max}^2$ is the larger of s_1^2 and s_2^2. If the variances are equal, this quantity has an F distribution with $n_1 - 1$ and $n_2 - 1$ degrees of freedom.

It is generally not recommended that the results of the F test be used to decide whether to use the regular t test or the modified t' on a single set of data. The modified t' (Satterthwaite's procedure) is the more conservative approach to use if there is doubt about the equality of the variances.

Examples

Example 1

This example, taken from Conover and Iman (1983, p. 294), involves scores on arithmetic tests of two grade-school classes. The question is whether a group taught by an experimental method has a higher mean score. Only the difference in means is output. The data are shown below.

Scores for Standard Group	Scores for Experimental Group
72	111
75	118
77	128
80	138
104	140
110	150
125	163
	164
	169

```
#include <imsls.h>

main()
{
#define N1_OBSERVATIONS 7
#define N2_OBSERVATIONS 9

    float  diff_means;
    float x1[N1_OBSERVATIONS] = {
        72.0, 75.0, 77.0, 80.0, 104.0, 110.0, 125.0};
    float x2[N2_OBSERVATIONS] = {
        111.0, 118.0, 128.0, 138.0, 140.0, 150.0, 163.0,
        164.0, 169.0};

                     /* Perform analysis */
    diff_means = imsls_f_normal_two_sample(N1_OBSERVATIONS, x1,
        N2_OBSERVATIONS, x2, 0);

                     /* Print results */
    printf("\nx1_mean - x2_mean = %5.2f\n", diff_means);
}
```

Output

```
x1_mean - x2_mean = -50.48
```

Example 2

The same data is used for this example as for the initial example. Here, the results of the *t* test are output. The variances of the two populations are assumed to be equal. It is seen from the output that there is strong reason to believe that the two means are different (*t* value of –4.804). Since the lower 97.5-percent confidence limit does not include 0, the null hypothesis is that $\mu_1 \le \mu_2$ would be rejected at the 0.05 significance level. (The closeness of the values of the sample variances provides some qualitative substantiation of the assumption of equal variances.)

```
#include <imsls.h>

main()
{
#define N1_OBSERVATIONS 7
#define N2_OBSERVATIONS 9

    int    df;
    float  diff_means, lower_limit, upper_limit, t, p_value, sp2;
    float x1[N1_OBSERVATIONS] = {
        72.0, 75.0, 77.0, 80.0, 104.0, 110.0, 125.0};
    float x2[N2_OBSERVATIONS] = {
        111.0, 118.0, 128.0, 138.0, 140.0, 150.0, 163.0,
        164.0, 169.0};

                     /* Perform analysis */
    diff_means = imsls_f_normal_two_sample(N1_OBSERVATIONS, x1,
        N2_OBSERVATIONS, x2,
        IMSLS_POOLED_VARIANCE, &sp2,
        IMSLS_CI_DIFF_FOR_EQUAL_VARS, &lower_limit, &upper_limit,
        IMSLS_T_TEST_FOR_EQUAL_VARS, &df, &t, &p_value,
        0);
```

```
                    /* Print results */
    printf("\nx1_mean - x2_mean = %5.2f\n", diff_means);
    printf("Pooled variance = %5.2f\n", sp2);
    printf("95%% CI for x1_mean - x2_mean is (%5.2f,%5.2f)\n",
        lower_limit, upper_limit);
    printf("df = %3d\n", df);
    printf("t = %5.2f\n", t);
    printf("p-value = %8.5f\n", p_value);
}
```

Output

```
x1_mean - x2_mean = -50.48
Pooled variance = 434.63
95% CI for x1_mean - x2_mean is (-73.01,-27.94)
df =  14
t = -4.80
p-value =  0.00028
```

table_oneway

Tallies observations into a one-way frequency table.

Synopsis

#include `<imsls.h>`

float `*imsls_f_table_oneway` (*int* `n_observations`, *float* `x[]`,
int `n_intervals`, ..., 0)

The type *double* function is `imsls_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 `<imsls.h>`

float `*imsls_f_table_oneway` (*int* `n_observations`, *float* `x[]`,
int `n_intervals`,

`IMSLS_DATA_BOUNDS`, *float* `*minimum`, *float* `*maximum`, *or*
`IMSLS_KNOWN_BOUNDS`, *float* `lower_bound`, *float* `upper_bound`,
or
`IMSLS_CUTPOINTS`, *float* `cutpoints[]`, *or*
`IMSLS_CLASS_MARKS`, *float* `class_marks[]`,
`IMSLS_RETURN_USER`, *float* `table[]`,
0)

Optional Arguments

`IMSLS_DATA_BOUNDS`, *float* `*minimum`, *float* `*maximum` (Output)
If none is specified or if `IMSLS_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 `IMSLS_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 length (`maximum` – `minimum`)/`n_intervals`.

or

`IMSLS_KNOWN_BOUNDS`, *float* `lower_bound`, *float* `upper_bound` (Input)
If `IMSLS_KNOWN_BOUNDS` is specified, two semi-infinite intervals are used as the initial and last intervals. 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

$$\frac{\texttt{upper\_bound-lower\_bound}}{\texttt{n\_intervals}-2}$$

and are open on the left and closed on the right. Argument `n_intervals` must be greater than or equal to 3 for this option.

or

`IMSLS_CUTPOINTS`, *float* `cutpoints[]` (Input)
If `IMSLS_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. Argument `n_interval` must be greater than or equal to 3 for this option.

or

`IMSLS_CLASS_MARKS`, *float* `class_marks[]` (Input)
If `IMSLS_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`. Each interval is assumed to have length `class_marks` [`1`] – `class_marks` [`0`]. Argument `n_intervals` must be greater than or equal to 2 for this option.

None or exactly one of the four optional arguments described above can be specified in order to define the intervals or bins for the one-way table.

`IMSLS_RETURN_USER`, *float* `table[]` (Output)
Counts are stored in the array `table` of length `n_intervals`, which is provided by the user.

Examples

Example 1

The data for this example is from Hinkley (1977) and Velleman and Hoaglin (1981). The measurements (in inches) are for precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
#include <imsls.h>
main()
{
    int     n_intervals=10;
    int     n_observations=30;
    float   *table;
    float   x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
                  2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
                  0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
                  1.89, 0.90, 2.05};
    table = imsls_f_table_oneway (n_observations, x, n_intervals, 0);
    imsls_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

In this example, `IMSLS_KNOWN_BOUNDS` is used, and `lower_bound` = 0.5 and `upper_bound` = 4.5 are set 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, 4.5], and (4.5, ∞].

```
#include <imsls.h>
main()
{
    int     n_observations=30;
    int     n_intervals=10;
    float   *table;
    float   lower_bound=0.5, upper_bound=4.5;
    float   x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
                   2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
                   0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
                   1.89, 0.90, 2.05};
    table = imsls_f_table_oneway (n_observations, x, n_intervals,
                                 IMSLS_KNOWN_BOUNDS, lower_bound,
                                 upper_bound,
                                 0);
    imsls_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

In this example, 10 class marks, 0.25, 0.75, 1.25, ..., 4.75, are input. 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 <imsls.h>
main()
{
    int        n_intervals=10;
    int        n_observations=30;
    double     *table;
    double     x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47,
                      1.43, 3.37, 2.20, 3.00, 3.09, 1.51, 2.10,
                      0.52, 1.62, 1.31, 0.32, 0.59, 0.81, 2.81,
                      1.87, 1.18, 1.35, 4.75, 2.48, 0.96,1.89,
                      0.90, 2.05};
    double     class_marks[] = {0.25, 0.75, 1.25, 1.75, 2.25,
                                 2.75, 3.25,3.75, 4.25, 4.75};
    table = imsls_d_table_oneway (n_observations, x, n_intervals,
                                 IMSLS_CLASS_MARKS, class_marks,
                                 0);
    imsls_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

In this example, cutpoints, 0.5, 1.0, 1.5, 2.0, ..., 4.5, are input to define the same 10 intervals as in Example 2. Here again, the initial and last intervals are semi-infinite intervals.

```
#include <imsls.h>
main()
{
    int        n_intervals=10;
    int        n_observations=30;
    double     *table;
    double     x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47,
                      1.43, 3.37, 2.20, 3.00, 3.09, 1.51, 2.10,
                      0.52, 1.62, 1.31, 0.32, 0.59, 0.81, 2.81,
                      1.87, 1.18, 1.35, 4.75, 2.48, 0.96, 1.89,
                      0.90, 2.05};
    double     cutpoints[] = {0.5, 1.0, 1.5, 2.0, 2.5,
                              3.0, 3.5, 4.0, 4.5};
    table = imsls_d_table_oneway (n_observations, x, n_intervals,
                                  IMSLS_CUTPOINTS, cutpoints,
                                  0);
    imsls_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
```

table_twoway

Tallies observations into two-way frequency table.

Synopsis

#include `<imsls.h>`

float `*imsls_f_table_twoway` (*int* `n_observations`, *float* `x[]`,
float `y[]`, *int* `nx`, *int* `ny`, ..., 0)

The type *double* function is `imsls_d_table_twoway`.

Required Arguments

int `n_observations` (Input)
: Number of observations.

float `x[]` (Input)
: Array of length `n_observations` containing the data for the first variable.

float `y[]` (Input)
: Array of length `n_observations` containing the data for the second variable.

int `nx` (Input)
: Number of intervals (bins) for variable `x`.

int `nx` (Input)
: Number of intervals (bins) for variable `y`.

Return Value

Pointer to an array of size `nx` by `ny` containing the counts.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_table_twoway` (*int* `n_observations`, *float* `x[]`,
    float `y[]`, *int* `nx`, *int* `ny`,
    `IMSLS_DATA_BOUNDS`, *float* `*xmin`, *float* `*xmax`, *float* `*ymin`,
        float `*ymax`, *or*
    `IMSLS_KNOWN_BOUNDS`, *float* `xlo`, *float* `xhi`, *float* `ylo`,
        float `yhi`, *or*
    `IMSLS_CUTPOINTS`, *float* `cx[]`, *float* `cy[]`, *or*
    `IMSLS_CLASS_MARKS`, *float* `cx[]`, *float* `cy[]`,
    `IMSLS_RETURN_USER`, *float* `table[]`,
    0)

Optional Arguments

`IMSLS_DATA_BOUNDS`, *float* `*xlo`, *float* `*xhi`, *float* `*ylo`, *float* `*yhi` (Output)
: If none is specified or if `IMSLS_DATA_BOUNDS` is specified, `n_intervals` intervals of equal length are used. Let `xmin` and `xmax` be the minimum and maximum values in `x`, respectively, with similar meanings for `ymin` and `ymax`. Then, `table[0]` is the tally of observations with the `x` value less than or equal to `xmin` + (`xmax` − `xmin`)/`nx`, and the `y` value less than or equal to

`ymin` + (`ymax` − `ymin`)/`ny`. When `IMSLS_DATA_BOUNDS` is explicitly specified, the minimum and maximum values in `x` and `y` are output in `xmin`, `xmax`, `ymin`, and `ymax`.

or

`IMSLS_KNOWN_BOUNDS`, *float* `xlo`, *float* `xhi`, *float* `ylo`, *float* `yhi` (Input)
Intervals of equal lengths are used just as in the case of `IMSLS_DATA_BOUNDS`, except the upper and lower bounds are taken as the user supplied variables `xlo`, `xhi`, `ylo`, and `yhi`, instead of the actual minima and maxima in the data. Therefore, the first and last intervals for both variables are semi-infinite in length. Arguments `nx` and `ny` must be greater than or equal to 3.

or

`IMSLS_CUTPOINTS`, *float* `cx[]`, *float* `cy[]` (Input)
If `IMSLS_CUTPOINTS` is specified, cutpoints (boundaries) must be provided in the arrays `cx` and `cy`, of length (`nx-1`) and (`ny-1`) respectively. The tally in `table[0]` is the number of observations for which the `x` value is less than or equal to `cx[0]`, and the `y` value is less than or equal to `cy[0]`. This option allows unequal interval lengths. Arguments `nx` and `ny` must be greater than or equal to 2.

or

`IMSLS_CLASS_MARKS`, *float* `cx[]`, *float* `cy[]` (Input)
If `IMSLS_CLASS_MARKS` is specified, *equally spaced* class marks in ascending order must be provided in the arrays `cx` and `cy`. The class marks are the midpoints of each interval. Each interval is taken to have length `cx[1]` − `cx[0]` in the `x` direction and `cy[1]` − `cy[0]` in the `y` direction. The total number of elements in `table` may be less than `n_observations`. Arguments `nx` and `ny` must be greater than or equal to 2.

None or exactly one of the four optional arguments described above can be specified in order to define the intervals or bins for the one-way table.

`IMSLS_RETURN_USER`, *float* `table[]` (Output)
Counts are stored in the array table of size `nx` by `ny`, which is provided by the user.

Examples

Example 1

The data for `x` in this example are the same as those used in the examples for `table_oneway`. The data for `y` were created by adding small integers to the data in `x`. This example uses the default tally method, `IMSLS_DATA_BOUNDS`, which may be appropriate when the range of the data is unknown.

```
#include <imsls.h>
main()
{
    int     nx = 5;
    int     ny = 6;
    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};
    float   y[] = {1.77, 3.74, 3.81, 2.20, 3.95, 4.20, 1.47, 3.43, 6.37,
                   3.20, 5.00, 6.09, 2.51, 4.10, 3.52, 2.62, 3.31, 3.32,
                   1.59, 2.81, 5.81, 2.87, 3.18, 4.35, 5.75, 4.48, 3.96,
                   2.89, 2.90, 5.05};
    table = imsls_f_table_twoway (n_observations, x, y, nx, ny, 0);
    imsls_f_write_matrix("counts", nx, ny, table,
        IMSLS_ROW_NUMBER_ZERO, IMSLS_COL_NUMBER_ZERO, 0);
  }
```

Output

```
                                   counts
           0            1            2            3            4            5
0          4            2            4            2            0            0
1          0            4            3            2            1            0
2          0            0            1            2            0            1
3          0            0            0            0            1            2
4          0            0            0            0            0            1
```

Example 2

In this example, `xlo`, `xhi`, `ylo`, and `yhi` are chosen so that the intervals will be 0 to 1, 1 to 2, and so on for `x`, and 1 to 2, 2 to 3, and so on for `y`.

```
#include <imsls.h>
main()
{
    int     nx = 5;
    int     ny = 6;
    int     n_observations=30;
    float   *table;
    float   xlo = 1.0;
    float   xhi = 4.0;
    float   ylo = 2.0;
    float   yhi = 6.0;
    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};
    float   y[] = {1.77, 3.74, 3.81, 2.20, 3.95, 4.20, 1.47, 3.43, 6.37,
                   3.20, 5.00, 6.09, 2.51, 4.10, 3.52, 2.62, 3.31, 3.32,
                   1.59, 2.81, 5.81, 2.87, 3.18, 4.35, 5.75, 4.48, 3.96,
                   2.89, 2.90, 5.05};
    table = imsls_f_table_twoway (n_observations, x, y, nx, ny,
        IMSLS_KNOWN_BOUNDS, xlo, xhi, ylo, yhi, 0);
    imsls_f_write_matrix("counts", nx, ny, table,
        IMSLS_ROW_NUMBER_ZERO, IMSLS_COL_NUMBER_ZERO, 0);
  }
```

Output

```
                              counts
            0           1           2           3           4           5
0           3           2           4           0           0           0
1           0           5           5           2           0           0
2           0           0           1           3           2           0
3           0           0           0           0           0           2
4           0           0           0           0           1           0
```

Example 3

In this example, the class boundaries are input in `cx` and `cy`. The same intervals are chosen as in Example 2, where the first element of `cx` and `cy` specify the first cutpoint *between* classes.

```
#include <imsls.h>
main()
{
    int     nx = 5;
    int     ny = 6;
    int     n_observations=30;
    float   *table;
    float   cmx[] = {0.5, 1.5, 2.5, 3.5, 4.5};
    float   cmy[] = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5};
    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};
    float   y[] = {1.77, 3.74, 3.81, 2.20, 3.95, 4.20, 1.47, 3.43, 6.37,
                  3.20, 5.00, 6.09, 2.51, 4.10, 3.52, 2.62, 3.31, 3.32,
                  1.59, 2.81, 5.81, 2.87, 3.18, 4.35, 5.75, 4.48, 3.96,
                  2.89, 2.90, 5.05};
    table = imsls_f_table_twoway (n_observations, x, y, nx, ny,
        IMSLS_CLASS_MARKS, cmx, cmy, 0);
    imsls_f_write_matrix("counts", nx, ny, table,
        IMSLS_ROW_NUMBER_ZERO, IMSLS_COL_NUMBER_ZERO, 0);
  }
```

Output

```
                              counts
            0           1           2           3           4           5
0           3           2           4           0           0           0
1           0           5           5           2           0           0
2           0           0           1           3           2           0
3           0           0           0           0           0           2
4           0           0           0           0           1           0
```

Example 4

This example, uses the `IMSLS_CUTPOINTS` tally option with cutpoints such that the intervals are specified as in the previous examples.

```
#include <imsls.h>
main()
{
    int     nx = 5;
    int     ny = 6;
    int     n_observations=30;
    float   *table;
    float   cpx[] = {1, 2, 3, 4};
    float   cpy[] = {2, 3, 4, 5, 6};
    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};
    float   y[] = {1.77, 3.74, 3.81, 2.20, 3.95, 4.20, 1.47, 3.43, 6.37,
                   3.20, 5.00, 6.09, 2.51, 4.10, 3.52, 2.62, 3.31, 3.32,
                   1.59, 2.81, 5.81, 2.87, 3.18, 4.35, 5.75, 4.48, 3.96,
                   2.89, 2.90, 5.05};
    table = imsls_f_table_twoway (n_observations, x, y, nx, ny,
        IMSLS_CUTPOINTS, cpx, cpy, 0);
    imsls_f_write_matrix("counts", nx, ny, table,
        IMSLS_ROW_NUMBER_ZERO, IMSLS_COL_NUMBER_ZERO, 0);
  }
```

Output

```
                                counts
            0           1           2           3           4           5
0           3           2           4           0           0           0
1           0           5           5           2           0           0
2           0           0           1           3           2           0
3           0           0           0           0           0           2
4           0           0           0           0           1           0
```

sort_data

Sorts observations by specified keys, with option to tally cases into a multi-way frequency table.

Synopsis

#include `<imsls.h>`

void `imsls_f_sort_data` (*int* `n_observations`, *int* `n_variables`, *float* `x[]`, *int* `n_keys`, ..., 0)

The type *double* function is `imsls_d_sort_data`.

Required Arguments

int `n_observations` (Input)
Number of observations (rows) in `x`.

int `n_variables` (Input)
Number of variables (columns) in `x`.

float `x[]` (Input/Output)

An `n_observations` × `n_variables` matrix containing the observations to be sorted. The sorted matrix is returned in `x` (exception: see optional argument `IMSLS_PASSIVE`).

int `n_keys` (Input)

Number of columns of `x` on which to sort. The first `n_keys` columns of `x` are used as the sorting keys (exception: see optional argument `IMSLS_INDICES_KEYS`).

Synopsis with Optional Arguments

```
#include <imsls.h>

void imsls_f_sort_data (int n_observations, int n_variables,
        float x[], int n_keys,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_INDICES_KEYS, int indices_keys[],
        IMSLS_FREQUENCIES, float frequencies[],
        IMSLS_ASCENDING, or
        IMSLS_DESCENDING,
        IMSLS_ACTIVE, or
        IMSLS_PASSIVE,
        IMSLS_PERMUTATION, int **permutation,
        IMSLS_PERMUTATION_USER, int permutation[],
        IMSLS_TABLE, int **n_values, float **values, float **table,
        IMSLS_TABLE_USER, int n_values[], float values[],
               float table[],

        IMSLS_LIST_CELLS, int *n_cells, float **list_cells,
               float **table_unbalanced,
        IMSLS_LIST_CELLS_USER, int *n_cells, float list_cells[],
               float table_unbalanced[],
        IMSLS_N, int *n_cells, int **n,
        IMSLS_N_USER, int *n_cells, int n[],
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)

Column dimension of `x`.
Default: `x_col_dim = n_variables`

`IMSLS_INDICES_KEYS`, *int* `indices_keys[]` (Input)

Array of length `n_keys` giving the column numbers of `x` which are to be used in the sort.
Default: `indices_keys` [ ] = 0, 1, …, `n_keys` – 1

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
: Array of length `n_observations` containing the frequency for each observation in `x`.
Default: `frequencies` [ ] = 1

`IMSLS_ASCENDING`, *or*

`IMSLS_DESCENDING`
: By default, or if `IMSLS_ASCENDING` is specified, the sort is in ascending order. If `IMSLS_DESCENDING` is specified, the sort is in descending order.

`IMSLS_ACTIVE`, *or*

`IMSLS_PASSIVE`
: By default, or if `IMSLS_ACTIVE` is specified, the sorted matrix is returned in `x`. If `IMSLS_PASSIVE` is specified, `x` is unchanged by `imsls_f_sort_data` (i.e., `x` becomes input only).

`IMSLS_PERMUTATION`, *int* `**permutation` (Output)
: Address of a pointer to an internally allocated array of length `n_observations` specifying the rearrangement (permutation) of the observations (rows).

`IMSLS_PERMUTATION_USER`, *int* `permutation[]` (Output)
: Storage for array `permutation` is provided by the user. See `IMSLS_PERMUTATION`.

`IMSLS_TABLE`, *int* `**n_values`, *float* `**values`, *float* `**table` (Output)
: Argument `n_values` is the address of a pointer to an internally allocated array of length `n_keys` containing in its *i*-th element ($i = 0, 1, \ldots,$ `n_keys` – 1), the number of levels or categories of the *i*-th classification variable (column).

Argument `values` is the address of a pointer to an internally allocated array of length
`n_values` [`0`] + `n_values` [`1`] + … + `n_values` [`n_keys` – `1`]
containing the values of the classification variables. The first `n_values` [`0`] elements of `values` contain the values for the first classification variable. The next `n_values` [`1`] contain the values for the second variable. The last `n_values` [`n_keys` – `1`] positions contain the values for the last classification variable.

Argument `table` is the address of a pointer to an internally allocated array of length `n_values` [`0`] × `n_values` [`1`] × … × `n_values` [`n_keys` – `1`] containing the frequencies in the cells of the table to be fit.

Empty cells are included in `table`, and each element of `table` is nonnegative. The cells of `table` are sequenced so that the first variable cycles through its `n_values` [`0`] categories one time, the second variable cycles through its `n_values` [`1`] categories `n_values` [`0`] times, the third variable cycles through its `n_values` [`2`] categories `n_values` [`0`] × `n_values` [`1`] times, etc., up to the `n_keys`-th variable, which cycles through its `n_values` [`n_keys` – 1] categories `n_values` [`0`] × `n_values` [`1`] × … × `n_values` [`n_keys` – 2] times.

`IMSLS_TABLE_USER`, *int* `n_values[]`, *float* `values[]`, *float* `table[]` (Output)
Storage for arrays `n_values`, `values`, and `table` is provided by the user. If the length of `table` is not known in advance, the upper bound for this length can be taken to be the product of the number of distinct values taken by all of the classification variables (since `table` includes the empty cells).

`IMSLS_LIST_CELLS`, *int* `*n_cells`, *float* `**list_cells`, *float* `**table_unbalanced` (Output)
Number of nonempty cells is returned by `n_cells`. Argument `list_cells` is an internally allocated array of size `n_cells` × `n_keys` containing, for each row, a list of the levels of `n_keys` corresponding classification variables that describe a cell.

Argument `table_unbalanced` is the address of a pointer to an array of length `n_cells` containing the frequency for each cell.

`IMSLS_LIST_CELLS_USER`, *int* `*n_cells`, *float* `list_cells[]`, *float* `table_unbalanced[]` (Output)
Storage for arrays `list_cells` and `table_unbalanced` is provided by the user. See `IMSLS_LIST_CELLS`.

`IMSLS_N`, *int* `*n_cells`, *int* `**n` (Output)
The integer `n_cells` returns the number of groups of different observations. A group contains observations (rows) in `x` that are equal with respect to the method of comparison.

Argument `n` is the address of the pointer to an internally allocated array of length `n_cells` containing the number of observations (rows) in each group.

The first `n` [0] rows of the sorted `x` are group number 1. The next `n` [1]rows of the sorted `x` are group number 2, etc. The last `n` [`n_cells` – 1] rows of the sorted `x` are group number `n_cells`.

`IMSLS_N_USER`, *int* `*n_cells`, *int* `n[]` (Output)
Storage for array `n_cells` is provided by the user. If the value of

n_cells is not known, n_observations can be used as an upper bound for the length of n. See IMSLS_N.

Description

Function imsls_f_sort_data can perform both a key sort and/or tabulation of frequencies into a multi-way frequency table.

Sorting

Function imsls_f_sort_data sorts the rows of real matrix x using a particular row in x as the keys. The sort is algebraic with the first key as the most significant, the second key as the next most significant, etc. When x is sorted in ascending order, the resulting sorted array is such that the following is true:

- For $i = 0, 1, \ldots,$ n_observations $- 2$,
  x $[i]$ [indices_keys $[0]] \le$ x $[i + 1]$ [indices_keys $[0]]$

- For $k = 1, \ldots,$ n_keys $- 1$, if
  x $[i]$ [indices_keys $[j]]$ = x $[i + 1]$ [indices_keys $[j]]$ for
  $j = 0, 1, \ldots, k - 1$, then
  x $[i]$ [indices_keys $[k]]$ = x $[i + 1]$ [indices_keys $[k]]$

The observations also can be sorted in descending order.

The rows of x containing the missing value code NaN in at least one of the specified columns are considered as an additional group. These rows are moved to the end of the sorted x.

The sorting algorithm is based on a quicksort method given by Singleton (1969) with modifications by Griffen and Redish (1970) and Petro (1970).

Frequency Tabulation

Function imsls_f_sort_data determines the distinct values in multivariate data and computes frequencies for the data. This function accepts the data in the matrix x, but performs computations only for the variables (columns) in the first n_keys columns of x (Exception: see optional argument IMSLS_INDICES_KEYS). In general, the variables for which frequencies should be computed are discrete; they should take on a relatively small number of different values. Variables that are continuous can be grouped first. The imsls_f_table_oneway function can be used to group variables and determine the frequencies of groups.

When IMSLS_TABLE is specified, imsls_f_sort_data fills the vector values with the unique values of the variables and tallies the number of unique values of each variable in the vector table. Each combination of one value from each variable forms a cell in a multi-way table. The frequencies of these cells are entered in table so that the first variable cycles through its values exactly once, and the last variable cycles through its values most rapidly. Some cells cannot correspond to any observations in the data; in other words, "missing cells" are included in table and have a value of 0.

When IMSLS_LIST_CELLS is specified, the frequency of each cell is entered in table_unbalanced so that the first variable cycles through its values exactly once and the last variable cycles through its values most rapidly. All cells have a frequency of at least 1, i.e., there is no "missing cell." The array list_cells can be considered "parallel" to table_unbalanced because row *i* of list_cells is the set of n_keys values that describes the cell for which row *i* of table_unbalanced contains the corresponding frequency.

Examples

Example 1

The rows of a 10 × 3 matrix x are sorted in ascending order using Columns 0 and 1 as the keys. There are two missing values (NaNs) in the keys. The observations containing these values are moved to the end of the sorted array.

```
#include <imsls.h>
#define N_OBSERVATIONS 10
#define N_VARIABLES    3
main()
{
    int     n_keys=2;
    float   x[N_OBSERVATIONS][N_VARIABLES] = {1.0, 1.0, 1.0,
                                              2.0, 1.0, 2.0,
                                              1.0, 1.0, 3.0,
                                              1.0, 1.0, 4.0,
                                              2.0, 2.0, 5.0,
                                              1.0, 2.0, 6.0,
                                              1.0, 2.0, 7.0,
                                              1.0, 1.0, 8.0,
                                              2.0, 2.0, 9.0,
                                              1.0, 1.0, 9.0};
    x[4][1]=imsls_f_machine(6);
    x[6][0]=imsls_f_machine(6);
    imsls_f_sort_data (N_OBSERVATIONS, N_VARIABLES, x, n_keys, 0);
    imsls_f_write_matrix("sorted x", N_OBSERVATIONS, N_VARIABLES,
                      (float *)x, 0);
  }
```

Output

```
              sorted x
            1           2           3
 1          1           1           1
 2          1           1           9
 3          1           1           3
 4          1           1           4
 5          1           1           8
 6          1           2           6
 7           2           1           2
 8           2           2           9
 9   .........           2           7
10           2   .........           5
```

Example 2

This example uses the same data as the previous example. The permutation of the rows is output in the array `permutation`.

```
#include <imsls.h>
#define N_OBSERVATIONS 10
#define N_VARIABLES 3
MAIN()
{
    int     n_keys=2;
    int     n_cells;
    int     *n;
    int     *permutation;
    float   x[N_OBSERVATIONS][N_VARIABLES]={1.0, 1.0, 1.0,
                                           2.0, 1.0, 2.0,
                                           1.0, 1.0, 3.0,
                                           1.0, 1.0, 4.0,
                                           2.0, 2.0, 5.0,
                                           1.0, 2.0, 6.0,
                                           1.0, 2.0, 7.0,
                                           1.0, 1.0, 8.0,
                                           2.0. 2.0, 9.0,
                                           1.0, 1.0, 9.0};
    x[4][1]=imsls_f_machine(6);
    x[6][0]=imsls_f_machine(6);
    imsls_f_sort_data (N_OBSERVATIONS, N_VARIABLES,
                      (float *)x, n_keys,
                      IMSLS_PASSIVE,
                      IMSLS_PERMUTATION, &permutation,
                      IMSLS_N, &n_cells, &n, 0};
    imsls_f_write_matrix("unchanged x ", N_OBSERVATIONS, N_VARIABLES,
                        (float *)x, 0);
    imsls_i_write_matrix("permutation", 1, N_OBSERVATIONS, permutation,
                        0);
    imsls_i_write_matrix("n", 1, n_cells, n, 0);
  }
```

Output

```
              unchanged x
             1           2           3
 1           1           1           1
 2           2           1           2
 3           1           1           3
 4           1           1           4
 5           2  ..........           5
 6           1           2           6
 7  ..........           2           7
 8           1           1           8
 9           2           2           9
10           1           1           9

              permutation
 1   2   3   4   5   6   7   8   9  10
 0   9   2   3   7   5   1   8   6   4

       n
 1   2   3   4
 5   1   1   1
```

Example 3

The table of frequencies for a data matrix of size 30 × 2 is output in the array `table`.

```
#include <imsls.h>
main()
{
    int     n_observations=30;
    int     n_variables=2;
    int     n_keys=2;
    int     *n_values;
    int     n_rows, n_columns;
    float   *values;
    float   *table;
    float   x[] = {0.5, 1.5,
                   1.5, 3.5,
                   0.5, 3.5,
                   1.5, 2.5,
                   1.5, 3.5,
                   1.5, 4.5,
                   0.5, 1.5,
                   1.5, 3.5,
                   3.5, 6.5,
                   2.5, 3.5,
                   2.5, 4.5,
                   3.5, 6.5,
                   1.5, 2.5,
                   2.5, 4.5,
                   0.5, 3.5,
                   1.5, 2.5,
                   1.5, 3.5,
                   0.5, 3.5,
                   0.5, 1.5,
                   0.5, 2.5,
                   2.5, 5.5,
                   1.5, 2.5,
                   1.5, 3.5,
                   1.5, 4.5,
                   4.5, 5.5,
                   2.5, 4.5,
                   0.5, 3.5,
                   1.5, 2.5,
                   0.5, 2.5,
                   2.5, 5.5};

   imsls_f_sort_data (n_observations, n_variables, x, n_keys,
                      IMSLS_PASSIVE,
                      IMSLS_TABLE, &n_values, &values, &table,
                      0);
   imsls_f_write_matrix("unchanged x", n_observations, n_variables,
                    x, 0);
  n_rows = n_values[0];
  n_columns = n_values[1];
  imsls_f_write_matrix("row values", 1, n_rows, values, 0);
  imsls_f_write_matrix("column values", 1, n_columns, &values[n_rows],
                   0);
  imsls_f_write_matrix("table", n_rows, n_columns, table, 0);
}
```

Output

```
       unchanged x
            1           2
 1        0.5         1.5
 2        1.5         3.5
 3        0.5         3.5
 4        1.5         2.5
 5        1.5         3.5
 6        1.5         4.5
 7        0.5         1.5
 8        1.5         3.5
 9        3.5         6.5
10        2.5         3.5
11        2.5         4.5
12        3.5         6.5
13        1.5         2.5
14        2.5         4.5
15        0.5         3.5
16        1.5         2.5
17        1.5         3.5
18        0.5         3.5
19        0.5         1.5
20        0.5         2.5
21        2.5         5.5
22        1.5         2.5
23        1.5         3.5
24        1.5         4.5
25        4.5         5.5
26        2.5         4.5
27        0.5         3.5
28        1.5         2.5
29        0.5         2.5
30        2.5         5.5

                         row values
        1           2           3           4           5
      0.5         1.5         2.5         3.5         4.5

                              column values
        1           2           3           4           5           6
      1.5         2.5         3.5         4.5         5.5         6.5

                                   table
            1           2           3           4           5           6
1           3           2           4           0           0           0
2           0           5           5           2           0           0
3           0           0           1           3           2           0
4           0           0           0           0           0           2
5           0           0           0           0           1           0
```

ranks

Computes the ranks, normal scores, or exponential scores for a vector of observations.

Synopsis

#include `<imsls.h>`

float `*imsls_f_ranks` (*int* `n_observations`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_ranks`.

Required Arguments

int `n_observations` (Input)
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** `imsls_f_ranks` (*int* `n_observations`, *float* `x[]`,
`IMSLS_AVERAGE_TIE`, *or*
`IMSLS_HIGHEST`, *or*
`IMSLS_LOWEST`, *or*
`IMSLS_RANDOM_SPLIT`,
`IMSLS_FUZZ`, *float* `fuzz_value`,
`IMSLS_RANKS`, *or*
`IMSLS_BLOM_SCORES`, *or*
`IMSLS_TUKEY_SCORES`, *or*
`IMSLS_VAN_DER_WAERDEN_SCORES`, *or*
`IMSLS_EXPECTED_NORMAL_SCORES`, *or*
`IMSLS_SAVAGE_SCORES`,
`IMSLS_RETURN_USER`, *float* `ranks[]`,
0)

Optional Arguments

`IMSLS_AVERAGE_TIE`, *or*

`IMSLS_HIGHEST`, *or*

`IMSLS_LOWEST`, *or*

`IMSLS_RANDOM_SPLIT`
Exactly one of these optional arguments can be used to change the method used to assign a score to tied observations.

Argument	**Method**
`IMSLS_AVERAGE_TIE`	average of the scores of the tied observations (default)
`IMSLS_HIGHEST`	highest score in the group of ties
`IMSLS_LOWEST`	lowest score in the group of ties
`IMSLS_RANDOM_SPLIT`	tied observations are randomly split using a random number generator

`IMSLS_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.
Default: `fuzz_value` = 0.0

`IMSLS_RANKS`, *or*

`IMSLS_BLOM_SCORES`, *or*

`IMSLS_TUKEY_SCORES`, *or*

`IMSLS_VAN_DER_WAERDEN_SCORES`, *or*

`IMSLS_EXPECTED_NORMAL_SCORES`, *or*

`IMSLS_SAVAGE_SCORES`
Exactly one of these optional arguments can be used to specify the type of values returned.

Argument	**Result**
`IMSLS_RANKS`	ranks (default)
`IMSLS_BLOM_SCORES`	Blom version of normal scores
`IMSLS_TUKEY_SCORES`	Tukey version of normal scores
`IMSLS_VAN_DER_WAERDEN_SCORES`	Van der Waerden version of normal scores
`IMSLS_EXPECTED_NORMAL_SCORES`	expected value of normal order statistics (for tied observations, the average of the expected normal scores)
`IMSLS_SAVAGE_SCORES`	Savage scores (the expected value of exponential order statistics)

`IMSLS_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, the output value depends on the option used to break ties.

Argument	Result
`IMSLS_AVERAGE_TIE`	`ranks[i]` = `ranks[j]` = 1.5
`IMSLS_HIGHEST`	`ranks[i]` = `ranks[j]` = 2.0
`IMSLS_LOWEST`	`ranks[i]` = `ranks[j]` = 1.0
`IMSLS_RANDOM_SPLIT`	`ranks[i]` = 1.0 and `ranks[j]` = 2.0
	or, randomly,
	`ranks[i]` = 2.0 and `ranks[j]` = 1.0

When the ties are resolved randomly, function `imsls_f_random_uniform` (Chapter 12) 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 `imsls_f_random_seed_set` (Chapter 12).

Scores

As an option, normal and other functions of the ranks can 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, function `imsls_f_normal_inverse_cdf` (Chapter 11), 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 `IMSLS_AVERAGE_TIE` 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 as follows:

$$\Phi^{-1}\left(\frac{r_i - 1/3}{n + 1/3}\right)$$

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 as follows:

$$\Phi^{-1}\left(\frac{r_i}{n+1}\right)$$

Ties are handled in the same way as for the Blom normal scores.

When option `IMSLS_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, 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 also Lehmann 1975). If the value in `x[i]` is the k-th smallest, 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 as follows:

$$\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1}$$

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 <imsls.h>

#define N_OBSERVATIONS          30

main()
{
    float       *ranks;
    float       x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                       3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                       1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                       4.75, 2.48, 0.96, 1.89, 0.90, 2.05};

    ranks = imsls_f_ranks(N_OBSERVATIONS, x, 0);
    imsls_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 the score options with the same data set, which contains some ties. Ties are handled in several different ways in this example.

```
#include <imsls.h>

#define N_OBSERVATIONS          30

void main()
{
    float       fuzz_value=0.0, score[4][N_OBSERVATIONS], *ranks;
    float       x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
                       3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
                       1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
                       4.75, 2.48, 0.96, 1.89, 0.90, 2.05};
    char        *row_labels[] = {"Blom", "Tukey", "Van der Waerden",
                                  "Expected Value"};

                                /* Blom scores using largest ranks */
                                /* for ties */
    imsls_f_ranks(N_OBSERVATIONS, x,
                  IMSLS_HIGHEST,
                  IMSLS_BLOM_SCORES,
                  IMSLS_RETURN_USER,   &score[0][0],
                0);
                               /* Tukey normal scores using smallest */
                               /* ranks for ties */
    imsls_f_ranks(N_OBSERVATIONS, x,
                IMSLS_LOWEST,
                IMSLS_TUKEY_SCORES,
                IMSLS_RETURN_USER,  &score[1][0],
                0);
                               /* Van der Waerden scores using */
                               /* randomly resolved ties */
    imsls_random_seed_set(123457);
    imsls_f_ranks(N_OBSERVATIONS, x,
                IMSLS_RANDOM_SPLIT,
                IMSLS_VAN_DER_WAERDEN_SCORES,
                IMSLS_RETURN_USER, &score[2][0],
                0);
                               /* Expected value of normal order */
                               /* statistics using averaging to */
                               /* break ties */
    imsls_f_ranks(N_OBSERVATIONS, x,
                IMSLS_EXPECTED_NORMAL_SCORES,
                IMSLS_RETURN_USER, &score[3][0],
                0);
    imsls_f_write_matrix("Normal Order Statistics", 4, N_OBSERVATIONS,
                 (float *)score,
                IMSLS_ROW_LABELS,   row_labels,
                IMSLS_WRITE_FORMAT, "%9.3f",
                0);
                               /* Savage scores using averaging */
                               /* to break ties */
    ranks = imsls_f_ranks(N_OBSERVATIONS, x,
                IMSLS_SAVAGE_SCORES,
                0);
    imsls_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
```

Chapter 2: Regression

Routines

Usage Notes

The regression models in this chapter include the simple and multiple linear regression models, the multivariate general linear model, the polynomial model, and the nonlinear regression model. Functions for fitting regression models, computing summary statistics from a fitted regression, computing diagnostics, and computing confidence intervals for individual cases are provided. This chapter also provides methods for building a model from a set of candidate variables.

Simple and Multiple Linear Regression

The simple linear regression model is

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \; i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable, the x_i's are the settings of the independent (explanatory) variable, β_0 and β_1 are the intercept and slope parameters (respectively) and the ε_i's are independently distributed normal errors, each with mean 0 and variance σ^2.

The multiple linear regression model is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable; the x_{i1}'s, x_{i2}'s, ..., x_{ik}'s are the settings of the k independent (explanatory) variables; β_0, β_1, ..., β_k are the regression coefficients; and the ε_i's are independently distributed normal errors, each with mean 0 and variance σ^2.

Function `imsls_f_regression` (page 64) fits both the simple and multiple linear regression models using a fast Given's transformation and includes an option for excluding the intercept β_0. The responses are input in array `y`, and the independent variables are input in array `x`, where the individual cases correspond to the rows and the variables correspond to the columns.

After the model has been fitted using `imsls_f_regression`, function `imsls_f_regression_summary` computes summary statistics and `imsls_f_regression_prediction` computes predicted values, confidence intervals, and case statistics for the fitted model. The information about the fit is communicated from `imsls_f_regression` to `imsls_f_regression_summary` (page 77) and `imsls_f_regression_prediction` (page 85) by passing an argument of structure type *Imsls_f_regression*.

No Intercept Model

Several functions provide the option for excluding the intercept from a model. In most practical applications, the intercept should be included in the model. For functions that use the sums of squares and crossproducts matrix as input, the no-intercept case can be handled by using the raw sums of squares and crossproducts matrix as input in place of the corrected sums of squares and crossproducts. The raw sums of squares and crossproducts matrix can be computed as $(x_1, x_2, ..., x_k, y)^T (x_1, x_2, ..., x_k, y)$.

Variable Selection

Variable selection can be performed by `imsls_f_regression_selection` (page 112), which computes all best-subset regressions, or by `imsls_f_regression_stepwise` (page 123), which computes stepwise regression. The method used by `imsls_f_regression_selection` is generally preferred over that used by `imsls_f_regression_stepwise` because `imsls_f_regression_selection` implicitly examines all possible models in the search for a model that optimizes some criterion while stepwise does not examine all possible models. However, the computer time and memory requirements for `imsls_f_regression_selection` can be much greater than that for `imsls_f_regression_stepwise` when the number of candidate variables is large.

Polynomial Model

The polynomial model is

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + ... + \beta_k x_i^k + \varepsilon_i \qquad i = 1, 2, ..., n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable; the x_i's are the settings of the independent (explanatory) variable; $\beta_0, \beta_1, ..., \beta_k$ are the regression coefficients; and the ε_i's are independently distributed normal errors each with mean 0 and variance σ^2.

Function `imsls_f_poly_regression` (page 132) fits a polynomial regression model with the option of determining the degree of the model and also produces summary information. Function `imsls_f_poly_prediction` (page 140) computes predicted values, confidence intervals, and case statistics for the model fit by `imsls_f_poly_regression`.

The information about the fit is communicated from `imsls_f_poly_regression` to `imsls_f_poly_prediction` by passing an argument of structure type *Imsls_f_poly_regression*.

Specification of X for the General Linear Model

Variables used in the general linear model are either continuous or classification variables. Typically, multiple regression models use continuous variables, whereas analysis of variance models use classification variables. Although the notation used to specify analysis of variance models and multiple regression models may look quite different, the models are essentially the same. The term "general linear model" emphasizes that a common notational scheme is used for specifying a model that may contain both continuous and classification variables.

A general linear model is specified by its effects (sources of variation). An effect is referred to in this text as a single variable or a product of variables. (The term "effect" is often used in a narrower sense, referring only to a single regression coefficient.) In particular, an "effect" is composed of one of the following:

1. a single continuous variable
2. a single classification variable
3. several different classification variables
4. several continuous variables, some of which may be the same
5. continuous variables, some of which may be the same, and classification variables, which must be distinct

Effects of the first type are common in multiple regression models. Effects of the second type appear as main effects in analysis of variance models. Effects of the third type appear as interactions in analysis of variance models. Effects of the fourth type appear in polynomial models and response surface models as powers and crossproducts of some basic variables. Effects of the fifth type appear in one-way analysis of covariance models as regression coefficients that indicate lack of parallelism of a regression function across the groups.

The analysis of a general linear model occurs in two stages. The first stage calls function `imsls_f_regressors_for_glm` to specify all regressors except the intercept. The second stage calls `imsls_f_regression`, at which point the model will be specified as either having (default) or not having an intercept.

For this discussion, define a variable `INTCEP` as follows:

Option	INTCEP	Action
`IMSLS_NO_INTERCEPT`	0	An intercept is not in the model.
`IMSLS_INTERCEPT` (default)	1	An intercept is in the model.

The remaining variables (`n_continuous`, `n_class`, `x_class_columns`, `n_effects`, `n_var_effects`, and `indices_effects`) are defined for function `imsls_f_regressors_for_glm`. All these variables have defaults except for `n_continuous` and `n_class`, both of which must be specified.
(See the documentation for `imsls_f_regressors_for_glm` on page 56 for a discussion of the defaults.) The meaning of each of these arguments is as follows:

`n_continuous` (Input)
Number of continuous variables.

`n_class` (Input)
Number of classification variables.

`x_class_columns` (Input)
: Index vector of length `n_class` containing the column numbers of `x` that are the classification variables.

`n_effects` (Input)
: Number of effects (sources of variation) in the model, excluding error.

`n_var_effects` (Input)
: Vector of length `n_effects` containing the number of variables associated with each effect in the model.

`indices_effects` (Input)
: Index vector of length `n_var_effects`(0) + `n_var_effects`(1) + ... + `n_var_effects` (`n_effects` – 1). The first `n_var_effects`(0) elements give the column numbers of `x` for each variable in the first effect; the next `n_var_effects`(1) elements give the column numbers for each variable in the second effect; and finally, the last `n_var_effects` (`n_effects` – 1) elements give the column numbers for each variable in the last effect.

Suppose the data matrix has as its first four columns two continuous variables in Columns 0 and 1 and two classification variables in Columns 2 and 3. The data might appear as follows:

Column 0	Column 1	Column 2	Column 3
11.23	1.23	1.0	5.0
12.12	2.34	1.0	4.0
12.34	1.23	1.0	4.0
4.34	2.21	1.0	5.0
5.67	4.31	2.0	4.0
4.12	5.34	2.0	1.0
4.89	9.31	2.0	1.0
9.12	3.71	2.0	1.0

Each distinct value of a classification variable determines a level. The classification variable in Column 2 has two levels. The classification variable in Column 3 has three levels. (Integer values are recommended, but not required, for values of the classification variables. The values of the classification variables corresponding to the same level must be identical.) Some examples of regression functions and their specifications are as follows:

	`INTCEP`	`n_class`	`x_class_columns`
$\beta_0 + \beta_1 x_1$	1	0	
$\beta_0 + \beta_1 x_1 + \beta_2 x_1^2$	1	0	
$\mu + \alpha_i$	1	1	2
$\mu + \alpha_i + \beta_j + \gamma_{ij}$	1	2	2, 3
μ_{ij}	0	2	2, 3
$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$	1	0	
$\mu + \alpha_i + \beta x_{1i} + \beta_i x_{1i}$	1	1	2

	`n_effects`	`n_var_effects`	`Indices_effects`
$\beta_0 + \beta_1 x_1$	1	1	0
$\beta_0 + \beta_1 x_1 + \beta_2 x_1^2$	2	1, 2	0, 0, 0
$\mu + \alpha_i$	1	1	2
$\mu + \alpha_i + \beta_j + \gamma_{ij}$	3	1, 1, 2	2, 3, 2, 3
μ_{ij}	1	2	2, 3
$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$	3	1, 1, 2	0, 1, 0, 1
$\mu + \alpha_i + \beta x_{1i} + \beta_i x_{1i}$	3	1, 1, 2	2, 0, 0, 2

Functions for Fitting the Model

Function `imsls_f_regression` (page 64) fits a multivariate general linear model, where regressors for the general linear model have been generated using function `imsls_f_regressors_for_glm`.

Linear Dependence and the *R* Matrix

Linear dependence of the regressors frequently arises in regression models—sometimes by design and sometimes by accident. The functions in this chapter are designed to handle linear dependence of the regressors; i.e., the

$n \times p$ matrix X (the matrix of regressors) in the general linear model can have rank less than p. Often, the models are referred to as non-full rank models.

As discussed in Searle (1971, Chapter 5), be careful to correctly use the results of the fitted non-full rank regression model for estimation and hypothesis testing. In the non-full rank case, not all linear combinations of the regression coefficients can be estimated. Those linear combinations that can be estimated are called "estimable functions." If the functions are used to attempt to estimate linear combinations that cannot be estimated, error messages are issued. A good general discussion of estimable functions is given by Searle (1971, pp. 180–188).

The check used by functions in this chapter for linear dependence is sequential. The j-th regressor is declared linearly dependent on the preceding $j - 1$ regressors if

$$1 - R^2_{j(1,2,\ldots,j-1)}$$

is less than or equal to `tolerance`. Here,

$$R_{j(1,2,\ldots,j-1)}$$

is the multiple correlation coefficient of the j-th regressor with the first $j - 1$ regressors. When a function declares the j-th regressor to be linearly dependent on the first $j - 1$, the j-th regression coefficient is set to 0. Essentially, this removes the j-th regressor from the model.

The reason a sequential check is used is that practitioners frequently include the preferred variables to remain in the model first. Also, the sequential check is based on many of the computations already performed as this does not degrade the overall efficiency of the functions. There is no perfect test for linear dependence when finite precision arithmetic is used. The optional argument `IMSLS_TOLERANCE` allows the user some control over the check for linear dependence. If a model is full rank, input `tolerance` = 0.0. However, `tolerance` should be input as approximately 100 times the machine epsilon. The machine epsilon is `imsls_f_machine`(4) in single precision and `imsls_d_machine`(4) in double precision. (See functions `imsls_f_machine` and `imsls_d_machine` in Chapter 14.)

Functions performing least squares are based on QR decomposition of X or on a Cholesky factorization R^TR of X^TX. Maindonald (1984, Chapters 1–5) discusses these methods extensively. The R matrix used by the regression function is a $p \times p$ upper-triangular matrix, i.e., all elements below the diagonal are 0. The signs of the diagonal elements of R are used as indicators of linearly dependent regressors and as indicators of parameter restrictions imposed by fitting a restricted model. The rows of R can be partitioned into three classes by the sign of the corresponding diagonal element:

1. A positive diagonal element means the row corresponds to data.

2. A negative diagonal element means the row corresponds to a linearly independent restriction imposed on the regression parameters by $AB = Z$ in a restricted model.

3. A zero diagonal element means a linear dependence of the regressors was declared. The regression coefficients in the corresponding row of $\hat{B}$ are set to 0. This represents an arbitrary restriction that is imposed to obtain a solution for the regression coefficients. The elements of the corresponding row of R also are set to 0.

Nonlinear Regression Model

The nonlinear regression model is

$$y_i = f(x_i;\theta) + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable, the x_i's are the known vectors of values of the independent (explanatory) variables, f is a known function of an unknown regression parameter vector θ, and the ε_i's are independently distributed normal errors each with mean 0 and variance σ^2.

Function `imsls_f_nonlinear_regression` (page 149) performs the least-squares fit to the data for this model.

Weighted Least Squares

Functions throughout the chapter generally allow weights to be assigned to the observations. The vector `weights` is used throughout to specify the weighting for each row of X.

Computations that relate to statistical inference—e.g., t tests, F tests, and confidence intervals—are based on the multiple regression model except that the variance of ε_i is assumed to equal σ^2 times the reciprocal of the corresponding weight.

If a single row of the data matrix corresponds to n_i observations, the vector `frequencies` can be used to specify the frequency for each row of X. Degrees of freedom for error are affected by frequencies but are unaffected by weights.

Summary Statistics

Function `imsls_f_regression_summary` can be used to compute and print statistics related to a regression for each of the q dependent variables fitted by `imsls_f_regression` (page 64). The summary statistics include the model analysis of variance table, sequential sums of squares and F-statistics, coefficient estimates, estimated standard errors, t-statistics, variance inflation factors, and estimated variance-covariance matrix of the estimated regression coefficients. Function `imsls_f_poly_regression` (page 132) includes most of the same functionality for polynomial regressions.

The summary statistics are computed under the model $y = X\beta + \varepsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors with rank $(X) = r$, β is the $p \times 1$ vector of regression coefficients, and ε is the $n \times 1$ vector of errors whose elements are independently normally distributed with mean 0 and variance σ^2/w_i.

Given the results of a weighted least-squares fit of this model (with the w_i's as the weights), most of the computed summary statistics are output in the following variables:

`anova_table`

One-dimensional array usually of length 15. In `imsls_f_regression_stepwise`, `anova_table` is of length 13 because the last two elements of the array cannot be computed from the input. The array contains statistics related to the analysis of variance. The sources of variation examined are the regression, error, and total. The first 10 elements of `anova_table` and the notation frequently used for these is described in the following table (here, `AOV` replaces `anova_table`):

Model Analysis of Variance Table

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	*F*	*p*-value
Regression	DFR = `AOV`[0]	SSR = `AOV`[3]	MSR = `AOV`[6]	`AOV`[8]	`AOV`[9]
Error	DFE = `AOV`[1]	SSE = `AOV`[4]	s^2 = `AOV`[7]		
Total	DFT = `AOV`[2]	SST = `AOV`[5]			

If the model has an intercept (default), the total sum of squares is the sum of squares of the deviations of y_i from its (weighted) mean $\bar{y}$—the so-called *corrected total sum of squares*, denoted by the following:

$$\text{SST} = \sum_{i=1}^{n} w_i \left(y_i - \bar{y} \right)^2$$

If the model does not have an intercept (`IMSLS_NO_INTERCEPT`), the total sum of squares is the sum of squares of y_i—the so-called *uncorrected total sum of squares*, denoted by the following:

$$\text{SST} = \sum_{i=1}^{n} w_i y_i^2$$

The error sum of squares is given as follows:

$$\text{SSE} = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i \right)^2$$

The error degrees of freedom is defined by DFE = $n - r$.

The estimate of σ^2 is given by s^2 = SSE/DFE, which is the error mean square.

The computed F statistic for the null hypothesis, $H_0{:}\beta_1 = \beta_2 = \ldots = \beta_k = 0$, versus the alternative that at least one coefficient is nonzero is given by $F = \text{MSR}/s^2$. The p-value associated with the test is the probability of an F larger than that computed under the assumption of the model and the null hypothesis. A small p-value (less than 0.05) is customarily used to indicate there is sufficient evidence from the data to reject the null hypothesis.

The remaining five elements in `anova_table` frequently are displayed together with the actual analysis of variance table. The quantities R-squared (R^2 = `anova_table`[10]) and adjusted R-squared

$$R_a^2 = \left(\texttt{anova\_table}[11]\right)$$

are expressed as a percentage and are defined as follows:

$$R^2 = 100(\text{SSR/SST}) = 100(1 - \text{SSE/SST})$$

$$R_a^2 = 100\max\left\{0, 1 - \frac{s^2}{\text{SST/DFT}}\right\}$$

The square root of s^2(s = `anova_table`[12]) is frequently referred to as the estimated standard deviation of the model error.

The overall mean of the responses $\bar{y}$ is output in `anova_table`[13].

The coefficient of variation (CV = `anova_table`[14]) is expressed as a percentage and defined by CV = $100s/\bar{y}$.

`coef_t_tests`
Two-dimensional matrix containing the regression coefficient vector $\hat{\beta}$ as one column and associated statistics (estimated standard error, t statistic and p-value) in the remaining columns.

`coef_covariances`
Estimated variance-covariance matrix of the estimated regression coefficients.

Tests for Lack-of-Fit

Tests for lack-of-fit are computed for the polynomial regression by the function `imsls_f_poly_regression` (page 132). The output array `ssq_lof` contains the lack-of-fit F tests for each degree polynomial 1, 2, ..., k, that is fit to the data. These tests are used to indicate the degree of the polynomial required to fit the data well.

Diagnostics for Individual Cases

Diagnostics for individual cases (observations) are computed by two functions in the regression chapter: `imsls_f_regression_prediction` for linear and nonlinear regressions and `imsls_f_poly_prediction` for polynomial regressions.

Statistics computed include predicted values, confidence intervals, and diagnostics for detecting outliers and cases that greatly influence the fitted regression.

The diagnostics are computed under the model $y = X\beta + \varepsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors with rank $(X) = r$, β is the $p \times 1$ vector of regression coefficients, and ε is the $n \times 1$ vector of errors whose elements are independently normally distributed with mean 0 and variance σ^2/w_i.

Given the results of a weighted least-squares fit of this model (with the w_i's as the weights), the following five diagnostics are computed:

1. leverage
2. standardized residual
3. jackknife residual
4. Cook's distance
5. DFFITS

The definition of these terms is given in the discussion that follows:

Let x_i be a column vector containing the elements of the i-th row of X. A case can be unusual either because of x_i or because of the response y_i. The *leverage* h_i is a measure of uniqueness of the x_i. The leverage is defined by

$$h_i = [x_i^T \left(X^T WX\right)^- x_i]w_i$$

where $W = \text{diag}(w_1, w_2, \ldots, w_n)$ and $(X^T WX)^-$ denotes a generalized inverse of $X^T WX$. The average value of the h_i's is r/n. Regression functions declare x_i unusual if $h_i > 2r/n$. Hoaglin and Welsch (1978) call a data point highly influential (i.e., a leverage point) when this occurs.

Let e_i denote the residual

$$y_i - \hat{y}_i$$

for the i-th case. The estimated variance of e_i is $(1 - h_i)s^2/w_i$, where s^2 is the residual mean square from the fitted regression. The i-th *standardized residual* (also called the internally studentized residual) is by definition

$$r_i = e_i \sqrt{\frac{w_i}{s^2\left(1-h_i\right)}}$$

and r_i follows an approximate standard normal distribution in large samples.

The i-th *jackknife residual* or *deleted residual* involves the difference between y_i and its predicted value, based on the data set in which the i-th case is deleted. This difference equals $e_i/(1 - h_i)$. The jackknife residual is obtained by standardizing this difference. The residual mean square for the regression in which the i-th case is deleted is as follows:

$$s_i^2 = \frac{(n-r)s^2 - w_i e_i^2/(1-h_i)}{n-r-1}$$

The jackknife residual is defined as

$$t_i = e_i \sqrt{\frac{w_i}{s_i^2 (1-h_i)}}$$

and t_i follows a t distribution with $n - r - 1$ degrees of freedom.

Cook's distance for the i-th case is a measure of how much an individual case affects the estimated regression coefficients. It is given as follows:

$$D_i = \frac{w_i h_i e_i^2}{rs^2 (1-h_i)^2}$$

Weisberg (1985) states that if D_i exceeds the 50-th percentile of the $F(r, n - r)$ distribution, it should be considered large. (This value is about 1. This statistic does not have an F distribution.)

DFFITS, like Cook's distance, is also a measure of influence. For the i-th case, DFFITS is computed by the formula below.

$$\text{DFFITS}_i = e_i \sqrt{\frac{w_i h_i}{s_i^2 (1-h_i)^2}}$$

Hoaglin and Welsch (1978) suggest that DFFITS greater than

$$2\sqrt{r/n}$$

is large.

Transformations

Transformations of the independent variables are sometimes useful in order to satisfy the regression model. The inclusion of squares and crossproducts of the variables

$$\left(x_1, x_2, x_1^2, x_2^2, x_1 x_2\right)$$

is often needed. Logarithms of the independent variables are used also. (See Draper and Smith 1981, pp. 218–222; Box and Tidwell 1962; Atkinson 1985, pp. 177–180; Cook and Weisberg 1982, pp. 78–86.)

When the responses are described by a nonlinear function of the parameters, a transformation of the model equation often can be selected so that the transformed model is linear in the regression parameters. For example, by taking natural logarithms on both sides of the equation, the exponential model

$$y = e^{\beta_0 + \beta_1 x_1} \varepsilon$$

can be transformed to a model that satisfies the linear regression model provided the ε_i's have a log-normal distribution (Draper and Smith, pp. 222–225).

When the responses are nonnormal and their distribution is known, a transformation of the responses can often be selected so that the transformed responses closely satisfy the regression model, assumptions. The square-root transformation for counts with a Poisson distribution and the arc-sine transformation for binomial proportions are common examples (Snedecor and Cochran 1967, pp. 325–330; Draper and Smith, pp. 237–239).

Alternatives to Least Squares

The method of least squares has desirable characteristics when the errors are normally distributed, e.g., a least-squares solution produces maximum likelihood estimates of the regression parameters. However, when errors are not normally distributed, least squares may yield poor estimators. Function `imsls_f_lnorm_regression` offers three alternatives to least squares methodology, Least Absolute Value , *Lp* Norm , and Least Maximum Value.

The least absolute value (LAV, *L*1) criterion yields the maximum likelihood estimate when the errors follow a Laplace distribution. Option `IMSLS_METHOD_LAV` (page 170) is often used when the errors have a heavy tailed distribution or when a fit is needed that is resistant to outliers.

A more general approach, minimizing the *Lp* norm ($p \leq 1$), is given by option `IMSLS_METHOD_LLP` (page 170). Although the routine requires about 30 times the `CPU` time for the case $p = 1$ than would the use of `IMSLS_METHOD_LAV`, the generality of `IMSLS_METHOD_LLP` allows the user to try several choices for $p \geq 1$ by simply changing the input value of *p* in the calling program. The `CPU` time decreases as *p* gets larger. Generally, choices of *p* between 1 and 2 are of interest. However, the *Lp* norm solution for values of *p* larger than 2 can also be computed.

The minimax (LMV, L_∞, Chebyshev) criterion is used by `IMSLS_METHOD_LMV` (page 170). Its estimates are very sensitive to outliers, however, the minimax estimators are quite efficient if the errors are uniformly distributed.

Missing Values

NaN (Not a Number) is the missing value code used by the regression functions. Use function `imsls_f_machine`(6), Chapter 14 (or function `imsls_d_machine`(6) with double-precision regression functions) to retrieve NaN. Any element of the data matrix that is missing must be set to `imsls_f_machine`(6) (or `imsls_d_machine`(6) for double precision). In fitting regression models, any observation containing NaN for the independent, dependent, weight, or frequency variables is omitted from the computation of the regression parameters.

regressors_for_glm

Generates regressors for a general linear model.

Synopsis

#include `<imsls.h>`

int `imsls_f_regressors_for_glm` (*int* `n_observations`, *float* `x[]`,
    int `n_class`, *int* `n_continuous`, ..., 0)

The type *double* function is `imsls_d_regressors_for_glm`.

Required Arguments

int `n_observations` (Input)
    Number of observations.

float `x[]` (Input)
    An `n_observations` × (`n_class` + `n_continuous`) array containing the data. The columns must be ordered such that the first `n_class` columns contain the class variables and the next `n_continuous` columns contain the continuous variables. (Exception: see optional argument `IMSLS_X_CLASS_COLUMNS`.)

int `n_class` (Input)
    Number of classification variables.

int `n_continuous` (Input)
    Number of continuous variables.

Return Value

An integer (`n_regressors`) indicating the number of regressors generated.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `imsls_f_regressors_for_glm` (*int* `n_observations`, *float* `x[]`,
    int `n_class`, *int* `n_continuous`,
    `IMSLS_X_COL_DIM`, *int* `x_col_dim`,
    `IMSLS_X_CLASS_COLUMNS`, *int* `x_class_columns[]`,
    `IMSLS_MODEL_ORDER`, *int* `model_order`,
    `IMSLS_INDICES_EFFECTS`, *int* `n_effects`,
        int `n_var_effects[]`, *int* `indices_effects[]`,
    `IMSLS_DUMMY`, *Imsls_dummy_method* `dummy_method`,

IMSLS_REGRESSORS, *float* **regressors,
IMSLS_REGRESSORS_USER, *float* regressors[],
IMSLS_REGRESSORS_COL_DIM, *int* regressors_col_dim,
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of x.
Default: x_col_dim = n_class + n_continuous

IMSLS_X_CLASS_COLUMNS, *int* x_class_columns[] (Input)
Index array of length n_class containing the column numbers of x that are the classification variables. The remaining variables are assumed to be continuous.
Default: x_class_columns = 0, 1, ..., n_class – 1

IMSLS_MODEL_ORDER, *int* model_order (Input)
Order of the model. Model order can be specified as 1 or 2. Use optional argument IMSLS_INDICES_EFFECTS to specify more complicated models.
Default: model_order = 1
or

IMSLS_INDICES_EFFECTS, *int* n_effects, *int* n_var_effects[],
int indices_effects[] (Input)
Variable n_effects is the number of effects (sources of variation) in the model. Variable n_var_effects is an array of length n_effects containing the number of variables associated with each effect in the model. Argument indices_effects is an index array of length n_var_effects[0] + n_var_effects[1] + … + n_var_effects (n_effects – 1). The first n_var_effects[0] elements give the column numbers of x for each variable in the first effect. The next n_var_effects[1] elements give the column numbers for each variable in the second effect. … The last n_var_effects [n_effects – 1] elements give the column numbers for each variable in the last effect.

IMSLS_DUMMY, *Imsls_dummy_method* dummy_method (Input)
Dummy variable option. Indicator variables are defined for each class variable as described in the "Description" section.

Dummy variables are then generated from the *n* indicator variables in one of the following three ways:

dummy_method	**Method**
IMSLS_ALL	The *n* indicator variables are the dummy variables (default).

`dummy_method`	**Method**
`IMSLS_LEAVE_OUT_LAST`	The dummies are the first $n - 1$ indicator variables.
`IMSLS_SUM_TO_ZERO`	The $n - 1$ dummies are defined in terms of the indicator variables so that for balanced data, the usual summation restrictions are imposed on the regression coefficients.

`IMSLS_REGRESSORS`, *float* `**regressors` (Output)
Address of a pointer to the internally allocated array of size `n_observations` × `n_regressors` containing the regressor variables generated from `x`.

`IMSLS_REGRESSORS_USER`, *float* `regressors[]` (Output)
Storage for array `regressors` is provided by the user. See `IMSLS_REGRESSORS`.

`IMSLS_REGRESSORS_COL_DIM`, *int* `regressors_col_dim` (Input)
Column dimension of `regressors`.
Default: `regressors_col_dim` = `n_regressors`

Description

Function `imsls_f_regressors_for_glm` generates regressors for a general linear model from a data matrix. The data matrix can contain classification variables as well as continuous variables. Regressors for effects composed solely of continuous variables are generated as powers and crossproducts. Consider a data matrix containing continuous variables as Columns 3 and 4. The effect indices (3, 3) generate a regressor whose *i*-th value is the square of the *i*-th value in Column 3. The effect indices (3, 4) generates a regressor whose *i*-th value is the product of the *i*-th value in Column 3 with the *i*-th value in Column 4.

Regressors for an effect (source of variation) composed of a single classification variable are generated using indicator variables. Let the classification variable A take on values $a_1, a_2, ..., a_n$. From this classification variable, `imsls_f_regressors_for_glm` creates n indicator variables. For $k = 1, 2, ..., n$, we have

$$I_k = \begin{cases} 1 \text{ if } A = a_k \\ 0 \text{ otherwise} \end{cases}$$

For each classification variable, another set of variables is created from the indicator variables. These new variables are called *dummy variables*. Dummy variables are generated from the indicator variables in one of three manners:

1. The dummies are the n indicator variables.
2. The dummies are the first $n - 1$ indicator variables.

3. The $n - 1$ dummies are defined in terms of the indicator variables so that for balanced data, the usual summation restrictions are imposed on the regression coefficients.

In particular, for `dummy_method` = `IMSLS_ALL`, the dummy variables are $A_k = I_k (k = 1, 2, ..., n)$. For `dummy_method` = `IMSLS_LEAVE_OUT_LAST`, the dummy variables are $A_k = I_k (k = 1, 2, ..., n - 1)$. For `dummy_method` = `IMSLS_SUM_TO_ZERO`, the dummy variables are $A_k = I_k - I_n (k = 1, 2, ..., n - 1)$. The regressors generated for an effect composed of a single-classification variable are the associated dummy variables.

Let m_j be the number of dummies generated for the j-th classification variable. Suppose there are two classification variables A and B with dummies

$$A_1, A_2, ..., A_{m_1}$$

and

$$B_1, B_2, ..., B_{m_2}$$

The regressors generated for an effect composed of two classification variables A and B are

$$\begin{aligned} A \otimes B &= \left(A_1, A_2, ..., A_{m_1}\right) \otimes \left(B_1, B_2, ..., B_{m_2}\right) \\ &= (A_1B_1, A_1B_2, ..., A_1B_{m_2}, A_2B_1, A_2B_2, ..., \\ &\quad A_2B_{m_2}, ..., A_{m_1}B_1, A_{m_1}B_2, ..., A_{m_1}B_{m_2}) \end{aligned}$$

More generally, the regressors generated for an effect composed of several classification variables and several continuous variables are given by the Kronecker products of variables, where the order of the variables is specified in `indices_effects`. Consider a data matrix containing classification variables in Columns 0 and 1 and continuous variables in Columns 2 and 3. Label these four columns A, B, X_1, and X_2. The regressors generated by the effect indices (0, 1, 2, 2, 3) are $A \otimes B \otimes X_1X_1X_2$.

Remarks

Let the data matrix $x = (A, B, X_1)$, where A and B are classification variables and X_1 is a continuous variable. The model containing the effects A, B, AB, X_1, AX_1, BX_1, and ABX_1 is specified as follows (use optional keyword `IMSLS_INDICES_EFFECTS`):

`n_class` = 2

`n_continuous` = 1

`n_effects` = 7

`n_var_effects` = (1, 1, 2, 1, 2, 2, 3)

`indices_effects` = (0, 1, 0, 1, 2, 0, 2, 1, 2, 0, 1, 2)

For this model, suppose that variable A has two levels, A_1 and A_2, and that variable B has three levels, B_1, B_2, and B_3. For each `dummy_method` option, the regressors in their order of appearance in `regressors` are given below.

`dummy_method`	**`regressors`**
`IMSLS_ALL`	$A_1, A_2, B_1, B_2, B_3, A_1B_1, A_1B_2, A_1B_3, A_2B_1, A_2B_2, A_2B_3, X_1, A_1X_1, A_2X_1, B_1X_1, B_2X_1, B_3X_1, A_1B_1X_1, A_1B_2X_1, A_1B_3X_1, A_2B_1X_1, A_2B_2X_1, A_2B_3X_1$
`IMSLS_LEAVE_OUT_LAST`	$A_1, B_1, B_2, A_1B_1, A_1B_2, X_1, A_1X_1, B_1X_1, B_2X_1, A_1B_1X_1, A_1B_2X_1$
`IMSLS_SUM_TO_ZERO`	$A_1 - A_2, B_1 - B_3, B_2 - B_3, (A_1 - A_2)(B_1 - B_2), (A_1 - A_2)(B_2 - B_3), X_1, (A_1 - A_2)X_1, (B_1 - B_3)X_1, (B_2 - B_3)X_1, (A_1 - A_2)(B_1 - B_2)X_1, (A_1 - A_2)(B_2 - B_3)X_1$

Within a group of regressors corresponding to an interaction effect, the indicator variables composing the regressors vary most rapidly for the last classification variable, next most rapidly for the next to last classification variable, etc.

By default, `imsls_f_regressors_for_glm` internally generates values for `n_effects`, `n_var_effects`, and `indices_effects`, which correspond to a first order model with NEF = `n_continuous` + `n_class`. The variables then are used to create the regressor variables. The effects are ordered such that the first effect corresponds to the first column of `x`, the second effect corresponds to the second column of `x`, etc. A second order model corresponding to the columns (variables) of `x` is generated if `IMSLS_MODEL_ORDER` with `model_order` = 2 is specified.

There are

$$\text{NEF} = \texttt{n\_class} + 2 * \texttt{n\_continuous} + \binom{\text{NVAR}}{2}$$

effects, where NVAR = `n_continuous` + `n_class`. The first NVAR effects correspond to the columns of `x`, such that the first effect corresponds to the first column of `x`, the second effect corresponds to the second column of `x`, ..., the NVAR-th effect corresponds to the NVAR-th column of `x` (i.e. `x`[NVAR − 1]). The next `n_continuous` effects correspond to squares of the continuous variables. The last

$$\binom{\text{NVAR}}{2}$$

effects correspond to the two-variable interactions.

- Let the data matrix `x` = (A, B, X_1), where A and B are classification variables and X_1 is a continuous variable. The effects generated and order of appearance is

$$A, B, X_1, X_1^2, AB, AX_1, BX_1$$

- Let the data matrix `x` = (A, X_1, X_2), where A is a classification variable and X_1 and X_2 are continuous variables. The effects generated and order of appearance is

$$A, X_1, X_2, X_1^2, X_2^2, AX_1, AX_2, X_1X_2$$

- Let the data matrix `x` = (X_1, A, X_2) (see `IMSLS_CLASS_COLUMNS`), where A is a classification variable and X_1 and X_2 are continuous variables. The effects generated and order of appearance is

$$X_1, A, X_2, X_1^2, X_2^2, X_1A, X_1X_2, AX_2$$

Higher-order and more complicated models can be specified using `IMSLS_INDICES_EFFECTS`.

Examples

Example 1

In the following example, there are two classification variables, A and B, with two and three values, respectively. Regressors for a one-way model (the default model order) are generated using the `IMSLS_ALL` dummy method (the default dummy method). The five regressors generated are A_1, A_2, B_1, B_2, and B_3.

```
#include <imsls.h>
void main() {
    int n_observations = 6;
    int n_class = 2;
    int n_cont  = 0;
    int n_regressors;
    float x[12] = {
        10.0,  5.0,
        20.0, 15.0,
        20.0, 10.0,
        10.0, 10.0,
        10.0, 15.0,
        20.0,  5.0};

   n_regressors = imsls_f_regressors_for_glm (n_observations, x,
       n_class, n_cont, 0);

   printf("Number of regressors = %3d\n", n_regressors);
}
```

Output

```
Number of regressors =   5
```

Example 2

In this example, a two-way analysis of covariance model containing all the interaction terms is fit. First, `imsls_f_regressors_for_glm` is called to produce a matrix of regressors, `regressors`, from the data `x`. Then, `regressors` is used as the input matrix into `imsls_f_regression` to produce the final fit. The regressors, generated using `dummy_method` = `IMSLS_LEAVE_OUT_LAST`, are the model whose mean function is

$$\mu + \alpha_i + \beta_j + \Upsilon_{ij} + \delta x_{ij} + \zeta_i x_{ij} + \eta j x_{ij} + \theta_{ij} x_{ij} \qquad i = 1, 2; j = 1, 2, 3$$

where $\alpha_2 = \beta_3 = \Upsilon_{21} = \Upsilon_{22} = \Upsilon_{23} = \zeta_2 = \eta_3 = \theta_{21} = \theta_{22} = \theta_{23} = 0$.

```
#include <imsls.h>
void main() {
#define N_OBSERVATIONS 18
    int n_class = 2;
    int n_cont  = 1;
    float anova[15], *regressors;
    int n_regressors;
    float x[54] = {
        1.0, 1.0, 1.11,
        1.0, 1.0, 2.22,
        1.0, 1.0, 3.33,
        1.0, 2.0, 1.11,
        1.0, 2.0, 2.22,
        1.0, 2.0, 3.33,
        1.0, 3.0, 1.11,
        1.0, 3.0, 2.22,
        1.0, 3.0, 3.33,
        2.0, 1.0, 1.11,
        2.0, 1.0, 2.22,
        2.0, 1.0, 3.33,
        2.0, 2.0, 1.11,
        2.0, 2.0, 2.22,
        2.0, 2.0, 3.33,
        2.0, 3.0, 1.11,
        2.0, 3.0, 2.22,
        2.0, 3.0, 3.33};
   float y[N_OBSERVATIONS] = {
       1.0, 2.0, 2.0, 4.0, 4.0, 6.0,
       3.0, 3.5, 4.0, 4.5, 5.0, 5.5,
       2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
   int class_col[2] = {0,1};
   int  n_effects = 7;
   int n_var_effects[7] = {1, 1, 2, 1, 2, 2, 3};
   int indices_effects[12] = {0, 1, 0, 1, 2, 0, 2, 1, 2, 0, 1, 2};
   float *coef;
   char      *reg_labels[] = {
        " ", "Alpha1", "Beta1", "Beta2", "Gamma11", "Gamma12",
        "Delta", "Zeta1", "Eta1", "Eta2", "Theta11", "Theta12"};
   char      *labels[] = {
        "degrees of freedom for the model",
        "degrees of freedom for error",
        "total (corrected) degrees of freedom",
        "sum of squares for the model",
        "sum of squares for error",
        "total (corrected) sum of squares",
        "model mean square", "error mean square",
        "F-statistic", "p-value",
        "R-squared (in percent)","adjusted R-squared (in percent)",
        "est. standard deviation of the model error",
        "overall mean of y",
        "coefficient of variation (in percent)"};

   n_regressors = imsls_f_regressors_for_glm (N_OBSERVATIONS, x,
       n_class, n_cont,
       IMSLS_X_CLASS_COLUMNS, class_col,
       IMSLS_DUMMY, IMSLS_LEAVE_OUT_LAST,
       IMSLS_INDICES_EFFECTS, n_effects, n_var_effects, indices_effects,
       IMSLS_REGRESSORS, &regressors,
       0);

   printf("Number of regressors = %3d", n_regressors);

   imsls_f_write_matrix ("regressors", N_OBSERVATIONS, n_regressors,
   regressors,
       IMSLS_COL_LABELS, reg_labels,
       0);

   coef = imsls_f_regression (N_OBSERVATIONS, n_regressors, regressors,
   y,
       IMSLS_ANOVA_TABLE_USER, anova,
       0);

   imsls_f_write_matrix ("* * * Analysis of Variance * * *\n", 15, 1,
        anova,
        IMSLS_ROW_LABELS,   labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);

}
```

Output

```
Number of regressors =  11
                               regressors
        Alpha1       Beta1       Beta2     Gamma11     Gamma12       Delta
 1        1.00        1.00        0.00        1.00        0.00        1.11
 2        1.00        1.00        0.00        1.00        0.00        2.22
 3        1.00        1.00        0.00        1.00        0.00        3.33
 4        1.00        0.00        1.00        0.00        1.00        1.11
 5        1.00        0.00        1.00        0.00        1.00        2.22
 6        1.00        0.00        1.00        0.00        1.00        3.33
 7        1.00        0.00        0.00        0.00        0.00        1.11
 8        1.00        0.00        0.00        0.00        0.00        2.22
 9        1.00        0.00        0.00        0.00        0.00        3.33
10        0.00        1.00        0.00        0.00        0.00        1.11
11        0.00        1.00        0.00        0.00        0.00        2.22
12        0.00        1.00        0.00        0.00        0.00        3.33
13        0.00        0.00        1.00        0.00        0.00        1.11
14        0.00        0.00        1.00        0.00        0.00        2.22
15        0.00        0.00        1.00        0.00        0.00        3.33
16        0.00        0.00        0.00        0.00        0.00        1.11
```

```
17        0.00        0.00        0.00        0.00        0.00        2.22
18        0.00        0.00        0.00        0.00        0.00        3.33

        Zeta1        Eta1        Eta2     Theta11     Theta12
 1        1.11        1.11        0.00        1.11        0.00
 2        2.22        2.22        0.00        2.22        0.00
 3        3.33        3.33        0.00        3.33        0.00
 4        1.11        0.00        1.11        0.00        1.11
 5        2.22        0.00        2.22        0.00        2.22
 6        3.33        0.00        3.33        0.00        3.33
 7        1.11        0.00        0.00        0.00        0.00
 8        2.22        0.00        0.00        0.00        0.00
 9        3.33        0.00        0.00        0.00        0.00
10        0.00        1.11        0.00        0.00        0.00
11        0.00        2.22        0.00        0.00        0.00
12        0.00        3.33        0.00        0.00        0.00
13        0.00        0.00        1.11        0.00        0.00
14        0.00        0.00        2.22        0.00        0.00
15        0.00        0.00        3.33        0.00        0.00
16        0.00        0.00        0.00        0.00        0.00
17        0.00        0.00        0.00        0.00        0.00
18        0.00        0.00        0.00        0.00        0.00


           * * * Analysis of Variance * * *

degrees of freedom for the model               11.0000
degrees of freedom for error                    6.0000
total (corrected) degrees of freedom           17.0000
sum of squares for the model                   43.9028
sum of squares for error                        0.8333
total (corrected) sum of squares               44.7361
model mean square                               3.9912
error mean square                               0.1389
F-statistic                                    28.7364
p-value                                         0.0003
R-squared (in percent)                         98.1372
adjusted R-squared (in percent)                94.7221
est. standard deviation of the model error      0.3727
overall mean of y                               3.9722
coefficient of variation (in percent)           9.3821
```

regression

Fits a multivariate linear regression model using least squares.

Synopsis

#include `<imsls.h>`

float `*imsls_f_regression` (*int* `n_rows`, *int* `n_independent`, *float* `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_regression`.

Required Arguments

int `n_rows` (Input)
Number of rows in `x`.

int `n_independent` (Input)
Number of independent (explanatory) variables.

float `x[]` (Input)
Array of size `n_rows` × `n_independent` containing the independent (explanatory) variables(s). The *i*-th column of *x* contains the *i*-th independent variable.

float `y[]` (Input)
Array of size `n_rows` × `n_dependent` containing the dependent (response) variables(s). The *i*-th column of `y` contains the *i*-th dependent variable. See optional argument `IMSLS_N_DEPENDENT` to set the value of `n_dependent`.

Return Value

If the optional argument `IMSLS_NO_INTERCEPT` is not used, `regression` returns a pointer to an array of length `n_dependent` × (`n_independent` + 1) containing a least-squares solution for the regression coefficients. The estimated intercept is the initial component of each row, where the *i*-th row contains the regression coefficients for the *i*-th dependent variable.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float *imsls_f_regresssion (int n_rows, int n_independent,
        float x[], float y[],
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_Y_COL_DIM, int y_col_dim,
        IMSLS_N_DEPENDENT, int n_dependent,
        IMSLS_X_INDICES, int indind[], int inddep[], int ifrq,
                int iwt,
        IMSLS_IDO, int ido,
        IMSLS_ROWS_ADD, or
        IMSLS_ROWS_DELETE,
        IMSLS_INTERCEPT, or
        IMSLS_NO_INTERCEPT,
        IMSLS_TOLERANCE, float tolerance,
        IMSLS_RANK, int *rank,
        IMSLS_COEF_COVARIANCES, float **coef_covariances,
        IMSLS_COEF_COVARIANCES_USER, float coef_covariances[],
        IMSLS_COV_COL_DIM, int cov_col_dim,
        IMSLS_X_MEAN, float **x_mean,
        IMSLS_X_MEAN_USER, float x_mean[],
        IMSLS_RESIDUAL, float **residual,
```

IMSLS_RESIDUAL_USER, *float* residual[],
IMSLS_ANOVA_TABLE, *float* **anova_table,
IMSLS_ANOVA_TABLE_USER, *float* anova_table[],
IMSLS_FREQUENCIES, *float* frequencies[],
IMSLS_WEIGHTS, *float* weights[],
IMSLS_REGRESSION_INFO,
        Imsls_f_regression **regression_info,
IMSLS_RETURN_USER, *float* coefficients[],
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of x.
Default: x_col_dim = n_independent

IMSLS_Y_COL_DIM, *int* y_col_dim (Input)
Column dimension of y.
Default: y_col_dim = n_dependent

IMSLS_N_DEPENDENT, *int* n_dependent (Input)
Number of dependent variables. Input matrix y must be declared of size n_rows by n_dependent, where column *i* of y contains the *i*-th dependent variable.
Default: n_dependent = 1

IMSLS_X_INDICES, *int* indind[], *int* inddep, *int* ifrq, *int* iwt (Input)
This argument allows an alternative method for data specification. Data (independent, dependent, frequencies, and weights) is all stored in the data matrix x. Argument y, and keywords IMSLS_FREQUENCIES and IMSLS_WEIGHTS are ignored.

Each of the four arguments contains indices indicating column numbers of x in which particular types of data are stored. Columns are numbered 0 … x_col_dim − 1.

Parameter indind contains the indices of the independent variables..

Parameter inddep contains the indices of the dependent variables.

Parameters ifrq and iwt contain the column numbers of x in which the frequencies and weights, respectively, are stored. Set ifrq = −1 if there will be no column for frequencies. Set iwt = −1 if there will be no column for weights. Weights are rounded to the nearest integer. Negative weights are not allowed.

Note that required input argument y is not referenced, and can be declared a vector of length 1.

IMSLS_IDO, *int* ido (Input)
Processing option.

ido	Action
0	This is the only invocation; all the data are input at once. (Default)
1	This is the first invocation with this data; additional calls will be made. Initialization and updating for the `n_rows` observations of `x` will be performed.
2	This is an intermediate invocation; updating for the `n_rows` observations of `x` will be performed.
3	This is the final invocation of this function. Updating for the data in `x` and wrap-up computations are performed. Workspace is released. No further call to `regression` with `ido` greater than 1 should be made without first calling `regression` with `ido` = 1

Default: `ido` = 0

`IMSLS_ROWS_ADD`, *or*
`IMSLS_ROWS_DELETE`
By default (or if `IMSLS_ROWS_ADD` is specified), the observations in x are added to the discriminant statistics. If `IMSLS_ROWS_DELETE` is specified, then the observations are deleted.

If `ido` = 0, these optional arguments are ignored (data is always added if there is only one invocation).

`IMSLS_INTERCEPT`, *or*
`IMSLS_NO_INTERCEPT`
`IMSLS_INTERCEPT` is the default where 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 `IMSLS_NO_INTERCEPT` is specified, the intercept term

$$\left(\hat{\beta}_0\right)$$

is omitted from the model and the return value from regression is a pointer to an array of length `n_dependent` × `n_independent`.

`IMSLS_TOLERANCE`, *float* `tolerance` (Input)
Tolerance used in determining linear dependence. For `regression`, `tolerance` = 100 × `imsls_f_machine`(4) is the default choice. For `imsls_d_regression`, `tolerance` = 100 × `imsls_d_machine`(4) is the default. (See `imsls_f_machine` Chapter 14.)

`IMSLS_RANK`, *int* `*rank` (Output)
Rank of the fitted model is returned in `*rank`.

IMSLS_COEF_COVARIANCES, *float* **coef_covariances (Output)
Address of a pointer to the n_dependent × m × m internally allocated array containing the estimated variances and covariances of the estimated regression coefficients. Here, m is the number of regression coefficients in the model. If IMSLS_NO_INTERCEPT is specified, n = n_independent; otherwise, m = n_independent + 1.

The first $m \times m$ elements contain the matrix for the first dependent variable, the next $m \times m$ elements contain the matrix for the next dependent variable, ... and so on.

IMSLS_COEF_COVARIANCES_USER, *float* coef_covariances[] (Output)
Storage for arrays coef_covariances is provided by the user. See IMSLS_COEF_COVARIANCES.

IMSLS_COV_COL_DIM, *int* cov_col_dim (Input)
Column dimension of array coef_covariances.
Default: cov_col_dim = m, where m is the number of regression coefficients in the model

IMSLS_X_MEAN, *float* **x_mean (Output)
Address of a pointer to the internally allocated array containing the estimated means of the independent variables.

IMSLS_X_MEAN_USER, *float* x_mean[] (Output)
Storage for array x_mean is provided by the user.
See IMSLS_X_MEAN.

IMSLS_RESIDUAL, *float* **residual (Output)
Address of a pointer to the internally allocated array of size n_rows by n_dependent containing the residuals. Residuals may not be requested if ido > 0.

IMSLS_RESIDUAL_USER, *float* residual[] (Output)
Storage for array residual is provided by the user.
See IMSLS_RESIDUAL.

IMSLS_ANOVA_TABLE, *float* **anova_table (Output)
Address of a pointer to the internally allocated array of size 15 × n_dependent containing the analysis of variance table for each dependent variable. The *i*-th column corresponds to the analysis for the *i*-th dependent variable.

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

Element	Analysis of Variance Statistics
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)

The anova statistics may not be requested if `ido` > 0.

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for array `anova_table` is provided by the user. See `IMSLS_ANOVA_TABLE`.

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of length `n_rows` containing the frequency for each observation.
Default: `frequencies[]` = 1

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_rows` containing the weight for each observation.
Default: `weights[]` = 1

`IMSLS_REGRESSION_INFO`, *Imsls_f_regression* `**regression_info` (Output)
Address of the pointer to an internally allocated structure of type *Imsls_f_regression* containing information about the regression fit. This structure is required as input for functions `imsls_f_regression_prediction` and `imsls_f_regression_summary`.

`IMSLS_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 `IMSLS_NO_INTERCEPT` is specified, the array requires `n_dependent` $\times$ n units of memory, where n = `n_independent`; otherwise, n = `n_independent` + 1.

Description

Function `imsls_f_regression` fits a multivariate multiple linear regression model with or without an intercept. The multiple linear regression model is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the y_i's are the responses or values of the dependent variable; the x_{i1}'s, x_{i2}'s, …, x_{ik}'s are the settings of the k (input in `n_independent`) independent variables; β_0, β_1, …, β_k are the regression coefficients whose estimated values are to be output by `imsls_f_regression`; and the ε_i's are independently distributed normal errors each with mean 0 and variance s^2. Here, n is the sum of the frequencies for all nonmissing observations, i.e.,

$$\left(n = \sum_{i=0}^{\texttt{n\_rows}-1} f_i \right)$$

where f_i is equal to `frequencies[`i`]` if optional argument `IMSLS_FREQUENCIES` is specified and equal to 1.0 otherwise. Note that by default, β_0 is included in the model.

More generally, `imsls_f_regression` fits a multivariate regression model. See the chapter introduction for a description of the multivariate model.

Function `imsls_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 `IMSLS_ANOVA_TABLE` (or `IMSLS_ANOVA_TABLE_USER`) is specified and is computed as follows:

$$SSE = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i \right)^2$$

Another analysis of variance statistic is the total sum of squares. By default, the total sum of squares is the sum of squares of the deviations of y_i from its mean

$$\bar{y}$$

the so-called *corrected total sum of squares*. This statistic is computed as follows:

$$SST = \sum_{i=1}^{n} w_i \left(y_i - \overline{y} \right)^2$$

When `IMSLS_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 follows:

$$\text{SST} = \sum_{i=1}^{n} w_i y_i^2$$

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, `imsls_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 x_i represent the mean vector for the independent variables given the data for rows 1, 2, …, i. The current mean vector is defined as follows:

$$\overline{x}_i = \frac{\sum_{j=1}^{i} w_j f_j x_j}{\sum_{j=1}^{i} w_j f_j}$$

where the w_j's and the f_j's are the weights and frequencies. The i-th row of data has

$$\overline{x}_i$$

subtracted from it and is multiplied by

$$w_i f_i \frac{a_i}{a_{i-1}}$$

where

$$a_i = \sum_{j=1}^{i} w_j f_j$$

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:

$$\sum_{i=1}^{n} w_i f_i \left(x_i - \bar{x}_n\right)\left(x_i - \bar{x}_n\right)^T = \sum_{i=2}^{n} \frac{a_i}{a_{i-1}} w_i f_i \left(x_i - \bar{x}_i\right)\left(x_i - \bar{x}_i\right)^T$$

An orthogonal reduction on the centered matrix is computed. When the final computations are performed, the intercept estimate and the first row and column of the estimated covariance matrix of the estimated coefficients are updated (if `IMSLS_COEF_COVARIANCES` or `IMSLS_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, `imsls_f_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 0.
- Two or more regressors are constant.

$$\sqrt{1 - R^2_{i \cdot 1,2,\ldots,i-1}}$$

is less than or equal to `tolerance`. Here,

$$R_{i \cdot 1,2,\ldots,i-1}$$

is the multiple correlation coefficient of the *i*-th independent variable with the first *i* – 1 independent variables. If no intercept is in the model, the multiple correlation coefficient is computed without adjusting for the mean.

On completion of the final computations, if the *i*-th regressor is declared to be linearly dependent upon the previous *i* – 1 regressors, 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 `IMSLS_COEF_COVARIANCES` or `IMSLS_COEF_COVARIANCES_USER` is specified) are set to 0. Finally, if a linear dependence is declared, an informational (error) message, code `IMSLS_RANK_DEFICIENT`, is issued indicating the model is not full rank.

Examples

Example 1

A regression model

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \varepsilon_i \qquad i = 1, 2, \ldots, 9$$

is fitted to data taken from Maindonald (1984, pp. 203–204).

```
#include <imsls.h>

#define INTERCEPT       1
#define N_INDEPENDENT   3
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_OBSERVATIONS  9

main()
{
    float       *coefficients;
    float       x[][N_INDEPENDENT] = {7.0, 5.0, 6.0,
                                      2.0,-1.0, 6.0,
                                      7.0, 3.0, 5.0,
                                     -3.0, 1.0, 4.0,
                                      2.0,-1.0, 0.0,
                                      2.0, 1.0, 7.0,
                                     -3.0,-1.0, 3.0,
                                      2.0, 1.0, 1.0,
                                      2.0, 1.0, 4.0};
    float       y[] = {7.0,-5.0, 6.0, 5.0, 5.0, -2.0, 0.0, 8.0, 3.0};

    coefficients = imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
                                     (float *)x, y, 0);
    imsls_f_write_matrix("Least-Squares Coefficients", 1, N_COEFFICIENTS,
                        coefficients,
                        IMSLS_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

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i \quad i = 1, 2, \ldots, 4$$

and weights $1/i^2$ discussed by Maindonald (1984, pp. 67–68).

In the example, `IMSLS_WEIGHTS` is specified. The minimum sum of squares for error in terms of the original untransformed regressors and responses for this weighted regression is

$$\text{SSE}=\sum_{i=1}^{4} w_i \left(y_i - \hat{y}_i\right)^2$$

where $w_i = 1/i^2$, represented in the C code as array w.

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

#define N_INDEPENDENT   2
#define N_COEFFICIENTS  N_INDEPENDENT + 1
```

```
#define N_OBSERVATIONS  4

main()
{
    int         i;
    float       *coefficients, w[N_OBSERVATIONS], anova_table[15],
                power;
    float       x[][N_INDEPENDENT] = {
                    -2.0, 0.0,
                    -1.0, 2.0,
                     2.0, 5.0,
                     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)"};

                                /* Calculate weights */
    power = 0.0;
    for (i = 0;  i < N_OBSERVATIONS;  i++)  {
        power += 1.0;
        w[i] = 1.0 / (power*power);
    }

                                /*Perform analysis */
    coefficients = imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
        (float *) x, y,
        IMSLS_WEIGHTS, w,
        IMSLS_ANOVA_TABLE_USER, anova_table,
        0);

                                /* Print results */
    imsls_f_write_matrix("Least Squares Coefficients", 1,
        N_COEFFICIENTS, coefficients, 0);
    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
        anova_table,
        IMSLS_ROW_LABELS, anova_row_labels,
        IMSLS_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             2.00
degrees of freedom for error                  1.00
total (uncorrected) degrees of freedom        3.00
sum of squares for regression                 7.68
sum of squares for error                      1.01
total (uncorrected) sum of squares            8.69
regression mean square                        3.84
error mean square                             1.01
F-statistic                                   3.79
p-value                                       0.34
R-squared (in percent)                       88.34
adjusted R-squared (in percent)              65.03
est. standard deviation of model error        1.01
overall mean of y                            -1.51
coefficient of variation (in percent)       -66.55
```

Example 3

A multivariate regression is performed for a data set with two dependent variables. Also, usage of the keyword `IMSLS_X_INDICES` is demonstrated. Note that the required input variable `y` is not referenced and is declared as a pointer to a float.

```
#include <imsls.h>

#define INTERCEPT       1
#define N_INDEPENDENT   3
#define N_DEPENDENT     2
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_OBSERVATIONS  9

main()
{
    float  coefficients[N_DEPENDENT*N_COEFFICIENTS];
    float  *dummy;
    float  scpe[N_DEPENDENT*N_DEPENDENT];
    float  anova_table[15*N_DEPENDENT];
    static float   x[] =       { 7.0, 5.0, 6.0,  7.0,  1.0,
                                  2.0,-1.0, 6.0, -5.0,  4.0,
                                  7.0, 3.0, 5.0,  6.0, 10.0,
                                 -3.0, 1.0, 4.0,  5.0,  5.0,
                                  2.0,-1.0, 0.0,  5.0, -2.0,
                                  2.0, 1.0, 7.0, -2.0,  4.0,
                                 -3.0,-1.0, 3.0,  0.0, -6.0,
                                  2.0, 1.0, 1.0,  8.0,  2.0,
                                  2.0, 1.0, 4.0,  3.0,  0.0};
    int    ifrq = -1, iwt=-1;
    static int indind[N_INDEPENDENT] = {0, 1, 2};
    static int inddep[N_DEPENDENT] = {3, 4};
    char   *fmt = "%10.4f";
    char   *anova_row_labels[] = {
                   "d.f. regression",
                   "d.f. error",
                   "d.f. total (uncorrected)",
                   "ssr",
                   "sse",
                   "sst (uncorrected)",
                   "msr",
                   "mse", "F-statistic",
                   "p-value", "R-squared (in percent)",
                   "adj. R-squared (in percent)",
                   "est. s.t.d. of model error",
                   "overall mean of y",
                   "coefficient of variation (in percent)"};

    imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
        (float *) x, dummy,
        IMSLS_X_COL_DIM, N_INDEPENDENT+N_DEPENDENT,
        IMSLS_N_DEPENDENT, N_DEPENDENT,
        IMSLS_X_INDICES, indind, inddep, ifrq, iwt,
        IMSLS_SCPE_USER, scpe,
        IMSLS_ANOVA_TABLE_USER, anova_table,
        IMSLS_RETURN_USER, coefficients,
        0);

    imsls_f_write_matrix("Least Squares Coefficients", N_DEPENDENT,
        N_COEFFICIENTS, coefficients,
        IMSLS_COL_NUMBER_ZERO, 0);

    imsls_f_write_matrix("SCPE", N_DEPENDENT, N_DEPENDENT, scpe,
        IMSLS_WRITE_FORMAT, "%10.4f", 0);

    imsls_f_write_matrix("* * * Analysis of Variance * * *\n",
        15, N_DEPENDENT,
        anova_table,
        IMSLS_ROW_LABELS, anova_row_labels,
        IMSLS_WRITE_FORMAT, "%10.2f",
        0);


}
```

Output

```
            Least Squares Coefficients
             0           1           2           3
1        7.733      -0.200       2.333      -1.667
2       -1.633       0.400       0.167       0.667

           SCPE
             1           2
1       4.0000     20.0000
2      20.0000    110.0000

     * * * Analysis of Variance * * *

                              1           2
d.f. regression            3.00        3.00
d.f. error                 5.00        5.00
d.f. total (uncorre        8.00        8.00
   cted)
ssr                      152.00       56.00
sse                        4.00      110.00
```

```
sst (uncorrected)        156.00      166.00
msr                       50.67       18.67
mse                        0.80       22.00
F-statistic               63.33        0.85
p-value                    0.00        0.52
R-squared (in             97.44       33.73
   percent)
adj. R-squared            95.90        0.00
   (in percent)
est. s.t.d. of             0.89        4.69
   model error
overall mean of y          3.00        2.00
coefficient of            29.81      234.52
   variation (in
   percent)
```

Warning Errors

`IMSLS_RANK_DEFICIENT`	The model is not full rank. There is not a unique least-squares solution.

Fatal Errors

`IMSLS_BAD_IDO_6`	"ido" = #. Initial allocations must be performed by making a call to function regression with "ido" = 1.
`IMSLS_BAD_IDO_7`	"ido" = #. A new analysis may not begin until the previous analysis is terminated by a call to function regression with "ido" = 3.

regression_summary

Produces summary statistics for a regression model given the information from the fit.

Synopsis

#include `<imsls.h>`

void `imsls_f_regression_summary`
  (*Imsls_f_regression* `*regression_info`, ..., 0)

The type double function is `imsls_d_regression_summary`.

Required Argument

Imsls_f_regression `*regression_info` (Input)
  Pointer to a structure of type *Imsls_f_regression* containing information about the regression fit. See `imsls_f_regression`.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_regression_summary`
    `(`*Imsls_f_regression* `*regression_info`,
    `IMSLS_INDEX_REGRESSION`, *int* `idep`,
    `IMSLS_COEF_T_TESTS`, *float* `**`*coef_t_tests*
    `IMSLS_COEF_T_TESTS_USER`, *float* `coef_t_tests[]`,
    `IMSLS_COEF_COL_DIM`, *int* `coef_col_dim`,
    `IMSLS_COEF_VIF`, *float* `**coef_vif`,
    `IMSLS_COEF_VIF_USER`, *float* `coef_vif[]`,
    `IMSLS_COEF_COVARIANCES`, *float* `**coef_covariances`,
    `IMSLS_COEF_COVARIANCES_USER`, *float* `coef_covariances[]`,
    `IMSLS_COEF_COV_COL_DIM`, *int* `coef_cov_col_dim`,
    `IMSLS_ANOVA_TABLE`, *float* `**anova_table`,
    `IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]`,
    0)

Optional Arguments

`IMSLS_INDEX_REGRESSION`, *int* `idep` (Input)
Given a multivariate regression fit, this option allows the user to specify for which regression summary statistics will be computed.
Default: `idep` = 0

`IMSLS_COEF_T_TESTS`, *float* `**coef_t_tests` (Output)
Address of a pointer to the `npar` × 4 array containing statistics relating to the regression coefficients, where *npar* is equal to the number of parameters in the model.

Each row (for each dependent variable) corresponds to a coefficient in the model, where *npar* is the number of parameters in the model. Row i + *intcep* corresponds to the i-th independent variable, where *intcep* is equal to 1 if an intercept is in the model and 0 otherwise, for $i = 0, 1, 2, \ldots, npar - 1$.

The statistics in the columns are as follows:

Column	Description
0	coefficient estimate
1	estimated standard error of the coefficient estimate
2	*t*-statistic for the test that the coefficient is 0
3	*p*-value for the two-sided *t* test

`IMSLS_COEF_T_TESTS_USER`, *float* `coef_t_tests[]` (Output)
Storage for array `coef_t_tests` is provided by the user. See `IMSLS_COEF_T_TESTS`.

`IMSLS_COEF_COL_DIM`, *int* `coef_col_dim` (Input)
Column dimension of `coef_t_tests`.
Default: `coef_col_dim` = 4

`IMSLS_COEF_VIF`, *float* `**coef_vif` (Output)
Address of a pointer to an internally allocated array of length *npar* containing the variance inflation factor, where *npar* is the number of parameters. The $i + intcep$-th column corresponds to the *i*-th independent variable, where $i = 0, 1, 2, \ldots, npar - 1$, and *intcep* is equal to 1 if an intercept is in the model and 0 otherwise.

The square of the multiple correlation coefficient for the *i*-th regressor after all others can be obtained from `coef_vif` by

$$1.0 - \frac{1.0}{\texttt{coef\_vif}[i]}$$

If there is no intercept, or there is an intercept and $j = 0$, the multiple correlation coefficient is not adjusted for the mean.

`IMSLS_COEF_VIF_USER`, *float* `coef_vif[]` (Output)
Storage for array `coef_t_tests` is provided by the user. See `IMSLS_COEF_VIF`.

`IMSLS_COEF_COVARIANCES`, *float* `**coef_covariances` (Output)
An *npar* by *npar* (where *npar* is equal to the number of parameters in the model) array that is the estimated variance-covariance matrix of the estimated regression coefficients when *R* is nonsingular and is from an unrestricted regression fit. See “Remarks” on page 82 for an explanation of `coef_covariances` when *R* is singular and is from a restricted regression fit.

`IMSLS_COEF_COVARIANCES_USER`, *float* `coef_covariances[]` (Output)
Storage for coef_covariances is provided by the user. See `IMSLS_COEF_COVARIANCES`.

`IMSLS_COEF_COV_COL_DIM`, *int* `coef_cov_col_dim` (Input)
Column dimension of `coef_covariances`.
Default: `coef_cov_col_dim` = the number of parameters in the model

`IMSLS_ANOVA_TABLE`, *float* `**anova_table` (Output)
Address of a pointer to the array of size 15 containing the analysis of variance table.

Row	**Analysis of Variance Statistic**
0	degrees of freedom for the model
1	degrees of freedom for error
2	total (corrected) degrees of freedom
3	sum of squares for the model
4	sum of squares for error
5	total (corrected) sum of squares
6	model mean square
7	error mean square
8	overall F-statistic
9	p-value
10	R^2(in percent)
11	adjusted R^2 (in percent)
12	estimate of the standard deviation
13	overall mean of y
14	coefficient of variation (in percent)

If the model has an intercept, the regression and total are corrected for the mean; otherwise, the regression and total are not corrected for the mean, and `anova_table`[13] and `anova_table`[14] are set to NaN.

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for array `anova_table` is provided by the user. See `IMSLS_ANOVA_TABLE`.

Description

Function `imsls_f_regression_summary` computes summary statistics from a fitted general linear model. The model is $y = X\beta + \varepsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors, β is the $p \times 1$ vector of regression coefficients, and ε is the $n \times 1$ vector of errors whose elements are each independently distributed with mean 0 and variance σ^2. Function `regression` can be used to compute the fit of the model. Next, `imsls_f_regression_summary` uses the results of this fit to compute summary statistics, including analysis of variance, sequential sum of squares, t tests, and an estimated variance-covariance matrix of the estimated regression coefficients.

Some generalizations of the general linear model are allowed. If the i-th element of ε has variance of

$$\frac{\sigma^2}{w_i}$$

and the weights w_i are used in the fit of the model, `imsls_f_regression_summary` produces summary statistics from the weighted least-squares fit. More generally, if the variance-covariance matrix of ε is $\sigma^2 V$, `imsls_f_regression_summary` can be used to produce summary statistics from the generalized least-squares fit. Function `regression` can be used to perform a generalized least-squares fit, by regressing y^* on X^* where $y^* = (T^{-1})^T y$, $X^* = (T^{-1})^T X$ and T satisfies $T^T T = V$.

The sequential sum of squares for the i-th regression parameter is given by

$$\left(R\hat{\beta}\right)_i^2$$

The regression sum of squares is given by the sum of the sequential sums of squares. If an intercept is in the model, the regression sum of squares is adjusted for the mean, i.e.,

$$\left(R\hat{\beta}\right)_0^2$$

is not included in the sum.

The estimate of σ^2 is s^2 (stored in `anova_table`[7]) that is computed as SSE/DFE.

If R is nonsingular, the estimated variance-covariance matrix of

$$\hat{\beta}$$

(stored in `coef_covariances`) is computed by $s^2 R^{-1} (R^{-1})^T$.

If R is singular, corresponding to rank(X) < p, a generalized inverse is used. For a matrix G to be a g_i (i = 1, 2, 3, or 4) inverse of a matrix A, G must satisfy conditions j (for $j \le i$) for the Moore-Penrose inverse but generally must fail conditions k (for $k > i$). 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

In the case where R is singular, the method for obtaining `coef_covariances` follows the discussion of Maindonald (1984, pp. 101–103). Let Z be the diagonal matrix with diagonal elements defined by the following:

$$z_{ii} = \begin{cases} 1 \text{ if } r_{ii} \neq 0 \\ 0 \text{ if } r_{ii} = 0 \end{cases}$$

Let G be the solution to $RG = Z$ obtained by setting the i-th ($\{i : r_{ii} = 0\}$) row of G to 0. Argument `coef_covariances` is set to s^2GG^T. (G is a g_3 inverse of R, represented by,

$$R^{g_3}$$

the result

$$R^{g_3} R^{g_3^T}$$

is a symmetric g_2 inverse of $R^TR = X^TX$. See Sallas and Lionti 1988.)

Note that argument `coef_covariances` can be used only to get variances and covariances of estimable functions of the regression coefficients, i.e., nonestimable functions (linear combinations of the regression coefficients not in the space spanned by the nonzero rows of R) must not be used. See, for example, Maindonald (1984, pp. 166–168) for a discussion of estimable functions.

The estimated standard errors of the estimated regression coefficients (stored in Column 1 of `coef_t_tests`) are computed as square roots of the corresponding diagonal entries in `coef_covariances`.

For the case where an intercept is in the model, put $\bar{R}$ equal to the matrix R with the first row and column deleted. Generally, the variance inflation factor (VIF) for the i-th regression coefficient is computed as the product of the i-th diagonal element of R^TR and the i-th diagonal element of its computed inverse. If an intercept is in the model, the VIF for those coefficients not corresponding to the intercept uses the diagonal elements of $\bar{R}^T\bar{R}$ (see Maindonald 1984, p. 40).

Remarks

When R is nonsingular and comes from an unrestricted regression fit, `coef_covariances` is the estimated variance-covariance matrix of the estimated regression coefficients, and `coef_covariances` = (SSE/DFE) (R^TR). Otherwise, variances and covariances of estimable functions of the regression coefficients can be obtained using `coef_covariances`, and `coef_covariances` = (SSE/DFE) (GDG^T). Here, D is the diagonal matrix with diagonal elements equal to 0 if the corresponding rows of R are restrictions and with diagonal elements equal to 1 otherwise. Also, G is a particular generalized inverse of R.

Example

```
#include <imsls.h>

main()
{
#define INTERCEPT       1
#define N_INDEPENDENT   4
#define N_OBSERVATIONS  13
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_DEPENDENT     1

    Imsls_f_regression   *regression_info;
    float       *anova_table, *coef_t_tests, *coef_vif,
                *coefficients, *coef_covariances;
    float       x[][N_INDEPENDENT] = {
        7.0, 26.0,  6.0, 60.0,
        1.0, 29.0, 15.0, 52.0,
       11.0, 56.0,  8.0, 20.0,
       11.0, 31.0,  8.0, 47.0,
        7.0, 52.0,  6.0, 33.0,
       11.0, 55.0,  9.0, 22.0,
        3.0, 71.0, 17.0,  6.0,
        1.0, 31.0, 22.0, 44.0,
        2.0, 54.0, 18.0, 22.0,
       21.0, 47.0,  4.0, 26.0,
        1.0, 40.0, 23.0, 34.0,
       11.0, 66.0,  9.0, 12.0,
       10.0, 68.0,  8.0, 12.0};
    float        y[] = {78.5, 74.3, 104.3, 87.6, 95.9, 109.2,
       102.7, 72.5, 93.1, 115.9, 83.8, 113.3, 109.4};
    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)"};

                                /* Fit the regression model */
    coefficients = imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
        (float *)x, y,
        IMSLS_REGRESSION_INFO, &regression_info,
        0);
```

```
                              /* Generate summary statistics */
    imsls_f_regression_summary (regression_info,
        IMSLS_ANOVA_TABLE, &anova_table,
        IMSLS_COEF_T_TESTS, &coef_t_tests,
        IMSLS_COEF_VIF, &coef_vif,
        IMSLS_COEF_COVARIANCES, &coef_covariances,
        0);

                              /* Print results */
    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
        anova_table,
        IMSLS_ROW_LABELS, anova_row_labels,
        IMSLS_WRITE_FORMAT, "%10.2f", 0);

    imsls_f_write_matrix("* * * Inference on Coefficients * * *\n",
        N_COEFFICIENTS, 4, coef_t_tests,
        IMSLS_WRITE_FORMAT, "%10.2f", 0);

    imsls_f_write_matrix("* * * Variance Inflation Factors * * *\n",
        N_COEFFICIENTS, 1, coef_vif,
        IMSLS_WRITE_FORMAT, "%10.2f", 0);

    imsls_f_write_matrix("* * * Variance-Covariance Matrix * * *\n",
        N_COEFFICIENTS, N_COEFFICIENTS,
        coef_covariances,
        IMSLS_WRITE_FORMAT, "%10.2f", 0);
}
```

Output

```
         * * * Analysis of Variance * * *
degrees of freedom for regression           4.00
degrees of freedom for error                8.00
total (uncorrected) degrees of freedom     12.00
sum of squares for regression            2667.90
sum of squares for error                   47.86
total (uncorrected) sum of squares       2715.76
regression mean square                    666.97
error mean square                           5.98
F-statistic                               111.48
p-value                                     0.00
R-squared (in percent)                     98.24
adjusted R-squared (in percent)            97.36
est. standard deviation of model error      2.45
overall mean of y                          95.42
coefficient of variation (in percent)       2.56

     * * * Inference on Coefficients * * *

           1           2           3           4
1      62.41       70.07        0.89        0.40
2       1.55        0.74        2.08        0.07
3       0.51        0.72        0.70        0.50
4       0.10        0.75        0.14        0.90
5      -0.14        0.71       -0.20        0.84

* * * Variance Inflation Factors * * *
```

```
               1    10668.53
               2       38.50
               3      254.42
               4       46.87
               5      282.51


             * * * Variance-Covariance Matrix * * *

               1           2           3           4           5
1      4909.95      -50.51      -50.60      -51.66      -49.60
2       -50.51        0.55        0.51        0.55        0.51
3       -50.60        0.51        0.52        0.53        0.51
4       -51.66        0.55        0.53        0.57        0.52
5       -49.60        0.51        0.51        0.52        0.50
```

regression_prediction

Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_regression_prediction`
(*Imsls_f_regression* `*regression_info`, *int* `n_predict`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_regression_prediction`.

Required Argument

Imsls_f_regression `*regression_info` (Input)
Pointer to a structure of type *Imsls_f_regression* containing information about the regression fit. See `imsls_f_regression` (page 64).

int `n_predict` (Input)
Number of rows in `x`.

float `x[]` (Input)
Array of size `n_predict` by the number of independent variables containing the combinations of independent variables in each row for which calculations are to be performed.

Return Value

Pointer to an internally allocated array of length `n_predict` containing the predicted values.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float *imsls_f_regression_prediction
        (Imsls_f_regression *regression_info, int n_predict, float x[],
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_Y_COL_DIM, int y_col_dim,
        IMSLS_INDEX_REGRESSION, int idep,
        IMSLS_X_INDICES, int indind[], int inddep[], int ifrq,
                int iwt,
        IMSLS_WEIGHTS, float weights[],
        IMSLS_CONFIDENCE, float confidence,
        IMSLS_SCHEFFE_CI, float **lower_limit,
                float **upper_limit,
        IMSLS_SCHEFFE_CI_USER, float lower_limit[],
                float upper_limit[],
        IMSLS_POINTWISE_CI_POP_MEAN, float **lower_limit,
                float **upper_limit,
        IMSLS_POINTWISE_CI_POP_MEAN_USER, float lower_limit[],
                float upper_limit[],
        IMSLS_POINTWISE_CI_NEW_SAMPLE, float **lower_limit,
                float **upper_limit,
        IMSLS_POINTWISE_CI_NEW_SAMPLE_USER,
                float lower_limit[], float upper_limit[],
        IMSLS_LEVERAGE, float **leverage,
        IMSLS_LEVERAGE_USER, float leverage[],
        IMSLS_RETURN_USER, float y_hat[],
        IMSLS_Y, float y[],
        IMSLS_RESIDUAL, float **residual,
        IMSLS_RESIDUAL_USER, float residual[],
        IMSLS_STANDARDIZED_RESIDUAL,
                float **standardized_residual,
        IMSLS_STANDARDIZED_RESIDUAL_USER,
                float standardized_residual[],
        IMSLS_DELETED_RESIDUAL, float **deleted_residual,
        IMSLS_DELETED_RESIDUAL_USER, float deleted_residual[],
        IMSLS_COOKSD, float **cooksd,
        IMSLS_COOKSD_USER, float cooksd[],
        IMSLS_DFFITS, float **dffits,
        IMSLS_DFFITS_USER, float dffits[],
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)

Number of columns in `x`.
Default: `x_col_dim` is equal to the number of independent variables, which is input from the structure `regression_info`

`IMSLS_Y_COL_DIM`, *int* `y_col_dim` (Input)
Number of columns in `y`.
Default: `y_col_dim` = 1

`IMSLS_INDEX_REGRESSION`, *int* `idep` (Input)
Given a multivariate regression fit, this option allows the user to specify for which regression statistics will be computed.
Default: `idep` = 0

`IMSLS_X_INDICES`, *int* `indind[]`, *int* `inddep`, *int* `ifrq`, *int* `iwt` (Input)
This argument allows an alternative method for data specification. Data (independent, dependent, frequencies, and weights) is all stored in the data matrix `x`. Argument `y`, and keyword `IMSLS_WEIGHTS` are ignored.

Each of the four arguments contains indices indicating column numbers of `x` in which particular types of data are stored. Columns are numbered 0, …, `x_col_dim` – 1.

Parameter `indind` contains the indices of the independent variables.

Parameter `inddep` contains the indices of the dependent variables. If there is to be no dependent variable, this must be indicated by setting the first element of the vector to −1.

Parameters `ifrq` and `iwt` contain the column numbers of `x` in which the frequencies and weights, respectively, are stored. Set `ifrq` = −1 if there will be no column for frequencies. Set `iwt` = −1 if there will be no column for weights. Weights are rounded to the nearest integer. Negative weights are not allowed.

Note that frequencies are not referenced by function `regression_prediction`, and is included here only for the sake of keyword consistency.

Finally, note that `IMSLS_X_INDICES` and `IMSLS_Y` are mutually exclusive keywords, and may not be specified in the same call to `regression_prediction`.

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_predict` containing the weight for each row of `x`. The computed prediction interval uses SSE/(DFE`*weights`[*i*]) for the estimated variance of a future response.
Default: `weights[]` = 1

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Confidence level for both two-sided interval estimates on the mean and for two-sided prediction intervals, in percent. Argument `confidence` must be in the range [0.0, 100.0). For one-sided intervals with confidence level `onecl`, where 50.0 ≤ `onecl` < 100.0, set `confidence` = 100.0 – 2.0`*` (100.0 – `onecl`).
Default: `confidence` = 95.0

`IMSLS_SCHEFFE_CI`, *float* `**lower_limit`, *float* `**upper_limit` (Output)
Array `lower_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the lower confidence limits of Scheffé confidence intervals corresponding to the rows of `x`. Array `upper_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the upper confidence limits of Scheffé confidence intervals corresponding to the rows of `x`.

`IMSLS_SCHEFFE_CI_USER`, *float* `lower_limit[]`, *float* `upper_limit[]` (Output)
Storage for arrays `lower_limit` and `upper_limit` is provided by the user. See `IMSLS_SCHEFFE_CI`.

`IMSLS_POINTWISE_CI_POP_MEAN`, *float* `**lower_limit`, *float* `**upper_limit` (Output)
Array `lower_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the lower-confidence limits of the confidence intervals for two-sided interval estimates of the means, corresponding to the rows of x. Array `upper_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the upper-confidence limits of the confidence intervals for two-sided interval estimates of the means, corresponding to the rows of `x`.

`IMSLS_POINTWISE_CI_POP_MEAN_USER`, *float* `lower_limit[]`, *float* `upper_limit[]` (Output)
Storage for arrays `lower_limit` and `upper_limit` is provided by the user. See `IMSLS_POINTWISE_CI_POP_MEAN`.

`IMSLS_POINTWISE_CI_NEW_SAMPLE`, *float* `**lower_limit`, *float* `**upper_limit` (Output)
Array `lower_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the lower-confidence limits of the confidence intervals for two-sided prediction intervals, corresponding to the rows of `x`. Array `upper_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the upper-confidence limits of the confidence intervals for two-sided prediction intervals, corresponding to the rows of `x`.

`IMSLS_POINTWISE_CI_NEW_SAMPLE_USER`, *float* `lower_limit[]`, *float* `upper_limit[]` (Output)
Storage for arrays `lower_limit` and `upper_limit` is provided by the user. See `IMSLS_POINTWISE_CI_NEW_SAMPLE`.

`IMSLS_LEVERAGE`, *float* `**leverage` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the leverages.

`IMSLS_LEVERAGE_USER`, *float* `leverage[]` (Output)
Storage for array leverage is provided by the user. See `IMSLS_LEVERAGE`.

`IMSLS_RETURN_USER`, *float* `y_hat[]` (Output)
Storage for array `y_hat` is provided by the user. The length `n_predict` array contains the predicted values.

`IMSLS_Y`, *float* `y[]` (Input)
Array of length `n_predict` containing the observed responses.

Note: `IMSLS_Y` (or `IMSLS_X_INDICES`) must be specified if any of the following optional arguments are specified.

`IMSLS_RESIDUAL`, *float* `**residual` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the residuals.

`IMSLS_RESIDUAL_USER`, *float* `residual[]` (Output)
Storage for array residual is provided by the user. See `IMSLS_RESIDUAL`.

`IMSLS_STANDARDIZED_RESIDUAL`, *float* `**standardized_residual` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the standardized residuals.

`IMSLS_STANDARDIZED_RESIDUAL_USER`, *float* `standardized_residual[]` (Output)
Storage for array `standardized_residual` is provided by the user. See `IMSLS_STANDARDIZED_RESIDUAL`.

`IMSLS_DELETED_RESIDUAL`, *float* `**deleted_residual` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the deleted residuals.

`IMSLS_DELETED_RESIDUAL_USER`, *float* `deleted_residual[]` (Output)
Storage for array `deleted_residual` is provided by the user. See `IMSLS_DELETED_RESIDUAL`.

`IMSLS_COOKSD`, *float* `**cooksd` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the Cook's *D* statistics.

`IMSLS_COOKSD_USER`, *float* `cooksd[]` (Output)
Storage for array cooksd is provided by the user. See `IMSLS_COOKSD`.

`IMSLS_DFFITS`, *float* `**dffits` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the DFFITS statistics.

`IMSLS_DFFITS_USER`, *float* `dffits[]` (Output)
Storage for array `dffits` is provided by the user. See `IMSLS_DFFITS`.

Description

The general linear model used by function `imsls_f_regression_prediction` is

$$y = X\beta + \varepsilon$$

where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors, β is the $p \times 1$ vector of regression coefficients, and ε is the $n \times 1$ vector of errors whose elements are independently normally distributed with mean 0 and the variance below.

$$\frac{\sigma^2}{w_i}$$

From a general linear model fit using the w_i's as the weights, function `imsls_f_regression_prediction` computes confidence intervals and statistics for the individual cases that constitute the data set. Let x_i be a column vector containing elements of the i-th row of X. Let $W = \text{diag}(w_1, w_2, \ldots, w_n)$. The leverage is defined as

$$h_i = \left(x_i^T \left(X^T W X\right)^{-}\right) x_i w_i$$

Put $D = \text{diag}(d_1, d_2, \ldots, d_n)$ with $d_j = 1$ if the j-th diagonal element of R is positive and 0 otherwise. The leverage is computed as $h_i = (a^T D a)\, w_i$ where a is a solution to $R^T a = x_i$. The estimated variance of

$$\hat{y} = x_i^T \hat{B}$$

is given by the following:

$$\frac{h_i s^2}{w_i}$$

where

$$s^2 = \frac{\text{SSE}}{\text{DFE}}$$

The computation of the remainder of the case statistics follow easily from their definitions. See case diagnostics (page 53).

Informational errors can occur if the input matrix `x` is not consistent with the information from the fit (contained in `regression_info`), or if excess rounding has occurred. The warning error `IMSLS_NONESTIMABLE` arises when `x` contains a row not in the space spanned by the rows of R. An examination of the model that was fitted and the `x` for which diagnostics are to be computed is required in order to ensure that only linear combinations of the regression coefficients that can be estimated from the fitted model are specified in `x`. For further details, see the discussion of estimable functions given in Maindonald (1984, pp. 166–168) and Searle (1971, pp. 180–188).

Often predicted values and confidence intervals are desired for combinations of settings of the independent variables not used in computing the regression fit. This can be accomplished by defining a new data matrix. Since the information about the model fit is input in `regression_info`, it is not necessary to send in the data set used for the original calculation of the fit, i.e., only variable combinations for which predictions are desired need be entered in x.

Examples

Example 1

```
#include <imsls.h>

main()
{
#define INTERCEPT       1
#define N_INDEPENDENT   4
#define N_OBSERVATIONS  13
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_DEPENDENT     1

    float       *y_hat, *coefficients;
    Imsls_f_regression   *regression_info;
    float       x[][N_INDEPENDENT] = {
        7.0, 26.0,  6.0, 60.0,
        1.0, 29.0, 15.0, 52.0,
       11.0, 56.0,  8.0, 20.0,
       11.0, 31.0,  8.0, 47.0,
        7.0, 52.0,  6.0, 33.0,
       11.0, 55.0,  9.0, 22.0,
        3.0, 71.0, 17.0,  6.0,
        1.0, 31.0, 22.0, 44.0,
        2.0, 54.0, 18.0, 22.0,
       21.0, 47.0,  4.0, 26.0,
        1.0, 40.0, 23.0, 34.0,
       11.0, 66.0,  9.0, 12.0,
       10.0, 68.0,  8.0, 12.0};
    float        y[] = {78.5, 74.3, 104.3, 87.6, 95.9, 109.2,
       102.7, 72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

                            /* Fit the regression model */
    coefficients = imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
        (float *)x, y,
        IMSLS_REGRESSION_INFO, &regression_info,
        0);

                            /* Generate case statistics */
    y_hat = imsls_f_regression_prediction(regression_info,
        N_OBSERVATIONS, (float*)x, 0);

                            /* Print results */
    imsls_f_write_matrix("Predicted Responses", 1, N_OBSERVATIONS,
        y_hat, 0);
}
```

Output

```
                    Predicted Responses
         1           2           3           4           5           6
      78.5        72.8       106.0        89.3        95.6       105.3

         7           8           9          10          11          12
```

```
     104.1        75.7        91.7       115.6        81.8       112.3

        13
     111.7
```

Example 2

```
#include <imsls.h>

main()
{
#define INTERCEPT       1
#define N_INDEPENDENT   4
#define N_OBSERVATIONS  13
#define N_COEFFICIENTS  (INTERCEPT + N_INDEPENDENT)
#define N_DEPENDENT     1

    float       *y_hat, *leverage, *residual, *standardized_residual,
                *deleted_residual, *dffits, *cooksd, *mean_lower_limit,
                *mean_upper_limit, *new_sample_lower_limit,
                *new_sample_upper_limit, *scheffe_lower_limit,
                *scheffe_upper_limit, *coefficients;
    Imsls_f_regression   *regression_info;
    float       x[][N_INDEPENDENT] = {
        7.0, 26.0,  6.0, 60.0,
        1.0, 29.0, 15.0, 52.0,
       11.0, 56.0,  8.0, 20.0,
       11.0, 31.0,  8.0, 47.0,
        7.0, 52.0,  6.0, 33.0,
       11.0, 55.0,  9.0, 22.0,
        3.0, 71.0, 17.0,  6.0,
        1.0, 31.0, 22.0, 44.0,
        2.0, 54.0, 18.0, 22.0,
       21.0, 47.0,  4.0, 26.0,
        1.0, 40.0, 23.0, 34.0,
       11.0, 66.0,  9.0, 12.0,
       10.0, 68.0,  8.0, 12.0};
    float        y[] = {78.5, 74.3, 104.3, 87.6, 95.9, 109.2,
       102.7, 72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

                                /* Fit the regression model */
    coefficients = imsls_f_regression(N_OBSERVATIONS, N_INDEPENDENT,
        (float *)x, y,
        IMSLS_REGRESSION_INFO, &regression_info,
        0);

                                /* Generate the case statistics */
    y_hat = imsls_f_regression_prediction(regression_info,
        N_OBSERVATIONS, (float*)x,
        IMSLS_Y,                        y,
        IMSLS_LEVERAGE,                 &leverage,
        IMSLS_RESIDUAL,                 &residual,
        IMSLS_STANDARDIZED_RESIDUAL,    &standardized_residual,
        IMSLS_DELETED_RESIDUAL,         &deleted_residual,
        IMSLS_COOKSD,                   &cooksd,
        IMSLS_DFFITS,                   &dffits,
        IMSLS_POINTWISE_CI_POP_MEAN,    &mean_lower_limit,
                                        &mean_upper_limit,
        IMSLS_POINTWISE_CI_NEW_SAMPLE, &new_sample_lower_limit,
                                       &new_sample_upper_limit,
        IMSLS_SCHEFFE_CI,              &scheffe_lower_limit,
                                       &scheffe_upper_limit,
        0);

                                /* Print results */
    imsls_f_write_matrix("Predicted Responses", 1, N_OBSERVATIONS,
        y_hat, 0);
    imsls_f_write_matrix("Residuals", 1, N_OBSERVATIONS, residual, 0);
    imsls_f_write_matrix("Standardized Residuals", 1, N_OBSERVATIONS,
        standardized_residual, 0);
    imsls_f_write_matrix("Leverages", 1, N_OBSERVATIONS, leverage, 0);
    imsls_f_write_matrix("Deleted Residuals", 1, N_OBSERVATIONS,
        deleted_residual, 0);
    imsls_f_write_matrix("Cooks D", 1, N_OBSERVATIONS, cooksd, 0);
    imsls_f_write_matrix("DFFITS", 1, N_OBSERVATIONS, dffits, 0);
    imsls_f_write_matrix("Scheffe Lower Limit", 1, N_OBSERVATIONS,
        scheffe_lower_limit, 0);
    imsls_f_write_matrix("Scheffe Upper Limit", 1, N_OBSERVATIONS,
        scheffe_upper_limit, 0);
    imsls_f_write_matrix("Population Mean Lower Limit", 1,
        N_OBSERVATIONS, mean_lower_limit, 0);
    imsls_f_write_matrix("Population Mean Upper Limit", 1,
        N_OBSERVATIONS, mean_upper_limit, 0);
    imsls_f_write_matrix("New Sample Lower Limit", 1, N_OBSERVATIONS,
        new_sample_lower_limit, 0);
    imsls_f_write_matrix("New Sample Upper Limit", 1, N_OBSERVATIONS,
        new_sample_upper_limit, 0);
}
```

Output

```
                         Predicted Responses
         1            2            3            4            5            6
      78.5         72.8        106.0         89.3         95.6        105.3

         7            8            9           10           11           12
     104.1         75.7         91.7        115.6         81.8        112.3

        13
     111.7

                              Residuals
         1            2            3            4            5            6
     0.005        1.511       -1.671       -1.727        0.251        3.925

         7            8            9           10           11           12
    -1.449       -3.175        1.378        0.282        1.991        0.973

        13
    -2.294

                       Standardized Residuals
         1            2            3            4            5            6
     0.003        0.757       -1.050       -0.841        0.128        1.715

         7            8            9           10           11           12
    -0.744       -1.688        0.671        0.210        1.074        0.463
```

```
      13
  -1.124

                              Leverages
       1           2           3           4           5           6
  0.5503      0.3332      0.5769      0.2952      0.3576      0.1242

       7           8           9          10          11          12
  0.3671      0.4085      0.2943      0.7004      0.4255      0.2630

      13
  0.3037

                          Deleted Residuals
       1           2           3           4           5           6
   0.003       0.735      -1.058      -0.824       0.120       2.017

       7           8           9          10          11          12
  -0.722      -1.967       0.646       0.197       1.086       0.439

      13
  -1.146

                               Cooks D
       1           2           3           4           5           6
  0.0000      0.0572      0.3009      0.0593      0.0018      0.0834

       7           8           9          10          11          12
  0.0643      0.3935      0.0375      0.0207      0.1708      0.0153

      13
  0.1102

                               DFFITS
       1           2           3           4           5           6
   0.003       0.519      -1.236      -0.533       0.089       0.759

       7           8           9          10          11          12
  -0.550      -1.635       0.417       0.302       0.935       0.262

      13
  -0.757
                         Scheffe Lower Limit
       1           2           3           4           5           6
    70.7        66.7        98.0        83.6        89.4       101.6

       7           8           9          10          11          12
    97.8        69.0        86.0       106.8        75.0       106.9

      13
   105.9

                         Scheffe Upper Limit
       1           2           3           4           5           6
    86.3        78.9       113.9        95.0       101.9       109.0

       7           8           9          10          11          12
   110.5        82.4        97.4       124.4        88.7       117.7
```

```
        13
     117.5

                  Population Mean Lower Limit
         1            2            3            4            5            6
      74.3         69.5        101.7         86.3         92.3        103.3

         7            8            9           10           11           12
     100.7         72.1         88.7        110.9         78.1        109.4

        13
     108.6

                  Population Mean Upper Limit
         1            2            3            4            5            6
      82.7         76.0        110.3         92.4         99.0        107.3

         7            8            9           10           11           12
     107.6         79.3         94.8        120.3         85.5        115.2

        13
     114.8

                    New Sample Lower Limit
         1            2            3            4            5            6
      71.5         66.3         98.9         82.9         89.1         99.3

         7            8            9           10           11           12
      97.6         69.0         85.3        108.3         75.1        106.0

        13
     105.3

                    New Sample Upper Limit
         1            2            3            4            5            6
      85.5         79.3        113.1         95.7        102.2        111.3

         7            8            9           10           11           12
     110.7         82.4         98.1        123.0         88.5        118.7

        13
     118.1
```

Warning Errors

`IMSLS_NONESTIMABLE`	Within the preset tolerance, the linear combination of regression coefficients is nonestimable.
`IMSLS_LEVERAGE_GT_1`	A leverage (= #) much greater than 1.0 is computed. It is set to 1.0.
`IMSLS_DEL_MSE_LT_0`	A deleted residual mean square (= #) much less than 0 is computed. It is set to 0.

Fatal Errors

`IMSLS_NONNEG_WEIGHT_REQUEST_2`	The weight for row # was #. Weights must be nonnegative.

hypothesis_partial

Constructs an equivalent completely testable multivariate general linear hypothesis $H\beta U = G$ from a partially testable hypothesis $H_p\beta U = G_p$.

Synopsis

#include `<imsls.h>`

int `imsls_f_hypothesis_partial`
    (*Imsls_f_regression* `*regression_info`, *int* `nhp`, *float* `hp[]`, ..., 0)

The type *double* function is `imsls_d_hypothesis_partial`.

Required Argument

Imsls_f_regression `*regression_info` (Input)
    Pointer to a structure of type *Imsls_f_regression* containing information about the regression fit. See function `imsls_f_regression` (page 64).

int `nhp` (Input)
    Number of rows in the hypothesis matrix, `hp`.

float `hp[]` (Input)
    The H_p array of size `nhp` by *n_coefficients* with each row corresponding to a row in the hypothesis and containing the constants that specify a linear combination of the regression coefficients. Here, *n_coefficients* is the number of coefficients in the fitted regression model.

Return Value

Number of rows in the completely testable hypothesis, `nh`. This value is also the degrees of freedom for the hypothesis. The value `nh` classifies the hypothesis $H_p\beta U = G_p$ as nontestable (`nh` = 0), partially testable (0 < `nh` < `rank_hp`) or completely testable (0 < `nh` = `rank_hp`), where `rank_hp` is the rank of H_p (see keyword `IMSLS_RANK_HP`).

Synopsis with Optional Arguments

#include `<imsls.h>`

int `imsls_f_hypothesis_partial`
    (*Imsls_f_regression* `*regression_info`, *int* `nhp`, *float* `hp[]`,
    `IMSLS_GP`, *float* `gp[]`,
    `IMSLS_U`, *int* `nu`, *float* `u[]`,

`IMSLS_RANK_HP`, *int* `rank_hp`
`IMSLS_H_MATRIX`, *float* `**h`,
`IMSLS_H_MATRIX_USER`, *float* `h[]`,
`IMSLS_G`, *float* `**g`,
`IMSLS_G_USER`, *float* `g[]`,
0)

Optional Arguments

`IMSLS_GP`, *float* `gp[]` (Input)
Array of size `nhp` by `nu` containing the G_p matrix, the null hypothesis values. By default, each value of G_p is equal to 0.

`IMSLS_U`, *int* `nu`, *float* `u[]` (Input)
Argument `nu` is the number of linear combinations of the dependent variables to be considered. The value `nu` must be greater than 0 and less than or equal to *n_dependent*.

Argument `u` contains the *n_dependent* by `nu` U matrix for the test $H_pBU = G_p$. This argument is not referenced by `imsls_f_hypothesis_partial` and is included only for consistency with functions `imsls_f_hypothesis_scph` and `imsls_f_hypothesis_test`. A dummy array of length 1 may be substituted for this argument.

Default: `nu` = *n_dependent* and `u` is the identity matrix.

`IMSLS_RANK_HP`, *int*`*rank_hp` (Output)
Rank of H_p.

`IMSLS_H_MATRIX`, *float* `**h` (Output)
Address of a pointer to the internally allocated array of size `nhp` by *n_parameters* containing the H matrix. Each row of `h` corresponds to a row in the completely testable hypothesis and contains the constants that specify an estimable linear combination of the regression coefficients.

`IMSLS_H_MATRIX_USER`, *float* `h[]` (Output)
Storage for array `h` is provided by the user. See `IMSLS_H`.

`IMSLS_G`, *float* `**g` (Output)
Address of a pointer to the internally allocated array of size `nph` ny `n_dependent` containing the G matrix. The elements of `g` contain the null hypothesis values for the completely testable hypothesis.

`IMSLS_G_USER`, *float* `g[]` (Output)
Storage for array `g` is provided by the user. See `IMSLS_G`.

Description

Once a general linear model $y = X\beta + \varepsilon$ is fitted, particular hypothesis tests are frequently of interest. If the matrix of regressors X is not full rank (as evidenced by the fact that some diagonal elements of the R matrix output from the fit are equal to zero), methods that use the results of the fitted model to compute the hypothesis sum of squares (see function `imsls_f_hypothesis_scph`, page 101) require specification in the hypothesis of only linear combinations of the regression parameters that are estimable. A linear combination of regression parameters $c^T\beta$ is *estimable* if there exists some vector a such that $c^T = a^TX$, i.e., c^T is in the space spanned by the rows of X. For a further discussion of estimable functions, see Maindonald (1984, pp. 166–168) and Searle (1971, pp. 180–188). Function `imsls_f_hypothesis_partial` is only useful in the case of non-full rank regression models, i.e., when the problem of estimability arises.

Peixoto (1986) noted that the customary definition of testable hypothesis in the context of a general linear hypothesis test $H\beta = g$ is overly restrictive. He extended the notion of a testable hypothesis (a hypothesis composed of estimable functions of the regression parameters) to include partially testable and completely testable hypothesis. A hypothesis $H\beta = g$ is *partially testable* if the intersection of the row space H (denoted by $\Re(H)$) and the row space of X ($\Re(X)$) is not essentially empty and is a proper subset of $\Re(H)$, i.e., $\{0\} \subset \Re(H) \cap \Re(X) \subset \Re(H)$. A hypothesis $H\beta = g$ is completely testable if $\{0\} \subset \Re(H) \cap \Re(H) \subset \Re(X)$. Peixoto also demonstrated a method for converting a partially testable hypothesis to one that is completely testable so that the usual method for obtaining sums of squares for the hypothesis from the results of the fitted model can be used. The method replaces H_p in the partially testable hypothesis $H_p\beta = g_p$ by a matrix H whose rows are a basis for the intersection of the row space of H_p and the row space of X. A corresponding conversion of the null hypothesis values from g_p to g is also made. A sum of squares for the completely testable hypothesis can then be computed (see function `imsls_f_hypothesis_scph`, page 101). The sum of squares that is computed for the hypothesis $H\beta = g$ equals the difference in the error sums of squares from two fitted models—the restricted model with the partially testable hypothesis $H_p\beta = g_p$ and the unrestricted model.

For the general case of the multivariate model $Y = X\beta + \varepsilon$ with possible linear equality restrictions on the regression parameters, `imsls_f_hypothesis_partial` converts the partially testable hypothesis $H_p\beta = g_p$ to a completely testable hypothesis $H\beta U = G$. For the case of the linear model with linear equality restrictions, the definitions of the estimable functions, nontestable hypothesis, partially testable hypothesis, and completely testable hypothesis are similar to those previously given for the unrestricted model with the exception that $\Re(X)$ is replaced by $\Re(R)$ where R is the upper triangular matrix based on the linear equality restrictions. The nonzero rows of R form a basis for the rowspace of the matrix $(X^T, A^T)^T$. The rows of H form an orthonormal basis for the intersection of two subspaces—the subspace spanned by the rows of H_p and the subspace spanned by the rows of R. The algorithm used for computing the intersection of these two subspaces is based on an algorithm for computing angles between linear subspaces due to Björk and Golub (1973). (See also Golub and Van Loan 1983, pp. 429–430). The method is closely related to a canonical correlation analysis discussed by Kennedy and Gentle (1980, pp. 561–565). The algorithm is as follows:

1. Compute a QR factorization of

$$H_P^T$$

with column permutations so that

$$H_P^T = Q_1 R_1 P_1^T$$

Here, P_1 is the associated permutation matrix that is also an orthogonal matrix. Determine the rank of H_p as the number of nonzero diagonal elements of R_1, for example n_1. Partition $Q_1 = (Q_{11}, Q_{12})$ so that Q_{11} is the first n_1 column of Q_1. Set `rank_hp` = n.

2. Compute a QR factorization of the transpose of the R matrix (input through `regression_info`) with column permuations so that

$$R^T = Q_2 R_2 P_2^T$$

Determine the rank of R from the number of nonzero diagonal elements of R, for example n_2. Partition $Q_2 = (Q_{21}, Q_{22})$ so that Q_{21} is the first n_2 columns of Q_2.

3. Form

$$A = Q_{11}^T Q_{21}$$

4. Compute the singular values of A

$$\sigma_1 \ge \sigma_2 \ge \ldots \ge \sigma_{\min(n_1, n_2)}$$

and the left singular vectors W of the singular value decomposition of A so that

$$W^T A V = \operatorname{diag}\left(\sigma_1, \ldots \sigma_{\min(n_1, n_2)}\right)$$

If $\sigma_1 < 1$, then the dimension of the intersection of the two subspaces is $s = 0$. Otherwise, assume the dimension of the intersection to be s if $\sigma_s = 1 > \sigma_{s+1}$. Set `nh` = s.

5. Let W_1 be the first s columns of W. Set $H = (Q_1 W_1)^T$.

6. Assume R_{11} to be a `nhp` by `nhp` matrix related to R_1 as follows: If `nhp` < *n_parameters*, R_{11} equals the first `nhp` rows of R_1. Otherwise, R_{11} contains R_1 in its first *n_parameters* rows and zeros in the remaining rows. Compute a solution Z to the linear system

$$R_{11}^T Z = P_1^T G_p$$

If this linear system is delcared inconsistent, an error message with error code equal to 2 is issued.

7. Partition

$$Z^T = \left(Z_1^T, Z_2^T\right)$$

so that Z_1 is the first n_1 rows of Z. Set

$$G = W_1^T Z_1$$

The degrees of freedom (`nh`) classify the hypothesis $H_p \beta U = G_p$ as nontestable (`nh` = 0), partially testable (0 < `nh` < `rank_hp`), or completely testable (0 < `nh` = `rank_hp`).

For further details concerning the algorithm, see Sallas and Lionti (1988).

Example

A one-way analysis-of-variance model discussed by Peixoto (1986) is fitted to data. The model is

$$y_{ii} = \mu + \alpha_i + \varepsilon_{ii} \qquad (i, j) = (1, 1)\ (2, 1)\ (2, 2)$$

The model is fitted using function `imsls_f_regression` (page 64). The partially testable hypothesis

$$H_0 : \substack{\alpha_1 = 5 \\ \alpha_2 = 3}$$

is converted to a completely testable hypothesis.

```
#include <imsls.h>
#define N_ROWS 3
#define N_INDEPENDENT 1
#define N_DEPENDENT 1
#define N_PARAMETERS 3
#define NHP 2

main() {
    Imsls_f_regression *info;
    int    n_class = 1;
    int    n_continuous = 0;
    int    nh, nreg, rank_hp;
    float  *coefficients, *x, *g, *h;
    static float   z[N_ROWS*N_INDEPENDENT] = { 1, 2, 2 };
    static float   y[] = {17.3, 24.1, 26.3};
    static float   gp[] = {5, 3};
    static float   hp[NHP*N_PARAMETERS] = {0, 1, 0,
                                           0, 0, 1};

    nreg = imsls_f_regressors_for_glm(N_ROWS, z,
        n_class, n_continuous,
        IMSLS_REGRESSORS, &x, 0);

    coefficients = imsls_f_regression(N_ROWS, nreg, x, y,
        IMSLS_N_DEPENDENT, N_DEPENDENT,
        IMSLS_REGRESSION_INFO, &info,
        0);

    nh = imsls_f_hypothesis_partial(info, NHP, hp,
        IMSLS_GP, gp,
        IMSLS_H_MATRIX, &h,
        IMSLS_G, &g,
        IMSLS_RANK_HP, &rank_hp, 0);

    if (nh == 0) {
        printf("Nontestable Hypothesis\n");
    } else if (nh < rank_hp) {
        printf("Partially Testable Hypothesis\n");
    } else {
        printf("Completely Testable Hypothesis\n");
    }

    imsls_f_write_matrix("H Matrix", nh, N_PARAMETERS, h, 0);

    imsls_f_write_matrix("G", nh, N_DEPENDENT, g, 0);

    free(coefficients);
    free(info);
    free(x);
    free(h);
    free(g);
}
```

Output

```
Partially Testable Hypothesis

             H Matrix
         1           2           3
    0.0000      0.7071     -0.7071

     G
     1.414
```

Warning Errors

`IMSLS_HYP_NOT_CONSISTENT`	The hypothesis is inconsistent within the computed tolerance.

hypothesis_scph

Computes the matrix of sums of squares and crossproducts for the multivariate general linear hypothesis $H\beta U = G$ given the regression fit.

Synopsis

#include `<imsls.h>`

float `*imsls_f_hypothesis_scph`
(*Imsls_f_regression* `*regression_info`, *int* `nh`, *float* `h[]`,
float `*dfh`, ..., 0)

The type *double* function is `imsls_d_hypothesis_scph`.

Required Argument

Imsls_f_regression `*regression_info` (Input)
: Pointer to a structure of type *Imsls_f_regression* containing information about the regression fit. See function `imsls_f_regression` (page 64).

int `nh` (Input)
: Number of rows in the hypothesis matrix, `h`.

float `h[]` (Input)
: The *H* array of size `nh` by *n_coefficients* with each row corresponding to a row in the hypothesis and containing the constants that specify a linear combination of the regression coefficients. Here, *n_coefficients* is the number of coefficients in the fitted regression model.

float `*dfh` (Output)
: Degrees of freedom for the sums of squares and crossproducts matrix. This is equal to the rank of input matrix `h`.

Return Value

Array of size `nu` by `nu` containing the sums of squares and crossproducts attributable to the hypothesis.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_regression_scph`
(*Imsls_f_regression* `*regression_info`, *int* `nh`, *float* `h[]`,
float `*dfh`,
`IMSLS_G`, *float* `g[]`,
`IMSLS_U`, *int* `nu`, *float* `u[]`,
`IMSLS_RETURN_USER`, `scph[]`,
0)

Optional Arguments

`IMSLS_G`, *float* `g[]` (Input)
: Array of size `nh` by `nu` containing the *G* matrix, the null hypothesis values. By default, each value of *G* is equal to 0.

`IMSLS_U`, *int* `nu`, *float* `u[]` (Input)
: Argument `nu` is the number of linear combinations of the dependent variables to be considered. The value `nu` must be greater than 0 and less than or equal to *n_dependent*.

Argument `u` contains the *n_dependent* by `nu` *U* matrix for the test $H_p\beta U = G_p$.

Default: `nu` = *n_dependent* and `u` is the identity matrix

`IMSLS_RETURN_USER`, *float* `scph[]` (Output)
If specified, the sums of squares and crossproducts matrix is stored in array `scph` provided by the user, where `scph` is of size `nu` by `nu`.

Description

Function `imsls_f_hypothesis_scph` computes the matrix of sums of squares and crossproducts for the general linear hypothesis $H\beta U = G$ for the multivariate general linear model $Y = X\beta + \varepsilon$.

The rows of H must be linear combinations of the rows of R, i.e., $H\beta = G$ must be completely testable. If the hypothesis is not completely testable, function `imsls_f_hypothesis_partial` (page 96) can be used to construct an equivalent completely testable hypothesis.

Computations are based on an algorithm discussed by Kennedy and Gentle (1980, p. 317) that is extended by Sallas and Lionti (1988) for mulitvariate non-full rank models with possible linear equality restrictions. The algorithm is as follows:

1. Form $W = H\hat{\beta}U - G$.

2. Find C as the solution of $R^T C = H^T$. If the equations are declared inconsistent within a computed tolerance, a warning error message is issued that the hypothesis is not completely testable.

3. For all rows of R corresponding to restrictions, i.e., containing negative diagonal elements from a restricted least-squares fit, zero out the corresponding rows of C, i.e., from DC.

4. Decompose DC using Householder transformations and column pivoting to yield a square, upper triangular matrix T with diagonal elements of nonincreasing magnitude and permutation matrix P such that

$$DCP = Q\begin{bmatrix} T \\ 0 \end{bmatrix}$$

   where Q is an orthogonal matrix.

5. Determine the rank of T, say r. If $t_{11} = 0$, then $r = 0$. Otherwise, the rank of T is r if

$$| t_{rr} | > | t_{11} | \varepsilon \geq | t_{r+1,\, r+1} |$$

   where $\varepsilon = 10.0 \times$ `imsls_f_machine`(4)
   ($10.0 \times$ `imsls_d_machine`(4) for the double-precision version).

   Then, zero out all rows of T below r. Set the degrees of freedom for the hypothesis, `dfh`, to r.

6. Find V as a solution to $T^T V = P^T W$. If the equations are inconsistent, a warning error message is issued that the hypothesis is inconsistent within a computed tolerance, i.e., the linear system

$$H\beta U = G$$

$$A\beta = Z$$

does not have a solution for β.

Form $V^T V$, which is the required matrix of sum of squares and crossproducts, `scph`.

In general, the two warning errors described above are serious user errors that require the user to correct the hypothesis before any meaningful sums of squares from this function can be computed. However, in some cases, the user may know the hypothesis is consistent and completely testable, but the checks in `imsls_f_hypothesis_scph` are too tight. For this reason, `imsls_f_hypothesis_scph` continues with the calculations.

Function `imsls_f_hypothesis_scph` gives a matrix of sums of squares and crossproducts that could also be obtained from separate fittings of the two models:

$$Y^{\neq} = X\beta^{\neq} + \varepsilon^{\neq} \qquad (1)$$

$$A\beta^{\neq} = Z^{\neq}$$

$$H\beta^{\neq} = G$$

and

$$Y^{\neq} = X\beta^{\neq} + \varepsilon^{\neq} \qquad (2)$$

$$A\beta^{\neq} = Z^{\neq}$$

where $Y^{\neq} = YU$, $\beta^{\neq} = \beta U$, $\varepsilon^{\neq} = \varepsilon U$, and $Z^{\neq} = ZU$. The error sum of squares and crossproducts matrix for (1) minus that for (2) is the matrix sum of squares and crossproducts output in `scph`. Note that this approach avoids the question of testability.

Example

The data for this example are from Maindonald (1984, pp. 203–204). A multivariate regression model containing two dependent variables and three independent variables is fit using function `imsls_f_regression` and the results stored in the structure info. The sum of squares and crossproducts matrix, `scph`, is then computed by calling `imsls_f_hypothesis_scph` for the test that the third independent variable is in the model (determined by the specification of `h`). The degrees of freedom for `scph` also is computed.

```
#include <imsls.h>
main()
{
    Imsls_f_regression *info;
    float   *coefficients, *scph;
    float   dfh;
    float   x[]     = { 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, 1.0,
                     -5.0, 4.0,
                      6.0, 10.0,
                      5.0, 5.0,
                      5.0, -2.0,
                     -2.0, 4.0,
                      0.0, -6.0,
                      8.0, 2.0,
                      3.0, 0.0 };
    int     n_observations = 9;
    int     n_independent = 3;
    int     n_dependent = 2;
    int     nh = 1;
    float h[]       = { 0, 0, 0, 1 };

    coefficients = imsls_f_regression(n_observations, n_independent,
        x, y,
        IMSLS_N_DEPENDENT, n_dependent,
        IMSLS_REGRESSION_INFO, &info,
        0);

    scph = imsls_f_hypothesis_scph(info, nh, h, &dfh, 0);

    printf("Degrees of Freedom Hypothesis = %4.0f\n", dfh);

    imsls_f_write_matrix("Sum of Squares and Crossproducts",
        n_dependent, n_dependent, scph,
        IMSLS_NO_COL_LABELS, IMSLS_NO_ROW_LABELS,
        0);

}
```

Output

```
Degrees of Freedom Hypothesis =    1

Sum of Squares and Crossproducts
            100         -40
            -40          16
```

Warning Errors

`IMSLS_HYP_NOT_TESTABLE`	The hypothesis is not completely testable within the computed tolerance. Each row of "h" must be a linear combination of the rows of "r".
`IMSLS_HYP_NOT_CONSISTENT`	The hypothesis is inconsistent within the computed tolerance.

hypothesis_test

Performs tests for a multivariate general linear hypothesis $H\beta U = G$ given the hypothesis sums of squares and crossproducts matrix S_H.

Synopsis

#include `<imsls.h>`

float `imsls_f_hypothesis_test` (*Imsls_f_regression* `*regression_info`,
float `dfh`, *float* `*scph`, ..., 0)

The type *double* function is `imsls_d_hypothesis_test`.

Required Argument

Imsls_f_regression `*regression_info` (Input)
Pointer to a structure of type *Imsls_f_regression* containing information about the regression fit. See function `imsls_f_regression`.

float `dfh` (Input)
Degrees of freedom for the sums of squares and crossproducts matrix.

float `*scph` (Input)
Array of size `nu` by `nu` containing S_H, the sums of squares and crossproducts attributable to the hypothesis.

Return Value

The *p*-value corresponding to Wilks' lambda test.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_hypothesis_test` (*Imsls_f_regression* `*regression_info`,
float `dfh`, *float* `*scph`,
`IMSLS_U`, *int* `nu`, *float* `u[]`,
`IMSLS_WILK_LAMBDA`, *float* `*value`, *float* `*p_value`,
`IMSLS_ROY_MAX_ROOT`, *float* `*value`, *float* `*p_value`,
`IMSLS_HOTELLING_TRACE`, *float* `*value`, *float* `*p_value`,
`IMSLS_PILLAI_TRACE`, *float* `*value`, *float* `*p_value`,
0)

Optional Arguments

`IMSLS_U`, *int* `nu`, *float* `u[]` (Input)
Argument `nu` is the number of linear combinations of the dependent variables to be considered. The value `nu` must be greater than 0 and less than or equal to *n_dependent*. Argument `u` contains the *n_dependent* by `nu` U matrix for the test $H_p\beta U = G_p$.
Default: `nu` = *n_dependent* and `u` is the identity matrix

`IMSLS_WILK_LAMBDA`, *float* `*value`, *float* `*p_value` (Output)
Wilk's lamda and *p*-value.

`IMSLS_ROY_MAX_ROOT`, *float* `*value`, *float* `*p_value` (Output)
Roy's maximum root criterion and *p*-value.

`IMSLS_HOTELLING_TRACE`, *float* `*value`, *float* `*p_value` (Output)
Hotelling's trace and *p*-value.

`IMSLS_PILLAI_TRACE`, *float* `*value`, *float* `*p_value` (Output)
Pillai's trace and *p*-value.

Description

Function `imsls_f_hypothesis_test` computes test statistics and *p*-values for the general linear hypothesis $H\beta U = G$ for the multivariate general linear model.

The hypothesis sum of squares and crossproducts matrix input in `scph` is

$$S_H = \left(H\hat{\beta}U - G\right)^T \left(C^T DC\right)^- \left(H\hat{\beta}U - G\right)$$

where C is a solution to $R^T C = H$ and where D is a diagonal matrix with diagonal elements

$$d_{ii} = \begin{cases} 1 & \text{if } r_{ii} > 0 \\ 0 & \text{otherwise} \end{cases}$$

See the section "Linear Dependence and the *R* Matrix" in the introduction (page 48).

The error sum of squares and crossproducts matrix for the model $Y = X\beta + \varepsilon$ is

$$\left(Y - X\hat{\beta}\right)^T \left(Y - X\hat{\beta}\right)$$

which is input in `regression_info`. The error sum of squares and crossproducts matrix for the hypothesis $H\beta U = G$ computed by `imsls_f_hypothesis_test` is

$$S_E = U^T \left(Y - X\hat{\beta}\right)^T \left(Y - X\hat{\beta}\right) U$$

Let p equal the order of the matrices S_E and S_H, i.e.,

$$p = \begin{Bmatrix} \text{NU} & \text{if NU} > 0 \\ \text{NDEP} & \text{otherwise} \end{Bmatrix}$$

Let q (stored in `dfh`) be the degrees of freedom for the hypothesis. Let v (input in `regression_info`) be the degrees of freedom for error. Function `imsls_f_hypothesis_test` computed three test statistics based on eigenvalues λ_i ($i = 1, 2, \ldots, p$) of the generalized eigenvalue problem $S_H x = \lambda S_E x$. These test statistics are as follows:

Wilk's lambda

$$\Lambda = \frac{\det(S_E)}{\det(S_H + S_E)} = \prod_{i=1}^{p} \frac{1}{1+\lambda_i}$$

The associated p-value is based on an approximation discussed by Rao (1973, p. 556). The statistic

$$F = \frac{ms - pq/2+1}{pq} \frac{1-\Lambda^{1/s}}{\Lambda^{1/s}}$$

has an approximate F distribution with pq and $ms - pq / 2 + 1$ numerator and denominator degrees of freedom, respectively, where

$$s = \begin{cases} 1 & \text{if } p = 1 \text{ or } q = 1 \\ \sqrt{\dfrac{p^2q^2-4}{p^2+q^2-5}} & \text{otherwise} \end{cases}$$

and

$$m = \upsilon - \frac{(p+q-1)}{2}$$

The F test is exact if min $(p, q) \le 2$ (Kshirsagar, 1972, Theorem 4, p. 299–300).

Roy's maximum root

$$c = \max \lambda_i \qquad \text{over all } i$$

where c is output as `value`. The p-value is based on the approximation

$$F = \frac{\upsilon + q - s}{s} c$$

where s = max (p, q) has an approximate F distribution with s and $\upsilon + q - s$ numerator and denominator degrees of freedom, respectively. The F test is exact if $s = 1$; the p-value is also exact. In general, the value output in `p_value` is lower bound on the actual p-value.

Hotelling's trace

$$U = \text{tr}\left(HE^{-1}\right) = \sum_{i=1}^{p} \lambda_i$$

U is output as `value`. The p-value is based on the approximation of McKeon (1974) that supersedes the approximation of Hughes and Saw (1972). McKeon's approximation is also discussed by Seber (1984, p. 39). For

$$b = 4 + \frac{pq+2}{\dfrac{(\upsilon+q-p-1)(\upsilon-1)}{(\upsilon-p-3)(\upsilon-p)}}$$

the *p*-value is based on the result that

$$F = \frac{b(\upsilon - p - 1)}{(b-2)\,pq} U$$

has an approximate *F* distribution with *pq* and *b* degrees of freedom. The test is exact if min $(p, q) = 1$. For $\upsilon \le p + 1$, the approximation is not valid, and `p_value` is set to NaN.

These three test statistics are valid when S_E is positive definite. A necessary condition for S_E to be positive definite is $\upsilon \ge p$. If S_E is not positive definite, a warning error message is issued, and both `value` and `p_value` are set to NaN.

Because the requirement $\upsilon \ge p$ can be a serious drawback, `imsls_f_hypothesis_test` computes a fourth test statistic based on eigenvalues θ_i ($i = 1, 2, \ldots, p$) of the generalized eigenvalue problem $S_H w = \theta(S_H + S_E)\, w$. This test statistic requires a less restrictive assumption—$S_H + S_E$ is positive definite. A necessary condition for $S_H + S_E$ to be positive definite is $\upsilon + q \ge p$. If S_E is positive definite, `imsls_f_hypothesis_test` avoids the computation of the generalized eigenvalue problem from scratch. In this case, the eigenvalues θ_i are obtained from λ_i by

$$\theta_i = \frac{\lambda_i}{1 + \lambda_i}$$

The fourth test statistic is as follows:

Pillai's trace

$$V = \operatorname{tr}\left[S_H \left(S_H + S_E\right)^{-1} \right] = \sum_{i=1}^{p} \theta_i$$

V is output as `value`. The *p*-value is based on an approximation discussed by Pillai (1985). The statistic

$$F = \frac{2n + s + 1}{2m + s + 1} \frac{V}{s - V}$$

has an approximate *F* distribution with $s(2m + s + 1)$ and $s(2n + s + 1)$ numerator and denominator degrees of freedom, respectively, where

$$s = \min(p, q)$$

$$m = ½(|p - q| - 1)$$

$$n = ½(\upsilon - p - 1)$$

The *F* test is exact if min $(p, q) = 1$.

Examples

Example 1

The data for this example are from Maindonald (1984, p. 203−204). A multivariate regression model containing two dependent variables and three independent variables is fit using function `imsls_f_regression` and the results stored in the structure `regression_info`. The sum of squares and crossproducts matrix, `scph`, is then computed with a call to `imsls_f_hypothesis_scph` for the test that the third independent variable is in the model (determined by specification of `h`). Finally, function `imsls_f_hypothesis_test` is called to compute the *p*-value for the test statistic (Wilk's lambda).

```
#include <imsls.h>
main()
{
    Imsls_f_regression *info;
    float   *coefficients, *scph;
    float   dfh, p_value;
    float   x[]     = { 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, 1.0,
                       -5.0, 4.0,
                        6.0, 10.0,
                        5.0, 5.0,
                        5.0, -2.0,
                       -2.0, 4.0,
                        0.0, -6.0,
                        8.0, 2.0,
                        3.0, 0.0 };
    int     n_observations = 9;
    int     n_independent = 3;
    int     n_dependent = 2;
    int     nh = 1;
    float h[]       = { 0, 0, 0, 1 };

    coefficients = imsls_f_regression(n_observations, n_independent,
        x, y,
        IMSLS_N_DEPENDENT, n_dependent,
        IMSLS_REGRESSION_INFO, &info,
        0);

    scph = imsls_f_hypothesis_scph(info, nh, h, &dfh, 0);

    p_value = imsls_f_hypothesis_test(info, dfh, scph, 0);

    printf("P-value = %10.6f\n", p_value);

}
```

Output

```
P-value =   0.000010
```

Example 2

This example is the same as the first example, but more statistics are computed. Also, the *U* matrix, `u`, is explicitly specified as the identity matrix (which is the same default configuration of *U*).

```
#include <imsls.h>
main()
{
    Imsls_f_regression *info;
    float   *coefficients, *scph;
    float   dfh, p_value;
    float   x[]     = { 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, 1.0,
                       -5.0, 4.0,
                        6.0, 10.0,
                        5.0, 5.0,
                        5.0, -2.0,
                       -2.0, 4.0,
                        0.0, -6.0,
                        8.0, 2.0,
                        3.0, 0.0 };
    int     n_observations = 9;
    int     n_independent = 3;
    int     n_dependent = 2;
    int     nh = 1;
    float   h[]     = { 0, 0, 0, 1 };
    int     nu = 2;
    float   u[4]={1, 0, 0, 1};
    float   v1, v2, v3, v4, p1, p2, p3, p4;

    coefficients = imsls_f_regression(n_observations, n_independent,
        x, y,
        IMSLS_N_DEPENDENT, n_dependent,
        IMSLS_REGRESSION_INFO, &info,
        0);

    scph = imsls_f_hypothesis_scph(info, nh, h, &dfh, 0);

    p_value = imsls_f_hypothesis_test(info, dfh, scph,
        IMSLS_U, nu, u,
        IMSLS_WILK_LAMBDA, &v1, &p1,
        IMSLS_ROY_MAX_ROOT, &v2, &p2,
        IMSLS_HOTELLING_TRACE, &v3, &p3,
        IMSLS_PILLAI_TRACE, &v4, &p4,
        0);
    printf("Wilk      value = %10.6f   p-value = %10.6f\n", v1, p1);
    printf("Roy       value = %10.6f   p-value = %10.6f\n", v2, p2);
    printf("Hotelling value = %10.6f   p-value = %10.6f\n", v3, p3);
    printf("Pillai    value = %10.6f   p-value = %10.6f\n", v4, p4);
}
```

Output

```
Wilk      value =   0.003149   p-value =   0.000010
Roy       value = 316.600861   p-value =   0.000010
Hotelling value = 316.600861   p-value =   0.000010
Pillai    value =   0.996851   p-value =   0.000010
```

Warning Errors

`IMSLS_SINGULAR_1`	"u"*"scpe"*"u" is singular. Only Pillai's trace can be computed. Other statistics are set to NaN.

Fatal Errors

`IMSLS_NO_STAT_1`	"scpe" + "scph" is singular. No tests can be computed.
`IMSLS_NO_STAT_2`	No statistics can be computed. Iterations for eigenvalues for the generalized eigenvalue problem "scph"*x = (lambda)*("scph"+"scpe")*x failed to converge.
`IMSLS_NO_STAT_3`	No statistics can be computed. Iterations for eigenvalues for the generalized eigenvalue problem "scph" *x = (lambda)*("scph"+"u"*"scpe"*"u")*x failed to converge.
`IMSLS_SINGULAR_2`	"u"*"scpe"*"u" + "scph" is singular. No tests can be computed.
`IMSLS_SINGULAR_TRI_MATRIX`	The input triangular matrix is singular. The index of the first zero diagonal element is equal to #.

regression_selection

Selects the best multiple linear regression models.

Synopsis

#include `<imsls.h>`

void `imsls_f_regression_selection` (*int* `n_rows`, *int* `n_candidate`,
float `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_regression_selection`.

Required Arguments

int `n_rows` (Input)
Number of observations or rows in `x` and `y`.

int `n_candidate` (Input)
Number of candidate variables (independent variables) or columns in `x`. `n_candidate` must be greater than 2.

float `x[]` (Input)
Array of size `n_rows` × `n_candidate` containing the data for the candidate variables.

float `y[]` (Input)
Array of length `n_rows` containing the responses for the dependent variable.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_regression_selection` (*int* `n_rows`, *int* `n_candidate`,
float `x[]`, *float* `y[]`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_PRINT`, *or*
`IMSLS_NO_PRINT`,
`IMSLS_WEIGHTS`, *float* `weights[]`,
`IMSLS_FREQUENCIES`, *float* `frequencies[]`,
`IMSLS_R_SQUARED`, *int* `max_subset_size`, *or*
`IMSLS_ADJ_R_SQUARED`, *or*
`IMSLS_MALLOWS_CP`,
`IMSLS_MAX_N_BEST`, *int* `max_n_best`,
`IMSLS_MAX_N_GOOD_SAVED`, *int* `max_n_good_saved`,
`IMSLS_CRITERIONS`, *int* `**index_criterions`,
float `**criterions`,
`IMSLS_CRITERIONS_USER`, *int* `index_criterions[]`,
float `criterions[]`,
`IMSLS_INDEPENDENT_VARIABLES`, *int* `**index_variables`,
int `**independent_variables`,
`IMSLS_INDEPENDENT_VARIABLES_USER`,
int `index_variables[]`,
int `independent_variables[]`,
`IMSLS_COEF_STATISTICS`, *int* `**index_coefficients`,
float `**coefficients`,
`IMSLS_COEF_STATISTICS_USER`, *int* `index_coefficients[]`,
float `coefficients[]`,

IMSLS_INPUT_COV, *int* n_observations, *float* cov[],
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
The column dimension of x.
Default: x_col_dim = n_candidate

IMSLS_PRINT
Printing is performed. This is the default.
or

IMSLS_NO_PRINT
Printing is not performed.

IMSLS_WEIGHTS, *float* weights[] (Input)
Array of length n_rows containing the weight for each row of x.
Default: weights[] = 1

IMSLS_FREQUENCIES, *float* frequencies[] (Input)
Array of length n_rows containing the frequency for each row of x.
Default: frequencies[] = 1

IMSLS_R_SQUARED, *int* max_subset_size (Input)
The R^2 criterion is used, where subset sizes
1, 2, ..., max_subset_size are examined.
This option is the default with max_subset_size = n_candidate.
or

IMSLS_ADJ_R_SQUARED
The adjusted R^2 criterion is used, where subset sizes
1, 2, ..., n_candidate are examined.
or

IMSLS_MALLOWS_CP
Mallows C_p criterion is used, where subset sizes
1, 2, ..., n_candidate are examined.

IMSLS_MAX_N_BEST, *int* max_n_best (Input)
Number of best regressions to be found. If the R^2 criterions are selected, the max_n_best best regressions for each subset size examined are found. If the adjusted R^2 or Mallows C_p criterion is selected, the max_n_best overall regressions are found.
Default: max_n_best = 1

IMSLS_MAX_N_GOOD_SAVED, *int* max_n_good_saved (Input)
Maximum number of good regressions of each subset size to be saved in finding the best regressions. Argument max_n_good_saved must be greater than or equal to max_n_best. Normally, max_n_good_saved should be less than or equal to 10. It doesn't ever need to be larger than the maximum number of subsets for any subset size. Computing time

required is inversely related to `max_n_good_saved`.
Default: `max_n_good_saved` = 10

`IMSLS_CRITERIONS`, *int* `**index_criterions`, *float* `**criterions` (Output)
Argument `index_criterions` is the address of a pointer to the internally allocated array of length *nsize* + 1(where *nsize* is equal to `max_subset_size` if optional argument `IMSLS_R_SQUARED` is specified; otherwise, *nsize* is equal to `n_candidate`) containing the locations in `criterions` of the first element for each subset size. For `I` = 0, 1, ..., *nsize* –1, element numbers `index_criterions[I]`, `index_criterions[I]` + 1, ..., `index_criterions[I + 1]` – 1 of `criterions` correspond to the (`I` + 1)-st subset size. Argument `criterions` is the address of a pointer to the internally allocated array of length max (`index_criterions` [*nsize*] – 1 , `n_candidate`) containing in its first `index_criterions` [*nsize*] – 1 elements the criterion values for each subset considered, in increasing subset size order.

`IMSLS_CRITERIONS_USER`, *int* `index_criterions[]`, *float* `criterions[]` (Output)
Storage for arrays `index_criterions` and `criterions` is provided by the user. An upper bound on the length of `criterions` is max(`max_n_good_saved` × *nsize*, `n_candidate`). See `IMSLS_CRITERIONS`.

`IMSLS_INDEPENDENT_VARIABLES`, *int* `**index_variables`, *int* `**independent_variables` (Output)
Argument `index_variables` is the address of a pointer to the internally allocated array of length *nsize* + 1 (where *nsize* is equal to `max_subset_size` if optional argument `IMSLS_R_SQUARED` is specified; otherwise, *nsize* is equal to `n_candidate`) containing the locations in `independent_variables` of the first element for each subset size. For `I` = 0, 1, ..., *nsize* – 1, element numbers `index_variables[I]`, `index_variables[I]` + 1, ..., `index_variables[I + 1]` – 1 of `independent_variables` correspond to the (`I+1`)-st subset size. Argument `independent_variables` is the address of a pointer to the internally allocated array of length `index_variables` [*nsize*] – 1 containing the variable numbers for each subset considered and in the same order as in `criterions`.

`IMSLS_INDEPENDENT_VARIABLES_USER`, *int* `index_variables[]`, *int* `independent_variables[]` (Output)
Storage for arrays `index_variables` and `independent_variables` is provided by the user. An upper bound for the length of `independent_variables` is as follows:

$$\frac{\texttt{max\_n\_good\_saved} \times nsize \times (nsize + 1)}{2}$$

where *nsize* is equal to `max_subset_size`.

See `IMSLS_INDEPENDENT_VARIABLES`.

`IMSLS_COEF_STATISTICS`, *int* `**index_coefficients`, *float* `**coefficients` (Output)

Argument `index_coefficients` is the address of a pointer to the internally allocated array of length *ntbest* + 1 containing the locations in `coefficients` or the first row for each of the best regressions. Here, *ntbest* is the total number of best regression found and is equal to `max_subset_size` × `max_n_best` if `IMSLS_R_SQUARED` is specified, equal to `max_n_best` if either `IMSLS_MALLOWS_CP` or `IMSLS_ADJ_R_SQUARED` is specified, and equal to `max_n_best` × `n_candidate`, otherwise. For `I` = 0, 1, ..., *ntbest* – 1, rows `index_coefficients[I]`, `index_coefficients[I]` + 1, ..., `index_coefficients[I` + 1] – 1 of `coefficients` correspond to the (`I` + 1)-st regression. Argument `coefficients` is the address of a pointer to the internally allocated array of size (`index_coefficients` [*ntbest*] – 1) × 5 containing statistics relating to the regression coefficients of the best models. Each row corresponds to a coefficient for a particular regression. The regressions are in order of increasing subset size. Within each subset size, the regressions are ordered so that the better regressions appear first. The statistic in the columns are as follows (inferences are conditional on the selected model):

Column	Description
0	variable number
1	coefficient estimate
2	estimated standard error of the estimate
3	*t*-statistic for the test that the coefficient is 0
4	*p*-value for the two-sided *t* test

`IMSLS_COEF_STATISTICS_USER`, *int* `index_coefficients[]`, *float* `coefficients[]` (Output)

Storage for arrays `index_coefficients` and `coefficients` is provided by the user. See `IMSLS_COEF_STATISTICS`.

`IMSLS_INPUT_COV`, *int* `n_observations`, *float* `cov[]` (Input)

Argument `n_observations` is the number of observations associated with array `cov`. Argument `cov` is an (`n_candidate` + 1) by (`n_candidate` + 1) array containing a variance-covariance or sum of squares and crossproducts matrix, in which the last column must correspond to the dependent variable. Array `cov` can be computed using `imsls_f_covariances`. Arguments `x` and `y`, and optional arguments `frequencies` and `weights` are not accessed when this option is specified. Normally, `imsls_f_regression_selection` computes `cov` from the input data matrices `x` and `y`. However, there may be cases when the user will wish to calculate the covariance matrix and manipulate it before calling `imsls_f_regression_selection`. See the description section below for a discussion of such cases.

Description

Function `imsls_f_regression_selection` finds the best subset regressions for a regression problem with `n_candidate` independent variables. Typically, the intercept is forced into all models and is not a candidate variable. In this case, a sum of squares and crossproducts matrix for the independent and dependent variables corrected for the mean is computed internally. There may be cases when it is convenient for the user to calculate the matrix; see the description of optional argument `IMSLS_INPUT_COV`.

"Best" is defined, on option, by one of the following three criteria:

- R^2 (in percent)

$$R^2 = 100\,(1 - \frac{\mathrm{SSE}_p}{\mathrm{SST}})$$

- R_a^2 (adjusted R^2 in percent)

$$R_a^2 = 100\left[1 - (\frac{n-1}{n-p})\frac{\mathrm{SSE}_p}{\mathrm{SST}}\right]$$

  Note that maximizing the criterion is equivalent to minimizing the residual mean square:

$$\frac{\mathrm{SSE}_p}{(n-p)}$$

- Mallows' C_p statistic

$$C_p = \frac{\mathrm{SSE}_p}{s^2_{\texttt{n\_candidate}}} + 2p - n$$

Here, n is equal to the sum of the frequencies (or `n_rows` if `IMSLS_FREQUENCIES` is not specified) and SST is the total sum of squares. SSE_p is the error sum of squares in a model containing p regression parameters including β_0 (or $p - 1$ of the `n_candidate` candidate variables). Variable

$$s^2_{\texttt{n\_candidate}}$$

is the error mean square from the model with all `n_candidate` variables in the model. Hocking (1972) and Draper and Smith (1981, pp. 296–302) discuss these criteria.

Function `imsls_f_regression_selection` is based on the algorithm of Furnival and Wilson (1974). This algorithm finds `max_n_good_saved` candidate regressions for each possible subset size. These regressions are used to identify a set of best regressions. In large problems, many regressions are not computed. They may be rejected without computation based on results for other subsets; this yields an efficient technique for considering all possible regressions.

There are cases when the user may want to input the variance-covariance matrix rather than allow the function `imsls_f_regression_selection` to calculate it. This can be accomplished using optional argument `IMSLS_INPUT_COV`. Three situations in which the user may want to do this are as follows:

1. The intercept is not in the model. A raw (uncorrected) sum of squares and crossproducts matrix for the independent and dependent variables is required. Argument `n_observations` must be set to 1 greater than the number of observations. Form $A^T A$, where $A = [A, Y]$, to compute the raw sum of squares and crossproducts matrix.
2. An intercept is a candidate variable. A raw (uncorrected) sum of squares and crossproducts matrix for the constant regressor (= 1.0), independent, and dependent variables is required for `cov`. In this case, `cov` contains one additional row and column corresponding to the constant regressor. This row/column contains the sum of squares and crossproducts of the constant regressor with the independent and dependent variables. The remaining elements in `cov` are the same as in the previous case. Argument `n_observations` must be set to 1 greater than the number of observations.
3. There are *m* variables to be forced into the models. A sum of squares and crossproducts matrix adjusted for the *m* variables is required (calculated by regressing the candidate variables on the variables to be forced into the model). Argument `n_observations` must be set to *m* less than the number of observations.

Programming Notes

Function `imsls_f_regression_selection` can save considerable CPU time over explicitly computing all possible regressions. However, the function has some limitations that can cause unexpected results for users who are unaware of the limitations of the software.

1. For `n_candidate` $+ 1 > -\log_2(\varepsilon)$, where ε is `imsls_f_machine`(4) (`imsls_d_machine`(4) for double precision; see Chapter 14), some results can be incorrect. This limitation arises because the possible models indicated (the model numbers 1, 2, ..., $2^{\text{n\_candidate}}$) are stored as floating-point values; for sufficiently large `n_candidate`, the model numbers cannot be stored exactly. On many computers, this means `imsls_f_regression_selection` (for `n_candidate` > 24) and `imsls_d_regression_selection` (for `n_candidate` > 49) can produce incorrect results.

2. Function `imsls_f_regression_selection` eliminates some subsets of candidate variables by obtaining lower bounds on the error sum of squares from fitting larger models. First, the full model containing all `n_candidate` is fit sequentially using a forward stepwise procedure in which one variable enters the model at a time, and criterion values and model numbers for all the candidate variables that can enter at each step are stored. If linearly dependent variables are removed from the full model, error `IMSLS_VARIABLES_DELETED` is issued. If this error is issued, some submodels that contain variables removed from the full model because of linear dependency can be overlooked if they have not already been identified during the initial forward stepwise procedure. If error `IMSLS_VARIABLES_DELETED` is issued and you want the variables that were removed from the full model to be considered in smaller models, you can rerun the program with a set of linearly independent variables.

Examples

Example 1

This example uses a data set from Draper and Smith (1981, pp. 629–630). Function `imsls_f_regression_selection` is invoked to find the best regression for each subset size using the R^2 criterion. By default, the function prints the results.

```
#include <imsls.h>
#define N_OBSERVATIONS 13
#define N_CANDIDATE    4
main()
{
    float  x[N_OBSERVATIONS][N_CANDIDATE] =
        {7., 26.,  6., 60.,
         1., 29., 15., 52.,
        11., 56.,  8., 20.,
        11., 31.,  8., 47.,
         7., 52.,  6., 33.,
        11., 55.,  9., 22.,
         3., 71., 17.,  6.,
         1., 31., 22., 44.,
         2., 54., 18., 22.,
        21., 47.,  4., 26.,
         1., 40., 23., 34.,
        11., 66.,  9., 12.,
        10., 68.,  8., 12.};
    float  y[N_OBSERVATIONS] = {78.5, 74.3, 104.3, 87.6, 95.9,
        109.2, 102.7,  72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

    imsls_f_regression_selection(N_OBSERVATIONS, N_CANDIDATE, x, y, 0);
}
```

Output

```
 Regressions with   1 variable(s) (R-squared)
```

```
        Criterion         Variables
             67.5          4
             66.6          2
             53.4          1
             28.6          3


 Regressions with   2 variable(s) (R-squared)

        Criterion         Variables
             97.9          1  2
             97.2          1  4
             93.5          3  4
               68          2  4
             54.8          1  3


 Regressions with   3 variable(s) (R-squared)

        Criterion         Variables
             98.2          1  2  4
             98.2          1  2  3
             98.1          1  3  4
             97.3          2  3  4


 Regressions with   4 variable(s) (R-squared)

        Criterion         Variables
             98.2          1  2  3  4


      Best Regression with   1 variable(s) (R-squared)
Variable  Coefficient  Standard Error  t-statistic  p-value
       4      -0.7382          0.1546       -4.775   0.0006



      Best Regression with   2 variable(s) (R-squared)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.468          0.1213        12.10   0.0000
       2        0.662          0.0459        14.44   0.0000



      Best Regression with   3 variable(s) (R-squared)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.452          0.1170        12.41   0.0000
       2        0.416          0.1856         2.24   0.0517
       4       -0.237          0.1733        -1.36   0.2054



      Best Regression with   4 variable(s) (R-squared)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.551          0.7448        2.083   0.0708
       2        0.510          0.7238        0.705   0.5009
       3        0.102          0.7547        0.135   0.8959
       4       -0.144          0.7091       -0.203   0.8441
```

Example 2

This example uses the same data set as the first example, but Mallow's C_p statistic is used as the criterion rather than R^2. Note that when Mallow's C_p statistic (or adjusted R^2) is specified, the variable `max_n_best` indicates the *total* number of "best" regressions (rather than indicating the number of best regressions *per subset size*, as in the case of the R^2 criterion). In this example, the three best regressions are found to be (1, 2), (1, 2, 4), and (1, 2, 3).

```
#include <imsls.h>
#define N_OBSERVATIONS 13
#define N_CANDIDATE    4
main()
{
    float  x[N_OBSERVATIONS][N_CANDIDATE] =
        {7., 26.,  6., 60.,
         1., 29., 15., 52.,
        11., 56.,  8., 20.,
        11., 31.,  8., 47.,
         7., 52.,  6., 33.,
        11., 55.,  9., 22.,
         3., 71., 17.,  6.,
         1., 31., 22., 44.,
         2., 54., 18., 22.,
        21., 47.,  4., 26.,
         1., 40., 23., 34.,
        11., 66.,  9., 12.,
        10., 68.,  8., 12.};
    float  y[N_OBSERVATIONS] = {78.5, 74.3, 104.3, 87.6, 95.9,
        109.2, 102.7,  72.5, 93.1, 115.9, 83.8, 113.3, 109.4};
    int    max_n_best = 3;

    imsls_f_regression_selection(N_OBSERVATIONS, N_CANDIDATE,
        (float *) x, y,
        IMSLS_MALLOWS_CP,
        IMSLS_MAX_N_BEST,   max_n_best,
        0);
}
```

Output

```
1

 Regressions with   1 variable(s) (Mallows  CP)
        Criterion         Variables
              139          4
              142          2
              203          1
              315          3


 Regressions with   2 variable(s) (Mallows  CP)

        Criterion         Variables
             2.68          1  2
              5.5          1  4
             22.4          3  4
              138          2  4
              198          1  3


 Regressions with   3 variable(s) (Mallows  CP)

        Criterion         Variables
             3.02          1  2  4
             3.04          1  2  3
              3.5          1  3  4
             7.34          2  3  4


 Regressions with   4 variable(s) (Mallows  CP)

        Criterion         Variables
                5          1  2  3  4
1

     Best Regression with   2 variable(s) (Mallows CP)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.468          0.1213        12.10   0.0000
       2        0.662          0.0459        14.44   0.0000



     Best Regression with   3 variable(s) (Mallows CP)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.452          0.1170        12.41   0.0000
       2        0.416          0.1856         2.24   0.0517
       4       -0.237          0.1733        -1.36   0.2054


    2nd Best Regression with   3 variable(s) (Mallows CP)
Variable  Coefficient  Standard Error  t-statistic  p-value
       1        1.696          0.2046         8.29   0.0000
       2        0.657          0.0442        14.85   0.0000
       3        0.250          0.1847         1.35   0.2089
```

Warning Errors

`IMSLS_VARIABLES_DELETED`	At least one variable is deleted from the full model because the variance-covariance matrix “cov” is singular.

Fatal Errors

`IMSLS_NO_VARIABLES`	No variables can enter any model.

regression_stepwise

Builds multiple linear regression models using forward selection, backward selection, or stepwise selection.

Synopsis

#include `<imsls.h>`

void `imsls_f_regression_stepwise` (*int* `n_rows`, *int* `n_candidate`,
float `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_regression_stepwise`.

Required Arguments

int `n_rows` (Input)
Number of rows in `x` and the number of elements in `y`.

int `n_candidate` (Input)
Number of candidate variables (independent variables) or columns in `x`.

float `x[]` (Input)
Array of size `n_rows` × `n_candidate` containing the data for the candidate variables.

float `y[]` (Input)
Array of length `n_rows` containing the responses for the dependent variable.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_regression_stepwise` (*int* `n_rows`, *int* `n_candidate`,
float `x[]`, *float* `y[]`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_WEIGHTS`, *float* `weights[]`,
`IMSLS_FREQUENCIES`, *float* `frequencies[]`,
`IMSLS_FIRST_STEP`, *or*
`IMSLS_INTERMEDIATE_STEP`, *or*
`IMSLS_LAST_STEP`, *or*
`IMSLS_ALL_STEPS`,
`IMSLS_N_STEPS`, *int* `n_steps`,
`IMSLS_FORWARD`, *or*
`IMSLS_BACKWARD`, *or*
`IMSLS_STEPWISE`,
`IMSLS_P_VALUE_IN`, *float* `p_value_in`,
`IMSLS_P_VALUE_OUT`, *float* `p_value_out`,
`IMSLS_TOLERANCE`, *float* `tolerance`,
`IMSLS_ANOVA_TABLE`, *float* `**anova_table`,
`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]`,

`IMSLS_COEF_T_TESTS`, *float* `**coef_t_tests`,
`IMSLS_COEF_T_TESTS_USER`, *float* `coef_t_tests[]`,
`IMSLS_COEF_VIF`, *float* `**coef_vif`,
`IMSLS_COEF_VIF_USER`, *float* `coef_vif[]`,
`IMSLS_LEVEL`, *int* `level[]`,
`IMSLS_FORCE`, *int* `n_force`,
`IMSLS_IEND`, *int* `*iend`,
`IMSLS_SWEPT_USER`, *int* `swept[]`,
`IMSLS_HISTORY_USER`, *float* `history[]`,
`IMSLS_COV_SWEPT_USER`, *float* `*covs`
`IMSLS_INPUT_COV`, *int* `n_observations`, *float* `*cov`,
0)

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of `x`.
Default: `x_col_dim` = `n_candidate`

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_rows` containing the weight for each row of `x`.
Default: `weights[]` = 1

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of length `n_rows` containing the frequency for each row of `x`.
Default: `frequencies[]` = 1

`IMSLS_FIRST_STEP`, *or*
`IMSLS_INTERMEDIATE_STEP`, *or*
`IMSLS_LAST_STEP`, *or*
`IMSLS_ALL_STEPS`
One or none of these options can be specified. If none of these is specified, the action defaults to `IMSLS_ALL_STEPS`.

Argument	Action
`IMSLS_FIRST_STEP`	This is the first invocation; additional calls will be made. Initialization and stepping is performed.
`IMSLS_INTERMEDIATE_STEP`	This is an intermediate invocation. Stepping is performed.
`IMSLS_LAST_STEP`	This is the final invocation. Stepping and wrap-up computations are performed.
`IMSLS_ALL_STEPS`	This is the only invocation. Initialization, stepping, and wrap-up computations are performed.

IMSLS_N_STEPS, *int* n_steps (Input)

For nonnegative n_steps, n_steps steps are taken. If n_steps = −1, stepping continues until completion.

IMSLS_FORWARD, *or*
IMSLS_BACKWARD, *or*
IMSLS_STEPWISE

One or none of these options can be specified. If none is specified, the action defaults to IMSLS_BACKWARD.

Keyword	**Action**
IMSLS_FORWARD	An attempt is made to add a variable to the model. A variable is added if its *p*-value is less than p_value_in. During initialization, only the forced variables enter the model.
IMSLS_BACKWARD	An attempt is made to remove a variable from the model. A variable is removed if its *p*-value exceeds p_value_out. During initialization, all candidate independent variables enter the model.
IMSLS_STEPWISE	A backward step is attempted. If a variable is not removed, a forward step is attempted. This is a stepwise step. Only the forced variables enter the model during initialization.

IMSLS_P_VALUE_IN, *float* p_value_in (Input)

Largest *p*-value for variables entering the model. Variables with *p*-values less than p_value_in may enter the model.
Default: p_value_in = 0.05

IMSLS_P_VALUE_OUT, *float* p_value_out (Input)

Smallest *p*-value for removing variables. Variables with p_values greater than p_value_out may leave the model. Argument p_value_out must be greater than or equal to p_value_in. A common choice for p_value_out is 2*p_value_in.
Default: p_value_out = 0.10

IMSLS_TOLERANCE, *float* tolerance (Input)

Tolerance used in determining linear dependence.
Default: tolerance = 100*eps, where eps = imsls_f_machine(4) for single precision and eps = imsls_d_machine(4) for double precision

IMSLS_ANOVA_TABLE, *float* **anova_table (Output)

Address of a pointer to the internally allocated array containing the analysis of variance table. The analysis of variance statistics are as follows:

Element	Analysis of Variance Statistic
0	degrees of freedom for regression
1	degrees of freedom for error
2	total degrees of freedom
3	sum of squares for regression
4	sum of squares for error
5	total sum of squares
6	regression mean square
7	error mean square
8	*F*-statistic
9	*p*-value
10	R^2 (in percent)
11	adjusted R^2 (in percent)
12	estimate of the standard deviation

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for `anova_table` is provided by the user. See `IMSLS_ANOVA_TABLE`.

`IMSLS_COEF_T_TESTS`, *float* `**coef_t_tests` (Output)
Address to a pointer to the internally allocated array containing statistics relating to the regression coefficient for the final model in this invocationing. The rows correspond to the `n_candidate` independent variables. The rows are in the same order as the variables in `x` (or, if `IMSLS_INPUT_COV` is specified, the rows are in the same order as the variables in `cov`). Each row corresponding to a variable not in the model contains statistics for a model which includes the variables of the final model and the variable corresponding to the row in question.

Column	Description
0	coefficient estimate
1	estimated standard error of the coefficient estimate
2	*t*-statistic for the test that the coefficient is 0
3	*p*-value for the two-sided t test

`IMSLS_COEF_T_TESTS_USER`, *float* `coef_t_tests[]` (Output)
Storage for array `coef_t_tests` is provided by the user. See `IMSLS_COEF_T_TESTS`.

`IMSLS_COEF_VIF`, *float* `**coef_vif` (Output)
Address to a pointer to the internally allocated array containing variance inflation factors for the final model in this invocation. The elements correspond to the `n_candidate` dependent variables. The elements are in the same order as the variables in `x` (or, if `IMSLS_INPUT_COV` is specified, the elements are in the same order as the variables in `cov`). Each element corresponding to a variable not in the model contains statistics for a model which includes the variables of the final model and the variables corresponding to the element in question.

The square of the multiple correlation coefficient for the *I*-th regressor after all others can be obtained from `coef_vif[I]` by the following formula:

$$1.0 - \frac{1.0}{\text{VIF}}$$

`IMSLS_COEF_VIF_USER`, *float* `coef_vif[]` (Output)
Storage for array `coef_vif` is provided by the user. See `IMSLS_COEF_VIF`.

`IMSLS_LEVEL`, *int* `level[]` (Input)
Array of length `n_candidate` + 1 containing levels of priority for variables entering and leaving the regression. Each variable is assigned a positive value which indicates its level of entry into the model. A variable can enter the model only after all variables with smaller nonzero levels of entry have entered. Similarly, a variable can only leave the model after all variables with higher levels of entry have left. Variables with the same level of entry compete for entry (deletion) at each step. Argument `level[I]` = 0 means the `I`-th variable is never to enter the model. Argument `level[I]` = −1 means the `I`-th variable is the dependent variable. Argument `level[n_candidate]` must correspond to the dependent variable, except when `IMSLS_INPUT_COV` is specified.
Default: 1, 1, ..., 1, −1 where −1 corresponds to `level[n_candidate]`

`IMSLS_FORCE`, *int* `n_force` (Input)
Variable with levels 1, 2, ..., `n_force` are forced into the model as independent variables. See `IMSLS_LEVEL`.

`IMSLS_IEND`, *int* `*iend` (Output)
Variable which indicates whether additional steps are possible.

iend	**Meaning**
0	Additional steps may be possible.
1	No additional steps are possible.

`IMSLS_SWEPT_USER`, *int* `swept[]` (Output)
A user-allocated array of length `n_candidate` + 1 with information to indicate the independent variables in the model. Argument `swept[n_candidate]` usually corresponds to the dependent variable. See `IMSLS_LEVEL`.

swept[*i*]	**Status of *i*-th Variable**
−1	Variable *i* is not in model.
1	Variable *i* is in model.

`IMSLS_HISTORY_USER`, *float* `history[]` (Output)
User-allocated array of length `n_candidate` + 1 containing the recent history of the independent variables. Element `history[n_candidate]` usually corresponds to the dependent variable. See `IMSLS_LEVEL`.

history[*i*]	**Status of *i*-th Variable**
0.0	Variable has never been added to model.
0.5	Variable was added into the model during initialization.
$k > 0.0$	Variable was added to the model during the *k*-th step.
$k < 0.0$	Variable was deleted from model during the *k*-th step.

`IMSLS_COV_SWEPT_USER`, *float* `*covs` (Output)
User-allocated array of length (`n_candidate` + 1) × (`n_candidate` + 1) that results after `cov` has been swept on the columns corresponding to the variables in the model. The estimated variance-covariance matrix of the estimated regression coefficients in the final model can be obtained by extracting the rows and columns of `covs` corresponding to the independent variables in the final model and multiplying the elements of this matrix by `anova_table[7]`.

`IMSLS_INPUT_COV`, *int* `n_observations` *float* `*cov` (Input)
An (`n_candidate` + 1) by (`n_candidate` + 1) array containing a

variance-covariance or sum of squares and crossproducts matrix, in which the last column must correspond to the dependent variable. Argument `n_observations` is an integer specifying the number of observations associated with `cov`. Argument `cov` can be computed using `imsls_f_covariances`. Arguments `x`, `y`, `weights`, and `frequencies` are not accessed when this option is specified.

By default, `imsls_regression_stepwise` computes `cov` from the input data matrices `x` and `y`.

Description

Function `imsls_f_regression_stepwise` builds a multiple linear regression model using forward selection, backward selection, or forward stepwise (with a backward glance) selection. Function `imsls_f_regression_stepwise` is designed so the user can monitor, and perhaps change, the variables added (deleted) to (from) the model after each step. In this case, multiple calls to `imsls_f_regression_stepwise` (using optional arguments `IMSLS_FIRST_STEP`, `IMSLS_INTERMEDIATE_STEP`, ..., `IMSLS_LAST_STEP`) are made. Alternatively, `imsls_f_regression_stepwise` can be invoked once (default, or specify optional argument `IMSLS_ALL_STEPS`) in order to perform the stepping until a final model is selected.

Levels of priority can be assigned to the candidate independent variables (use optional argument `IMSLS_LEVEL`). All variables with a priority level of 1 must enter the model before variables with a priority level of 2. Similarly, variables with a level of 2 must enter before variables with a level of 3, etc. Variables also can be forced into the model (see optional argument `IMSLS_FORCE`). Note that specifying optional argument `IMSLS_FORCE` without also specifying optional argument `IMSLS_LEVEL` will result in all variables being forced into the model.

Typically, the intercept is forced into all models and is not a candidate variable. In this case, a sum-of-squares and crossproducts matrix for the independent and dependent variables corrected for the mean is required. Other possibilities are as follows:

1. The intercept is not in the model. A raw (uncorrected) sum-of-squares and crossproducts matrix for the independent and dependent variables is required as input in `cov` (see optional argument `IMSLS_INPUT_COV`). Argument `n_observations` must be set to one greater than the number of observations.

2. An intercept is a candidate variable. A raw (uncorrected) sum-of-squares and crossproducts matrix for the constant regressor (=1), independent and dependent variables are required for `cov`. In this case, `cov` contains one additional row and column corresponding to the constant regressor. This row/column contains the sum-of-squares and crossproducts of the constant regressor with the independent and dependent variables. The remaining elements in `cov` are the same as in the previous case.

Argument `n_observations` must be set to one greater than the number of observations.

The stepwise regression algorithm is due to Efroymson (1960). Function `imsls_f_regression_stepwise` uses sweeps of the covariance matrix (input in `cov`, if optional argument `IMSLS_INPUT_COV` is specified, or generated internally by default) to move variables in and out of the model (Hemmerle 1967, Chapter 3). The SWEEP operator discussed in Goodnight (1979) is used. A description of the stepwise algorithm is also given by Kennedy and Gentle (1980, pp. 335–340). The advantage of stepwise model building over all possible regression (see function `imsls_f_regression_selection`, page 112) is that it is less demanding computationally when the number of candidate independent variables is very large. However, there is no guarantee that the model selected will be the best model (highest R^2) for any subset size of independent variables.

Example

This example uses a data set from Draper and Smith (1981, pp. 629–630). Backwards stepping is performed by default.

```
#include <imsls.h>
#define N_OBSERVATIONS 13
#define N_CANDIDATE    4
main()
{
    char           *labels[] = {
                    "degrees of freedom for regression",
                    "degrees of freedom for error",
                    "total degrees of freedom",
                    "sum of squares for regression",
                    "sum of squares for error",
                    "total 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 within error"
    };
    char           *c_labels[] = {
                    "variable",
                    "estimate",
                    "s.e.",
                    "t",
                    "prob > t"
    };
    float  *aov, *tt;
    float  x[N_OBSERVATIONS][N_CANDIDATE] =
        {7., 26.,  6., 60.,
         1., 29., 15., 52.,
        11., 56.,  8., 20.,
        11., 31.,  8., 47.,
         7., 52.,  6., 33.,
        11., 55.,  9., 22.,
         3., 71., 17.,  6.,
         1., 31., 22., 44.,
         2., 54., 18., 22.,
        21., 47.,  4., 26.,
         1., 40., 23., 34.,
        11., 66.,  9., 12.,
        10., 68.,  8., 12.};
    float  y[N_OBSERVATIONS] = {78.5, 74.3, 104.3, 87.6, 95.9,
        109.2, 102.7,  72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

    imsls_f_regression_stepwise(N_OBSERVATIONS, N_CANDIDATE, x, y,
        IMSLS_ANOVA_TABLE, &aov,
        IMSLS_COEF_T_TESTS, &tt,
        0);

    imsls_f_write_matrix("* * * Analysis of Variance * * *\n",
        13, 1, aov,
        IMSLS_ROW_LABELS, labels,
        IMSLS_WRITE_FORMAT, "%9.2f",
        0);

    imsls_f_write_matrix("* * * Inference on Coefficients * * *\n",
        4, 4, tt,
        IMSLS_COL_LABELS, c_labels,
        IMSLS_WRITE_FORMAT, "%9.2f",
        0);

    return;
}
```

Output

```
          * * * Analysis of Variance * * *

degrees of freedom for regression            2.00
degrees of freedom for error                10.00
total degrees of freedom                    12.00
sum of squares for regression             2657.86
sum of squares for error                    57.90
total sum of squares                      2715.76
regression mean square                    1328.93
error mean square                            5.79
F-statistic                                229.50
p-value                                      0.00
R-squared (in percent)                      97.87
adjusted R-squared (in percent)             97.44
est. standard deviation of within error      2.41

        * * * Inference on Coefficients * * *

variable   estimate       s.e.          t   prob > t
       1       1.47       0.12      12.10       0.00
       2       0.66       0.05      14.44       0.00
       3       0.25       0.18       1.35       0.21
       4      -0.24       0.17      -1.36       0.21
```

Warning Errors

`IMSLS_LINEAR_DEPENDENCE_1`	Based on "tolerance" = #, there are linear dependencies among the variables to be forced.

Fatal Errors

`IMSLS_NO_VARIABLES_ENTERED`	No variables entered the model. All elements of "anova_table" are set to NaN.

poly_regression

Performs a polynomial least-squares regression.

Synopsis

#include `<imsls.h>`

float `*imsls_f_poly_regression` (*int* `n_observations`, *float* `x[]`,
float `y[]`, *int* `degree`, ..., 0)

The type *double* function is `imsls_d_poly_regression`.

Required Arguments

int `n_observations` (Input)
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)
Degree of the polynomial.

Return Value

A pointer to the array of size `degree` + 1 containing the coefficients of the fitted polynomial. If a fit cannot be computed, NULL is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_poly_regression` (*int* `n_observations`, *float* `x[]`,
float `y[]`, *int* `degree`,
`IMSLS_WEIGHTS`, *float* `weights[]`,
`IMSLS_SSQ_POLY`, *float* `**ssq_poly`,
`IMSLS_SSQ_POLY_USER`, *float* `ssq_poly[]`,
`IMSLS_SSQ_POLY_COL_DIM`, *int* `ssq_poly_col_dim`,

```
IMSLS_SSQ_LOF, float **ssq_lof,
IMSLS_SSQ_LOF_USER, float ssq_lof[],
IMSLS_SSQ_LOF_COL_DIM, int ssq_lof_col_dim,
IMSLS_X_MEAN, float *x_mean,
IMSLS_X_VARIANCE, float *x_variance,
IMSLS_ANOVA_TABLE, float **anova_table,
IMSLS_ANOVA_TABLE_USER, float anova_table[],
IMSLS_DF_PURE_ERROR, int *df_pure_error,
IMSLS_SSQ_PURE_ERROR, float *ssq_pure_error,
IMSLS_RESIDUAL, float **residual,
IMSLS_RESIDUAL_USER, float residual[],
IMSLS_POLY_REGRESSION_INFO,
        Imsls_f_poly_regression **poly_info,
IMSLS_RETURN_USER, float coefficients[],
0)
```

Optional Arguments

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array with `n_observations` components containing the array of weights for the observation.
Default: `weights[]` = 1

`IMSLS_SSQ_POLY`, *float* `**ssq_poly` (Output)
Address of a pointer to the internally allocated array containing the sequential sums of squares and other statistics. Row *i* corresponds to x^i, $i = 0, \ldots,$ `degree` − 1, and the columns are described as follows:

Column	Description
0	degrees of freedom
1	sums of squares
2	*F*-statistic
3	*p*-value

`IMSLS_SSQ_POLY_USER`, *float* `ssq_poly[]` (Output)
Storage for array `ssq_poly` is provided by the user. See `IMSLS_SSQ_POLY`.

`IMSLS_SSQ_POLY_COL_DIM`, *int* `ssq_poly_col_dim` (Input)
Column dimension of `ssq_poly`.
Default: `ssq_poly_col_dim` = 4

`IMSLS_SSQ_LOF`, *float* `**ssq_lof` (Output)
Address of a pointer to the internally allocated array containing the lack-of-fit statistics. Row *i* corresponds to x^i, $i = 0, \ldots,$ `degree` − 1, and the columns are described in the following table:

Column	Description
0	degrees of freedom
1	lack-of-fit sums of squares
2	*F*-statistic for testing lack-of-fit for a polynomial model of degree *i*
3	*p*-value for the test

`IMSLS_SSQ_LOF_USER`, *float* `ssq_lof[]` (Output)
Storage for array `ssq_lof` is provided by the user. See `IMSLS_SSQ_LOF`.

`IMSLS_SSQ_LOF_COL_DIM`, *int* `ssq_lof_col_dim` (Input)
Column dimension of `ssq_lof`.
Default: `ssq_lof_col_dim` = 4

`IMSLS_X_MEAN`, *float* `*x_mean` (Output)
Mean of *x*.

`IMSLS_X_VARIANCE`, *float* `*x_variance` (Output)
Variance of *x*.

`IMSLS_ANOVA_TABLE`, *float* `**anova_table` (Output)
Address of a pointer to the array containing the analysis of variance table.

Column	Description
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

Column	Description
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)

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for `anova_table` is provided by the user. See `IMSLS_ANOVA_TABLE`.

`IMSLS_DF_PURE_ERROR`, *int* `*df_pure_error` (Output)
If specified, the degrees of freedom for pure error are returned in `df_pure_error`.

`IMSLS_SSQ_PURE_ERROR`, *float* `*ssq_pure_error` (Output)
If specified, the sums of squares for pure error are returned in `ssq_pure_error`.

`IMSLS_RESIDUAL`, *float* `**residual` (Output)
Address of a pointer to the array containing the residuals.

`IMSLS_RESIDUAL_USER`, *float* `residual[]` (Output)
Storage for array `residual` is provided by the user. See `IMSLS_RESIDUAL`.

`IMSLS_POLY_REGRESSION_INFO`, *Imsls_f_poly_regression* `**poly_info` (Output)
Address of a pointer to an internally allocated structure containing the information about the polynomial fit required as input for IMSL function `imsls_f_poly_prediction`.

`IMSLS_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

Function `imsls_f_poly_regression` computes estimates of the regression coefficients in a polynomial (curvilinear) regression model. In addition to the computation of the fit, `imsls_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,

$$R^2 = \frac{\sum w_i \left(\hat{y}_i - \overline{y}\right)^2}{\sum w_i \left(y_i - \overline{y}\right)^2} 100\%$$

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 percent, inclusive. $R^2 = 100$ percent 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).

Function `imsls_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 is also given.

```
#include <imsls.h>

#define DEGREE          2
#define NOBS           14

main()
{
    float       *coefficients;
    float       x[] = {0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 4.0,
                      4.0, 5.0, 5.0, 6.0, 6.0, 7.0, 7.0};
    float       y[] = {508.1, 498.4, 568.2, 577.3, 651.7, 657.0, 755.3,
                      758.9, 787.6, 792.1, 841.4, 831.8, 854.7, 871.4};

    coefficients = imsls_f_poly_regression (NOBS, x, y, DEGREE, 0);
    imsls_f_write_matrix("Least-Squares Polynomial Coefficients",
                        DEGREE + 1, 1, coefficients,
                        IMSLS_ROW_NUMBER_ZERO,
                        0);
}
```

Output

```
Least-Squares Polynomial Coefficients
            0       503.3
            1        78.9
            2        -4.0
```

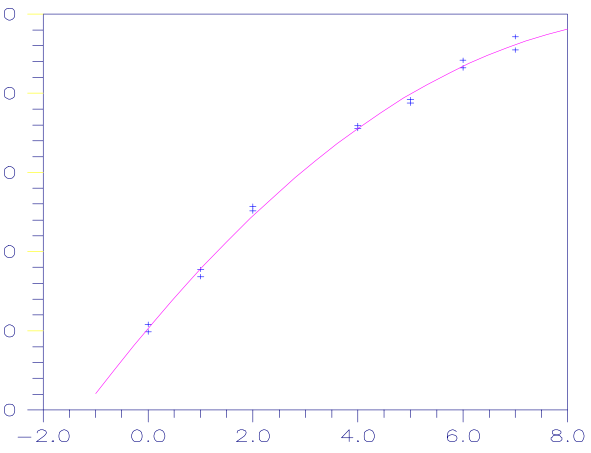

Figure 2-1   A Polynomial Fit

Example 2

This example is a continuation of the initial example. Here, many optional arguments are used.

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

#define DEGREE            2
#define NOBS             14

void main()
{
    int         iset = 1, dfpe;
    float       *coefficients, *anova_table, sspe, *ssqpoly, *ssqlof;
    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 = imsls_f_poly_regression(NOBS, x, y, DEGREE,
                                       IMSLS_SSQ_POLY, &ssqpoly,
                                       IMSLS_SSQ_LOF, &ssqlof,
                                       IMSLS_ANOVA_TABLE, &anova_table,
                                       IMSLS_DF_PURE_ERROR, &dfpe,
                                       IMSLS_SSQ_PURE_ERROR, &sspe,
                                       0);
    imsls_write_options(-1, &iset);
    imsls_f_write_matrix("Least Squares Polynomial Coefficients",
                                        1, DEGREE + 1,
                        coefficients,
                        IMSLS_COL_LABELS, coef_clab,
                        0);
    coef_rlab[0] = coef_clab[2];
    coef_rlab[1] = coef_clab[3];
    imsls_f_write_matrix("Sequential Statistics", DEGREE, 4, ssqpoly,
                        IMSLS_COL_LABELS, stat_clab,
                        IMSLS_ROW_LABELS, coef_rlab,
                        IMSLS_WRITE_FORMAT, "%3.1f%8.1f%6.1f%6.4f",
                        0);
    imsls_f_write_matrix("Lack-of-Fit Statistics", DEGREE, 4, ssqlof,
                        IMSLS_COL_LABELS, stat_clab,
                        IMSLS_ROW_LABELS, coef_rlab,
                        IMSLS_WRITE_FORMAT,  "%3.1f%8.1f%6.1f%6.4f",
                        0);
    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
                                                        anova_table,
                        IMSLS_ROW_LABELS, anova_rlab,
                        IMSLS_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

`IMSLS_CONSTANT_YVALUES`	The *y* values are constant. A zero-order polynomial is fit. High order coefficients are set to zero.
`IMSLS_FEW_DISTINCT_XVALUES`	There are too few distinct *x* values to fit the desired degree polynomial. High order coefficients are set to zero.
`IMSLS_PERFECT_FIT`	A perfect fit was obtained with a polynomial of degree less than `degree`. High order coefficients are set to zero.

Fatal Errors

`IMSLS_NONNEG_WEIGHT_REQUEST_2`	All weights must be nonnegative.
`IMSLS_ALL_OBSERVATIONS_MISSING`	Each (*x, y*) point contains NaN. There are no valid data.
`IMSLS_CONSTANT_XVALUES`	The *x* values are constant.

poly_prediction

Computes predicted values, confidence intervals, and diagnostics after fitting a polynomial regression model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_poly_prediction` (*Imsls_f_poly_regression* `*poly_info`,
    int `n_predict`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_poly_prediction`.

Required Arguments

Imsls_f_poly_regression `*poly_info` (Input)
    Pointer to a structure of type *Imsls_f_poly_regression*. See function `imsls_f_poly_regression` (page 132).

int `n_predict` (Input)
    Length of array `x`.

float `x[]` (Input)
    Array of length `n_predict` containing the values of the independent variable for which calculations are to be performed.

Return Value

A pointer to an internally allocated array of length `n_predict` containing the predicted values.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_poly_prediction` (*Imsls_f_poly_regression* `*poly_info`,
    int `n_predict`, *float* `x[]`,
    `IMSLS_CONFIDENCE`, *float* `confidence`,
    `IMSLS_WEIGHTS`, *float* `weights[]`,
    `IMSLS_SCHEFFE_CI`, *float* `**lower_limit`,
        float `**upper_limit`,
    `IMSLS_SCHEFFE_CI_USER`, *float* `lower_limit[]`,
        float `upper_limit[]`,
    `IMSLS_POINTWISE_CI_POP_MEAN`, *float* `**lower_limit`,
        float `**upper_limit`,
    `IMSLS_POINTWISE_CI_POP_MEAN_USER`, *float* `lower_limit[]`,
        float `upper_limit[]`,
    `IMSLS_POINTWISE_CI_NEW_SAMPLE`, *float* `**lower_limit`,
        float `**upper_limit`,
    `IMSLS_POINTWISE_CI_NEW_SAMPLE_USER`,
        float `lower_limit[]`,

```
        float upper_limit[],
    IMSLS_LEVERAGE, float **leverage,
    IMSLS_LEVERAGE_USER, float leverage[],
    IMSLS_RETURN_USER, float y_hat[],
    IMSLS_Y, float y[],
    IMSLS_RESIDUAL, float **residual,
    IMSLS_RESIDUAL_USER, float residual[],
    IMSLS_STANDARDIZED_RESIDUAL,
        float **standardized_residual,
    IMSLS_STANDARDIZED_RESIDUAL_USER,
        float standardized_residual[],
    IMSLS_DELETED_RESIDUAL, float **deleted_residual,
    IMSLS_DELETED_RESIDUAL_USER, float deleted_residual[],
    IMSLS_COOKSD, float **cooksd,
    IMSLS_COOKSD_USER, float cooksd[],
    IMSLS_DFFITS, float **dffits,
    IMSLS_DFFITS_USER, float dffits[],
    0)
```

Optional Arguments

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
: Confidence level for both two-sided interval estimates on the mean and for two-sided prediction intervals in percent. Argument `confidence` must be in the range [0.0, 100.0). For one-sided intervals with confidence level `onecl`, where 50.0 ≤ `onecl` < 100.0, set `confidence` = 100.0 – 2.0 * (100.0 – `onecl`).
Default: `confidence` = 95.0

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
: Array of length `n_predict` containing the weight for each row of `x`. The computed prediction interval uses SSE/(DFE*`weights[i]`) for the estimated variance of a future response.
Default: `weights[]` = **1**

`IMSLS_SCHEFFE_CI`, *float* `**lower_limit`, *float* `**upper_limit`
: (Output)
Array `lower_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the lower confidence limits of Scheffé confidence intervals corresponding to the rows of `x`. Array `upper_limit` is the address of a pointer to an internally allocated array of length `n_predict` containing the upper confidence limits of Scheffé confidence intervals corresponding to the rows of `x`.

`IMSLS_SCHEFFE_CI_USER`, *float* `lower_limit[]`, *float* `upper_limit[]`
: (Output)
Storage for arrays lower_limit and upper_limit is provided by the user. See `IMSLS_SCHEFFE_CI`.

IMSLS_POINTWISE_CI_POP_MEAN, *float* **lower_limit,
float **upper_limit (Output)
Array lower_limit is the address of a pointer to an internally allocated array of length n_predict containing the lower confidence limits of the confidence intervals for two-sided interval estimates of the means, corresponding to the rows of x. Array upper_limit is the address of a pointer to an internally allocated array of length n_predict containing the upper confidence limits of the confidence intervals for two-sided interval estimates of the means, corresponding to the rows of x.

IMSLS_POINTWISE_CI_POP_MEAN_USER, *float* lower_limit[],
float upper_limit[] (Output)
Storage for arrays lower_limit and upper_limit is provided by the user. See IMSLS_POINTWISE_CI_POP_MEAN.

IMSLS_POINTWISE_CI_NEW_SAMPLE, *float* **lower_limit,
float **upper_limit (Output)
Array lower_limit is the address of a pointer to an internally allocated array of length n_predict containing the lower confidence limits of the confidence intervals for two-sided prediction intervals, corresponding to the rows of x. Array upper_limit is the address of a pointer to an internally allocated array of length n_predict containing the upper confidence limits of the confidence intervals for two-sided prediction intervals, corresponding to the rows of x.

IMSLS_POINTWISE_CI_NEW_SAMPLE_USER, *float* lower_limit[],
float upper_limit[] (Output)
Storage for arrays lower_limit and upper_limit is provided by the user. See IMSLS_POINTWISE_CI_NEW_SAMPLE.

IMSLS_LEVERAGE, *float* **leverage (Output)
Address of a pointer to an internally allocated array of length n_predict containing the leverages.

IMSLS_LEVERAGE_USER, *float* leverage[] (Output)
Storage for array leverage is provided by the user. See IMSLS_LEVERAGE.

IMSLS_RETURN_USER, *float* y_hat[] (Output)
Storage for array y_hat is provided by the user. The length n_predict array contains the predicted values.

IMSLS_Y *float* y[] (Input)
Array of length n_predict containing the observed responses.

Note: IMSLS_Y must be specified if any of the following optional arguments are specified.

IMSLS_RESIDUAL, *float* **residual (Output)
Address of a pointer to an internally allocated array of length n_predict containing the residuals.

`IMSLS_RESIDUAL_USER`, *float* `residual[]` (Output)
Storage for array `residual` is provided by the user. See `IMSLS_RESIDUAL`.

`IMSLS_STANDARDIZED_RESIDUAL`, *float* `**standardized_residual` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the standardized residuals.

`IMSLS_STANDARDIZED_RESIDUAL_USER`, *float* `standardized_residual[]` (Output)
Storage for array `standardized_residual` is provided by the user. See `IMSLS_STANDARDIZED_RESIDUAL`.

`IMSLS_DELETED_RESIDUAL`, *float* `**deleted_residual` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the deleted residuals.

`IMSLS_DELETED_RESIDUAL_USER`, *float* `deleted_residual[]` (Output)
Storage for array `deleted_residual` is provided by the user. See `IMSLS_DELETED_RESIDUAL`.

`IMSLS_COOKSD`, *float* `**cooksd` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the Cook's *D* statistics.

`IMSLS_COOKSD_USER`, *float* `cooksd[]` (Output)
Storage for array `cooksd` is provided by the user. See `IMSLS_COOKSD`.

`IMSLS_DFFITS`, *float* `**dffits` (Output)
Address of a pointer to an internally allocated array of length `n_predict` containing the DFFITS statistics.

`IMSLS_DFFITS_USER`, *float* `dffits[]` (Output)
Storage for array `dffits` is provided by the user. See `IMSLS_DFFITS`.

Description

Function `imsls_f_poly_prediction` assumes a polynomial model

$$y_i = \beta_0 + \beta_1 x_i + ..., \beta_k x_i^k + \varepsilon_i \qquad i = 1, 2, ..., n$$

where the observed values of the y_i's constitute the response, the x_i's are the settings of the independent variable, the β_j's are the regression coefficients and the ε_i's are the errors that are independently distributed normal with mean 0 and the following variance:

$$\frac{\sigma^2}{w_i}$$

Given the results of a polynomial regression, fitted using orthogonal polynomials and weights w_i, function `imsls_f_poly_prediction` produces predicted values, residuals, confidence intervals, prediction intervals, and diagnostics for outliers and in influential cases.

Often, a predicted value and confidence interval are desired for a setting of the independent variable not used in computing the regression fit. This is accomplished by simply using a different `x` matrix when calling `imsls_f_poly_prediction` than was used for the fit (function `imsls_f_poly_regression`). See Example 1on page 136.

Results from function `imsls_f_poly_regression`, which produces the fit using orthogonal polynomials, are used for input by the structure `poly_info`. The fitted model from `imsls_f_poly_regression` is

$$\hat{y}_i = \hat{\alpha}_0 p_0 \left(z_i\right) + \hat{\alpha}_1 p_1 \left(z_i\right) + \ldots + \hat{\alpha}_k p_k \left(z_i\right)$$

where the z_i's are settings of the independent variable x scaled to the interval [−2, 2] and the p_j (z)'s are the orthogonal polynomials. The $X^T X$ matrix for this model is a diagonal matrix with elements d_j. The case statistics are easily computed from this model and are equal to those from the original polynomial model with β_j's as the regression coefficients.

The leverage is computed as follows:

$$h_i = w_i \sum_{j=0}^{k} d_j^{-1} p_j^2 \left(z_i\right)$$

The estimated variance of

$$\hat{y}_i$$

is given by the following:

$$\frac{h_i s^2}{w_i}$$

The computation of the remainder of the case statistics follows easily from the definitions. See "Diagnostics for Individual Cases" (page 53) for the definition of the case diagnostics.

Often, predicted values and confidence intervals are desired for combinations of settings of the independent variables not used in computing the regression fit. This can be accomplished by defining a new data matrix. Since the information about the model fit is input in `poly_info`, it is not necessary to send in the data set used for the original calculation of the fit, i.e., only variable combinations for which predictions are desired need be entered in `x`.

Examples

Example 1

A polynomial model is fit to the 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 dispensers. Responses for 14 similar cafeterias are in the data set.

```
#include <imsls.h>

main()
{
    Imsls_f_poly_regression *poly_info;
    float     *y_hat, *coefficients;
    int       n_observations = 14;
    int       degree = 2;
    int       n_predict = 8;
    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};
    float     x2[] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};

    /* Generate the polynomial regression fit*/
    coefficients = imsls_f_poly_regression (n_observations, x, y,
        degree, IMSLS_POLY_REGRESSION_INFO, &poly_info, 0);

    /* Compute predicted values */
    y_hat = imsls_f_poly_prediction(poly_info, n_predict, x2, 0);

    /* Print predicted values */
    imsls_f_write_matrix("Predicted Values", 1, n_predict, y_hat, 0);

    free(coefficients);
    free(y_hat);
    return;
}
```

Output

```
                        Predicted Values
         1            2            3            4            5            6
     503.3        578.3        645.4        704.4        755.6        798.8

         7            8
     834.1        861.4
```

Example 2

Predicted values, confidence intervals, and diagnostics are computed for the data set described in the first example.

```
#include <imsls.h>

main()
{
#define N_PREDICT 14
    Imsls_f_poly_regression *poly_info;
    float     *coefficients, y_hat[N_PREDICT],
              lower_ci[N_PREDICT], upper_ci[N_PREDICT],
              lower_pi[N_PREDICT], upper_pi[N_PREDICT],
              s_residual[N_PREDICT], d_residual[N_PREDICT],
              leverage[N_PREDICT], cooksd[N_PREDICT],
            dffits[N_PREDICT], lower_scheffe[N_PREDICT],
            upper_scheffe[N_PREDICT];
    int     n_observations = N_PREDICT;
    int     degree = 2;
    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};

    /* Generate the polynomial regression fit*/
    coefficients = imsls_f_poly_regression (n_observations, x, y,
        degree, IMSLS_POLY_REGRESSION_INFO, &poly_info, 0);

    /* Compute predicted values and case statistics */
    imsls_f_poly_prediction(poly_info, N_PREDICT, x,
        IMSLS_RETURN_USER, y_hat,
        IMSLS_POINTWISE_CI_POP_MEAN_USER, lower_ci, upper_ci,
        IMSLS_POINTWISE_CI_NEW_SAMPLE_USER, lower_pi, upper_pi,
        IMSLS_Y, y,
        IMSLS_STANDARDIZED_RESIDUAL_USER, s_residual,
        IMSLS_DELETED_RESIDUAL_USER, d_residual,
        IMSLS_LEVERAGE_USER, leverage,
        IMSLS_COOKSD_USER, cooksd,
        IMSLS_DFFITS_USER, dffits,
        IMSLS_SCHEFFE_CI_USER, lower_scheffe, upper_scheffe,
        0);

    /* Print results */
    imsls_f_write_matrix("Predicted Values", 1, N_PREDICT, y_hat, 0);
    imsls_f_write_matrix("Lower Scheffe CI", 1, N_PREDICT,
        lower_scheffe, 0);
    imsls_f_write_matrix("Upper Scheffe CI", 1, N_PREDICT,
        upper_scheffe, 0);
    imsls_f_write_matrix("Lower CI", 1, N_PREDICT, lower_ci, 0);
    imsls_f_write_matrix("Upper CI", 1, N_PREDICT, upper_ci, 0);
    imsls_f_write_matrix("Lower PI", 1, N_PREDICT, lower_pi, 0);
    imsls_f_write_matrix("Upper PI", 1, N_PREDICT, upper_pi, 0);
    imsls_f_write_matrix("Standardized Residual", 1, N_PREDICT,
        s_residual, 0);
    imsls_f_write_matrix("Deleted Residual", 1, N_PREDICT,
        d_residual, 0);
    imsls_f_write_matrix("Leverage", 1, N_PREDICT, leverage, 0);
    imsls_f_write_matrix("Cooks Distance", 1, N_PREDICT, cooksd, 0);
    imsls_f_write_matrix("DFFITS", 1, N_PREDICT, dffits, 0);


    free(coefficients);
    return;

}
```

Output

```
                       Predicted Values
         1           2           3           4           5           6
     503.3       503.3       578.3       578.3       645.4       645.4

         7           8           9          10          11          12
     755.6       755.6       798.8       798.8       834.1       834.1
     13          14
  861.4       861.4

                       Lower Scheffe CI
      1           2           3           4           5           6
  489.8       489.8       569.5       569.5       636.5       636.5

      7           8           9          10          11          12
  745.7       745.7       790.2       790.2       825.5       825.5

     13          14
  847.7       847.7

                       Upper Scheffe CI
      1           2           3           4           5           6
  516.9       516.9       587.1       587.1       654.2       654.2

      7           8           9          10          11          12
  765.5       765.5       807.4       807.4       842.7       842.7

     13          14
  875.1       875.1

                           Lower CI
      1           2           3           4           5           6
  492.8       492.8       571.5       571.5       638.4       638.4

      7           8           9          10          11          12
  747.9       747.9       792.1       792.1       827.4       827.4

     13          14
  850.7       850.7
                           Upper CI
      1           2           3           4           5           6
  513.9       513.9       585.2       585.2       652.3       652.3

      7           8           9          10          11          12
  763.3       763.3       805.5       805.5       840.8       840.8

     13          14
  872.1       872.1

                           Lower PI
      1           2           3           4           5           6
  482.8       482.8       559.3       559.3       626.4       626.4

      7           8           9          10          11          12
  736.3       736.3       779.9       779.9       815.2       815.2

     13          14
  840.8       840.8

                           Upper PI
      1           2           3           4           5           6
  523.9       523.9       597.3       597.3       664.3       664.3

      7           8           9          10          11          12
  774.9       774.9       817.7       817.7       853.0       853.0
```

```
       13          14
    882.1       882.1

                       Standardized Residual
        1           2           3           4           5           6
    0.737      -0.766      -1.366      -0.137       0.859       1.575

        7           8           9          10          11          12
   -0.041       0.456      -1.507      -0.902       0.982      -0.308

       13          14
   -1.051       1.557

                         Deleted Residual
        1           2           3           4           5           6
    0.720      -0.751      -1.429      -0.131       0.848       1.707

        7           8           9          10          11          12
   -0.039       0.439      -1.613      -0.894       0.980      -0.295

       13          14
   -1.056       1.681

                             Leverage
        1           2           3           4           5           6
   0.3554      0.3554      0.1507      0.1507      0.1535      0.1535

        7           8           9          10          11          12
   0.1897      0.1897      0.1429      0.1429      0.1429      0.1429

       13          14
   0.3650      0.3650

                          Cooks Distance
        1           2           3           4           5           6
   0.0997      0.1080      0.1104      0.0011      0.0446      0.1500

        7           8           9          10          11          12
   0.0001      0.0162      0.1262      0.0452      0.0536      0.0053

       13          14
   0.2116      0.4644

                              DFFITS
        1           2           3           4           5           6
    0.535      -0.558      -0.602      -0.055       0.361       0.727

        7           8           9          10          11          12
   -0.019       0.212      -0.659      -0.365       0.400      -0.120

       13          14
   -0.801       1.274
```

Warning Errors

`IMSLS_LEVERAGE_GT_1`	A leverage (= #) much greater than one is computed. It is set to 1.0.

`IMSLS_DEL_MSE_LT_0`	A deleted residual mean square (= #) much less than zero is computed. It is set to zero.

Fatal Errors

`IMSLS_NEG_WEIGHT`	"weights[#]" = #. Weights must be nonnegative.

nonlinear_regression

Fits a multivarite nonlinear regression model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_nonlinear_regression` (*float* `fcn()`,
    int `n_parameters`, *int* `n_observations`, *int* `n_independent`,
    float `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_nonlinear_regression`.

Required Arguments

float `fcn` (*int* `n_independent`, *float* `xi[]`, *int* `n_parameters`,
    float `theta[]`)
    User-supplied function to evaluate the function that defines the nonlinear regression problem where `xi` is an array of length `n_independent` at which point the function is evaluated and `theta` is an array of length `n_parameters` containing the current values of the regression coefficients. Function `fcn` returns a predicted value at the point `xi`. In the following, $f(x_i;\theta)$, or just f_i, denotes the value of this function at the point x_i, for a given value of θ. (Both x_i and θ are arrays.)

int `n_parameters` (Input)
    Number of parameters to be estimated.

int `n_observations` (Input)
    Number of observations.

int `n_independent` (Input)
    Number of independent variables.

float `x[]` (Input)
    Array of size `n_observations` by `n_independent` containing the matrix of independent (explanatory) variables.

float `y[]` (Input)
    Array of length `n_observations` containing the dependent (response) variable.

Return Value

A pointer to an array of length `n_parameters` containing a solution, $\hat{\theta}$ for the nonlinear regression coefficients. To release this space, use `free`. If no solution can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_nonlinear_regression (float fcn(),
        int n_parameters, int n_observations, int n_independent,
        float x[], float y[],
        IMSLS_THETA_GUESS, float theta_guess[],
        IMSLS_JACOBIAN, void jacobian(),
        IMSLS_THETA_SCALE, float theta_scale[],
        IMSLS_GRADIENT_EPS, float gradient_eps,
        IMSLS_STEP_EPS, float step_eps,
        IMSLS_SSE_REL_EPS, float sse_rel_eps,
        IMSLS_SSE_ABS_EPS, float sse_abs_eps,
        IMSLS_MAX_STEP, float max_step,
        IMSLS_INITIAL_TRUST_REGION, float trust_region,
        IMSLS_GOOD_DIGIT, int ndigit,
        IMSLS_MAX_ITERATIONS, int max_itn,
        IMSLS_MAX_SSE_EVALUATIONS, int max_sse_eval,
        IMSLS_MAX_JACOBIAN_EVALUATIONS, int max_jacobian,
        IMSLS_TOLERANCE, float tolerance,
        IMSLS_PREDICTED, float **predicted,
        IMSLS_PREDICTED_USER, float predicted[],
        IMSLS_RESIDUAL, float **residual,
        IMSLS_RESIDUAL_USER, float residual[],
        IMSLS_R, float **r,
        IMSLS_R_USER, float r[],
        IMSLS_R_COL_DIM, int r_col_dim,
        IMSLS_R_RANK, int *rank,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_DF, int *df,
        IMSLS_SSE, float *sse,
        IMSLS_RETURN_USER, float theta_hat[],
        IMSLS_FCN_W_DATA, void fcn(),void *data,
        IMSLS_JACOBIAN_W_DATA, void jacobian(),void *data,
        0)
```

Optional Arguments

`IMSLS_THETA_GUESS`, *float* `theta_guess[]` (Input)
    Array with `n_parameters` components containing an initial guess.
    Default: `theta_guess[]` = 0

`IMSLS_JACOBIAN`, *void* `jacobian` (*int* `n_independent`, *float* `xi[]`, *int* `n_parameters`, *float* `theta[]`, *float* `fjac[]`) (Input/Output)
User-supplied function to compute the *i*-th row of the Jacobian, where the `n_independent` data values corresponding to the *i*-th row are input in `xi`. Argument `theta` is an array of length `n_parameters` containing the regression coefficients for which the Jacobian is evaluated, `fjac` is the computed `n_parameters` row of the Jacobian for observation *i* at `theta`. Note that each derivative $\partial f(x_i)/\partial\theta_j$ should be returned in `fjac [j – 1]` for $j = 1, 2, ...,$ `n_parameters`.

`IMSLS_THETA_SCALE`, *float* `theta_scale[]` (Input)
Array with `n_parameters` components containing the scaling array for θ. Array `theta_scale` is used mainly in scaling the gradient and the distance between two points. See keywords `IMSLS_GRADIENT_EPS` and `IMSLS_STEP_EPS` for more detail.
Default: `theta_scale[]` = **1**

`IMSLS_GRADIENT_EPS`, *float* `gradient_eps` (Input)
Scaled gradient tolerance. The *j*-th component of the scaled gradient at θ is calculated as

$$\frac{|g_j| * \max(|\theta_j|, 1/t_j)}{\frac{1}{2}\|F(\theta)\|_2^2}$$

where $g = \nabla F(\theta)$, t = `theta_scale`, and

$$\|F(\theta)\|_2^2 = \sum_{i=1}^{n}\left(y_i - f(x_i;\theta)\right)^2$$

The value $F(\theta)$ is the sum of the squared residuals, SSE, at the point θ.
Default:

$$\texttt{grad\_tol} = \sqrt{\varepsilon}$$

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

`IMSLS_STEP_EPS`, *float* `step_eps` (Input)
Scaled step tolerance. The *j*-th component of the scaled step from points θ and θ′ is computed as

$$\frac{|\theta_j - \theta'_j|}{\max(|\theta_j|, 1/t_j)}$$

where t = `theta_scale`
Default: `step_eps` = $\varepsilon^{2/3}$,where ε is the machine precision

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

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

`IMSLS_MAX_STEP`, *float* `max_step` (Input)
: Maximum allowable step size.
Default: `max_step` = 1000 max (ε_1, ε_2), where $\varepsilon_1 = (t^T\theta_0)^{1/2}$, $\varepsilon_2 = \|t\|_2$, t = `theta_scale`, and θ_0 = `theta_guess`

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

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

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

`IMSLS_MAX_SSE_EVALUATIONS`, *int* `max_sse_eval` (Input)
: Maximum number of SSE function evaluations.
Default: `max_sse_eval` = 400

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

`IMSLS_TOLERANCE`, *float* `tolerance` (Input)
: False convergence tolerance.
Default: `tolerance` = 100* eps, where eps = `imsls_f_machine`(4) if single precision and eps = `imsls_d_machine`(4) if double precision

`IMSLS_PREDICTED`, *float* `**predicted` (Output)
: Address of a pointer to a real internally allocated array of length `n_observations` containing the predicted values at the approximate solution.

`IMSLS_PREDICTED_USER`, *float* `predicted[]` (Output)
: Storage for array `predicted` is provided by the user. See `IMSLS_PREDICTED`.

`IMSLS_RESIDUAL`, *float* `**residual` (Output)
: Address of a pointer to a real internally allocated array of length `n_observations` containing the residuals at the approximate solution.

`IMSLS_RESIDUAL_USER`, *float* `residual[]` (Output)
: Storage for array `residual` is provided by the user. See `IMSLS_RESIDUAL`.

IMSLS_R, *float* **r (Output)
Address of a pointer to an internally allocated array of size n_parameters × n_parameters containing the *R* matrix from a *QR* decomposition of the Jacobian.

IMSLS_R_USER, *float* r[] (Output)
Storage for array r is provided by the user. See IMSLS_R.

IMSLS_R_COL_DIM, *int* r_col_dim (Input)
Column dimension of array r.
Default: r_col_dim = n_parameters

IMSLS_R_RANK, *int* *rank (Output)
Rank of r. Argument rank less than n_parameters may indicate the model is overparameterized.

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of x.
Default: x_col_dim = n_independent

IMSLS_DF, *int* *df (Output)
Degrees of freedom.

IMSLS_SSE, *float* *sse (Output)
Residual sum of squares.

IMSLS_RETURN_USER, *float* theta_hat[] (Output)
User-allocated array of length n_parameters containing the estimated regression coefficients.

IMSLS_FCN_W_DATA, *float* fcn (*int* n_independent, *float* xi[], *int* n_parameters, *float* theta[]), *void* *data, (Input)
User-supplied function to evaluate the function that defines the nonlinear regression problem, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

IMSLS_JACOBIAN_W_DATA, *void* jacobian (*int* n_independent, *float* xi[], *int* n_parameters, *float* theta[], *float* fjac[]), *void* *data, (Input)
User-supplied function to compute the *i*-th row of the Jacobian, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Function imsls_f_nonlinear_regression fits a nonlinear regression model using least squares. The nonlinear regression model is

$$y_i = f(x_i; \theta) + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable, the known x_i's are the vectors of the values of the independent (explanatory) variables, θ is the vector of p regression parameters, and the ε_i's are independently distributed normal errors with mean 0 and variance σ^2. For this model, a least-squares estimate of θ is also a maximum likelihood estimate of θ.

The residuals for the model are as follows:

$$e_i(\theta) = y_i - f(x_i;\, \theta) \qquad i = 1, 2, ..., n$$

A value of θ that minimizes

$$\sum_{i=1}^{n} \left[e_i(\theta)\right]^2$$

is a least-squares estimate of θ. Function `imsls_f_nonlinear_regression` is designed so that the values of the function $f(x_i;\, \theta)$ are computed one at a time by a user-supplied function.

Function `imsls_f_nonlinear_regression` is based on MINPACK routines LMDIF and LMDER by Moré et al. (1980) that use a modified Levenberg-Marquardt method to generate a sequence of approximations to a minimum point. Let

$$\hat{\theta}_c$$

be the current estimate of θ. A new estimate is given by

$$\hat{\theta}_c + s_c$$

where s_c is a solution to the following:

$$(J(\hat{\theta}_c)^T J(\hat{\theta}_c) + \mu_c I)s_c = J(\hat{\theta}_c)^T e(\hat{\theta}_c)$$

Here

$$J(\hat{\theta}_c)$$

is the Jacobian evaluated at

$$\hat{\theta}_c$$

The algorithm uses a "trust region" approach with a step bound of δ_c. A solution of the equations is first obtained for

$$\mu_c = 0. \text{ If } \|s_c\|_2 < \delta_c$$

this update is accepted; otherwise, μ_c is set to a positive value and another solution is obtained. The method is discussed by Levenberg (1944), Marquardt (1963), and Dennis and Schnabel (1983, pp. 129–147, 218–338).

If a user-supplied function is specified in `IMSLS_JACOBIAN`, the Jacobian is computed analytically; otherwise, forward finite differences are used to estimate the Jacobian numerically. In the latter case, especially if type *float* is used, the estimate of the Jacobian may be so poor that the algorithm terminates at a noncritical point. In such instances, the user should either supply a Jacobian function, use type *double*, or do both.

Programming Notes

Nonlinear regression allows substantial flexibility over linear regression because the user can specify the functional form of the model. This added flexibility can cause unexpected convergence problems for users that are unaware of the limitations of the software. Also, in many cases, there are possible remedies that may not be immediately obvious. The following is a list of possible convergence problems and some remedies. There is not a one-to-one correspondence between the problems and the remedies. Remedies for some problems also may be relevant for the other problems.

1. A local minimum is found. Try a different starting value. Good starting values often can be obtained by fitting simpler models. For example, for a nonlinear function

   $$f(x;\theta) = \theta_1 e^{\theta_2 x}$$

   good starting values can be obtained from the estimated linear regression coefficients

   $$\hat{\beta}_0$$

   and

   $$\hat{\beta}_1$$

   from a simple linear regression of ln *y* on ln *x*. The starting values for the nonlinear regression in this case would be

   $$\theta_1 = e^{\hat{\beta}_0} \text{ and } \theta_2 = \hat{\beta}_1$$

   If an approximate linear model is not clear, then simplify the model by reducing the number of nonlinear regression parameters. For example, some nonlinear parameters for which good starting values are known could be set to these values in order to simplify the model for computing starting values for the remaining parameters.

2. The estimate of θ is incorrectly returned as the same or very close to the initial estimate. This occurs often because of poor scaling of the problem, which might result in the residual sum of squares being either very large or very small relative to the precision of the computer. The optional arguments allow control of the scaling.

3. The model is discontinuous as a function of θ. (The function *f*(*x*;θ) can be a discontinuous function of *x*.)

4. Overflow occurs during the computations. Make sure the user-supplied functions do not overflow at some value of θ.

5. The estimate of θ is going to infinity. A parameterization of the problem in terms of reciprocals may help.
6. Some components of θ are outside known bounds. This can sometimes be handled by making a function that produces artificially large residuals outside of the bounds (even though this introduces a discontinuity in the model function).

Examples

Example 1

In this example (Draper and Smith 1981, p. 518), the following nonlinear model is fit:

$$Y = \alpha + \left(0.49 - \alpha\right) e^{-\beta(X-8)} + \varepsilon$$

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

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

void main ()
{
#define N_OBSERVATIONS 4
    int         n_independent  = 1;
    int         n_parameters   = 2;
    float       *theta_hat;
    float       x[N_OBSERVATIONS][1] = {10.0, 20.0, 30.0, 40.0};
    float       y[N_OBSERVATIONS] = {0.48, 0.42, 0.40, 0.39};

                                /* Nonlinear regression */
    theta_hat = imsls_f_nonlinear_regression(fcn, n_parameters,
        N_OBSERVATIONS, n_independent, (float *)x, y, 0);

                                /* Print estimates */
    imsls_f_write_matrix("estimated coefficients", 1, n_parameters,
        theta_hat, 0);

}                               /* End of main */

float fcn(int n_independent, float x[], int n_parameters, float theta[])
{
    return (theta[0] + (0.49 - theta[0])*exp(theta[1]*(x[0] - 8)));
}                               /* End of fcn */
```

Output

```
estimated coefficients
         1           2
    0.3807     -0.0794
```

Example 2

Consider the nonlinear regression model and data set discussed by Neter et al. (1983, pp. 475–478):

$$y_i = \theta_1 e^{\theta_2 x_i} + \varepsilon_i$$

There are two parameters and one independent variable. The data set considered consists of 15 observations.

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

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

void main()
{
#define N_OBSERVATIONS 15
    int             n_independent=1;
    int             n_parameters= 2;
    float           *theta_hat, *r, *y_hat;
    float           grad_eps = 1.0e-3;
    float           theta_guess[2] = {60.0, -0.03};
    float           y[N_OBSERVATIONS] = {
                        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           x[N_OBSERVATIONS] = {
                         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 };
    char            *fmt="%12.5e";

                                /* Nonlinear regression */
    theta_hat = imsls_f_nonlinear_regression(fcn, n_parameters,
        N_OBSERVATIONS,  n_independent, x, y,
        IMSLS_THETA_GUESS, theta_guess,
        IMSLS_GRADIENT_EPS, grad_eps,
        IMSLS_R, &r,
        IMSLS_PREDICTED, &y_hat,
        IMSLS_JACOBIAN, jacobian,
        0);

                                /* Print results */
    imsls_f_write_matrix("Estimated coefficients", 1, n_parameters,
        theta_hat, 0);

    imsls_f_write_matrix("Predicted values", 1, N_OBSERVATIONS,
        y_hat, 0);

    imsls_f_write_matrix("R matrix", n_parameters, n_parameters,
        r, IMSLS_WRITE_FORMAT, "%10.2f", 0);

}                               /* End of main */


float fcn(int n_independent, float x[], int n_parameters, float theta[])
```

```
{
    return (theta[0]*exp(x[0]*theta[1]));
}                               /* End of fcn */

void jacobian(int n_independent, float x[], int n_parameters,
    float theta[], float fjac[])
{
    fjac[0] = exp(theta[1]*x[0]);
    fjac[1] = theta[0]*x[0]*exp(theta[1]*x[0]);
}
                              /* End of jacobian */
```

Output

```
Estimated coefficients
         1           2
     58.61       -0.04

                           Predicted values
         1           2           3           4           5           6
     54.15       48.08       44.42       39.45       33.67       27.62

         7           8           9          10          11          12
     20.94       17.18       15.26       13.02        9.87        7.48

        13          14          15
      7.19        5.45        4.47

        R matrix
            1           2
1        1.87     1139.93
2        0.00     1139.80
```

Informational Errors

`IMSLS_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_eps" is too big.

Warning Errors

`IMSLS_LITTLE_FCN_CHANGE`	Both the actual and predicted relative reductions in the function are less than or equal to the relative function tolerance.
`IMSLS_TOO_MANY_ITN`	Maximum number of iterations exceeded.
`IMSLS_TOO_MANY_FCN_EVAL`	Maximum number of function evaluations exceeded.

`IMSLS_TOO_MANY_JACOBIAN_EVAL`	Maximum number of Jacobian evaluations exceeded.
`IMSLS_UNBOUNDED`	Five consecutive steps have been taken with the maximum step length.
`IMSLS_FALSE_CONVERGENCE`	The iterates appear to be converging to a noncritical point.

nonlinear_optimization

Fits data to a nonlinear model (possibly with linear constraints) using the successive quadratic programming algorithm (applied to the sum of squared errors, $sse = \Sigma(y_i - f(x_i; \theta))^2$) and either a finite difference gradient or a user-supplied gradient.

Synopsis

#include `<imsls.h>`

float `*imsls_f_nonlinear_optimization` (*float* `fcn()`,
int `n_parameters`, *int* `n_observations`, *int* `n_independent`,
float `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_nonlinear_optimization`.

Required Arguments

float `fcn` (*int* `n_independent`, *float* `xi[]`, *int* `n_parameters`,
float `theta[]`)
User-supplied function to evaluate the function that defines the nonlinear regression problem where `xi` is an array of length `n_independent` at which point the function is evaluated and `theta` is an array of length `n_parameters` containing the current values of the regression coefficients. Function `fcn` returns a predicted value at the point `xi`. In the following, $f(x_i; \theta)$, or just f_i, denotes the value of this function at the point x_i, for a given value of θ. (Both x_i and θ are arrays.)

int `n_parameters` (Input)
Number of parameters to be estimated.

int `n_observations` (Input)
Number of observations.

int `n_independent` (Input)
Number of independent variables.

float `*x` (Input)
Array of size `n_observations` by `n_independent` containing the matrix of independent (explanatory) variables.

float `y[]` (Input)
Array of length `n_observations` containing the dependent (response) variable.

Return Value

A pointer to an array of length `n_parameters` containing a solution, $\hat{\theta}$ for the nonlinear regression coefficients. To release this space, use `free`. If no solution can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_nonlinear_optimization (float fcn(),
       int n_parameters, int n_observations, int
       n_independent, float x[], float y[],
       IMSLS_THETA_GUESS, float theta_guess[],
       IMSLS_JACOBIAN, void jacobian(),
       IMSLS_SIMPLE_LOWER_BOUNDS, float theta_lb[],
       IMSLS_SIMPLE_UPPER_BOUNDS, float theta_ub[],
       IMSLS_LINEAR_CONSTRAINTS, int n_constraints,
              int n_equality, float a[], float b[],
       IMSLS_FREQUENCIES, float frequencies,
       IMSLS_WEIGHTS, float weights,
       IMSLS_ACC, float acc,
       IMSLS_MAX_SSE_EVALUATIONS, int *max_sse_eval,
       IMSLS_PRINT_LEVEL, int print_level,
       IMSLS_STOP_INFO, int *stop_info,
       IMSLS_ACTIVE_CONSTRAINTS_INFO, int *n_active,
              int **indices_active, float **multiplier,
       IMSLS_ACTIVE_CONSTRAINTS_INFO_USER, int *n_active,
              int indices_active[], float multiplier[],
       IMSLS_PREDICTED, float **predicted,
       IMSLS_PREDICTED_USER, float predicted[],
       IMSLS_RESIDUAL, float **residual,
       IMSLS_RESIDUAL_USER, float residual[],
       IMSLS_SSE, float *sse,
       IMSLS_RETURN_USER, float theta_hat[],
       IMSLS_FCN_W_DATA, float fcn(), void *data,
       IMSLS_JACOBIAN_W_DATA, float jacobian(), void *data,
       0)
```

Optional Arguments

`IMSLS_THETA_GUESS`, *float* `theta_guess[]` (Input)
Array with `n_parameters` components containing an initial guess.
Default: `theta_guess`[] = 0

`IMSLS_JACOBIAN`, *void* `jacobian` (*int* `n_independent`, *float* `xi[]`, *int* `n_parameters`, *float* `theta[]`, *float* `fjac[]`) (Input/Output)
User-supplied function to compute the *i*-th row of the Jacobian, where the `n_independent` data values corresponding to the *i*-th row are input in `xi`. Argument `theta` is an array of length `n_parameters` containing the regression coefficients for which the Jacobian is evaluated, `fjac` is

the computed `n_parameters` row of the Jacobian for observation *i* at `theta`. Note that each derivative $f(x_i)/\theta$ should be returned in `fjac[j-1]` for $i = 1, 2, ...,$ `n_parameters`. Further note that in order to maintain consistency with the other nonlinear solver, `nonlinear_regression`, the Jacobian values must be specified as the *negative* of the calculated derivatives.

`IMSLS_SIMPLE_LOWER_BOUNDS`, *float* `theta_lb[]` (Input)
Vector of length `n_parameters` containing the lower bounds on the parameters; choose a very large negative value if a component should be unbounded below or set `theta_lb[i]` = `theta_ub[i]` to freeze the *i*-th variable.
Default: All parameters are bounded below by -10^6.

`IMSLS_SIMPLE_UPPER_BOUNDS`, *float* `theta_ub[]` (Input)
Vector of length `n_parameters` containing the upper bounds on the parameters; choose a very large value if a component should be unbounded above or set `theta_lb[i]` = `theta_ub[i]` to freeze the *i*-th variable.
Default: All parameters are bounded above by 10^6.

`IMSLS_LINEAR_CONSTRAINTS`, *int* `n_constraints`, *int* `n_equality`, *float* `a[]`, *float* `b[]` (Input)
Argument `n_constraints` is the total number of linear constraints (excluding simple bounds). Argument `n_equality` is the number of these constraints which are *equality* constraints; the remaining `n_constraints` – `n_equality` constraints are *inequality* constraints. Argument `a` is a `n_constraints` by `n_parameters` array containing the equality constraint gradients in the first `n_equality` rows, followed by the inequality constraint gradients. Argument `b` is a vector of length `n_constraints` containing the right-hand sides of the linear constraints.

Specifically, the constraints on θ are:
$a_{i1}\,\theta_1 + ... + a_{ij}\,\theta_j = b_i$ for $i = 1$, `n_equality` and $j = 1$, `n_parameter`, and

$a_{k1}\,\theta_1 + ... + a_{kj}\,\theta_j \le b_k$ for $k =$ `n_equality` + 1, `n_constraints` and $j = 1$, `n_parameter`.

Default: There are no default linear constraints.

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of length `n_observations` containing the frequency for each observation.
Default: `frequencies[]` = 1

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_observations` containing the weight for each observation.
Default: `weights[]` = 1

IMSLS_ACC, *float* acc (Input)

The nonnegative tolerance on the first order conditions at the calculated solution.

IMSLS_MAX_SSE_EVALUATIONS, *int* *max_sse_eval (Input/Output)

On input max_sse_eval is the maximum number of sse evaluations allowed. On output, max_sse_eval contains the actual number of sse evaluations needed.
Default: max_sse_eval = 400

IMSLS_PRINT_LEVEL, *int* print_level (Input)

Argument print_level specifies the frequency of printing during execution. If print_level = 0, there is no printing. Otherwise, after ensuring feasibility, information is printed every print_level iterations and whenever an internal tolerance (called *tol*) is reduced. The printing provides the values of theta and the sse and gradient at the value of theta. If print_level is negative, this information is augmented by the current values of indices_active, multiplier, and *reskt*, where *reskt* is the Kuhn-Tucker residual vector at theta.

IMSLS_STOP_INFO, *int* *stop_info (Output)

Argument stop_info will have one of the following integer values to indicate the reason for leaving the routine:

stop_info	Reason for leaving routine
1	θ is feasible, and the condition that depends on acc is satisfied.
2	θ is feasible, and rounding errors are preventing further progress.
3	θ is feasible, but sse fails to decrease although a decrease is predicted by the current gradient vector.
4	The calculation cannot begin because a contains fewer than n_constraints constraints or because the lower bound on a variable is greater than the upper bound.
5	The equality constraints are inconsistent. These constraints include any components of $\hat{\theta}$ that are frozen by setting theta_lb[i] equal to theta_ub[i].
6	The equality constraints and the bound on the variables are found to be inconsistent.
7	There is no possible θ that satisfies all of the constraints.

stop_info	Reason for leaving routine
8	Maximum number of sse evaluations (max_sse_eval) is exceeded.
9	θ is determined by the equality constraints.

IMSLS_ACTIVE_CONSTRAINTS_INFO, *int* *n_active, *int* **indices_active, *float* **multiplier (Output)
Argument n_active returns the final number of active constraints. Argument indices_active is the address of a pointer to an internally allocated integer array of length n_active containing the indices of the final active constraints. Argument multiplier is the address of a pointer to an internally allocated real array of length n_active containing the Lagrange multiplier estimates of the final active constraints.

IMSLS_ACTIVE_CONSTRAINTS_INFO_USER, *int* *n_active, *int* indices_active[], *float* multiplier[] (Output)
Storage for arrays indices_active and multiplier are provided by the user. The maximum length needed for these arrays is n_constraints. See IMSLS_ACTIVE_CONSTRAINTS_INFO.

IMSLS_PREDICTED, *float* **predicted (Output)
Address of a pointer to a real internally allocated array of length n_observations containing the predicted values at the approximate solution.

IMSLS_PREDICTED_USER, *float* predicted[] (Output)
Storage for array predicted is provided by the user. See IMSLS_PREDICTED.

IMSLS_RESIDUAL, *float* **residual (Output)
Address of a pointer to a real internally allocated array of length n_observations containing the residuals at the approximate solution.

IMSLS_RESIDUAL_USER, *float* residual[] (Output)
Storage for array residual is provided by the user. See IMSLS_RESIDUAL.

IMSLS_SSE, *float* *sse (Output)
Residual sum of squares.

IMSLS_RETURN_USER, *float* theta_hat[] (Output)
User-allocated array of length n_parameters containing the estimated regression coefficients.

IMSLS_FCN_W_DATA, *float* fcn (*int* n_independent, *float* xi[], *int* n_parameters, *float* theta[]), *void* *data, (Input)
User-supplied function to evaluate the function that defines the nonlinear regression problem, which also accepts a pointer to data that is supplied

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

`IMSLS_JACOBIAN_W_DATA`, *void* `jacobian` (*int* `n_independent`, *float* `xi[]`, *int* `n_parameters`, *float* `theta[]`, *float* `fjac[]`), *void* `*data`, (Input)
User-supplied function to compute the *i*-th row of the Jacobian, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Function `imsls_f_nonlinear_optimization` is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form min $f(\theta)$, $\theta \in \Re$, subject to

$$A_1\theta = b_1$$

$$A_2\theta \le b_2$$

$$\theta_l \le \theta \le \theta_u$$

given the vectors b_1, b_2, θ_l, and θ_u and the matrices A_1 and A_2.

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

$$A_1\theta = b_1$$

Next, θ^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 θ^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

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

subject to

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

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

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

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

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

and

$$(d^k)^T \nabla f(\theta^k + \alpha^k d^k) \ge 0.7\, (d^k)^T \nabla f(\theta^k)$$

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

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

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

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

The iteration repeats until the stopping criterion

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

is satisfied; here, τ is a user-supplied tolerance. For more details, see Powell (1988, 1989).

Since 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. Also, whenever the exact gradient can be easily provided, the gradient should be passed to `imsls_f_nonlinear_optimization` using the optional argument `IMSLS_JACOBIAN`.

Examples

Example 1

In this example, a data set is fitted to the nonlinear model function

$$y_i = \sin(\theta_0 x_i) + \varepsilon_i$$

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

float fcn(int n_independent, float x[], int n_parameters, float theta[]);

main()
{
    int     n_parameters   =  1;
    int     n_observations = 11;
    int     n_independent  =  1;
    float   *theta_hat;
    float   x[11] = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
                     0.7, 0.8, 0.9, 1.0};
    float   y[15] = {0.05, 0.21, 0.67, 0.72, 0.98, 0.94,
                     1.00, 0.73, 0.44, 0.36, 0.02};

    theta_hat =
        imsls_f_nonlinear_optimization(fcn, n_parameters,
                                       n_observations, n_independent, x, y,
                                       0);

    imsls_f_write_matrix("Theta Hat", 1, n_parameters, theta_hat, 0);

    free(theta_hat);
}

float fcn(int n_independent, float x[], int n_parameters, float theta[])
{
   return sin(theta[0]*x[0]);
}
```

Output

```
 Theta Hat

     3.161
```

Example 2

Draper and Smith (1981, p. 475) state a problem due to Smith and Dubey. [H. Smith and S. D. Dubey (1964), "Some reliability problems in the chemical industry", Industrial Quality Control, 21 (2), 1964, pp. 64–70] A certain product must have 50% available chlorine at the time of manufacture. When it reaches the customer 8 weeks later, the level of available chlorine has dropped to 49%. It was known that the level should stabilize at about 30%. To predict how long the chemical would last at the customer site, samples were analyzed at different times. It was postulated that the following nonlinear model should fit the data.

$$y_i = \theta_0 + \left(0.49 - \theta\right) e^{-\theta(x_i - 8)} + \varepsilon_i$$

Since the chlorine level will stabilize at about 30%, the initial guess for theta1 is 0.30. Using the last data point ($x = 42$, $y = 0.39$) and $\theta_0 = 0.30$ and the above nonlinear equation, an estimate for θ_1 of 0.02 is obtained.

The constraints that $\theta_0 \geq = 0$ and $\theta_1 \geq = 0$ are also imposed. These are equivalent to requiring that the level of available chlorine always be positive and never increase with time.

The Jacobian of the nonlinear model equation is also used.

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


float fcn(int n_independent, float x[], int n_parameters, float theta[]);
void jacobian(int n_independent, float x[], int n_parameters,
              float theta[],
float fjac[]);
main()
{
    int     n_parameters   =  2;
    int     n_observations = 44;
    int     n_independent  =  1;
    float   *theta_hat;
    float   x[44] = {
        8.0, 8.0, 10.0, 10.0, 10.0, 10.0, 12.0, 12.0, 12.0,
        12.0, 14.0, 14.0, 14.0, 16.0, 16.0, 16.0, 18.0, 18.0, 20.0,
        20.0, 20.0, 22.0, 22.0, 22.0, 24.0, 24.0, 24.0, 26.0, 26.0,
        26.0, 28.0, 28.0, 30.0, 30.0, 30.0, 32.0, 32.0, 34.0, 36.0,
        36.0, 38.0, 38.0, 40.0, 42.0};
    float   y[44] = {
        .49, .49, .48, .47, .48, .47, .46, .46, .45, .43, .45,
        .43, .43, .44, .43, .43, .46, .45, .42, .42, .43, .41, .41,
        .4, .42, .4, .4, .41, .4, .41, .41, .4, .4, .4, .38, .41,
        .4, .4, .41, .38, .4, .4, .39, .39};
    float   guess[2] =  {0.30, 0.02};
    float   xlb[2] = {0.0, 0.0};
    float   sse;

    theta_hat =
        imsls_f_nonlinear_optimization(fcn, n_parameters, n_observations,
                                       n_independent, x, y,
                                       IMSLS_THETA_GUESS, guess,
                                       IMSLS_SIMPLE_LOWER_BOUNDS, xlb,
                                       IMSLS_JACOBIAN, jacobian,
                                       IMSLS_SSE, &sse,
                                       0);
    imsls_f_write_matrix("Theta Hat", 1, 2, theta_hat, 0);
    free(theta_hat);
}

float fcn(int n_independent, float x[], int n_parameters, float theta[])
{
    return  theta[0] + (0.49-theta[0])*exp(-theta[1]*(x[0]-8.0));
}



void jacobian(int n_independent, float x[], int n_parameters,
              float theta[],
float fjac[])
```

```
{
    fjac[0] = -1.0 + exp(-theta[1]*(x[0]-8.0));
    fjac[1] = (0.49-theta[0])*(x[0]-8.0) * exp(-theta[1]*(x[0]-8.0));
}
```

Output

```
      Theta Hat

        1           2

    0.3901      0.1016
```

Fatal Errors

`IMSLS_BAD_CONSTRAINTS_1`	The equality constraints are inconsistent.
`IMSLS_BAD_CONSTRAINTS_2`	The equality constraints and the bounds on the variables are found to be inconsistent.
`IMSLS_BAD_CONSTRAINTS_3`	No vector “theta” satisfies all of the constraints. Specifically, the current active constraints prevent any change in “theta” that reduces the sum of constraint violations.
`IMSLS_BAD_CONSTRAINTS_4`	The variables are determined by the equality constraints.
`IMSLS_TOO_MANY_ITERATIONS_1`	Number of function evaluations exceeded “maxfcn” = #.

Lnorm_regression

Fits a multiple linear regression model using criteria other than least squares. Namely, `imsls_f_Lnorm_regression` allows the user to choose Least Absolute Value (L_1), Least L$_p$ norm (L_p), or Least Maximum Value (Minimax or L_∞) method of multiple linear regression.

Synopsis

#include `<imsls.h>`

float `*imsls_f_Lnorm_regression` (*int* `n_rows`, *int* `n_independent`,
float `x[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_Lnorm_regression`.

Required Arguments

int `n_rows` (Input)
: Number of rows in `x`.

int `n_independent` (Input)
: Number of independent (explanatory) variables.

float `x[]` (Input)
: Array of size `n_rows` × `n_independent` containing the independent (explanatory) variables(s). The *i*-th column of *x* contains the *i*-th independent variable.

float `y[]` (Input)
: Array of size `n_rows` containing the dependent (response) variable.

Return Value

`imsls_f_Lnorm_regression` returns a pointer to an array of length `n_independent` + 1 containing a least absolute value solution for the regression coefficients. The estimated intercept is the initial component of the array, where the *i*-th component contains the regression coefficients for the *i*-th dependent variable. If the optional argument `IMSLS_NO_INTERCEPT` is used then the *(i-1)*-s*t* component contains the regression coefficients for the *i*-th dependent variable. `imsls_f_Lnorm_regression` returns the L_p norm or least maximum value solution for the regression coefficients when appropriately specified in the optional argument list.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_Lnorm__regression(int n_rows, int n_independent,
      float x[], float y[],
      IMSLS_METHOD_LAV,
      IMSLS_METHOD_LLP, float p,
      IMSLS_METHOD_LMV,
      IMSLS_X_COL_DIM, int x_col_dim,
      IMSLS_INTERCEPT,
      IMSLS_NO_INTERCEPT,
      IMSLS_RANK, int *rank,
      IMSLS_ITERATIONS, int *iterations,
      IMSLS_N_ROWS_MISSING, int *n_rows_missing,
      IMSLS_TOLERANCE, float tolerence,
      IMSLS_SEA, float *sum_lav_error,
      IMSLS_MAX_RESIDUAL,  float *max_residual,
      IMSLS_R, float **R_matrix,
      IMSLS_R_USER, float R_matrix[],
      IMSLS_DEGREES_OF_FREEDOM, float df_error,
      IMSLS_RESIDUALS, float **residual,
      IMSLS_RESIDUALS_USER, float residual[],
      IMSLS_SCALE, float *square_of_scale,
      IMSLS_RESIDUALS_LP_NORM, float *Lp_norm_residual,
      IMSLS_EPS, float epsilon,
      IMSLS_WEIGHTS,  float weights[],
      IMSLS_FREQUENCIES, float  frequencies[],
      IMSLS_RETURN_USER, float coefficients[],
      0)
```

Optional Arguments

`IMSLS_METHOD_LAV`, *or*

`IMSLS_METHOD_LLP`, *float* `p`, (Input) *or*

`IMSLS_METHOD_LMV`,
By default (or if `IMSLS_METHOD_LAV` is specified) the function fits a multiple linear regression model using the least absolute values criterion.

`IMSLS_METHOD_LLP` requires the argument *p*, for $p \geq 1$, and fits a multiple linear regression model using the L_p norm criterion.

`IMSLS_METHOD_LMV` fits a multiple linear regression model using the minimax criterion.

`IMSLS_WEIGHTS`, *float* `weights[]`, (Input)
Array of size `n_rows` containing the weights for the independent (explanatory) variable.

`IMSLS_FREQUENCIES`, *float* `frequencies[]`, (Input)
Array of size `n_rows` containing the frequencies for the independent (explanatory) variable.

`IMSLS_X_COL_DIM`, *int* `x_col_dim`, (Input)
Leading dimension of `x` exactly as specified in the dimension statement in the calling program.

`IMSLS_INTERCEPT`, *or*
`IMSLS_NO_INTERCEPT`,
`IMSLS_INTERCEPT` is the default where the fitted value for observation *i* is

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_k x_k$$

where k = `n_independent`. If `IMSLS_NO_INTERCEPT` is specified, the intercept term

$$\left(\hat{\beta}_0\right)$$

is omitted from the model and the return value from regression is a pointer to an array of length `n_independent`.

`IMSLS_RANK`, *int* `*rank`, (Output)
Rank of the fitted model is returned in `*rank`.

`IMSLS_ITERATIONS`, *int* `*iterations`, (Output)
Number of iterations performed.

`IMSLS_N_ROWS_MISSING`, *int* `*n_rows_missing`, (Output)
Number of rows of data containing NaN (not a number) for the dependent or independent variables. If a row of data contains NaN for any of these variables, that row is excluded from the computations.

`IMSLS_RETURN_USER`, *float* `coefficients[]` (Output)
Storage for array `coefficients` is provided by the user.
See Return Value.

If `IMSLS_METHOD_LAV` is specified:
`IMSLS_SEA`, *float* `sum_lav_error`, (Output)
Sum of the absolute value of the errors.

If `IMSLS_METHOD_LMV` is specified:
`IMSLS_MAX_RESIDUAL`, *float* `max_residual`, (Output)
Magnitude of the largest residual.

If `IMSLS_METHOD_LLP` is specified:
`IMSLS_TOLERANCE`, *float* `tolerence`, (Input)
Tolerance used in determining linear dependence.
`tolerence = 100 * imsls_f_machine(4)` is the default.
For more details see Chapter 14, "Utilities" function `imsls_f_machine`.

`IMSLS_R`, *float* `**R_matrix`, (Output)
Upper triangular matrix of dimension (number of coeffieciencts by number of coeffecients) containing the R matrix from a QR decomposition of the matrix of regressors.

`IMSLS_R_USER`, *float* `R_matrix[]`, (Output)
Storage for array `R_matrix` is provided by the user. See `IMSLS_R`..

`IMSLS_DEGREES_OF_FREEDOM`, *float* `df_error`, (Output)
Sum of the frequencies minus `*rank`. In least squares fit (p =2) `df_error` is called the degrees of freedom of error.

`IMSLS_RESIDUALS`, *float* `**residual`, (Output)
Address of a pointer to an array (of length equal to the number of observations) containing the residuals.

`IMSLS_RESIDUALS_USER`, *float* `residual[]`, (Output)
Storage for array residual is provided by the user.
See `IMSLS_RESIDUALS`.

`IMSLS_SCALE,` *float* `*square_of_scale,` (Output)
Square of the scale constant used in an *Lp* analysis. An estimated asymptotic variance-covariance matrix of the regression coefficients is `square_of_scale` * $(R^TR)^{-1}$.

`IMSLS_RESIDUALS_LP_NORM,` *float* `*Lp_norm_residual,` (Output)
L_p norm of the residuals.

`IMSLS_EPS,` *float* `epsilon,` (Input)
Convergence criterion. If the maximum relative difference in residuals from the `k-th` to `(k+1)`-st iterations is less than `epsilon`, convergence is declared. `epsilon = 100 * machine(4)` is the default.

Description

Least Absolute Value Criterion

Function `imsls_f_Lnorm_regression` computes estimates of the regression coefficients in a multiple linear regression model. For optional argument `IMSLS_LAV` (default), the criterion satisfied is the minimization of the sum of the absolute values of the deviations of the observed response y_i from the fitted response

$$\hat{y}_i$$

for a set on *n* observations. Under this criterion, known as the L_1 or LAV (least absolute value) criterion, the regression coefficient estimates minimize

$$\sum_{i=0}^{n-1} \left| y_i - \hat{y}_i \right|$$

The estimation problem can be posed as a linear programming problem. The special nature of the problem, however, allows for considerable gains in efficiency by the modification of the usual simplex algorithm for linear programming. These modifications are described in detail by Barrodale and Roberts (1973, 1974).

In many cases, the algorithm can be made faster by computing a least-squares solution prior to the invocation of `IMSLS_LAV`. This is particularly useful when a least-squares solution has already been computed. The procedure is as follows:

1. Fit the model using least squares and compute the residuals from this fit.
2. Fit the residuals from Step 1 on the regressor variables in the model using `IMSLS_LAV`.

3 Add the two estimated regression coefficient vectors from Steps 1 and 2. The result is an L_1 solution.

When multiple solutions exist for a given problem, option `IMSLS_LAV` may yield different estimates of the regression coefficients on different computers, however, the sum of the absolute values of the residuals should be the same (within rounding differences). The informational error indicating nonunique solutions may result from rounding accumulation. Conversely, because of rounding the error may fail to result even when the problem does have multiple solutions.

L_p Norm Criterion

Optional argument `IMSLS_LLP` computes estimates of the regression coefficients in a multiple linear regression model $y = X\beta + \varepsilon$ under the criterion of minimizing the L_p norm of the deviations for $i = 0, \ldots, n$-1 of the observed response y_i from the fitted response

$$\hat{y}_i$$

for a set on n observations and for $p \geq 1$. For the case when `IMSLS_WEIGHTS AND IMSLS_FREQUENCIES` are not supplied, the estimated regression coefficient vector,

$$\hat{\beta}$$

(output in `coefficients []`) minimizes the L_p norm

$$\left(\sum_{i=0}^{n-1} \left| y_i - \hat{y}_i \right|^P \right)^{1/p}$$

The choice $p = 1$ yields the maximum likelihood estimate for β when the errors have a Laplace distribution. The choice $p = 2$ is best for errors that are normally distributed. Sposito (1989, pages 36–40) discusses other reasonable alternatives for p based on the sample kurtosis of the errors.

Weights are useful if the errors in the model have known unequal variances

$$\sigma_i^2$$

In this case, the weights should be taken as

$$w_i = 1/\sigma_i^2$$

Frequencies are useful if there are repetitions of some observations in the data set. If a single row of data corresponds to n_i observations, set the frequency $f_i = n_i$. In general, `IMSLS_LLP` minimizes the L_p norm

$$\left(\sum_{i=0}^{n-1} f_i \left| \sqrt{w_i} \left(y_i - \hat{y}_i \right) \right|^p \right)^{1/p}$$

The asymptotic variance-covariance matrix of the estimated regression coefficients is given by

$$\text{asy.var}(\hat{\beta}) = \lambda^2 (R^T R)^{-1}$$

where R is from the QR decomposition of the matrix of regressors (output in R-Matrix) ere an estimate of λ^2 is output in square_of_scale.

In the discussion that follows, we will first present the algorithm with frequencies and weights all taken to be one. Later, we will present the modifications to handle frequencies and weights different from one.

Option call IMSLS_LLP uses Newton's method with a line search for $p > 1.25$ and, for $p \le 1.25$, uses a modification due to Ekblom (1973, 1987) in which a series of perturbed problems are solved in order to guarantee convergence and increase the convergence rate. The cutoff value of 1.25 as well as some of the other implementation details given in the remaining discussion were investigated by Sallas (1990) for their effect on CPU times.

In each case, for the first iteration a least-squares solution for the regression coefficients is computed using routine imsls_f_regression (page 64). If $p = 2$, the computations are finished. Otherwise, the residuals from the k-th iteration,

$$e_i^{(k)} = y_i - \hat{y}_i^{(k)}$$

are used to compute the gradient and Hessian for the Newton step for the $(k + 1)$-st iteration for minimizing the p-th power of the L_p norm. (The exponent $1/p$ in the L_p norm can be omitted during the iterations.)

For subsequent iterations, we first discuss the $p > 1.25$ case. For $p > 1.25$, the gradient and Hessian at the $(k + 1)$-st iteration depend upon

$$z_i^{(k+1)} = \left|e_i^{(k)}\right|^{p-1} \operatorname{sign}\left(e_i^{(k)}\right)$$

and

$$v_i^{(k+1)} = \left|e_i^{(k)}\right|^{p-2}$$

In the case $1.25 < p < 2$ and

$$e_i^{(k)} = 0,\ v_i^{(k+1)}$$

and the Hessian are undefined; and we follow the recommendation of Merle and Spath (1974). Specifically, we modify the definition of

$$v_i^{(k+1)}$$

to the following:

$$v_i^{(k+1)} = \begin{cases} \tau^{p-2} & \text{if } p < 2 \text{ and } \left|e_i^{(k)}\right| < \tau \\ \left|e_i^{(k)}\right|^{p-2} & \text{otherwise} \end{cases}$$

where τ equals 100 * imsls_f_machine(4) (or 100.0 * imsls_d_machine(4) for the double precision version) times the square root of the residual mean square from the least-squares fit. (See routines imsls_f_machine and `imsls_d_machine` which are documented in the section “Machine-Dependent Constants” in Reference Material.)

Let $V^{(k+1)}$ be a diagonal matrix with diagonal entries

$$v_i^{(k+1)}$$

and let $z^{(k+1)}$ be a vector with elements

$$z_i^{(k+1)}$$

In order to compute the step on the (*k* + 1)-st iteration, the *R* from the *QR* decomposition of

$$[V^{(k+1)}]^{1/2}X$$

is computed using fast Givens transformations. Let

$$R^{(k+1)}$$

denote the upper triangular matrix from the *QR* decomposition. The linear system

$$[R(^{k}+1)]^{T}R(^{k}+1)d(^{k}+1)= X^{T} z(^{k}+1)$$

is solved for

$$d^{(k+1)}$$

where $R^{(k+1)}$ is from the *QR* decomposition of $V^{(k+1)}]^{1/2}X$. The step taken on the (*k* + 1)-st iteration is

$$\hat{\beta}^{(k+1)} = \hat{\beta}^{(k)} + \alpha^{(k+1)} \frac{1}{p-1} d^{(k+1)}$$

The first attempted step on the (*k* + 1)-st iteration is with $\alpha^{(k+1)} = 1$. If all of the

$$e_i^{(k)}$$

are nonzero, this is exactly the Newton step. See Kennedy and Gentle (1980, pages 528−529) for further discussion.

If the first attempted step does not lead to a decrease of at least one-tenth of the predicted decrease in the *p*-th power of the L_p norm of the residuals, a backtracking linesearch procedure is used. The backtracking procedure uses a one-dimensional quadratic model to estimate the backtrack constant *p*. The value of *p* is constrained to be no less that 0.1. An approximate upper bound for *p* is 0.5. If after 10 successive backtrack attempts, $\alpha^{(k)} = p_1 p_2 \ldots p_{10}$ does not produce a step with a sufficient decrease, then `imsls_f_Lnorm_regression` issues a message with error code 5. For further details on the backtrack line-search procedure, see Dennis and Schnabel (1983, pages 126−127).

Convergence is declared when the maximum relative change in the residuals from one iteration to the next is less than or equal to `epsilon`. The relative change

$$\delta_i^{(k+1)}$$

in the *i*-th residual from iteration *k* to iteration *k* + 1 is computed as follows:

$$\delta_i^{(k+1)} = \begin{cases} 0 & \text{if } e_i^{(k+1)} = e_i^{(k)} = 0 \\ \left|e_i^{(k+1)} - e_i^{(k)}\right| / \max(\left|\mathrm{e}_i^{(k)}\right|, \left|e_i^{(k+1)}\right|, s) & \text{otherwise} \end{cases}$$

where *s* is the square root of the residual mean square from the least-squares fit on the first iteration.

For the case $1 \le p \le 1.25$, we describe the modifications to the previous procedure that incorporate Ekblom's (1973) results. A sequence of perturbed problems are solved with a successively smaller perturbation constant *c*. On the first iteration, the least-squares problem is solved. This corresponds to an infinite *c*. For the second problem, *c* is taken equal to *s*, the square root of the residual mean square from the least-squares fit. Then, for the (*j* + 1)-st problem, the value of *c* is computed from the previous value of *c* according to

$$c_{j+1} = c_j / 10^{5p-4}$$

Each problem is stated as

$$Minimize \sum_{i=0}^{n-1} (e_i^2 + c^2)^{p/2}$$

For each problem, the gradient and Hessian on the (*k* + 1)-st iteration depend upon

$$z_i^{(k+1)} = e_i^{(k)} r_i^{(k)}$$

and

$$v_i^{(k+1)} = \left[1 + \frac{(p-2)(e_i^{(k)})^2}{(e_i^{(k)})^2 + c^2} \right] r_i^{(k)}$$

where

$$r_i^{(k)} = \left[(e_i^{(k)})^2 + c^2 \right]^{(p-2)/2}$$

The linear system $[R^{(k+1)}]^T R^{(k+1)} d^{(k+1)} = X^T z^{(k+1)}$ is solved for $d^{(k+1)}$ where $R^{(k+1)}$ is from the *QR* decomposition of $[V^{(k+1)}]^{1/2} X$. The step taken on the (*k* + 1)-st iteration is

$$\hat{\beta}^{(k+1)} = \hat{\beta}^{(k)} + \alpha^{(k+1)} d^{(k+1)}$$

where the first attempted step is with $\alpha^{(k+1)} = 1$. If necessary, the backtracking line-search procedure discussed earlier is used.

Convergence for each problem is relaxed somewhat by using a convergence epsilon equal to max(`epsilon`, 10^{-j}) where $j = 1, 2, 3, \ldots$ indexes the problems ($j = 0$ corresponds to the least-squares problem).

After the convergence of a problem for a particular c, Ekblom's (1987) extrapolation technique is used to compute the initial estimate of β for the new problem. Let $R^{(k)}$,

$$v_i^{(k)}, e_i^{(k)}$$

and c be from the last iteration of the last problem. Let

$$t_i = \frac{(p-2)v_i^{(k)}}{(e_i^{(k)})^2 + c^2}$$

and let t be the vector with elements t_i. The initial estimate of β for the new problem with perturbation constant $0.01c$ is

$$\hat{\beta}^{(0)} = \hat{\beta}^{(k)} + \Delta c d$$

where $\Delta c = (0.01c - c) = -0.99c$, and where d is the solution of the linear system $[R^{(k)}]^T R^{(k)} d = X^T t$.

Convergence of the sequence of problems is declared when the maximum relative difference in residuals from the solution of successive problems is less than `epsilon`.

The preceding discussion was limited to the case for which `weights[i]` = 1 and `frequencies[i]` = 1, i.e., the weights and frequencies are all taken equal to one. The necessary modifications to the preceding algorithm to handle weights and frequencies not all equal to one are as follows:

1. Replace

   $$e_i^{(k)} \text{ by } \sqrt{w_i}e_i^{(k)}$$

   in the definitions of

   $$z_i^{(k+1)}, v_i^{(k+1)}, \delta_i^{(k+1)}$$

   and t_i.

2. Replace

   $$z_i^{(k+1)} \text{ by } f_i\sqrt{w_i}z_i^{(k+1)}, v_i^{(k+1)} \text{ by } f_i w_i v_i^{(k+1)}, \text{ and } t_i^{(k+1)} \text{ by } f_i\sqrt{w_i}t_i^{(k+1)}$$

These replacements have the same effect as multiplying the i-th row of X and y by

$$\sqrt{w_i}$$

and repeating the row f_i times except for the fact that the residuals returned by `imsls_f_Lnorm_regression` are in terms of the original y and X.

Finally, R and an estimate of λ^2 are computed. Actually, R is recomputed because on output it corresponds to the R from the initial QR decomposition for least squares. The formula for the estimate of λ^2 depends on p.

For $p = 1$, the estimator for λ^2 is given by (McKean and Schrader 1987)

$$\hat{\lambda}^2 = \left[\frac{\sqrt{\mathrm{DFE}}\,(\tilde{e}_{(\mathrm{DFE}-k+1)} - \tilde{e}_{(k)})}{2z_{0.975}} \right]^2$$

with

$$k = \frac{\mathrm{DFE} + k}{2} - z_{0.975}\sqrt{\frac{\mathrm{DFE}}{4}}$$

where $z_{0.975}$ is the 97.5 percentile of the standard normal distribution, and where

$$\tilde{\varepsilon}_{(m)}(m = 1, 2, \ldots, DFE)$$

are the ordered residuals where rank zero residuals are excluded. Note that

$$DFE = \sum\nolimits_{i=0}^{n-1} f_i - \mathrm{rank}$$

For $p = 2$, the estimator of λ^2 is the customary least-squares estimator given by

$$s^2 = \frac{SSE}{DFE} = \frac{\sum_{i=0}^{n-1} f_i w_i (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1} f_i - \mathrm{rank}}$$

For $1 < p < 2$ and for $p > 2$, the estimator for λ^2 is given by (Gonin and Money 1989)

$$\hat{\omega}_p^2 = \frac{m_{2p-2}}{\left[(p-1)m_{p-2}\right]^2}$$

with

$$m_r = \frac{\sum_{i=1}^{n} f_i \left|\sqrt{w_i}\,(y_i - \hat{y}_i)\right|^r}{\sum_{i=0}^{n-1} f_i}$$

Least Minimum Value Criterion (minimax)

Optional call `IMSLS_LMV` computes estimates of the regression coefficients in a multiple linear regression model. The criterion satisfied is the minimization of the maximum deviation of the observed response y_i from the fitted response $\hat{y}_i$ for a set on n observations. Under this criterion, known as the minimax or LMV (least maximum value) criterion, the regression coefficient estimates minimize

$$\max_{0 \le i \le n-1} \left| y_i - \hat{y}_i \right|$$

The estimation problem can be posed as a linear programming problem. A dual simplex algorithm is appropriate, however, the special nature of the problem allows for considerable gains in efficiency by modification of the dual simplex iterations so as to move more rapidly toward the optimal solution. The modifications are described in detail by Barrodale and Phillips (1975).

When multiple solutions exist for a given problem, `IMSLS_LMV` may yield different estimates of the regression coefficients on different computers, however, the largest residual in absolute value should have the same absolute value (within rounding differences). The informational error indicating nonunique solutions may result from rounding accumulation. Conversely, because of rounding, the error may fail to result even when the problem does have multiple solutions.

Example 1

A straight line fit to a data set is computed under the LAV criterion.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
    float xx[] = {1.0, 4.0, 2.0, 2.0, 3.0, 3.0, 4.0, 5.0};
    float yy[] = {1.0, 5.0, 0.0, 2.0, 1.5, 2.5, 2.0, 3.0};
    float sea;
    int irank, iter, nrmiss;

    float *coefficients = NULL;

    coefficients = imsls_f_Lnorm_regression(8, 1, xx, yy,
                                  IMSLS_SEA, &sea,
                                  IMSLS_RANK, &irank,
                                  IMSLS_ITERATIONS, &iter,
                                  IMSLS_N_ROWS_MISSING, &nrmiss,0);

    printf("B = %6.2f\t%6.2f\n\n", coefficients[0], coefficients[1]);
    printf("Rank of Regressors Matrix   = %3d\n", irank);
    printf("Sum Absolute Value of Error = %8.4f\n", sea);
    printf("Number of Iterations        = %3d\n", iter);
    printf("Number of Rows Missing      = %3d\n", nrmiss);

}
```

Output

```
B =    0.50        0.50
Rank of Regressors Matrix        =     2
Sum Absolute Value of Error      =     6.00000
Number of Iterations             =     2
Number of Rows Missing           =     0
```

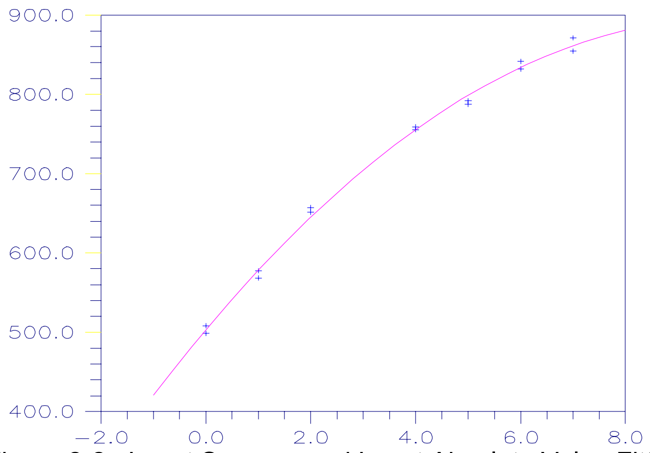

Figure 2-2   Least Squares and Least Absolute Value Fitted Lines

Example 2

Different straight line fits to a data set are computed under the criterion of minimizing the L_p norm by using p equal to 1, 1.5, 2.0 and 2.5.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
    float xx[] = {1.0, 4.0, 2.0, 2.0, 3.0, 3.0, 4.0, 5.0};
    float yy[] = {1.0, 5.0, 0.0, 2.0, 1.5, 2.5, 2.0, 3.0};
    float p, tolerance, convergence_eps, square_of_scale, df_error,&
                                                Lp_norm_residual;
    float R_matrix[4], residuals[8];
    int   i, irank, iter, nrmiss;

    int   n_row=2;
    int   n_col=2;

    float *coefficients = NULL;

    tolerance = 100*imsls_f_machine(4);
    convergence_eps = 0.001;
    p = 1.0;
    for(i=0; i<4; i++)
    {
    coefficients = imsls_f_Lnorm_regression(8, 1, xx, yy,
                             IMSLS_METHOD_LLP, p,
                             IMSLS_EPS, convergence_eps,
                             IMSLS_RANK, &irank,
                             IMSLS_ITERATIONS, &iter,
                             IMSLS_N_ROWS_MISSING, &nrmiss,
                             IMSLS_R_USER, R_matrix,
                             IMSLS_DEGREES_OF_FREEDOM, &df_error,
                             IMSLS_RESIDUALS_USER, residuals,
                             IMSLS_SCALE, &square_of_scale,
                             IMSLS_RESIDUALS_LP_NORM, &Lp_norm_residual,
                              0);
    printf("Coefficients = %6.2f\t%6.2f\n\n", coefficients[0], coefficients[1]);
    printf("Residuals = %6.2f\t%6.2f\t%6.2f\t%6.2f\t%6.2f\t%6.2f\t%6.2f\t%6.2f\n\n",
             residuals[0], residuals[1], residuals[2], residuals[3],
             residuals[4], residuals[5], residuals[6], residuals[7]);
    printf("P                          = %5.3f\n", p);
    printf("Lp norm of the residuals    = %5.3f\n", Lp_norm_residual);
    printf("Rank of Regressors Matrix   = %3d\n", irank);
    printf("Degrees of Freedom Error    = %5.3f\n", df_error);
    printf("Number of Iterations        = %3d\n", iter);
    printf("Number of Missing Values    = %3d\n", nrmiss);
    printf("Square of Scale Constant    = %5.3f\n", square_of_scale);

    imsls_f_write_matrix("R Matrix\n", n_row, n_col, R_matrix, 0);
    printf("-----------------------------------------------------\n\n");
    p += 0.5;
    }

}
```

Output

```
Coefficients    0.50    0.50
Residuals    0.00    2.50   -1.50    0.50   -0.50    0.50   -0.50   0.00

p                                1.00
Lp norm of the residuals         6.00
Rank of the matrix of regressors    2
Degrees of freedom error         6.00
Number of iterations                8
Number of missing values            0
Square of the scale constant     6.25

   R matrix
        1       2
1   2.828   8.485
2   0.000   3.464

------------------------------------------------------------------------

Coefficients    0.39    0.55

Residuals    0.06    2.39   -1.50    0.50   -0.55    0.45   -0.61   -0.16
p                                1.50
Lp norm of the residuals         3.71
Rank of the matrix of regressors    2
Degrees of freedom error         6.00
Number of iterations                6
Number of missing values            0
Square of the scale constant     1.06

   R matrix
  1       2
1   2.828   8.485
2   0.000   3.464

------------------------------------------------------------------------
```

```
Coefficients   -0.12    0.75
Residuals    0.38    2.12   -1.38    0.62   -0.62    0.38   -0.88   -0.62

p                                2.00
Lp norm of the residuals         2.94
Rank of the matrix of regressors    2
Degrees of freedom error         6.00
Number of iterations                1
Number of missing values            0
Square of the scale constant     1.44

   R matrix
        1       2
1   2.828   8.485
2   0.000   3.464

------------------------------------------------------------------------

Coefficients   -0.44    0.87
Residuals    0.57    1.96   -1.30    0.70   -0.67    0.33   -1.04   -0.91
p                                2.50
Lp norm of the residuals         2.54
Rank of the matrix of regressors    2
Degrees of freedom error         6.00
Number of iterations                4
Number of missing values            0
Square of the scale constant     0.79

   R matrix
        1       2
1   2.828   8.485
2   0.000   3.464
```

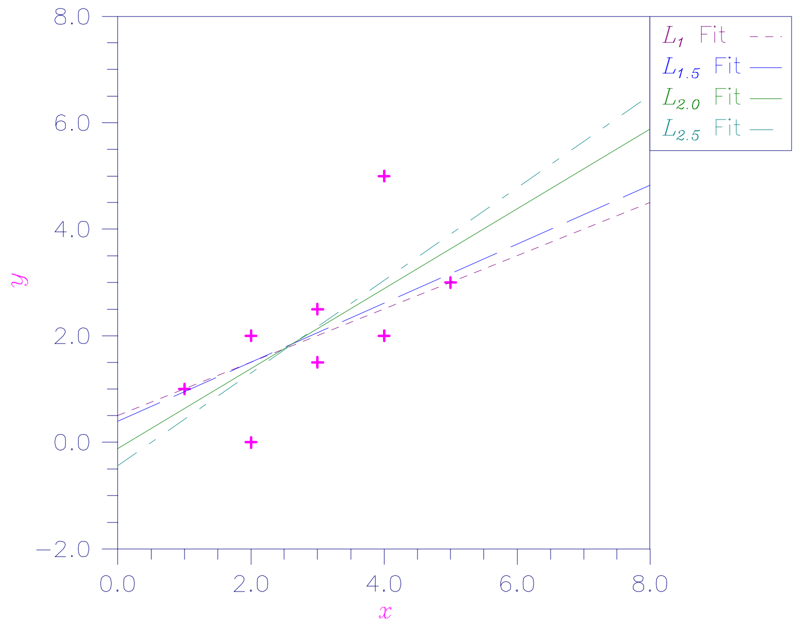

Figure 2-3 Various L_p Fitted Lines

Example 3

A straight line fit to a data set is computed under the LMV criterion.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
    float xx[] = {0.0, 1.0, 2.0, 3.0, 4.0, 4.0, 5.0};
    float yy[] = {0.0, 2.5, 2.5, 4.5, 4.5, 6.0, 5.0};
    float max_residual;
    int irank, iter, nrmiss;

    float *coefficients = NULL;

    coefficients = imsls_f_Lnorm_regression(7, 1, xx, yy,
                                  IMSLS_METHOD_LMV,
                                  IMSLS_MAX_RESIDUAL, &max_residual,
                                   IMSLS_RANK, &irank,
                                   IMSLS_ITERATIONS, &iter,
                                   IMSLS_N_ROWS_MISSING, &nrmiss,
                                   0);
    printf("B = %6.2f\t%6.2f\n\n", coefficients[0], coefficients[1]);
    printf("Rank of Regressors Matrix     = %3d\n", irank);
    printf("Magnitude of Largest Residual = %8.4f\n", max_residual);
    printf("Number of Iterations          = %3d\n", iter);
    printf("Number of Rows Missing        = %3d\n", nrmiss);

}
```

Output

```
B =    1.00          1.00
Rank of Regressors Matrix          =   2
Magnitude of Largest Residual      =  1.00000
Number of Iterations               =   3
Number of Rows Missing             =   0
```

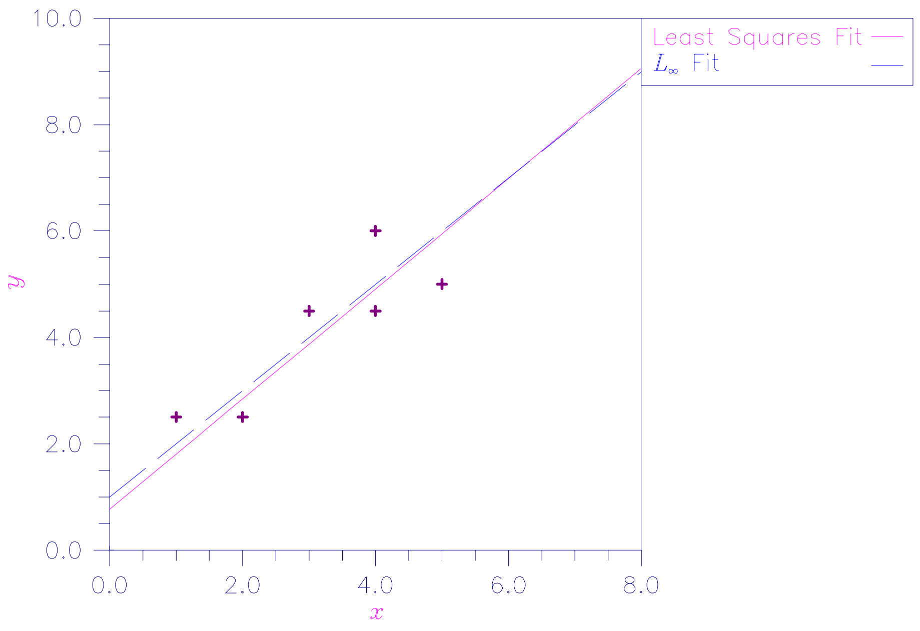

Figure 2-4 Least Squares and Least Maximum Value Fitted Lines

Chapter 3: Correlation and Covariance

Routines

Usage Notes

This chapter is concerned with measures of correlation for bivariate data as follows:

- The usual multivariate measures of correlation and covariance for continuous random variables are produced by routine `imsls_f_covariances`.
- For data grouped by some auxiliary variable, routine `imsls_f_pooled_covariances` can be used to compute the pooled covariance matrix along with the means for each group.
- Partial correlations or covariances are computed by `imsls_f_partial_correlations`.
- Function `imsls_f_robust_covariances` computes robust M-estimates of the mean and covariance matrix from a matrix of observations.

covariances

Computes the sample variance-covariance or correlation matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_covariances` (*int* `n_rows`, *int* `n_variables`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_covariances`.

Required Arguments

int `n_rows` (Input)
Number of rows in `x`.

int `n_variables` (Input)
Number of variables.

float `x[]` (Input)
Array of size `n_rows` × `n_variables` containing the data.

Return Value

If no optional arguments are used, `imsls_f_covariances` returns a pointer to an `n_variables` × `n_variables` array containing the sample variance-covariance matrix of the observations. The rows and columns of this array correspond to the columns of `x`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_covariances` (*int* `n_rows`, *int* `n_variables`, *float* `x[]`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_MISSING_VALUE_METHOD`, *int* `missing_value_method`,
`IMSLS_INCIDENCE_MATRIX`, *int* `**incidence_matrix`,
`IMSLS_INCIDENCE_MATRIX_USER`, *int* `incidence_matrix[]`,
`IMSLS_N_OBSERVATIONS`, *int* `*n_observations`,
`IMSLS_VARIANCE_COVARIANCE_MATRIX`, *or*
`IMSLS_CORRECTED_SSCP_MATRIX`, *or*
`IMSLS_CORRELATION_MATRIX`, *or*
`IMSLS_STDEV_CORRELATION_MATRIX`,
`IMSLS_MEANS`, *float* `**means`,
`IMSLS_MEANS_USER`, *float* `means[]`,
`IMSLS_COVARIANCE_COL_DIM`, *int* `covariance_col_dim`,
`IMSLS_FREQUENCIES`, *float* `frequencies[]`,
`IMSLS_WEIGHTS`, *float* `weights[]`,
`IMSLS_SUM_WEIGHTS`, *float* `*sumwt`,
`IMSLS_N_ROWS_MISSING`, *int* `*nrmiss`,
`IMSLS_RETURN_USER`, *float* `covariance[]`,
0)

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of array `x`.
Default: `x_col_dim` = `n_variables`

`IMSLS_MISSING_VALUE_METHOD`, *int* `missing_value_method` (Input)
Method used to exclude missing values in `x` from the computations, where NaN is interpreted as the missing value code. See function `imsls_f_machine`/`imsls_d_machine` (Chapter 14). The methods are as follows:

missing_value_method	**Action**
0	The exclusion is listwise. (The entire row of `x` is excluded if any of the values of the row is equal to the missing value code.)
1	Raw crossproducts are computed from all valid pairs and means, and variances are computed from all valid data on the individual variables. Corrected crossproducts, covariances, and correlations are computed using these quantities.
2	Raw crossproducts, means, and variances are computed as in the case of `missing_value_method` = 1. However, corrected crossproducts and covariances are computed only from the valid pairs of data. Correlations are computed using these covariances and the variances from all valid data.
3	Raw crossproducts, means, variances, and covariances are computed as in the case of `missing_value_method` = 2. Correlations are computed using these covariances, but the variances used are computed from the valid pairs of data.

`IMSLS_INCIDENCE_MATRIX`, *int* `**incidence_matrix` (Output)
Address of a pointer to an internally allocated array containing the incidence matrix. If `missing_value_method` is 0, `incidence_matrix` is 1 × 1 and contains the number of valid observations; otherwise, `incidence_matrix` is `n_variables` × `n_variables` and contains the number of pairs of valid observations used in calculating the crossproducts for covariance.

`IMSLS_INCIDENCE_MATRIX_USER`, *int* `incidence_matrix[]` (Output)
Storage for array `incidence_matrix` is provided by the user. See `IMSLS_INCIDENCE_MATRIX`.

`IMSLS_N_OBSERVATIONS`, *int* `*n_observations` (Output)
Sum of the frequencies. If `missing_value_method` is 0, observations with missing values are not included in `n_observations`; otherwise, all observations are included except for observations with missing values for the weight or the frequency.

`IMSLS_VARIANCE_COVARIANCE_MATRIX`, *or*
`IMSLS_CORRECTED_SSCP_MATRIX`, *or*
`IMSLS_CORRELATION_MATRIX`, *or*
`IMSLS_STDEV_CORRELATION_MATRIX`
Exactly one of these options can be used to specify the type of matrix to be computed.

Keyword	**Type of Matrix**
`IMSLS_VARIANCE_COVARIANCE_MATRIX`	variance-covariance matrix (default)
`IMSLS_CORRECTED_SSCP_MATRIX`	corrected sums of squares and crossproducts matrix
`IMSLS_CORRELATION_MATRIX`	correlation matrix
`IMSLS_STDEV_CORRELATION_MATRIX`	correlation matrix except for the diagonal elements which are the standard deviations

`IMSLS_MEANS`, *float* `**means` (Output)
Address of a pointer to the internally allocated array containing the means of the variables in `x`. The components of the array correspond to the columns of `x`.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
Storage for array means is provided by the user. See `IMSLS_MEANS`.

`IMSLS_COVARIANCE_COL_DIM`, *int* `covariance_col_dim` (Input)
Column dimension of array covariance if `IMSLS_RETURN_USER` is specified; otherwise, the column dimension of the return value.
Default: `covariance_col_dim` = `n_variables`

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of length `n_observations` containing the frequency for each observation.
Default: `frequencies` [ ] = 1

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_observations` containing the weight for each observation.
Default: `weights` [ ] = 1

`IMSLS_SUM_WEIGHTS`, *float* `*sum_wt` (Output)
Sum of the weights of all observations. If `missing_value_method` is equal to 0, observations with missing values are not included in `sum_wt`.

Otherwise, all observations are included except for observations with mssing values for the weight or the frequency.

`IMSLS_N_ROWS_MISSING`, *int* `*nrmiss` (Output)
Total number of observations that contain any missing values (NaN).

`IMSLS_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

Function `imsls_f_covariances` computes estimates of correlations, covariances, or sums of squares and crossproducts for a data matrix *x*. Weights and frequencies are allowed but not required.

The means, (corrected) sums of squares, and (corrected) sums of crossproducts are computed using the method of provisional means. Let x_{ki} denote the mean based on *i* observations for the *k*-th variable, f_i denote the frequency of the *i*-th observation, w_i denote the weight of the *i*-th observations, and 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 shown in the example below.

The means and crossproducts are initialized as follows:

$$x_{k0} = 0.0 \quad \text{for } k = 1, \ldots, p$$

$$c_{jk0} = 0.0 \text{ for } j, k = 1, \ldots, p$$

where *p* denotes the number of variables. Letting $x_{k,i+1}$ denote the *k*-th variable of observation $i + 1$, each new observation leads to the following updates for x_{ki} and c_{jki} using the update constant r_{i+1}:

$$r_{i+1} = \frac{f_{i+1} w_{i+1}}{\sum_{l=1}^{i+1} f_l w_l}$$

$$\bar{x}_{k,\,i+1} = \bar{x}_{ki} + \left(x_{k,\,i+1} - \bar{x}_{ki}\right) r_{i+1}$$

$$c_{jk,\,i+1} = c_{jki} + f_{i+1} w_{i+1} \left(x_{j,\,i+1} - \bar{x}_{ji}\right)\left(x_{k,\,i+1} - \bar{x}_{ki}\right)\left(1 - r_{i+1}\right)$$

The default value for weights and frequencies is 1. Means and variances are computed based on the valid data for each variable or, if required, based on all the valid data for each pair of variables.

Usage Notes

Function `imsls_f_covariances` defines a sample mean by

$$\bar{x}_k = \frac{\sum_{i=1}^{n} f_i w_i x_{ki}}{\sum_{i=1}^{n_r} f_i w_i}$$

where *n* is the number of observations.

The following formula defines the sample covariance, s_{jk}, between variables *j* and *k*:

$$s_{jk} = \frac{\sum_{i=1}^{n} f_i w_i \left(x_{ji} - \bar{x}_j\right)\left(x_{ki} - \bar{x}_k\right)}{\sum_{i=1}^{n} f_i - 1}$$

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

This example illustrates the use of `imsls_f_covariances` for the first 50 observations in the Fisher iris data (Fisher 1936). Note that the first variable is constant over the first 50 observations.

```
#include <imsls.h>

#define N_VARIABLES      5
#define N_OBSERVATIONS  50


main()
{
    float       *covariances, *means;
    float       x[] = {
        1.0, 5.1, 3.5, 1.4, .2,  1.0, 4.9, 3.0, 1.4, .2,
        1.0, 4.7, 3.2, 1.3, .2,  1.0, 4.6, 3.1, 1.5, .2,
        1.0, 5.0, 3.6, 1.4, .2,  1.0, 5.4, 3.9, 1.7, .4,
        1.0, 4.6, 3.4, 1.4, .3,  1.0, 5.0, 3.4, 1.5, .2,
        1.0, 4.4, 2.9, 1.4, .2,  1.0, 4.9, 3.1, 1.5, .1,
        1.0, 5.4, 3.7, 1.5, .2,  1.0, 4.8, 3.4, 1.6, .2,
        1.0, 4.8, 3.0, 1.4, .1,  1.0, 4.3, 3.0, 1.1, .1,
        1.0, 5.8, 4.0, 1.2, .2,  1.0, 5.7, 4.4, 1.5, .4,
        1.0, 5.4, 3.9, 1.3, .4,  1.0, 5.1, 3.5, 1.4, .3,
        1.0, 5.7, 3.8, 1.7, .3,  1.0, 5.1, 3.8, 1.5, .3,
        1.0, 5.4, 3.4, 1.7, .2,  1.0, 5.1, 3.7, 1.5, .4,
        1.0, 4.6, 3.6, 1.0, .2,  1.0, 5.1, 3.3, 1.7, .5,
        1.0, 4.8, 3.4, 1.9, .2,  1.0, 5.0, 3.0, 1.6, .2,
        1.0, 5.0, 3.4, 1.6, .4,  1.0, 5.2, 3.5, 1.5, .2,
        1.0, 5.2, 3.4, 1.4, .2,  1.0, 4.7, 3.2, 1.6, .2,
        1.0, 4.8, 3.1, 1.6, .2,  1.0, 5.4, 3.4, 1.5, .4,
        1.0, 5.2, 4.1, 1.5, .1,  1.0, 5.5, 4.2, 1.4, .2,
        1.0, 4.9, 3.1, 1.5, .2,  1.0, 5.0, 3.2, 1.2, .2,
        1.0, 5.5, 3.5, 1.3, .2,  1.0, 4.9, 3.6, 1.4, .1,
        1.0, 4.4, 3.0, 1.3, .2,  1.0, 5.1, 3.4, 1.5, .2,
        1.0, 5.0, 3.5, 1.3, .3,  1.0, 4.5, 2.3, 1.3, .3,
        1.0, 4.4, 3.2, 1.3, .2,  1.0, 5.0, 3.5, 1.6, .6,
        1.0, 5.1, 3.8, 1.9, .4,  1.0, 4.8, 3.0, 1.4, .3,
        1.0, 5.1, 3.8, 1.6, .2,  1.0, 4.6, 3.2, 1.4, .2,
        1.0, 5.3, 3.7, 1.5, .2,  1.0, 5.0, 3.3, 1.4, .2};

                                /* Perform analysis */
    covariances = imsls_f_covariances (N_OBSERVATIONS,
        N_VARIABLES, x, 0);

                                /* Print results */
    imsls_f_write_matrix ("The default case: variances/covariances",
        N_VARIABLES, N_VARIABLES, covariances,
        IMSLS_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, which uses the first 50 observations in the Fisher iris data, illustrates the use of optional arguments.

```
#include <imsls.h>

#define N_VARIABLES      5
#define N_OBSERVATIONS  50

main()
{
    char        *title;
    float       *means, *correlations;
    float       x[] = {
        1.0, 5.1, 3.5, 1.4, .2,  1.0, 4.9, 3.0, 1.4, .2,
        1.0, 4.7, 3.2, 1.3, .2,  1.0, 4.6, 3.1, 1.5, .2,
        1.0, 5.0, 3.6, 1.4, .2,  1.0, 5.4, 3.9, 1.7, .4,
        1.0, 4.6, 3.4, 1.4, .3,  1.0, 5.0, 3.4, 1.5, .2,
        1.0, 4.4, 2.9, 1.4, .2,  1.0, 4.9, 3.1, 1.5, .1,
        1.0, 5.4, 3.7, 1.5, .2,  1.0, 4.8, 3.4, 1.6, .2,
        1.0, 4.8, 3.0, 1.4, .1,  1.0, 4.3, 3.0, 1.1, .1,
        1.0, 5.8, 4.0, 1.2, .2,  1.0, 5.7, 4.4, 1.5, .4,
        1.0, 5.4, 3.9, 1.3, .4,  1.0, 5.1, 3.5, 1.4, .3,
        1.0, 5.7, 3.8, 1.7, .3,  1.0, 5.1, 3.8, 1.5, .3,
        1.0, 5.4, 3.4, 1.7, .2,  1.0, 5.1, 3.7, 1.5, .4,
        1.0, 4.6, 3.6, 1.0, .2,  1.0, 5.1, 3.3, 1.7, .5,
        1.0, 4.8, 3.4, 1.9, .2,  1.0, 5.0, 3.0, 1.6, .2,
        1.0, 5.0, 3.4, 1.6, .4,  1.0, 5.2, 3.5, 1.5, .2,
        1.0, 5.2, 3.4, 1.4, .2,  1.0, 4.7, 3.2, 1.6, .2,
        1.0, 4.8, 3.1, 1.6, .2,  1.0, 5.4, 3.4, 1.5, .4,
        1.0, 5.2, 4.1, 1.5, .1,  1.0, 5.5, 4.2, 1.4, .2,
        1.0, 4.9, 3.1, 1.5, .2,  1.0, 5.0, 3.2, 1.2, .2,
        1.0, 5.5, 3.5, 1.3, .2,  1.0, 4.9, 3.6, 1.4, .1,
        1.0, 4.4, 3.0, 1.3, .2,  1.0, 5.1, 3.4, 1.5, .2,
        1.0, 5.0, 3.5, 1.3, .3,  1.0, 4.5, 2.3, 1.3, .3,
        1.0, 4.4, 3.2, 1.3, .2,  1.0, 5.0, 3.5, 1.6, .6,
        1.0, 5.1, 3.8, 1.9, .4,  1.0, 4.8, 3.0, 1.4, .3,
        1.0, 5.1, 3.8, 1.6, .2,  1.0, 4.6, 3.2, 1.4, .2,
        1.0, 5.3, 3.7, 1.5, .2,  1.0, 5.0, 3.3, 1.4, .2};

                                /* Perform analysis */
    correlations = imsls_f_covariances (N_OBSERVATIONS,
        N_VARIABLES-1, x+1,
        IMSLS_STDEV_CORRELATION_MATRIX,
        IMSLS_X_COL_DIM, N_VARIABLES,
        IMSLS_MEANS, &means,
        0);

                                /* Print results */
    imsls_f_write_matrix ("Means\n", 1, N_VARIABLES-1, means, 0);
    title = "Correlations with Standard Deviations on the Diagonal\n";
    imsls_f_write_matrix (title, N_VARIABLES-1, N_VARIABLES-1,
        correlations, IMSLS_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

`IMSLS_CONSTANT_VARIABLE`	Correlations are requested, but the observations on one or more variables are constant. The corresponding correlations are set to NaN.
`IMSLS_INSUFFICIENT_DATA`	Variances and covariances are requested, but fewer than two valid observations are present for a variable. The pertinent statistics are set to NaN.

`IMSLS_ZERO_SUM_OF_WEIGHTS_2`	The sum of the weights is zero. The means, variances, and covariances are set to NaN.
`IMSLS_ZERO_SUM_OF_WEIGHTS_3`	The sum of the weights is zero. The means and correlations are set to NaN.
`IMSLS_TOO_FEW_VALID_OBS_CORREL`	Correlations are requested, but fewer than two valid observations are present for a variable. The pertinent correlation coefficients are set to NaN.

partial_covariances

Computes partial covariances or partial correlations from the covariance or correlation matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_partial_covariances` (*int* `n_independent`,
    int `n_dependent`, *float* `x`, ..., 0)

The type *double* function is `imsls_d_partial_covariances`.

Required Argument

int `n_independent` (Input)
: Number of "independent" variables to be used in the partial covariances/correlations. The partial covariances/correlations are the covariances/correlations between the dependent variables after removing the linear effect of the independent variables.

int `n_dependent` (Input)
: Number of variables for which partial covariances/correlations are desired (the number of "dependent" variables).

float `x` (Input)
: The $n \times n$ covariance or correlation matrix, where n = `n_independent` + `n_dependent`. The rows/columns must be ordered such that the first `n_independent` rows/columns contain the independent variables, and the last `n_dependent` row/columns contain the dependent variables. Matrix `x` must always be square symmetric.

Return Value

Matrix of size `n_dependent` by `n_dependent` containing the partial covariances (the default) or partial correlations (use keyword `IMSLS_PARTIAL_CORR`).

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_partial_covariances` (*int* `n_independent`,
    int `n_dependent`, *float* `x[]`,
    `IMSLS_X_COL_DIM`, *int* `x_col_dim`,
    `IMSLS_X_INDICES`, *int* `indices[]`,
    `IMSLS_PARTIAL_COV`, *or*
    `IMSLS_PARTIAL_CORR`,
    `IMSLS_TEST`, *int* `df`, *int* `*df_out`, *float* `**p_values`,
    `IMSLS_TEST_USER`, *int* df, *int* `*df_out`, *float* `p_values[]`,
    `IMSLS_RETURN_USER`, *float* `c[]`,
    0)

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Row/Column dimension of `x`.
Default: `x_col_dim` = `n_independent` + `n_dependent`.

`IMSLS_X_INDICES`, *int* `indices[]` (Input)
An array of length `x_col_dim` containing values indicating the status of the variable as in the following table:

indices[i]	Variable is...
−1	not used in analysis
0	dependent variable
1	independent variable

By default, the first `n_independent` elements of `indices` are equal to 1, and the last `n_dependent` elements are equal to 0.

`IMSLS_PARTIAL_COV`, *or*
`IMSLS_PARTIAL_CORR`,
By default, and if `IMSLS_PARTIAL_COV` is specified, partial covariances are calculated. Partial correlations are calculated if `IMSLS_PARTIAL_CORR` is specified.

`IMSLS_TEST`, *int* `df`, *int* `*df_out`, *float* `**p_values`
(Input, Output, Output)
Argument `df` is an input integer indicating the number of degrees of freedom associated with input matrix `x`. If the number of degrees of

freedom in x varies from element to element, then a conservative choice for df is the minimum degrees of freedom for all elements in x.

Argument df_out contains the number of degrees of freedom in the test that the partial covariances/correlations are zero. This value will usually be df – n_independent, but will be greater than this value if the independent variables are computationally linearly related.

Argument p_values is the address of a pointer to an internally allocated array of size n_dependent by n_dependent containing the *p*-values for testing the null hypothesis that the associated partial covariance/correlation is zero. It is assumed that the observations from which x was computed flows a multivariate normal distribution and that each element in x has df degrees of freedom.

IMSLS_TEST_USER, *int* df, *int* *df_out, *float* p_values[] (Input, Output, Output)
Storage for array p_values is provided by the user. See IMSLS_TEST above.

IMSLS_RETURN_USER, *float* c[] (Output)
If specified, c returns the partial covariances/correlations. Storage for array c is provided by the user.

Description

Function imsls_f_partial_covariances computed partial covariances or partial correlations from an input covariance or correlation matrix. If the "independent" variables (the linear "effect" of the independent variables is removed in computing the partial covariances/correlations) are linearly related to one another, imsls_f_partial_covariances detects the linearity and eliminates one or more of the independent variables from the list of independent variables. The number of variables eliminated, if any, can be determined from argument df_out.

Given a covariance or correlation matrix Σ partitioned as

$$\begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}$$

function imsls_f_partial_covariances computed the partial covariances (of the standardized variables if Σ is a correlation matrix) as

$$\Sigma_{22/1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}$$

If partial correlations are desired, these are computed as

$$P_{22/1} = \left[diag\left(\Sigma_{22/1}\right)\right]^{-1/2} \Sigma_{22/1} \left[diag\left(\Sigma_{22/1}\right)\right]^{-1/2}$$

where *diag* denotes the matrix containing the diagonal of its argument along its diagonal with zeros off the diagonal. If Σ_{11} is singular, then as many variables as required are deleted from Σ_{11} (and Σ_{12}) in order to eliminate the linear dependencies. The computations then proceed as above.

The *p*-value for a partial covariance tests the null hypothesis H_0: $\sigma_{ij|1} = 0$, where $\sigma_{ij|1}$ is the (i, j) element in matrix $\Sigma_{22|1}$. The *p*-value for a partial correlation tests the null hypothesis H_0: $\rho_{ij|1} = 0$, where $\rho_{ij|1}$ is the (i, j) element in matrix $P_{22|1}$. The *p*-values are returned in `p_values`. If the degrees of freedom for `x`, `df`, is not known, the resulting *p*-values may be useful for comparison, but they should not by used as an approximation to the actual probabilities.

Examples

Example 1

The following example computes partial covariances, scaled from a nine-variable correlation matrix originally given by Emmett (1949). The first three rows and columns contain the independent variables and the final six rows and columns contain the dependent variables.

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

main()
{
    float *pcov;
    float x[9][9] = {
        6.300, 3.050, 1.933, 3.365, 1.317, 2.293, 2.586, 1.242, 4.363,
        3.050, 5.400, 2.170, 3.346, 1.473, 2.303, 2.274, 0.750, 4.077,
        1.933, 2.170, 3.800, 1.970, 0.798, 1.062, 1.576, 0.487, 2.673,
        3.365, 3.346, 1.970, 8.100, 2.983, 4.828, 2.255, 0.925, 3.910,
        1.317, 1.473, 0.798, 2.983, 2.300, 2.209, 1.039, 0.258, 1.687,
        2.293, 2.303, 1.062, 4.828, 2.209, 4.600, 1.427, 0.768, 2.754,
        2.586, 2.274, 1.576, 2.255, 1.039, 1.427, 3.200, 0.785, 3.309,
        1.242, 0.750, 0.487, 0.925, 0.258, 0.768, 0.785, 1.300, 1.458,
        4.363, 4.077, 2.673, 3.910, 1.687, 2.754, 3.309, 1.458, 7.400};

    pcov = imsls_f_partial_covariances(3, 6, x, 0);

    imsls_f_write_matrix("Partial Covariances", 6, 6, pcov, 0);

    free(pcov);
    return;
}
```

Output

```
                          Partial Covariances
            1            2            3            4            5            6
1       0.000        0.000        0.000        0.000        0.000        0.000
2       0.000        0.000        0.000        0.000        0.000        0.000
3       0.000        0.000        0.000        0.000        0.000        0.000
4       0.000        0.000        0.000        5.495        1.895        3.084
5       0.000        0.000        0.000        1.895        1.841        1.476
6       0.000        0.000        0.000        3.084        1.476        3.403
```

Example 2

The following example computes partial correlations from a 9 variable correlation matrix originally given by Emmett (1949). The partial correlations between the remaining variables, after adjusting for variables 1, 3 and 9, are computed. Note in the output that the row and column labels are numbers, not variable numbers. The corresponding variable numbers would be 2, 4, 5, 6, 7 and 8, respectively.

```
#include <imsls.h>

main()
{
    float *pcorr, *pval;
    int   df;
    float x[9][9] = {
        1.0, 0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0, 0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0, .355, 0.27, 0.254, 0.452,  0.219, 0.504,
        0.471, 0.506, 0.355, 1.0, 0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27, 0.691, 1.0, 0.679,  0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0, 0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0, 0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0, 0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68, 0.47, 1.0};
    int indices[9] = {1, 0, 1, 0, 0, 0, 0, 0, 1};

    pcorr = imsls_f_partial_covariances(3, 6, &x[0][0],
                                    IMSLS_PARTIAL_CORR,
                                    IMSLS_X_INDICES, indices,
                                    IMSLS_TEST, 30, &df, &pval,
                                    0);

    printf ("The degrees of freedom are %d\n\n", df);
    imsls_f_write_matrix("Partial Correlations", 6, 6, pcorr, 0);
    imsls_f_write_matrix("P-Values", 6, 6, pval, 0);

    free(pcorr);
    free(pval);
    return;
}
```

Output

```
The degrees of freedom are 27

                          Partial Correlations
             1            2            3            4            5            6
1        1.000        0.224        0.194        0.211        0.125       -0.061
2        0.224        1.000        0.605        0.720        0.092        0.025
3        0.194        0.605        1.000        0.598        0.123       -0.077
4        0.211        0.720        0.598        1.000        0.035        0.086
5        0.125        0.092        0.123        0.035        1.000        0.062
6       -0.061        0.025       -0.077        0.086        0.062        1.000

                                P-Values
             1            2            3            4            5            6
```

```
1       0.0000      0.2525      0.3232      0.2801      0.5249      0.7576
2       0.2525      0.0000      0.0006      0.0000      0.6417      0.9000
3       0.3232      0.0006      0.0000      0.0007      0.5328      0.6982
4       0.2801      0.0000      0.0007      0.0000      0.8602      0.6650
5       0.5249      0.6417      0.5328      0.8602      0.0000      0.7532
6       0.7576      0.9000      0.6982      0.6650      0.7532      0.0000
```

Warning Errors

`IMSLS_NO_HYP_TESTS`	The input matrix "x" has # degrees of freedom, and the rank of the dependent variables is #. There are not enough degrees of freedom for hypothesis testing. The elements of "p_values" are set to NaN (not a number).

Fatal Errors

`IMSLS_INVALID_MATRIX_1`	The input matrix "x" is incorrectly specified. A computed correlation is greater than 1 for variables # and #.
`IMSLS_INVALID_PARTIAL`	A computed partial correlation for variables # and # is greater than 1. The input matrix "x" is not positive semi-definite.

pooled_covariances

Compute a pooled variance-covariance from the observations.

Synopsis

#include `<imsls.h>`

float `*imsls_f_pooled_covariances` (*int* `n_rows`, *int* `n_variables`,
float `*x`, *int* `n_groups`, ..., 0)

The type *double* function is `imsls_d_pooled_covariances`.

Required Argument

int `n_rows` (Input)
Number of rows observations) in the input matrix `x`.

int `n_variables` (Input)
Number of variables to be used in computing the covariance matrix.

float `*x` (Input)
A `n_rows` × `n_variables` + 1 matrix containing the data. The first `n_variables` columns correspond to the variables, and the last column (column `n_variables` must contain the group numbers).

int `n_groups` (Input)

Number of groups in the data.

Return Value

Matrix of size `n_variables` by `n_variables` containing the matrix of covariances.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_pooled_covariances (int n_rows, int n_variables,
        float x[], int n_groups,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_X_INDICES, int igrp, int ind[], int ifrq, int iwt,
        IMSLS_IDO, int ido,
        IMSLS_ROWS_ADD,
        IMSLS_ROWS_DELETE,
        IMSLS_GROUP_COUNTS, int **gcounts,
        IMSLS_GROUP_COUNTS_USER, int gcounts[],
        IMSLS_SUM_WEIGHTS, float **sum_weights,
        IMSLS_SUM_WEIGHTS_USER, float sum_weights[],
        IMSLS_MEANS_USER, float means[],
        IMSLS_U, float **u,
        IMSLS_U_USER, float u[],
        IMSLS_N_ROWS_MISSING, int *nrmiss,
        IMSLS_RETURN_USER, float c[],
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)

Default: `x_col_dim` = `n_variables` + 1

`IMSLS_X_INDICES`, *int* `igrp`, *int* `ind[]`, *int* `ifrq`, *int* `iwt` (Input)

Each of the four arguments contains indices indicating column numbers of `x` in which particular types of data are stored. Columns are numbered 0 ... `x_col_dim` − 1.

Parameter `igrp` contains the index for the column of `x` in which the group numbers are stored.

Parameter `ind` contains the indices of the variables to be used in the analysis.

Parameters `ifrq` and `iwt` contain the column numbers of `x` in which the frequencies and weights, respectively, are stored. Set `ifrq` = −1 if there will be no column for frequencies. Set `iwt` = −1 if there will be no column for weights. Weights are rounded to the nearest integer. Negative weights are not allowed.

Defaults: `igrp` = `n_variables`,
`ind`[ ] = 0, 1, …, `n_variables` – 1, `ifrq` = –1, and `iwt` = –1

`IMSLS_IDO`, *int* `ido` (Input)
Processing option.

ido	Action
0	This is the only invocation; all the data are input at once. (Default)
1	This is the first invocation with this data; additional calls will be made. Initialization and updating for the `n_rows` observations of `x` will be performed.
2	This is an intermediate invocation; updating for the `n_rows` observations of `x` will be performed.
3	All statistics are updated for the `n_rows` observations. The covariance matrix computed.

Default: `ido` = 0

`IMSLS_ROWS_ADD`, or
`IMSLS_ROWS_DELETE`
By default (or if `IMSLS_ROWS_ADD` is specified), the observations in `x` are added into the analysis. If `IMSLS_ROWS_DELETE` is specified, the observations are deleted from the analysis. If `ido` = 0, these optional arguments are ignored (data is always added if there is only one invocation).

`IMSLS_GROUP_COUNTS`, *int* `**gcounts` (Output)
Address of a pointer to an integer array of length `n_groups` containing the number of observations in each group. Array `gcounts` is updated when `ido` is equal to 0, 1, or 2.

`IMSLS_GROUP_COUNTS_USER`, *int* `gcounts[]` (Output)
Storage for integer array `gcounts` is provided by the user. See `IMSLS_GROUP_COUNTS`.

`IMSLS_SUM_WEIGHTS`, *float* `**sum_weights` (Output)
Address of a pointer to an array of length `n_groups` containing the sum of the weights times the frequencies in the groups.

`IMSLS_SUM_WEIGHTS_USER`, *float* `sum_weights[]` (Output)
Storage for array `sum_weights` is provided by the user. See `IMSLS_SUM_WEIGHTS`.

`IMSLS_MEANS`, *float* `**means` (Output)
Address of a pointer to an array of size `n_groups` × `n_variables`. The *i*-th row of `means` contains the group *i* variable means.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
Storage for array `means` is provided by the user. See `IMSLS_MEANS`.

`IMSLS_U`, *float* `**u` (Output)
Address of a pointer to an array of size `n_variables` × `n_variables` containing the lower matrix *U*, the lower triangular for the pooled sample cross-products matrix. *U* is computed from the pooled sample covariance matrix, *S* (See the "Description" section below), as $S = U^T U$.

`IMSLS_U_USER`, *float* `u[]` (Output)"
Storage for array `u` is provided by the user. See `IMSLS_U`.

`IMSLS_N_ROWS_MISSING`, *int* `*nrmiss` (Output)
Number of rows of data encountered in calls to `imsls_f_pooled_covariances` containing missing values (NaN) for any of the variables used.

`IMSLS_RETURN_USER`, *float* `c[]` (Output)
If specified, `c` returns the covariance matrix. Storage for array `c` is provided by the user.

Description

Function `imsls_f_pooled_covariances` computes the pooled variance-covariance matrix from a matrix of observations. The within-groups means are also computed. Listwise deletion of missing values is assumed so that all observations used are complete; in any row of `x`, if any element of the observation is missing, the row is not used. Function `imsls_f_pooled_covariances` should be used whenever the user suspects that the data has been sampled from populations with different means but identical variance-covariance matrices. If these assumptions cannot be made, a different variance-covariance matrix should be estimated within each group.

By default, all observations are processed in one call to `imsls_f_pooled_covariances`. The computations are the same as if `imsls_f_pooled_covariances` were consecutively called with `ido` equal to 1, 2, and 3. For brevity, the following discusses the computations with `ido` > 0.

When `ido` = 1 variables are initialized, workspace is allocated and input variables are checked for errrors.

If `n_rows` ≠ 0 (for any value of `ido`), the group observation totals, T_i, for $i = 1, \ldots, g$, where g is the number of groups, are updated for the `n_rows` observations in `x`. The group totals are computed as:

$$T_i = \sum_j w_{ij} f_{ij} x_{ij}$$

where w_{ij} is the observation weight, x_{ij} is the *j*-th observation in the *i*-th group, and f_{ij} is the observation frequency.

Modified Givens rotations are used in computed the Cholesky decomposition of the pooled sums of squares and crossproducts matrix. (Golub and Van Loan 1983).

The group means and the pooled sample covariance matrix S are computed from the intermediate results when `ido` = 3. These quantities are defined by

$$\overline{x_{i\bullet}} = \frac{T_i}{\sum_j w_i f_i}$$

$$S = \frac{1}{\sum_{ij} f_{ij} - g} \sum_{i,j} w_{ij} f_{ij} \left(x_{ij} - \bar{x}_{i\bullet} \right) \left(x_{ij} - \bar{x}_{ii\bullet} \right)^T$$

Examples

Example 1

The following example computes a pooled variance-covariance matrix. The last column of the data set is the group indicator.

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

main() {
    int nobs = 6;
    int nvar = 2;
    int n_groups = 2;
    float *cov;
    static float x[6][3] = {
        2.2, 5.6, 1,
        3.4, 2.3, 1,
        1.2, 7.8, 1,
        3.2, 2.1, 2,
        4.1, 1.6, 2,
        3.7, 2.2, 2};

    cov = imsls_f_pooled_covariances(nobs, nvar, &x[0][0], n_groups, 0);

    imsls_f_write_matrix("Pooled Covariance Matrix", nvar, nvar, cov, 0);
    free(cov);
}
```

Output

```
Pooled Covariance Matrix
            1           2
1       0.708      -1.575
2      -1.575       3.883
```

Example 2

The following example computes a pooled variance-covariance matrix for the Fisher iris data. To illustrate the use of the `ido` argument, multiple calls to `imsls_f_pooled_covariances` are made.

The first column of data is the group indicator, requiring either a permuation of the matrix or the use of the `IMSLS_X_INDICES` optional keyword. This exampe chooses the keyword method.

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

main() {
    int nobs = 150;
    int nvar = 4;
    int n_groups = 3;
    int igrp = 0;
    static int ind[4] = {1, 2, 3, 4};
    int ifrq = -1;
    int iwt = -1;
    float *x, cov[16];
    float *means;
    int i;

    /* Retrieve the Fisher iris data set */
    x = imsls_f_data_sets(3, 0);

    /* Initialize */
    imsls_f_pooled_covariances(0, nvar, x, n_groups,
        IMSLS_IDO, 1,
        IMSLS_RETURN_USER, cov,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);

    /* Add 10 rows at a time */
    for (i=0;i<15;i++) {
    imsls_f_pooled_covariances(10, nvar, (x+i*50), n_groups,
        IMSLS_IDO, 2,
        IMSLS_RETURN_USER, cov,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);
    }

    /* Calculate cov and free internal workspace */
    imsls_f_pooled_covariances(0, nvar, x, n_groups,
        IMSLS_IDO, 3,
        IMSLS_RETURN_USER, cov,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt,
        IMSLS_MEANS, &means, 0);

    imsls_f_write_matrix("Pooled Covariance Matrix", nvar, nvar, cov, 0);
    imsls_f_write_matrix("Means", n_groups, nvar, means, 0);

    free(means);
    free(x);
}
```

Output

```
            Pooled Covariance Matrix
            1           2           3           4
1      0.2650      0.0927      0.1675      0.0384
2      0.0927      0.1154      0.0552      0.0327
```

```
3      0.1675      0.0552      0.1852      0.0427
4      0.0384      0.0327      0.0427      0.0419


                     Means
             1           2           3           4
1        5.006       3.428       1.462       0.246
2        5.936       2.770       4.260       1.326
3        6.588       2.974       5.552       2.026
```

Warning Errors

`IMSLS_OBSERVATION_IGNORED`	In call #, row # of the matrix "x" has group number = #. The group number must be between 1 and #, the number of groups. This observation will be ignored.

Fatal Errors

`IMSLS_BAD_IDO_4`	"ido" = #. Initial allocations must be performed by making a call to `pooled_covariances` with "ido" = 1.
`IMSLS_BAD_IDO_5`	"ido" = #. A new analysis may not begin until the previous analysis is terminated by a call to `imsls_f_pooled_covariances` with "ido" equal to 3.

robust_covariances

Computes a robust estimate of a covariance matrix and mean vector.

Synopsis

#include `<imsls.h>`

float `*imsls_f_robust_covariances` (*int* `n_rows`, *int* `n_variables`,
float `*x`, *int* `n_groups`, ..., 0)

The type *double* function is `imsls_d_robust_covariances`.

Required Argument

int `n_rows` (Input)
Number of rows observations) in the input matrix `x`.

int `n_variables` (Input)
Number of variables to be used in computing the covariance matrix.

float `*x` (Input)
A `n_rows` by `n_variables` + 1 matrix containing the data. The first

n_variables columns correspond to the variables, and the last column (column n_variables) must contain the group numbers.

int n_groups (Input)
Number of groups in the data.

Return Value

Matrix of size n_variables by n_variables containing the matrix of covariances.

Synopsis with Optional Arguments

#include <imsls.h>

```
float *imsls_f_robust_covariances (int n_rows, int n_variables,
        float x[], int n_groups,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_X_INDICES, int igrp, int ind[], int ifrq, int iwt,
        IMSLS_INITIAL_EST_MEAN,
        IMSLS_INITIAL_EST_MEDIAN
        IMSLS_INITIAL_EST_INPUT, float input_means[],
                float input_cov[],
        IMSLS_ESTIMATION_METHOD, int method,
        IMSLS_PERCENTAGE, float percentage,
        IMSLS_MAX_ITERATIONS, int maxit,
        IMSLS_TOLERANCE, float tolerance,
        IMSLS_MINIMAX_WEIGHTS, float *a, float *b, float *c,
        IMSLS_GROUP_COUNTS, int **gcounts,
        IMSLS_GROUP_COUNTS_USER, int gcounts[],
        IMSLS_SUM_WEIGHTS, float **sum_weights,
        IMSLS_SUM_WEIGHTS_USER, float sum_weights[],
        IMSLS_MEANS, float **means,
        IMSLS_MEANS_USER, float means[],
        IMSLS_U, float **u,
        IMSLS_U_USER, float u[],
        IMSLS_BETA, float *beta,
        IMSLS_N_ROWS_MISSING, int *nrmiss,
        IMSLS_RETURN_USER, float c[],
        0)
```

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Row/Column dimension of x.
Default: x_col_dim = n_variables + 1

IMSLS_X_INDICES, *int* igrp, *int* ind[], *int* ifrq, *int* iwt (Input)
Each of the four arguments contains indices indicating column numbers

of x in which particular types of data are stored. Columns are numbered 0 … x_col_dim – 1.

Parameter igrp contains the index for the column of x in which the group numbers are stored.

Parameter ind contains the indices of the variables to be used in the analysis.

Parameters ifrq and iwt contain the column numbers of x in which the frequencies and weights, respectively, are stored. Set ifrq = –1 if there will be no column for frequencies. Set iwt = –1 if there will be no column for weights. Weights are rounded to the nearest integer. Negative weights are not allowed.

Defaults: igrp = n_variables,
ind [ ] = 0, 1, …, n_variables – 1, ifrq = –1, and iwt = –1

IMSLS_INITIAL_EST_MEAN, *or*
IMSLS_INITIAL_EST_MEDIAN, *or*
IMSLS_INITIAL_EST_INPUT, *float* *input_mean, *float* *input_cov (Input)

If IMSLS_INITIAL_EST_MEAN is specified, initial estimates are obtained as the usual estimate of a mean vector and of a covariance matrix.

If IMSLS_INITIAL_EST_MEDIAN is specified, initial estimates are based upon the median and interquartile range are used.

If IMSLS_INITIAL_EST_INPUT is specified, the initial estimates are specified in arrays input_mean and input_cov. Argument input_mean is an array of size n_groups by n_variables, and input_cov is an array of size n_variables by n_variables.

Default: IMSLS_INITIAL_EST_MEAN

IMSLS_ESTIMATION_METHOD, *int* method (Input)

Option parameter giving the algorithm to be used in computing the estimates.

method	**Method Used**
0	Huber's conjugate-gradient algorithm is used.
1	Stahel's algorithm is used.

`IMSLS_PERCENTAGE`, *float* `percentage` (Input)
: Percentage of gross errors expected in the data. Argument `percentage` must be in the range 0.0 to 100.0 and contains the percentage of outliers expected in the data. If the percentage of gross errors expected in the data is not known, a reasonable strategy is to choose a value of `percentage` that is such that larger values do not result in significant changes in the estimates.
Default: `percentage` = 5.0

`IMSLS_MAX_ITERATIONS`, *int* `maxit` (Input)
: Maximum number of iterations.
Default: `maxit` = 30

`IMSLS_TOLERANCE`, *float* `tolerance` (Input)
: Convergence criterion. When the maximum absolute change in a location or covariance estimate is less than `tolerance`, convergence is assumed.
Default: `tolerance` = 10^{-4}

`IMSLS_MINIMAX_WEIGHTS`, *float* `*a`, *float* `*b`, *float* `*c` (Output)
: Arguments `a`, `b`, and `c` contain the values for the parameters of the weighting function. See the "Description" section.

`IMSLS_GROUP_COUNTS`, *int* `**gcounts` (Output)
: Address of a pointer to an integer array of length `n_groups` containing the number of observations in each group.

`IMSLS_GROUP_COUNTS_USER`, *int* `gcounts[]` (Output)
: Storage for integer array `gcounts` is provided by the user. See `IMSLS_GROUP_COUNTS`.

`IMSLS_SUM_WEIGHTS`, *float* `**sum_weights` (Output)
: Address of a pointer to an array of length `n_groups` containing the sum of the weights times the frequencies in the groups.

`IMSLS_SUM_WEIGHTS_USER`, *float* `sum_weights[]`(Output)
: Storage for array `sum_weights` is provided by the user. See `IMSLS_SUM_WEIGHTS`.

`IMSLS_MEANS`, *float* `**means` (Output)
: Address of a pointer to an array of size `n_groups` by `n_variables`. The *i*-th row of `means` contains the group *i* variable means.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
: Storage for array means is provided by the user. See `IMSLS_MEANS`.

`IMSLS_U`, *float* `**u` (Output)
: Address of a pointer to an array of size `n_variables` by `n_variables` containing the lower matrix U, the lower triangular for the robust sample cross-products matrix. U is computed from the robust sample covariance matrix, S (See the "Description" section), as $S = U^T U$.

`IMSLS_U_USER`, *float* `u[]` (Output)

Storage for array `u` is provided by the user. See `IMSLS_U`.

`IMSLS_BETA`, *float* `*beta` (Output)

Argument `beta` contains the constant used to ensure that the estimated covariance matrix has unbiased expectation (for a given mean vector) for a multivariate normal density.

`IMSLS_N_ROWS_MISSING`, *int* `*nrmiss` (Output)

Number of rows of data encountered in calls to `robust_covariances` containing missing values (NaN) for any of the variables used.

`IMSLS_RETURN_USER`, *float* `c[]` (Output)

If specified, `c` returns the covariance matrix. Storage for array `c` is provided by the user.

Description

Function `imsls_f_robust_covariances` computes robust M-estimates of the mean and covariance matrix from a matrix of observations. A pooled estimate of the covariance matrix is computed when multiple groups are present in the input data. M-estimate weights are obtained using the "minimax" weights of Huber (1981, pp. 231-235), with `percentage` expected gross errors. Huber's (1981) weighting equations are given by:

$$u(r) = \begin{cases} \frac{a^2}{r^2} & r < a \\ 1 & a \le r \le b \\ \frac{b^2}{r^2} & r > b \end{cases}$$

$$w(r) = \min\left(1, \frac{c}{r}\right)$$

User specified observation weights and frequencies may be given for each row in `x`. Listwise deletion of missing values is assumed so that all observations used are "complete".

Let $f(x;\mu_i, \Sigma)$ denote the density of an observation p-vector `x` in population (group) i with mean vector μ_i, for $i = 1, \ldots, \tau$. Let the covariance matrix Σ be such that $\Sigma = R^T R$. If

$$y = R^{-T}(x - \mu_i)$$

then

$$g(y) = |\Sigma|^{1/2} f\left(R^T y + \mu_i; \mu_i, \Sigma\right)$$

It is assumed that $g(y)$ is a spherically symmetric density in p-dimensions.

In `imsls_f_robust_covariances`, Σ and μ_i are estimated as the solutions

$$\left(\hat{\Sigma}, \hat{\mu}_i\right)$$

of the estimation equations

$$\frac{1}{n}\sum_{j=1}^{n_i} f_{ig} w_{ij} w\left(r_{ij}\right) y_{ij} = 0$$

and

$$\frac{1}{n}\sum_{i=1}^{\tau}\sum_{j=1}^{n_i} f_{ij} w_{ij}\left[u\left(r_{ij}\right) y_{ij} y_{ij}^T - \beta I_p\right] = 0$$

where i indexes the τ groups, n_i, is the number of observations in group i, f_{ij} is the frequency for the j-th observation in group i, w_{ij} is the observation weight specified in column `iwt` of `x`, I_p is a $p \times p$ identity matrix,

$$r_{ij} = \sqrt{y_{ij}^T y_{ij}}$$

$w(r)$ and $u(r)$ are the weighting functions, and where β is a constant computed by the program to make the expected weighted Mahalanobis distance ($y^T y$) equal the expected Mahalanobis distance from a multivariate normal distribution (see Marazzi 1985). The constant β is described more fully below.

Function `imsls_f_robust_covariances` uses one of two algorithms for solving the estimation equations. The first algorithm is discussed in detail in Huber (1981) and is a variant of the conjugate gradient method. The second algorithm is due to Stahel (1981) and is discussed in detail by Marazzi (1985). In both algorithms, correction vectors T_{ki} for the group i means and correction matrix $W_k = I_p + U_k$ for the Cholesky factorization of Σ are found such that the updated mean vectors are given by

$$\hat{\mu}_{i,k+1} = \hat{\mu}_{i,k} + T_{ki}$$

and the updated matrix R is given as

$$\hat{R}_{k+1} = W_k \hat{R}_k$$

where k is the iteration number and

$$\hat{\Sigma}_k = R_k^T R_k$$

When all elements of U_k and T_{ki} are less than ε = `tolerance`, convergence is assumed.

Three methods for obtaining estimates are allowed. In the first method, the sample weighted estimate of Σ is computed. In the second method, estimates based upon the median and the interquartile range are used. Finally, in the last method, the user inputs initial estimates.

Function `imsls_f_robust_covariances` computes estimates based on the "minimax" weights discussed above. The constant β is chosen such that E

$(u(r)r_2) = \rho\beta$ where the expectation is with respect to a standard *p*-variate multivariate normal distribution. This yields estimates with the correct expectation for the multivariate normal distribution (for given mean vector). The expectation is computed via integration of estimated spline function. 200 knots are used on an equally apaced grid from 0.0 to the 99.999 percentile of

$$\chi_p^2$$

distribution. An error estimate is computed based upon 100 of these knots. If the estimated relative error is greater than 0.0001, a warning message is issued. If β is not computed accurately (i.e., if the warning message is issued), the computed esimates are still optimal, but the scale of the estimated covariance matrix may need to be multiplied by a constant in order for

$$\hat{\Sigma}$$

to have the correct multivariate normal covariance expectation.

Examples

Example 1

The following example computes a robust variance-covariance matrix. The last column of the data set is the group indicator.

```
#include <imsls.h>
#include <stdlib.h>
main()
{
    int nobs = 6;
    int nvar = 2;
    int n_groups = 2;
    float *cov;
    float x[18] = {
        2.2, 5.6, 1,
        3.4, 2.3, 1,
        1.2, 7.8, 1,
        3.2, 2.1, 2,
        4.1, 1.6, 2,
        3.7, 2.2, 2};

    cov = imsls_f_robust_covariances(nobs, nvar, x, n_groups, 0);

    imsls_f_write_matrix("Robust Covariance Matrix", nvar, nvar, cov,
        IMSLS_COL_NUMBER_ZERO,
        IMSLS_ROW_NUMBER_ZERO, 0);

    free(cov);
}
```

Output

```
Robust Covariance Matrix
           0           1
0      0.522      -1.160
```

```
1      -1.160       2.862
```

Example 2

The following example computes estimates of the pooled covariance matrix for the Fisher's iris data. For comparison, the estimates are first computed via function `imsls_f_pooled_covariances`. Function `imsls_f_robust_covariances` with `percentage` = 2.0 is then used to compute the robust estimates. As can be seen from the output, the resulting estimates are quite similar.

Next, three observations are made into outliers, and again, estimates are computed using functions `imsls_f_pooled_covariances` and `imsls_f_robust_covariances`. When outliers are present, the estimates of `imsls_f_pooled_covariances` are adversely affected, while the estimates produced by `imsls_f_robust_covariances` are close the estimates produced when no outliers are present.

```
include <imsls.h>
#include <stdlib.h>
main()
{
    int     nobs = 150;
    int     nvar = 4;
    int     n_groups = 3;
    float   percentage = 2.0;
    int     igrp = 0;
    int     ifrq = -1;
    int     iwt = -1;
    int     ind[4] = {1, 2, 3, 4};
    float   *x, cov[16], rbcov[16];

    x = imsls_f_data_sets(3, 0);

    imsls_f_pooled_covariances(nobs, nvar, x, n_groups,
        IMSLS_RETURN_USER, cov,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);

    imsls_f_write_matrix("Pooled Covariance with No Outliers", nvar, nvar,
                      cov,
        IMSLS_COL_NUMBER_ZERO,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_PRINT_UPPER, 0);

    imsls_f_robust_covariances(nobs, nvar, x, n_groups,
        IMSLS_RETURN_USER, rbcov,
        IMSLS_PERCENTAGE, percentage,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);

    imsls_f_write_matrix("Robust Covariance with No Outliers", nvar, nvar,
                      rbcov,
        IMSLS_COL_NUMBER_ZERO,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_PRINT_UPPER, 0);

    /* Add Outliers */
    x[1] = 100.0;
    x[19] = 100.0;
    x[497] = -100.0;

    imsls_f_pooled_covariances(nobs, nvar, x, n_groups,
        IMSLS_RETURN_USER, cov,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);

    imsls_f_write_matrix("Pooled Covariance with Outliers", nvar, nvar,
                         cov,
        IMSLS_COL_NUMBER_ZERO,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_PRINT_UPPER, 0);

    imsls_f_robust_covariances(nobs, nvar, x, n_groups,
        IMSLS_RETURN_USER, rbcov,
        IMSLS_PERCENTAGE, percentage,
        IMSLS_X_INDICES, igrp, ind, ifrq, iwt, 0);

    imsls_f_write_matrix("Robust Covariance with Outliers", nvar, nvar,
                         rbcov,
        IMSLS_COL_NUMBER_ZERO,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_PRINT_UPPER, 0);


    free(x);
}
```

Output

```
       Pooled Covariance with No Outliers
            0          1          2          3
0      0.2650     0.0927     0.1675     0.0384
1                 0.1154     0.0552     0.0327
2                            0.1852     0.0427
3                                       0.0419

       Robust Covariance with No Outliers
            0          1          2          3
0      0.2474     0.0872     0.1535     0.0360
1                 0.1073     0.0538     0.0322
2                            0.1705     0.0412
3                                       0.0401

         Pooled Covariance with Outliers
            0          1          2          3
0       60.43       0.30       0.13      -1.56
1                  70.53       0.17      -0.17
2                              0.19       0.07
3                                        66.38

         Robust Covariance with Outliers
            0          1          2          3
0      0.2555     0.0876     0.1553     0.0359
1                 0.1127     0.0545     0.0322
2                            0.1723     0.0412
```

```
3                                       0.0424
```

Warning Errors

`IMSLS_NO_CONVERGE_MAX_ITER`	Failure to converge within “maxit” = # iterations for at least one of the “nroot” = # roots.

Fatal Errors

`IMSLS_BAD_GROUP_2`	The group number for observation # is equal to #. It must be greater than or equal to one and less than or equal to #, the number of groups.

Chapter 4: Analysis of Variance and Designed Experiments

Routines

Usage Notes

The functions in this chapter cover a wide variety of commonly used experimental designs. They can be categorized, not only based upon the underlying experimental design that generated the user's data, but also on whether they provide support for missing values, factorial treatment structure, blocking and replication of the entire experiment, or multiple locations.

Typically, responses are stored in the input vector *y*. For a few functions, such as `imsls_f_anova_oneway` (page 230)and `imsls_f_anova_factorial` (page 239), the full set of model subscripts is not needed to identify each response. They assume the usual pattern, which requires that the last model subscript change most rapidly, followed by the model subscript next in line, and so forth, with the first subscript changing at the slowest rate. This pattern is referred to as *lexicographical ordering*.

However, for most of the functions in this chapter, one or more arrays are used to describe the experimental conditions associated with each value in the response input vector *y*. The function `imsls_f_split_plot` (page 316), for example, requires three additional input arrays: `split`, `whole` and `rep`. They are used to identify the split-plot, whole-plot and replicate number associated with each value in *y*.

Many of the functions described in this chapter permit users to enter missing data values using NaN (Not a Number) as the missing value code. Use function `imsls_f_machine` (or function `imsls_d_machine` with the double-precision) to retrieve NaN. Any element of *y* that is missing must be set to `imsls_f_machine`(6) or `imsls_d_machine`(6) (for double precision). See `imsls_f_machine` in Chapter 14, "Utilities" for a description. Functions `imsls_f_anova_factorial` (page 239), `imsls_f_anova_nested` (page 247)and `imsls_f_anova_balanced` (page 256) require complete, balanced data, and do not accept missing values.

As a diagnostic tool for validating model assumptions, some functions in this chapter perform a test for lack of fit when replicates are available in each cell of the experimental design..

Completely Randomized Experiments

Completely randomized experiments are analyzed using some variation of the one-way analysis of variance (Anova). A completely randomized design (CRD) is the simplest and most common example of a statistically designed experiment. Researchers using a CRD are interested in comparing the average effect of two or more treatments. In agriculture, treatments might be different plant varieties or fertilizers. In industry, treatments might be different product designs, different manufacturing plants, different methods for delivering the product, etc. In business, different business processes, such as different shipping methods or alternate approaches to a product repair process, might be considered treatments. Regardless of the area, the one thing they have in common is that random errors in the observations cause variations in differences between treatment observations, making it difficult to confirm the effectiveness of one treatment to another.

If observations on these treatments are completely independent then the design is referred to as a completely randomized design or CRD. The IMSL C Numerical Library has two routines for analysis of data from CRD: `imsls_f_anova_oneway` (page 230) and `imsls_f_crd_factorial` (page 267).

Both functions allow users to specify observations with missing values, have unequal group sizes, and output treatment means and standard deviations. The primary difference between the functions is that:

1. `imsls_f_anova_oneway` (page 230) conducts multiple comparisons of treatment functions; whereas `imsls_f_crd_factorial` (page 267) requires users to make a call to `imsls_f_multiple_comparisons` (page 385) to compare treatment means.
2. `imsls_f_crd_factorial` (page 267) can analyze treatments with a factorial treatment structure; whereas `imsls_f_anova_oneway` (page 230) does not analyze factorial structures.
3. `imsls_f_crd_factorial` (page 267) can analyze data from CRD experiments that are replicated across several blocks or locations. This can happen when the same experiment is repeated at different times or different locations.

Factorial Experiments

In some cases, treatments are identified by a combination of experimental factors. For example, in an octane study comparing several different gasolines, each gasoline could be developed using a combination of two additives, denoted below in Table 1, as Additive A and Additive B:

Treatment	Additive A	Additive B
1	No	No
2	Yes	No
3	No	Yes
4	Yes	Yes

Table 1 - A 2x2 Factorial Experiment

This is referred to as a 2x2 or 2^2 factorial experiment. There are 4 treatments involved in this study. One contains no additives, i.e. Treatment 1. Treatment 2 and 3 contain only one of the additives and treatment 4 contains both. A one-way anova, such as found in `anova_oneway` can analyze these data as 4 different treatments. Three functions, `imsls_f_crd_factorial` (page 267), `imsls_f_rcbd_factorial` (page 279) and `imsls_f_anova_factorial` (page 239) will analyze these data exploiting the factorial treatment structure. These functions allow users to answer structural questions about the treatments such as:

1. Are the average effects of the additives statistically significant? This is referred to as the factor main effects.
2. Is there an interaction effect between the additives. That is, is the effectiveness of an additive independent of the other?

Both `imsls_f_crd_factorial` (page 267) and `imsls_f_rcbd_factorial` (page 279) support analysis of a factorial experiment with missing values and multiple locations. The function `imsls_f_anova_factorial` (page 239) does not support analysis of experiments with missing values or experiments replicated over multiple locations. The main difference, as the names imply, between `imsls_f_crd_factorial` and `imsls_f_rcbd_factorial` is that `imsls_f_crd_factorial` assumes that treatments were completely randomized to experimental units. The `imsls_f_rcbd_factorial` routine assumes that treatments are blocked.

Blocking

Blocking is an important technique for reducing the impact of experimental error on the ability of the researcher to evaluate treatment differences. Usually this experimental error is caused by differences in location (spatial differences), differences in time (temporal differences) or differences in experimental units. Researchers refer to these as blocking factors. They are identifiable causes known to cause variation in observations between experimental units.

There are several functions that specifically support blocking in an experiment: `imsls_f_rcbd_factorial` (page 279), `imsls_f_lattice` (page 297), and `imsls_f_latin_square` (page 288). The first two functions, `imsls_f_rcbd_factorial` and `imsls_f_lattice`, support blocking on one factor.

A requirement of RCBD experiments is that every block must contain observations on every treatment. However, when the number of treatments (t) is greater than the block size (b), it is impossible to have every block contain observations on every treatment.

In this case, when $t > b$, an incomplete block design must be used instead of a RCBD. Lattice designs are a type of incomplete block design in which the number of treatments is equal to the square of an integer such as $t = 9$, 16, 25, etc. Lattice designs were originally described by Yates (1936). The function `imsls_f_lattice` (page 297) supports analysis of data from lattice experiments.

Besides the requirement that $t = k^2$, another characteristic of lattice experiments is that blocks be grouped into replicates, where each replicate contains one observation for every treatment. This forces the number of blocks in each replicate to be equal to the number of observations per block. That is, the number of blocks per replicate and the number of observations per block are both equal to $k = \sqrt{t}$.

In addition, the number of replicate groups in Lattice experiments is always less than or equal to $k + 1$. If it is equal to $k + 1$ then the design is referred to as a Balanced Lattice. If it is less than $k + 1$ then the design is referred to as a Partially Balanced Lattice. Tables of these experiments and their analysis are tabulated in Cochran & Cox (1950).

Consider, for example, a 3x3 balanced-lattice, i.e., *k*=3 and *t*=9. Notice that the number of replicates is $r = k + 1 = 4$. And the number of blocks per replicate and block size are both $k = 3$. The total number of blocks is equal to $b = \texttt{n\_locations} \cdot r \cdot (k - 1) + 1$. For a balanced-lattice, $b = r \cdot k = (k + 1) \cdot k = (\sqrt{t} + 1) \cdot \sqrt{t} = 4 \cdot 3 = 12$.

Replicate I	Replicate II
Block 1 (T1, T2, T3)	Block 4 (T1, T4, T7)
Block 2 (T4, T5, T6)	Block 5 (T2, T5, T8)
Block 3 (T7, T8, T9)	Block 6 (T3, T6, T9)
Replicate III	**Replicate IV**
Block 7 (T1, T5, T9)	Block 10 (T1, T6, T8)
Block 8 (T2, T6, T7)	Block 11 (T2, T4, T9)
Block 9 (T3, T4, T8)	Block 12 (T3, T5, T7)

Table 2 - A 3x3 Balanced-Lattice for Nine Treatments in Four Replicates.

The Anova table for a balanced-lattice experiment, takes the form shared with other balanced incomplete block experiments. In these experiments, the error term is divided into two components: the Inter-Block Error and the Intra-Block Error. For single and multiple locations, the general format of the Anova tables for Lattice experiments is illustrated in Table 3 and Table 4.

Source	DF	Sum of Squares	Mean Squares
REPLICATES	$t - 1$	SSR	MSR
TREATMENTS(unadj)	$t - 1$	SST	MST
TREATMENTS(adj)	$t - 1$	SSTa	MSTa
BLOCKS(adj)	$r \cdot (k - 1)$	SSBa	MSBa
INTRA-BLOCK ERROR	$(k - 1)(r \cdot k - k - 1)$	SSE	MSE
TOTAL	$r \cdot t - 1$	SSTot	

Table 3 – The Anova Table for a Lattice Experiment at One Location

	DF	Sum of Squares	Mean Squares
LOCATIONS	$p-1$	SSL	MSL
REPLICATES WITHIN LOCATIONS	$p\left(r-1\right)$	SSR	MSR
TREATMENTS(unadj)	$t-1$	SST	MST
TREATMENTS(adj)	$t-1$	SSTa	MSTa
BLOCKS(adj)	$p \cdot r(k-1)$	SSB	MSB
INTRA-BLOCK ERROR	$p(k-1)\left(r \cdot k-k-1\right)$	SSE	MSE
TOTAL	$p \cdot r \cdot t-1$	SSTot	

Table 4 – The Anova Table for a Lattice Experiment at Multiple Locations

Latin Square designs are very popular in cases where:

1. two blocking factors are involved
2. the two blocking factors do not interact with treatments, and
3. the number of blocks for each factor is equal to the number of treatments.

Consider an octane study involving 4 test vehicles tested in 4 bays with 4 test gasolines. This is a natural arrangement for a Latin square experiment. In this case there are 4 treatments, and two blocking factors, test vehicle and bay, each with 4 levels. The Latin Square for this example would look like the following arrangement.

		Test Vehicle			
		1	2	3	4
Test Bay	1	A	C	B	D
	2	D	B	A	C
	3	C	A	D	B
	4	B	D	C	A

Table 5. A Latin Square Design for *t*=4 Treatments

As illustrated above in Table 5, the letters A-D are used to denote the four test gasolines, or treatments. The assignment of each treatment to a particular test vehicle and test bay is described in Table 5. Gasoline A, for example, is tested in the following four vehicle/bay combinations: (1/1), (2/3), (3/2), and (4/4).

Notice that each treatment appears exactly once in every row and column. This balance, together with the assumed absence of interactions between treatments and the two blocking factors is characteristic of a Latin Square.

The corresponding Anova table for these data contains information on the blocking factors as well as treatment differences. Notice that the F-test for one of the two blocking factors, test vehicle, is statistically significant (p = 0.048); whereas the other, test bay, is not statistically significant (p=0.321).

Some researchers might use this as a basis to remove test bay as a blocking factor. In that case, the design can then be analyzed as a RCBD experiment since every treatment is repeated once and only once in every block, i.e., test vehicle.

Source	Degrees of Freedom	Sum of Squares	Mean Squares	F-Test	p-Value
Test Vehicle	3	1.5825	0.5275	4.83	0.048
Test Bay	3	0.0472	0.157	1.44	0.321
Gasoline	3	4.247	1.416	12.97	0.005
Error	6	0.655	0.109		
Total	15	6.9575			

Table 6 - Latin Square Anova Table for Octane Experiment

Multiple Locations

It is common for a researcher to repeat an experiment and then conduct an analysis of the data. In agricultural experiments, for example, it is common to repeat an experiment at several different farms. In other cases, a researcher may want to repeat an experiment at a specified frequency, such as week, month or year. If these repeated experiments are independent of one another then we can treat them as multiple locations.

Several of the functions in this chapter allow for multiple locations: `imsls_f_crd_factorial` (page 267), `imsls_f_rcbd_factorial` (page 279), `imsls_f_lattice` (page 297), `imsls_f_latin_square` (page 288), `imsls_f_split_plot` (page 316), `imsls_f_split_split_plot` (page 329), `imsls_f_strip_plot` (page 345), `strip_split_plot` (page 355). All of these functions allow for analysis of experiments replicated at multiple locations. By default they all treat locations as a random factor. Function `imsls_f_split_plot` also allows users to declare locations as a fixed effect.

Split-Plot Designs – Nesting and Restricted Randomization

Originally, split-plot designs were developed for testing agricultural treatments, such as varieties of wheat, different fertilizers or different insecticides. In these original experiments, growing areas were divided into plots. The major treatment factor, such as wheat variety, was randomly assigned to these plots. However, in addition to testing wheat varieties, they wanted to test another treatment factor such as fertilizer. This could have been done using a CRD or RCBD design. If a CRD design was used then treatment combinations would need to be randomly assigned to plots, such as shown below in Table 7.

CRD			
W3F2	W1F3	W4F1	W2F1
W2F3	W1F1	W1F3	W1F2
W2F2	W3F1	W2F1	W4F2
W3F2	W1F1	W2F3	W1F2
W4F1	W3F2	W3F2	W4F3
W4F3	W3F1	W2F2	W4F2

Table 7 – Completely Randomized Experiments –Both Factors Randomized

In the CRD illustration above, any plot could have any combination of wheat variety (W1, W2, W3 or W4) and fertilizer (F1, F2 or F3). There is no restriction on randomization in a CRD. Any of the $t = 4 \times 3 = 12$ treatments can appear in any of the 24 plots.

If a RCBD were used, all t=12 treatment combinations would need to be arranged in blocks similar to what is described in Table 8, which places one restriction on randomization.

RCBD				
BLOCK 1	W3F3	W1F3	W4F1	W4F3
	W2F3	W1F1	W3F2	W1F2
	W2F2	W3F1	W2F1	W4F2
BLOCK 2	W3F2	W1F1	W2F3	W1F2
	W4F1	W1F3	W3F2	W4F3
	W2F1	W3F1	W2F2	W4F2

Table 8 – Randomized Complete Block Experiments –
Both Factors Randomized Within a Block

The RCBD arrangement is basically a replicated CRD design with a randomization restriction that treatments are divided into two groups of replicates which are assigned to a block of land. Randomization of treatments only occurs within each block.

At first glance, a split-plot experiment could be mistaken for a RCBD experiment since it is also blocked. The split-plot arrangement with only one replicate for this experiment is illustrated below in Table 9. Notice that it appears as if levels of the fertilizer factor (F1, F2, and F3) are nested within wheat variety (W1, W2, W3 and W4), however that is not the case. Varieties were actually randomly assigned to one of four rows in the field. After randomizing wheat varieties, fertilizer was randomized within wheat variety.

Split-Plot Design				
Block 1	**W2**	W2F1	W2F3	W2F2
	W1	W1F3	W1F1	W1F2
	W4	W4F1	W4F3	W4F2
	W3	W3F2	W3F1	W3F3
Block 2	**W3**	W3F2	W3F1	W3F3
	W1	W1F3	W1F1	W1F2
	W4	W4F1	W4F3	W4F2
	W2	W2F1	W2F3	W2F2

Table 9 – A Split-Plot Experiment for Wheat (W) and Fertilizer (F)

The essential distinction between split-plot experiments and completely randomized or randomized complete block experiments is the presence of a second factor that is blocked, or nested, within each level of the first factor. This second factor is referred to as the split-plot factor, and the first is referred to as the whole-plot factor.

Both factors are randomized, but with a restriction on randomization of the second factor, the split-plot factor. Whole plots (wheat variety) are randomly assigned, without restriction to plots, or rows in this example. However, the randomization of split-plots (fertilizer) is restricted. It is restricted to random assignment within whole-plots.

Strip-Plot Designs

Strip-plot experiments look similar to split-plot experiments. In fact they are easily confused, resulting in incorrect statistical analyses. The essential distinction between strip-plot and split-plot experiments is the application of the second factor. In a split-plot experiment, levels of the second factor are nested within the whole-plot factor (see Table 11). In strip-plot experiments, the whole-plot factor is completely crossed with the second factor (see Table 10).

This occurs, for example, when an agricultural field is used as a block and the levels of the whole-plot factor are applied in vertical strips across the entire field. Levels of the second factor are assigned to horizontal strips across the same block.

		Whole-Plot Factor			
		A2	A1	A4	A3
Strip Plot	B3	A2B3	A1B3	A4B3	A3B3
	B1	A2B1	A1B1	A4B1	A3B1
	B2	A2B2	A1B2	A4B2	A3B2

Table 10 – Strip-Plot Experiments – Strip-Plots Completely Crossed

Whole Plot Factor			
A2	A1	A4	A3
A2B1	A1B3	A4B1	A3B3
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

Table 11 – Split-Plot Experiments – Split-Plots Nested within Strip-Plots

As described in the previous section, in a split-plot experiment the second experimental factor, referred to as the split-plot factor, is nested within the first factor, referred to as the whole-plot factor.

Consider, for example, the semiconductor experiment described in Figure 1, “Split-Plot Randomization” below. The wafers from each plater, the whole-plot factor, are divided into equal size groups and then randomly assigned to an etcher, the split-plot factor. Wafers from different platers are etched separately from those that went through another plating machine. Randomization occurred within each level of the whole-plot factor, i.e., plater.

Graphically, as shown below, this arrangement appears similar to a tree or hierarchical structure.

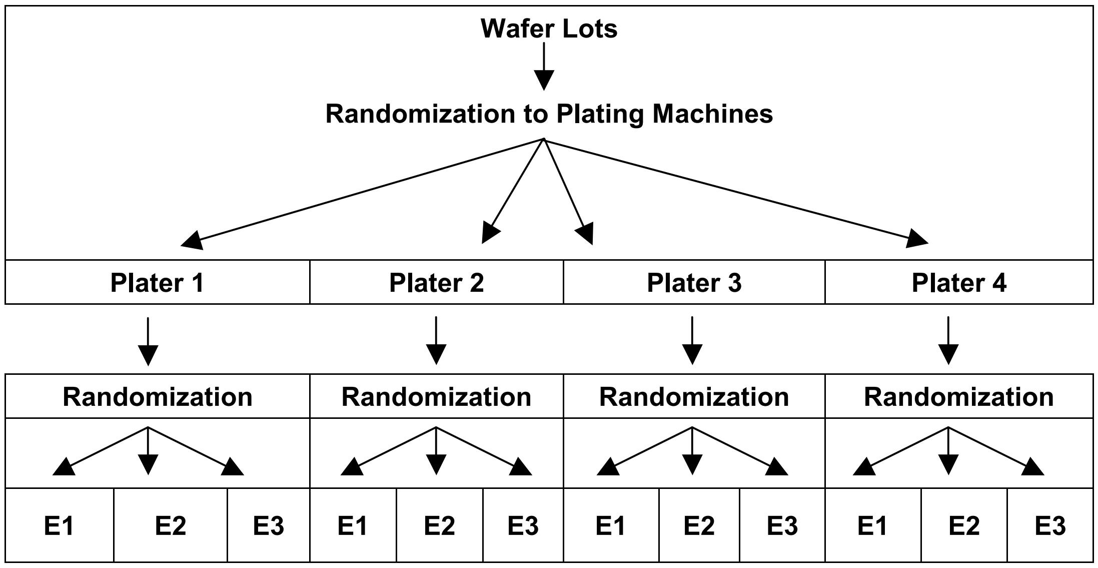

Figure 1 - Split-Plot Randomization

Notice that although there are only 3 etchers, 12 different runs are made using these etchers. The wafers randomly assigned to the first plater and first etcher are processed separately from the wafers assigned to other plating machines.

In a strip-plot experiment, the second randomization of the wafers to etchers occurs differently, see Figure 2, "Strip-Plot Semiconductor Experiment." Instead of randomizing the wafers from each plater to the three etchers and then running them separately from the wafers from another plater, the wafers from each plater are divided into three groups and then each randomly assigned to one of the three etchers. However, the wafers from all four plating machines assigned to the same etcher are run together.

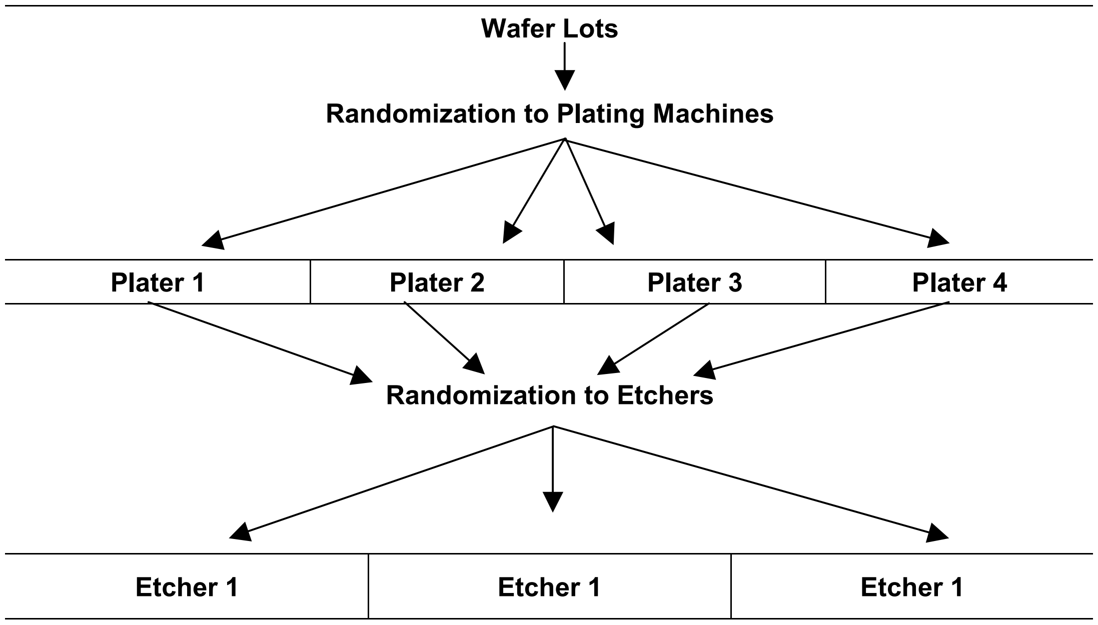

Figure 2 - Strip-Plot Semiconductor Experiment

Strip-plot experiments can be analyzed using `imsls_f_strip_plot` (page 345). Function `imsls_f_strip_plot` returns a strip-plot Anova table with the following general structure:

Source	DF	SS	MS	F-Test	p-Value
Blocks	1	0.0005	0.0005	0.955	0.431
Whole-Plots: Plating Machines	2	0.0139	0.0070	64.39	0.015
Whole-Plot Error	2	0.0002	0.0001	0.194	0.838
Strip-Plots: Etchers	1	0.0033	0.0033	100.0	0.060
Strip-Plot Error	1	<0.0001	<0.0001	0.060	0.830
Whole-Plot x Strip-Plot	2	0.0033	0.0017	2.970	0.251
Whole-Plot x Strip-Plot Error	2	0.0011	0.0006		
Total	11	0.0225			

Table 12 - Strip-Plot Anova Table for Semiconductor Experiment

Split-Split Plot and Strip-Split Plot Experiments

There are hundreds of other designs used in research and industry. The designs mentioned above are some of the most common. Other frequently used designs include variations of the split and strip-plot designs:

- Split-Split-Plot Experiments, and
- Strip-Split Plot Experiments.

The essential distinction between split-plot and split-split-plot experiments is the presence of a third factor that is blocked, or nested, within each level of the whole-plot and split-plot factors. This third factor is referred to as the sub-plot, factor. A split-plot experiment, see Table 12, has only two factors, denoted by A and B. The second factor is nested within the first factor. Randomization of the second factor, the split-plot factor, occurs within each level of the first factor.

Whole Plot Factor			
A2	**A1**	**A4**	**A3**
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B3

Table 13 - Split-Plot Experiment – Split-Plot B Nested within Whole-Plot A

On the other hand, a split-split plot experiment has three factors, illustrated in Table 14 by A, B and C. The second factor is nested within the first factor, and the third factor is nested within the second.

Whole Plot Factor A			
A2	**A1**	**A4**	**A3**
A2B3C2 A2B3C1	A1B2C1 A1B2C2	A4B1C2 A4B1C1	A3B3C2 A3B3C1
A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B3C2 A4B3C1	A3B2C2 A3B2C1
A2B2C2 A2B2C1	A1B3C1 A1B3C2	A4B2C1 A4B2C2	A3B1C2 A3B1C1

Table 14 – Split-Split Plot Experiment – Sub-Plot Factor C Nested Within Split-Plot Factor B, Nested Within Whole-Plot Factor A

Contrast the split-split plot experiment to the same experiment run using a strip-split plot design (see Table 15). In a strip-split plot experiment factor B is applied in strip across factor A; whereas, in a split-split plot experiment, factor B is randomly assigned to each level of factor A. In a strip-split plot experiment, the level of factor B is constant across a row; whereas in a split-split plot experiment, the levels of factor B change as you go across a row, reflecting the fact that for split-plot experiments, factor B is randomized within each level of factor A.

		Factor A Strip Plots			
		A2	**A1**	**A4**	**A3**
Factor B Strip Plots	**B3**	A2B3C2 A2B3C1	A1B3C1 A1B3C2	A4B3C2 A4B3C1	A3B3C2 A3B3C1
	B1	A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B1C2 A4B1C1	A3B1C2 A3B1C1
	B2	A2B2C2 A2B2C1	A1B2C1 A1B2C2	A4B2C1 A4B2C2	A3B2C2 A3B2C1

Table 15 – Strip-Split Plot Experiment, Split-Plots Nested Within Strip-Plot Factors A and B

In some studies, split-split-plot or strip-split-plot experiments are replicated at several locations. Functions `imsls_f_split_split_plot` (page 329) and `imsls_f_strip_split_plot` (page 355) can analyze these, even when the number of blocks or replicates at each location is different.

Validating Key Assumptions in Anova

The key output in the analysis of designed experiments is the F-tests in the Anova table for that experiment. The validity of these tests relies upon several key assumptions:

1. observational errors are independent of one another,
2. observational errors are Normally distributed, and
3. the variance of observational errors is homogeneous across treatments.

These are referred to as the independence, Normality and homogeneity of variance assumptions. All of these assumptions are evaluated by examining the properties of the residuals, which are estimates of the observational error for each observation. Residuals are calculated by taking the difference between each observed value in the series and its corresponding estimate. In most cases, the residual is the difference between the observed value and the mean for that treatment.

The independence assumption can be examined by evaluating the magnitude of the correlations among the residuals sorted in the order they were collected. The IMSL function `imsls_f_autocorrelation` (see Chapter 8, “Times Series and Forecasting”). can be used to obtain these correlations. The autocorrelations, to a maximum lag of about 20, can be examined to identify any that are statistically significant.

Residuals should be independent of one another, which implies that all autocorrelations with a lag of 1 or higher should be statistically equivalent to zero. If a statistically significant autocorrelation is found, leading a researcher to conclude that an autocorrelation is not equal to zero, then this would provide sufficient evidence to conclude that the observational errors are not independent of one another.

The second major assumption for analysis of variance is the Normality assumption. In the IMSL C Numerical Library, the function `imsls_f_normality_test` (see Chapter 7, "Tests of Goodness of Fit")can be used to determine whether the residuals are not Normally distributed. A small *p*-value from this test provides sufficient evidence to conclude that the observational errors are not Normally distributed.

The last assumption, *homogeneity of variance*, is evaluated by comparing treatment standard errors. This is equivalent to testing whether $\sigma_1 = \sigma_2 = \cdots = \sigma_t$, where σ_i is the standard deviation of the observational error for the ith treatment. This test can be conducted using `imsls_f_homogeneity` (page 378). To conduct this test, the residuals, and their corresponding treatment identifiers are passed into `imsls_f_homogeneity`. It calculates the *p*-values for both Bartlett's and Levene's tests for equal variance. If a *p*-value is below the stated significance level, a researcher would conclude that the within treatment variances are not homogeneous.

Missing Observations

Missing observations create problems with the interpretation and calculation of F-tests for designed experiments. The approach taken in the functions described in this chapter is to estimate missing values using the Yates method and then to compute the Anova table using these estimates.

Essentially the Yates method, implemented in `imsls_f_yates` (page 390), replaces missing observations with the values that minimize the error sum of squares in the Anova table. The Anova table is calculated using these estimates, with one modification. The total degrees of freedom and the error degrees of freedom are both reduced by the number of missing observations.

For simple cases, in which only one observation is missing, formulas have been developed for most designs. See Steel and Torrie (1960) and Cochran and Cox (1957) for a description of these formulas. However for more than one missing observation, a multivariate optimization is conducted to simultaneously estimate the missing values. For the simple case with only one missing value, this approach produces estimates identical to the published formulas for a single missing value.

A potential issue arises when the Anova table contains more than one form of error, such as split-plot and strip-plot designs. In every case, missing values are estimated by minimizing the last error term in the table.

anova_oneway

Analyzes a one-way classification model.

Synopsis

#include `<imsls.h>`

float `imsls_f_anova_oneway` (*int* `n_groups`, *int* `n[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_anova_oneway`

Required Arguments

int `n_groups` (Input)
Number of groups.

int `n[]` (Input)
Array of length `n_groups` containing the number of responses for each group.

float `y[]` (Input)
Array of length `n`[`0`] + `n`[`1`] + … + `n`[`n_group` – 1] containing the responses for each group.

Return Value

The *p*-value for the *F*-statistic.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_anova_oneway` (*int* `n_groups`, *int* `n[]`, *float* `y[]`,
`IMSLS_ANOVA_TABLE`, *float* `**anova_table`,
`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]`,
`IMSLS_GROUP_MEANS`, *float* `**means`,
`IMSLS_GROUP_MEANS_USER`, *float* `means[]`,
`IMSLS_GROUP_STD_DEVS`, *float* `**std_devs`,
`IMSLS_GROUP_STD_DEVS_USER`, *float* `std_devs[]`,
`IMSLS_GROUP_COUNTS`, *int* `**counts`,
`IMSLS_GROUP_COUNTS_USER`, *int* `counts[]`,
`IMSLS_CONFIDENCE`, *float* `confidence`,
`IMSLS_TUKEY`, *float* `**ci_diff_means`, *or*
`IMSLS_DUNN_SIDAK`, *float* `**ci_diff_means`, *or*
`IMSLS_BONFERRONI`, *float* `**ci_diff_means`, *or*
`IMSLS_SCHEFFE`, *float* `**ci_diff_means`, *or*
`IMSLS_ONE_AT_A_TIME`, *float* `**ci_diff_means`,
`IMSLS_TUKEY_USER`, *float* `ci_diff_means[]`, *or*
`IMSLS_DUNN_SIDAK_USER`, *float* `ci_diff_means[]`, *or*
`IMSLS_BONFERRONI_USER`, *float* `ci_diff_means[]`, *or*
`IMSLS_SCHEFFE_USER`, *float* `ci_diff_means[]`, *or*

`IMSLS_ONE_AT_A_TIME_USER`, *float* `ci_diff_means[]`,
0)

Optional Arguments

`IMSLS_ANOVA_TABLE`, *float* `**anova_table` (Output)
Address of a pointer to an internally allocated array of size 15 containing the analysis of variance table. The analysis of variance statistics are 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)

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for array `anova_table` is provided by the user. See `IMSLS_ANOVA_TABLE`.

`IMSLS_GROUP_MEANS`, *float* `**means` (Output)
Address of a pointer to an internally allocated array of length `n_groups` containing the group means.

`IMSLS_GROUP_MEANS_USER`, *float* `means[]` (Output)
Storage for array `means` is provided by the user. See `IMSLS_GROUP_MEANS`.

`IMSLS_GROUP_STD_DEVS`, *float* `**std_devs` (Output)
Address of a pointer to an internally allocated array of length `n_groups` containing the group standard deviations.

`IMSLS_GROUP_STD_DEVS_USER`, *float* `std_devs[]` (Output)
Storage for array `std_devs` is provided by the user. See `IMSLS_STD_DEVS`.

`IMSLS_GROUP_COUNTS`, *int* `**counts` (Output)
Address of a pointer to an internally allocated array of length `n_groups` containing the number of nonmissing observations for the groups.

`IMSLS_GROUP_COUNTS_USER`, *int* `counts[]` (Output)
Storage for array `counts` is provided by the user. See `IMSLS_COUNTS`.

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Confidence level for the simultaneous interval estimation.
If `IMSLS_TUKEY` is specified, `confidence` must be in the range [90.0, 99.0). Otherwise, confidence is in the range [0.0, 100.0).
Default: `confidence` = 95.0

`IMSLS_TUKEY`, *float* `**ci_diff_means` (Output), *or*
`IMSLS_DUNN_SIDAK`, *float* `**ci_diff_means` (Output), *or*
`IMSLS_BONFERRONI`, *float* `**ci_diff_means` (Output), *or*
`IMSLS_SCHEFFE`, *float* `**ci_diff_means` (Output), *or*
`IMSLS_ONE_AT_A_TIME`, *float* `**ci_diff_means` (Output)
Function `imsls_f_anova_oneway` computes the confidence intervals on all pairwise differences of means using any one of six methods: Tukey, Tukey-Kramer, Dunn-Šidák, Bonferroni, Scheffé, or Fisher's LSD (One-at-a-Time). If `IMSLS_TUKEY` is specified, the Tukey confidence intervals are calculated if the group sizes are equal; otherwise, the Tukey-Kramer confidence intervals are calculated.

On return, `ci_diff_means` contains the address of a pointer to a

$$\binom{\text{ngroups}}{2} \times 5$$

internally allocated array containing the statistics relating to the difference of means.

Column	**Description**
0	group number for the *i*-th mean
1	group number for the *j*-th mean
2	difference of means (*i*-th mean) − (*j*-th mean)
3	lower confidence limit for the difference
4	upper confidence limit for the difference

`IMSLS_TUKEY_USER`, *float* `ci_diff_means[]` (Output), *or*
`IMSLS_DUNN_SIDAK_USER`, *float* `ci_diff_means[]` (Output), *or*
`IMSLS_BONFERRONI_USER`, *float* `ci_diff_means[]` (Output), *or*
`IMSLS_SCHEFFE_USER`, *float* `ci_diff_means[]` (Output), *or*
`IMSLS_ONE_AT_A_TIME_USER`, *float* `ci_diff_means[]` (Output)
Storage for array `ci_diff_means` is provided by the user.

Description

Function `imsls_f_anova_oneway` performs an analysis of variance of responses from a oneway classification design. The model is

$$y_{ij} = \mu_i + \varepsilon_{ij} \qquad i = 1, 2, \ldots, k; j = 1, 2, \ldots, n_i$$

where the observed value y_{ij} constitutes the j-th response in the i-th group, μ_i denotes the population mean for the i-th group, and the ε_{ij} arguments are errors that are identically and independently distributed normal with mean 0 and variance σ^2. Function `imsls_f_anova_oneway` requires the y_{ij} observed responses as input into a single vector y with responses in each group occupying contiguous locations. The analysis of variance table is computed along with the group sample means and standard deviations. A discussion of formulas and interpretations for the one-way analysis of variance problem appears in most elementary statistics texts, e.g., Snedecor and Cochran (1967, Chapter 10).

Function `imsls_f_anova_oneway` computes simultaneous confidence intervals on all

$$k^* = \frac{k(k-1)}{2}$$

pairwise comparisons of k means μ_1 $\mu_2, \ldots, \mu_k$ in the one-way analysis of variance model. Any of several methods can be chosen. A good review of these methods is given by Stoline (1981). The methods are also discussed in many elementary statistics texts, e.g., Kirk (1982, pp. 114–127).

Let s^2 be the estimated variance of a single observation. Let v be the degrees of freedom associated with s^2. Let

$$\alpha = 1 - \frac{\texttt{confidence}}{100.0}$$

The methods are summarized as follows:

Tukey method: The Tukey method gives the narrowest simultaneous confidence intervals for all pairwise differences of means $\mu_i - \mu_j$ in balanced ($n_1 = n_2 = \ldots = n_k = n$) one-way designs. The method is exact and uses the Studentized range distribution. The formula for the difference $\mu_i - \mu_j$ is given by

$$\overline{y}_i - \overline{y}_j \pm q_{1-\alpha;k,v}\sqrt{\frac{s^2}{n}}$$

where $q_{1-\alpha;k,v}$ is the $(1 - \alpha)$ 100 percentage point of the Studentized range distribution with parameters k and v.

Tukey-Kramer method: The Tukey-Kramer method is an approximate extension of the Tukey method for the unbalanced case. (The method simplifies to the Tukey method for the balanced case.) The method always produces confidence intervals narrower than the Dunn-Šidák and Bonferroni methods. Hayter (1984) proved that the method is conservative, i.e., the method guarantees a confidence coverage of at least $(1 - \alpha)$ 100. Hayter's proof gave further support to earlier recommendations for its use (Stoline 1981). (Methods that are currently better are restricted to special cases and only offer improvement in severely unbalanced cases; see, for example, Spurrier and Isham 1985.) The formula for the difference $\mu_i - \mu_j$ is given by the following:

$$\bar{y}_i - \bar{y}_j \pm q_{1-\alpha;v,k}\sqrt{\frac{s^2}{2n_i}+\frac{s^2}{2n_j}}$$

Dunn-Šidák method: The Dunn-Šidák method is a conservative method. The method gives wider intervals than the Tukey-Kramer method. (For large *v* and small α and *k*, the difference is only slight.) The method is slightly better than the Bonferroni method and is based on an improved Bonferroni (multiplicative) inequality (Miller 1980, pp. 101, 254–255). The method uses the *t* distribution (see function `imsls_f_t_inverse_cdf`, Chapter 11, “Probability Distribution Functions and Inverses. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm t_{\frac{1}{2}+\frac{1}{2}(1-\alpha)^{1/k^*};v}\sqrt{\frac{s^2}{n_i}+\frac{s^2}{n_j}}$$

where $t_{f;v}$ is the 100*f* percentage point of the *t* distribution with ν degrees of freedom.

Bonferroni method: The Bonferroni method is a conservative method based on the Bonferroni (additive) inequality (Miller, p. 8). The method uses the *t* distribution. The formula for the difference $\mu_i - \mu_j$ is given by the following:

$$\bar{y}_i - \bar{y}_j \pm t_{1-\frac{\alpha}{2k^*};v}\sqrt{\frac{s^2}{n_i}+\frac{s^2}{n_j}}$$

Scheffé method: The Scheffé method is an overly conservative method for simultaneous confidence intervals on pairwise difference of means. The method is applicable for simultaneous confidence intervals on all contrasts, i.e., all linear combinations

$$\sum_{i=1}^{k} c_i \mu_i$$

where the following is true:

$$\sum_{i=1}^{k} c_i = 0$$

This method can be recommended here only if a large number of confidence intervals on contrasts in addition to the pairwise differences of means are to be constructed. The method uses the *F* distribution (see function `imsls_f_F_inverse_cdf`, Chapter 11, “Probabilty and Distribution Functions and Inverses”). The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm \sqrt{(k-1)F_{1-\alpha;k-1,v}(\frac{s^2}{n_i}+\frac{s^2}{n_j})}$$

where $F_{1-\alpha;(k-1),\nu}$ is the $(1-\alpha)$ 100 percentage point of the F distribution with $k-1$ and ν degrees of freedom.

One-at-a-Time *t* method (Fisher's LSD): The One-at-a-Time *t* method is appropriate for constructing a single confidence interval. The confidence percentage input is appropriate for one interval at a time. The method has been used widely in conjunction with the overall test of the null hypothesis $\mu_1 = \mu_2 = \ldots = \mu_k$ by the use of the F statistic. Fisher's LSD (least significant difference) test is a two-stage test that proceeds to make pairwise comparisons of means only if the overall F test is significant. Milliken and Johnson (1984, p. 31) recommend LSD comparisons after a significant F only if the number of comparisons is small and the comparisons were planned prior to the analysis. If many unplanned comparisons are made, they recommend Scheffé's method. If the F test is insignificant, a few planned comparisons for differences in means can still be performed by using either Tukey, Tukey-Kramer, Dunn-Šidák,or Bonferroni methods. Because the F test is insignificant, Scheffé's method does not yield any significant differences. The formula for the difference $\mu_i - \mu_j$ is given by the following:

$$\overline{y}_i - \overline{y}_j \pm t_{1-\frac{\alpha}{2};\nu}\sqrt{\frac{s^2}{n_i}+\frac{s^2}{n_j}}$$

Examples

Example 1

This example computes a one-way analysis of variance for data discussed by Searle (1971, Table 5.1, pp. 165–179). The responses are plant weights for six plants of three different types—three normal, two off-types, and one aberrant. The responses are given by type of plant in the following table:

Normal	Off-Type	Aberrant
101	84	32
105 94	88	

```
#include <imsls.h>
main()
{
    int     n_groups=3;
    int     n[] = {3, 2, 1};
    float   y[] = {101.0, 105.0, 94.0, 84.0, 88.0, 32.0};
    float   p_value;
    p_value = imsls_f_anova_oneway (n_groups, n, y, 0);
    printf ("p-value = %6.4f", p_value);
  }
```

Output

```
p-value = 0.002
```

Example 2

The data used in this example is the same as that used in the initial example. Here, the `anova_table` is printed.

```
#include <imsls.h>
main()
{
    int     n_groups=3;
    int     n[] = {3, 2, 1};
    float   y[] = {101.0, 105.0, 94.0, 84.0, 88.0, 32.0};
    float   p_value;
    float   *anova_table;
    char    *labels[] = {
                   "degrees of freedom for among groups",
                   "degrees of freedom for within groups",
                   "total (corrected) degrees of freedom",
                   "sum of squares for among groups",
                   "sum of squares for within groups",
                   "total (corrected) sum of squares",
                   "among mean square",
                   "within mean square", "F-statistic",
                   "p-value", "R-squared (in percent)",
                   "adjusted R-squared (in percent)",
                   "est. standard deviation of within error",
                   "overall mean of y",
                   "coefficient of variation (in percent)"};

                      /* Perform analysis */
    p_value = imsls_f_anova_oneway (n_groups, n, y,
        IMSLS_ANOVA_TABLE, &anova_table,
        0);
                      /* Print results */
    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
        anova_table,
        IMSLS_ROW_LABELS, labels,
        IMSLS_WRITE_FORMAT, "%9.2f",
        0);
}
```

Output

```
         * * * Analysis of Variance * * *
degrees of freedom for among groups          2.00
degrees of freedom for within groups         3.00
total (corrected) degrees of freedom         5.00
sum of squares for among groups           3480.00
sum of squares for within groups            70.00
total (corrected) sum of squares          3550.00
among mean square                         1740.00
within mean square                          23.33
F-statistic                                 74.57
p-value                                      0.00
R-squared (in percent)                      98.03
adjusted R-squared (in percent)             96.71
```

```
est. standard deviation of within error       4.83
overall mean of y                            84.00
coefficient of variation (in percent)         5.75
```

Example 3

Simultaneous confidence intervals are generated for the following measurements of cold-cranking power for five models of automobile batteries. Nelson (1989, pp. 232–241) provided the data and approach.

Model 1	Model 2	Model 3	Model 4	Model 5
41	42	27	48	28
43	43	26	45	32
42	46	28	51	37
46	38	27	46	25

The Tukey method is chosen for the analysis of pairwise comparisons, with a confidence level of 99 percent. The means and their confidence limits are output.

```
#include <imsls.h>

void main()
{

   int    n_groups = 5;
   int    n[] = {4, 4, 4, 4, 4};
   int    permute[] = {2, 3, 4, 0, 1};
   float  y[] = {41.0, 43.0, 42.0, 46.0, 42.0,
                 43.0, 46.0, 38.0, 27.0, 26.0,
                 28.0, 27.0, 48.0, 45.0, 51.0,
                 46.0, 28.0, 32.0, 37.0, 25.0};
   float  *anova_table, *ci_diff_means, tmp_diff_means[50];
   float  confidence = 99.0;
   char   *labels[] = {
                    "degrees of freedom for among groups",
                    "degrees of freedom for within groups",
                    "total (corrected) degrees of freedom",
                    "sum of squares for among groups",
                    "sum of squares for within groups",
                    "total (corrected) sum of squares",
                    "among mean square",
                    "within mean square", "F-statistic",
                    "p-value", "R-squared (in percent)",
                    "adjusted R-squared (in percent)",
                    "est. standard deviation of within error",
                    "overall mean of y",
                    "coefficient of variation (in percent)"};
   char   *mean_row_labels[] = {
                    "first and second",
                    "first and third",
                    "first and fourth",
                    "first and fifth",
                    "second and third",
                    "second and fourth",
                    "second and fifth",
                    "third and fourth",
                    "third and fifth",
                    "fourth and fifth"};
   char   *mean_col_labels[] = {
                    "Means",
                    "Difference of means",
                    "Lower limit",
                    "Upper limit"};
                        /* Perform analysis */

 imsls_f_anova_oneway(n_groups, n, y,
        IMSLS_ANOVA_TABLE, &anova_table,
        IMSLS_CONFIDENCE, confidence,
        IMSLS_TUKEY, &ci_diff_means,
        0);
                        /* Print anova_table */
   imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15,
        1, anova_table,
        IMSLS_ROW_LABELS, labels,
        IMSLS_WRITE_FORMAT, "%9.2f",
        0);
                      /* Permute ci_diff_means for printing */
   imsls_f_permute_matrix(10, 5, ci_diff_means, permute,
        IMSLS_PERMUTE_COLUMNS,
        IMSLS_RETURN_USER, tmp_diff_means,
        0);
                      /* Print ci_diff_means */
   imsls_f_write_matrix("* * * Differences in Means * * *\n", 10,
        3, tmp_diff_means,
        IMSLS_A_COL_DIM, 5,
        IMSLS_ROW_LABELS, mean_row_labels,
        IMSLS_COL_LABELS, mean_col_labels,
        IMSLS_WRITE_FORMAT, "%9.2f",
        0);
}
```

Output

```
         * * * Analysis of Variance * * *

degrees of freedom for among groups           4.00
degrees of freedom for within groups         15.00
total (corrected) degrees of freedom         19.00
sum of squares for among groups            1242.20
sum of squares for within groups            150.75
total (corrected) sum of squares           1392.95
among mean square                           310.55
within mean square                           10.05
F-statistic                                  30.90
p-value                                       0.00
R-squared (in percent)                       89.18
adjusted R-squared (in percent)              86.29
est. standard deviation of within error       3.17
overall mean of y                            38.05
coefficient of variation (in percent)         8.33

           * * * Differences in Means * * *

Means              Difference  Lower limit  Upper limit
```

```
                          of means
first and second              0.75          -8.05          9.55
first and third              16.00           7.20         24.80
first and fourth             -4.50         -13.30          4.30
first and fifth              12.50           3.70         21.30
second and third             15.25           6.45         24.05
second and fourth            -5.25         -14.05          3.55
second and fifth             11.75           2.95         20.55
third and fourth            -20.50         -29.30        -11.70
third and fifth              -3.50         -12.30          5.30
fourth and fifth             17.00           8.20         25.80
```

anova_factorial

Analyzes a balanced factorial design with fixed effects.

Synopsis

#include `<imsls.h>`

float `imsls_f_anova_factorial` (*int* `n_subscripts`, *int* `n_levels`,
    float `y[]`, ..., 0)

The type *double* function is `imsls_d_anova_factorial`

Required Arguments

int `n_subscripts` (Input)
    Number of subscripts. Number of factors in the model + 1 (for the error term).

int `n_levels` (Input)
    Array of length `n_subscripts` containing the number of levels for each of the factors for the first `n_subscripts` – 1 elements. `n_levels` [`n_subscripts` – 1] is the number of observations per cell.

float `y[]` (Input)
    Array of length `n_levels` [0]*`n_levels` [1]* … *`n_levels` [`n_subscripts` – 1] containing the responses. Argument *y* must not contain NaN for any of its elements, i.e., missing values are not allowed.

Return Value

The *p*-value for the overall *F* test.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_anova_factorial` (*int* `n_subscripts`, *int* `n_levels`,
    float `y[]`,
    `IMSLS_MODEL_ORDER`, *int* `model_order`,
    `IMSLS_PURE_ERROR`, *or*
    `IMSLS_POOL_INTERACTIONS`,

```
IMSLS_ANOVA_TABLE, float **anova_table,
IMSLS_ANOVA_TABLE_USER, float anova_table[],
IMSLS_TEST_EFFECTS, float **test_effects,
IMSLS_TEST_EFFECTS_USER, float test_effects[],
IMSLS_MEANS, float **means,
IMSLS_MEANS_USER, float means[],
0)
```

Optional Arguments

`IMSLS_MODEL_ORDER`, *int* `model_order` (Input)

Number of factors to be included in the highest-way interaction in the model. Argument `model_order` must be in the interval [1, `n_subscripts` – 1]. For example, a `model_order` of 1 indicates that a main effect model will be analyzed, and a `model_order` of 2 indicates that two-way interactions will be included in the model.
Default: `model_order` = `n_subscripts` – 1

`IMSLS_PURE_ERROR`, *or*
`IMSLS_POOL_INTERACTIONS` (Input)

`IMSLS_PURE_ERROR`, the default option, indicates factor `n_subscripts` is error. Its main effect and all its interaction effects are pooled into the error with the other (`model_order` + 1)-way and higher-way interactions. `IMSLS_POOL_INTERACTIONS` indicates factor `n_subscripts` is not error. Only (`model_order` + 1)-way and higher-way interactions are included in the error.

`IMSLS_ANOVA_TABLE`, *float* `**anova_table` (Output)

Address of a pointer to an internally allocated array of size 15 containing the analysis of variance table. 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)

Element	Analysis of Variance Statistics
11	adjusted R^2 (in percent)
12	estimate of the standard deviation
13	overall mean of y
14	coefficient of variation (in percent)

IMSLS_ANOVA_TABLE_USER, *float* anova_table[] (Output)

Storage for array anova_table is provided by the user. See IMSLS_ANOVA_TABLE.

IMSLS_TEST_EFFECTS, *float* **test_effects (Output)

Address of a pointer to an NEF × 4 internally allocated array containing a matrix containing statistics relating to the sums of squares for the effects in the model. Here,

$$\text{NEF} = \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{\min(n, |\texttt{model\_order}|)}$$

where n is given by n_subscripts if IMSLS_POOL_INTERACTIONS is specified; otherwise, n_subscripts − 1.

Suppose the factors are A, B, C, and error. With model_order = 3, rows 0 through NEF − 1 would correspond to A, B, C, AB, AC, BC, and ABC, respectively. The columns of test_effects are as follows:

Column	Description
0	degrees of freedom
1	sum of squares
2	F-statistic
3	p-value

IMSLS_TEST_EFFECTS_USER, *float* test_effects[] (Output)

Storage for array test_effects is provided by the user. See IMSLS_TEST_EFFECTS.

IMSLS_MEANS, *float* **means (Output)

Address of a pointer to an internally allocated array of length (n_levels [0] + 1) × (n_levels [1] + 1) × … × (n_levels[n − 1] + 1) containing the subgroup means.

See argument IMSLS_TEST_EFFECTS for a definition of n. If the factors are A, B, C, and error, the ordering of the means is grand mean, A means, B means, C means, AB means, AC means, BC means, and ABC means.

IMSLS_MEANS_USER, *float* means[] (Output)

Storage for array means is provided by the user. See IMSLS_MEANS.

Description

Function `imsls_f_anova_factorial` performs an analysis for an *n*-way classification design with balanced data. For balanced data, there must be an equal number of responses in each cell of the *n*-way layout. The effects are assumed to be fixed effects. The model is an extension of the two-way model to include *n* factors. The interactions (two-way, three-way, up to *n*-way) can be included in the model, or some of the higher-way interactions can be pooled into error. The argument `model_order` specifies the number of factors to be included in the highest-way interaction. For example, if three-way and higher-way interactions are to be pooled into error, set `model_order` = 2. (By default, `model_order` = `n_subscripts` – 1 with the last subscript being the error subscript.) Argument `IMSLS_PURE_ERROR` indicates there are repeated responses within the *n*-way cell; `IMSLS_POOL_INTERACTIONS_INTO_ERROR` indicates otherwise.

Function `imsls_f_anova_factorial` requires the responses as input into a single vector *y* in lexicographical order, so that the response subscript associated with the first factor varies least rapidly, followed by the subscript associated with the second factor, and so forth. Hemmerle (1967, Chapter 5) discusses the computational method.

Examples

Example 1

A two-way analysis of variance is performed with balanced data discussed by Snedecor and Cochran (1967, Table 12.5.1, p. 347). The responses are the weight gains (in grams) of rats that were fed diets varying in the source (A) and level (B) of protein. The model is

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk} \qquad i = 1, 2;\ j = 1, 2, 3;\ k = 1, 2, \ldots, 10$$

where

$$\sum_{i=1}^{2} \alpha_i = 0; \sum_{j=1}^{3} \beta_j = 0; \sum_{i=1}^{2} \gamma_{ij} = 0 \qquad \text{for } j = 1, 2, 3; \text{ and } \sum_{j=1}^{3} \gamma_{ij} = 0$$

for *i* = 1, 2. The first responses in each cell in the two-way layout are given in the following table:

	Protein Source (A)		
Protein Level (B)	**Beef**	**Cereal**	**Pork**
High	73, 102, 118, 104, 81, 107, 100, 87, 117, 111	98, 74, 56, 111, 95, 88, 82, 77, 86, 92	94, 79, 96, 98, 102, 102, 108, 91, 120, 105
Low	90, 76, 90, 64, 86, 51, 72, 90, 95, 78	107, 95, 97, 80, 98, 74, 74, 67, 89, 58	49, 82, 73, 86, 81, 97, 106, 70, 61, 82

```
#include <imsls.h>

void main ()
{
    int        n_subscripts= 3;
    int        n_levels[3] = {3,2,10};
    float      p_value;
    float      y[60] = {
        73.0, 102.0, 118.0, 104.0, 81.0,
        107.0, 100.0, 87.0, 117.0, 111.0,
        90.0, 76.0, 90.0, 64.0, 86.0,
        51.0, 72.0, 90.0, 95.0, 78.0,
        98.0, 74.0, 56.0, 111.0, 95.0,
        88.0, 82.0, 77.0, 86.0, 92.0,
        107.0, 95.0, 97.0, 80.0, 98.0,
        74.0, 74.0, 67.0, 89.0, 58.0,
        94.0, 79.0, 96.0, 98.0, 102.0,
        102.0, 108.0, 91.0, 120.0, 105.0,
        49.0, 82.0, 73.0, 86.0, 81.0,
        97.0, 106.0, 70.0, 61.0, 82.0};

    p_value = imsls_f_anova_factorial(n_subscripts, n_levels, y, 0);

    printf("P-value = %10.6f",p_value);
}
```

Output

```
P-value =   0.00229
```

Example 2

In this example, the same model and data is fit as in the initial example, but optional arguments are used for a more complete analysis.

```
#include <imsls.h>

void main ()
{
    int        n_subscripts= 3;
    int        n_levels[3] = {3,2,10};
    float      p_value;
    float      *test_effects, *means, *anova_table;
    float      y[60] = {
        73.0, 102.0, 118.0, 104.0, 81.0,
        107.0, 100.0, 87.0, 117.0, 111.0,
        90.0, 76.0, 90.0, 64.0, 86.0,
        51.0, 72.0, 90.0, 95.0, 78.0,
        98.0, 74.0, 56.0, 111.0, 95.0,
        88.0, 82.0, 77.0, 86.0, 92.0,
        107.0, 95.0, 97.0, 80.0, 98.0,
        74.0, 74.0, 67.0, 89.0, 58.0,
        94.0, 79.0, 96.0, 98.0, 102.0,
        102.0, 108.0, 91.0, 120.0, 105.0,
        49.0, 82.0, 73.0, 86.0, 81.0,
        97.0, 106.0, 70.0, 61.0, 82.0};
    char      *labels[] = {
        "degrees of freedom for the model",
        "degrees of freedom for error",
        "total (corrected) degrees of freedom",
        "sum of squares for the model",
        "sum of squares for error",
        "total (corrected) sum of squares",
        "model mean square", "error mean square",
        "F-statistic", "p-value",
        "R-squared (in percent)","Adjusted R-squared (in percent)",
        "est. standard deviation of the model error",
        "overall mean of y",
        "coefficient of variation (in percent)"};

    char      *test_row_labels[] = {"A", "B", "A*B"};
    char      *test_col_labels[] = {
        "Source", "DF", "Sum of\nSquares",
        "Mean\nSquare", "Prob. of\nLarger F"};

    char      *mean_row_labels[] = {
        "grand mean",
        "A1", "A2", "A3",
        "B1", "B2",
        "A1*B1", "A1*B2", "A2*B1", "A2*B2", "A3*B1", "A3*B2"};
                           /* Perform analysis */
    p_value = imsls_f_anova_factorial(n_subscripts, n_levels, y,
        IMSLS_ANOVA_TABLE,   &anova_table,
        IMSLS_TEST_EFFECTS,  &test_effects,
        IMSLS_MEANS,         &means,
        0);

    printf("P-value = %10.6f",p_value);
                           /* Print results */
    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
        anova_table,
        IMSLS_ROW_LABELS,   labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);

    imsls_f_write_matrix("* * * Variation Due to the Model * * *", 3, 4,
        test_effects,
        IMSLS_ROW_LABELS,   test_row_labels,
        IMSLS_COL_LABELS,   test_col_labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);

    imsls_f_write_matrix("* * * Subgroup Means * * *", 12, 1,
        means,
        IMSLS_ROW_LABELS,   mean_row_labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);
}
```

Output

```
P-value =   0.002299

          * * * Analysis of Variance * * *

degrees of freedom for the model                   5.0000
degrees of freedom for error                      54.0000
total (corrected) degrees of freedom              59.0000
sum of squares for the model                    4612.9346
sum of squares for error                       11585.9990
total (corrected) sum of squares               16198.9336
model mean square                                922.5869
error mean square                                214.5555
F-statistic                                        4.3000
p-value                                            0.0023
R-squared (in percent)                            28.4768
Adjusted R-squared (in percent)                   21.8543
est. standard deviation of the model error        14.6477
overall mean of y                                 87.8667
coefficient of variation (in percent)             16.6704


          * * * Variation Due to the Model * * *
Source          DF       Sum of          Mean     Prob. of
                        Squares        Square     Larger F
A           2.0000     266.5330        0.6211       0.5411
B           1.0000    3168.2678       14.7667       0.0003
A*B         2.0000    1178.1337        2.7455       0.0732


* * * Subgroup Means * * *
  grand mean      87.8667
  A1              89.6000
  A2              84.9000
  A3              89.1000
  B1              95.1333
  B2              80.6000
  A1*B1          100.0000
  A1*B2           79.2000
  A2*B1           85.9000
  A2*B2           83.9000
  A3*B1           99.5000
  A3*B2           78.7000
```

Example 3

This example performs a three-way analysis of variance using data discussed by John (1971, pp. 91–92). The responses are weights (in grams) of roots of carrots grown with varying amounts of applied nitrogen (*A*), potassium (*B*), and phosphorus (*C*). Each cell of the three-way layout has one response. Note that the

ABC interactions sum of squares, which is 186, is given incorrectly by John (1971, Table 5.2.) The three-way layout is given in the following table:

	A_0			A_1			A_2		
	B_0	B_1	B_2	B_0	B_1	B_2	B_0	B_1	B_2
C_0	88.76	91.41	97.85	94.83	100.49	99.75	99.90	100.23	104.51
C_1	87.45	98.27	95.85	84.57	97.20	112.30	92.98	107.77	110.94
C_2	86.01	104.20	90.09	81.06	120.80	108.77	94.72	118.39	102.87

```
#include <imsls.h>

void main ()
{
    int        n_subscripts= 3;
    int        n_levels[3] = {3,3,3};
    float      p_value;
    float      *test_effects, *anova_table;
    float      y[27] = {
         88.76, 87.45, 86.01, 91.41, 98.27, 104.2, 97.85, 95.85,
         90.09, 94.83, 84.57, 81.06, 100.49, 97.2, 120.8, 99.75,
         112.3, 108.77, 99.9, 92.98, 94.72, 100.23, 107.77, 118.39,
         104.51, 110.94, 102.87};
    char      *labels[] = {
        "degrees of freedom for the model",
        "degrees of freedom for error",
        "total (corrected) degrees of freedom",
        "sum of squares for the model",
        "sum of squares for error",
        "total (corrected) sum of squares",
        "model mean square", "error mean square",
        "F-statistic", "p-value",
        "R-squared (in percent)","Adjusted R-squared (in percent)",
        "est. standard deviation of the model error",
        "overall mean of y",
        "coefficient of variation (in percent)"};

    char      *test_row_labels[] = {"A", "B", "C", "A*B", "A*C", "B*C"};
    char      *test_col_labels[] = {
        "Source", "DF", "Sum of\nSquares",
        "Mean\nSquare", "Prob. of\nLarger F"};
                              /* Perform analysis */
    p_value = imsls_f_anova_factorial(n_subscripts, n_levels, y,
        IMSLS_ANOVA_TABLE,    &anova_table,
        IMSLS_TEST_EFFECTS,  &test_effects,
        IMSLS_POOL_INTERACTIONS,
        0);
                              /* Print results */
    printf("P-value = %10.6f",p_value);

    imsls_f_write_matrix("* * * Analysis of Variance * * *\n", 15, 1,
        anova_table,
        IMSLS_ROW_LABELS,   labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);

    imsls_f_write_matrix("* * * Variation Due to the Model * * *", 6, 4,
        test_effects,
        IMSLS_ROW_LABELS,   test_row_labels,
        IMSLS_COL_LABELS,   test_col_labels,
        IMSLS_WRITE_FORMAT, "%11.4f",
        0);

}
```

Output

```
P-value =   0.008299

           * * * Analysis of Variance * * *

degrees of freedom for the model                18.0000
degrees of freedom for error                     8.0000
total (corrected) degrees of freedom            26.0000
sum of squares for the model                  2395.7290
sum of squares for error                       185.7763
total (corrected) sum of squares              2581.5054
model mean square                              133.0961
error mean square                               23.2220
F-statistic                                      5.7315
p-value                                          0.0083
R-squared (in percent)                          92.8036
Adjusted R-squared (in percent)                 76.6116
est. standard deviation of the model error       4.8189
overall mean of y                               98.9619
coefficient of variation (in percent)            4.8695

          * * * Variation Due to the Model * * *
Source           DF        Sum of          Mean     Prob. of
                          Squares        Square     Larger F
A            2.0000      488.3678       10.5152       0.0058
B            2.0000     1090.6559       23.4832       0.0004
C            2.0000       49.1484        1.0582       0.3911
A*B          4.0000      142.5856        1.5350       0.2804
A*C          4.0000       32.3474        0.3482       0.8383
B*C          4.0000      592.6240        6.3800       0.0131
```

anova_nested

Analyzes a completely nested random model with possibly unequal numbers in the subgroups.

Synopsis

#include `<imsls.h>`

float `*imsls_f_anova_nested` (*int* `n_factors`, *int* `equal_option`, *int* `n_levels[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_anova_nested`.

Required Arguments

int `n_factors` (Input)
Number of factors (number of subscripts) in the model, including error.

int `equal_option` (Input)
Equal numbers option.

equal_option	**Description**
0	Unequal numbers in the subgroups
1	Equal numbers in the subgroups

int `n_levels[]` (Input)
Array with the number of levels.

If `equal_option` = 1, `n_levels` is of length `n_factors` and contains the number of levels for each of the factors. In this case, the following additional variables are referred to in the description of `anova_nested`:

Variable	**Description**
`LNL`	`n_levels[0] + n_levels[0] * n_levels[1] + ... + n_levels[0] * n_levels[1] * ... * n_levels[n_factors - 2]`
`LNLNF`	`n_levels[0] * n_levels[1] * ...* n_levels[n_factors - 2]`
`NOBS`	The number of observations. `NOBS` equals `n_levels[0] * n_levels[1] * ... * n_levels[n_factors-1].`

If `equal_option` = 0, `n_levels` contains the number of levels of each factor at each level of the factor in which it is nested. In this case, the following additional variables are referred to in the description of `anova_nested`:

Variable	**Description**
`LNL`	Length of `n_levels`.
`LNLNF`	Length of the subvector of `n_levels` for the last factor.
`NOBS`	Number of observations. `NOBS` equals the sum of the last `LNLNF` elements of `n_levels`.

For example, a random one-way model with two groups, five responses in the first group and ten in the second group, would have `LNL`= 3, `LNLNF`= 2, `NOBS` = 15, `n_levels[0] = 2, n_levels[1] = 5, and n_levels[2] = 10.`

float `y[]` (Input)
Array of length `NOBS` containing the responses. The elements of `Y` are ordered lexicographically, i.e., the last model subscript changes most

rapidly, the next to last model subscript changes the next most rapidly, and so forth, with the first subscript changing the slowest.

Return Value

The *p*-value for the F-statistic, `anova_table[9]`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_anova_nested` (*int* `n_factors`, *int* `equal_option`, *int* `n_levels[]`, *float* `y[]`,
`IMSLS_ANOVA_TABLE`, *float* `**anova_table`,
`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]`
`IMSLS_CONFIDENCE`, *float* `confidence`,
`IMSLS_VARIANCE_COMPONENTS`, *float* `**variance_components`,
`IMSLS_VARIANCE_COMPONENTS_USER`, *float* `variance_components[]`,
`IMSLS_EMS`, *float* `**expect_mean_sq`,
`IMSLS_EMS_USER`, *float* `expect_mean_sq[]`,
`IMSLS_Y_MEANS`, *float* `**y_means`,
`IMSLS_Y_MEANS_USER`, *float* `y_means[]`,
0)

Optional Arguments

`IMSLS_ANOVA_TABLE`, *float* `**anova_table`, (Output)
Address of a pointer to an internally allocated array of size 15 containing the analysis of variance table. The analysis of variance statistics are 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)

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for array anova_table is provided by the user.
See `IMSLS_ANOVA_TABLE`.

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Confidence level for two-sided interval estimates on the variance components, in percent. `confidence` percent confidence intervals are computed, hence, `confidence` must be in the interval `[0.0, 100.0)`. `confidence` often will be `90.0, 95.0, or 99.0`. For one-sided intervals with confidence level `ONECL`, `ONECL` in the interval `[50.0, 100.0)`, set `confidence = 100.0 - 2.0 * (100.0 - ONECL)`.
Default: `confidence = 95.0`

`IMSLS_VARIANCE_COMPONENTS`, *float* `**variance_components`, (Output)
Address to a pointer to an internally allocated array. `variance_components` is an `n_factors` by 9 matrix containing statistics relating to the particular variance components in the model. Rows of `variance_components` correspond to the `n_factors` factors. Columns of `variance_components` are as follows:

Column	**Description**
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F -statistic
5	*p*-value for F test
6	Variance component estimate
7	Percent of variance of variance explained by variance component
8	Lower endpoint for a confidence interval on the variance component
9	Upper endpoint for a confidence interval on the variance component

A test for the error variance equal to zero cannot be performed. `variance_components(n_factors, 4)` and `variance_components(n_factors, 5)` are set to NaN (not a number).

IMSLS_VARIANCE_COMPONENTS_USER, *float* variance_components[]
(Output) Storage for array variance_components is provided by the user. See IMSLS_VARIANCE_COMPONENTS.

IMSLS_EMS, *float* **expect_mean_sq, (Output)
Address to a pointer to an internally allocated array of length with expected mean square coefficients.

IMSLS_EMS_USER, *float* expect_mean_sq[], (Output)
Storage for array expect_mean_sq is provided by the user. See IMSLS_EMS.

IMSLS_Y_MEANS, *float* **y_means (Output)
Address to a pointer to an internally allocated array containing the subgroup means.

Equal options	**Length of y means**
0	1 + n_levels[0] + n_levels[1] + … n_levels[(LNL - LNLNF)-1] (See the description of argument n_levels for definitions of LNL and LNLNF.)
1	1 + n_levels[0] + n_levels[0] * n_levels[1] + … + n_levels[0]* n_levels[1] * … * n_levels[n_factors - 2]

If the factors are labeled *A*, *B*, *C*, and error, the ordering of the means is grand mean, *A* means, *AB* means, and then *ABC* means.

IMSLS_Y_MEANS_USER, *float* y_means[], Storage for array y_means
is provided by the user. See IMSLS_Y_MEANS

Description

Routine imsls_f_anova_nested analyzes a nested random model with equal or unequal numbers in the subgroups. The analysis includes an analysis of variance table and computation of subgroup means and variance component estimates. Anderson and Bancroft (1952, pages 325−330) discuss the methodology. The analysis of variance method is used for estimating the variance components. This method solves a linear system in which the mean squares are set to the expected mean squares. A problem that Hocking (1985, pages 324−330) discusses is that this method can yield negative variance component estimates. Hocking suggests a diagnostic procedure for locating the cause of a negative estimate. It may be necessary to reexamine the assumptions of the model.

Example 1

An analysis of a three-factor nested random model with equal numbers in the subgroups is performed using data discussed by Snedecor and Cochran (1967, Table 10.16.1, pages 285−288). The responses are calcium concentrations (in percent, dry basis) as measured in the leaves of turnip greens. Four plants are taken at random, then three leaves are randomly selected from each plant. Finally, from each selected leaf two samples are taken to determine calcium concentration. The model is

$$y_{ijk} = \mu + \alpha_i + \beta_{ij} + e_{ijk} \quad i = 1, 2, 3, 4; j = 1, 2, 3; k = 1, 2$$

where y_{ijk} is the calcium concentration for the k-th sample of the j-th leaf of the i-th plant, the α_i's are the plant effects and are taken to be independently distributed

$$N(0, \sigma^2)$$

the β_{ij}'s are leaf effects each independently distributed

$$N(0, \sigma_\beta^2)$$

and the ε_{ijk}'s are errors each independently distributed $N(0, \sigma^2)$. The effects are all assumed to be independently distributed. The data are given in the following table:

Plant	Leaf	Samples	
1	1	3.28	3.09
	2	3.52	3.48
	3	2.88	2.80
2	1	2.46	2.44
	2	1.87	1.92
	3	2.19	2.19
3	1	2.77	2.66
	2	3.74	3.44
	3	2.55	2.55
4	1	3.78	3.87
	2	4.07	4.12
	3	3.31	3.31

```
#include <imsls.h>
#include <stdio.h>
#define Mfloat float
void main()
{
       Mfloat pvalue, *aov, *varc, *ymeans, *ems;
       Mfloat y[] = {3.28, 3.09, 3.52, 3.48, 2.88, 2.80, 2.46, 2.44, 1.87,
              1.92, 2.19, 2.19, 2.77, 2.66, 3.74, 3.44, 2.55, 2.55, 3.78,
              3.87, 4.07, 4.12, 3.31, 3.31};
       int n_levels[] = {4, 3, 2};
```

```
char    *aov_labels[] = {
             "degrees of freedom for model",
             "degrees of freedom for error",
             "total (corrected) degrees of freedom",
             "sum of squares for model",
             "sum of squares for error",
             "total (corrected) sum of squares",
             "model mean square",
             "error mean square",
             "F-statistic",
             "p-value",
               "R-squared (in percent)",
             "adjusted R-squared (in percent)",
             "est. standard deviation of within error",
             "overall mean of y",
             "coefficient of variation (in percent)"};
char    *ems_labels[] = {
               "Effect A and Error",
               "Effect A and Effect B",
               "Effect A and Effect A",
               "Effect B and Error",
               "Effect B and Effect B",
               "Error and Error"};
char    *means_labels[] = {
               "Grand mean",
               " A means 1",
               " A means 2",
               " A means 3",
               " A means 4",
               "AB means 1 1",
               "AB means 1 2",
               "AB means 1 3",
               "AB means 2 1",
               "AB means 2 2",
               "AB means 2 3",
               "AB means 3 1",
               "AB means 3 2",
               "AB means 3 3",
               "AB means 4 1",
               "AB means 4 2",
               "AB means 4 3"};
char    *components_labels[] = {
             "degrees of freedom for A",
             "sum of squares for A",
             "mean square of A",
             "F-statistic for A",
             "p-value for A",
               "Estimate of A",
               "Percent Variation Explained by A",
               "95% Confidence Interval Lower Limit for A",
               "95% Confidence Interval Upper Limit for A",
               "degrees of freedom for B",
             "sum of squares for B",
             "mean square of B",
             "F-statistic for B",
             "p-value for B",
               "Estimate of B",
               "Percent Variation Explained by B",
               "95% Confidence Interval Lower Limit for B",
                    "95% Confidence Interval Upper Limit for B",
                    "degrees of freedom for Error",
                  "sum of squares for Error",
                  "mean square of Error",
                  "F-statistic for Error",
                  "p-value for Error",
                    "Estimate of Error",
                    "Percent Explained by Error",
                    "95% Confidence Interval Lower Limit for Error",
                    "95% Confidence Interval Upper Limit for Error"};

      pvalue = imsls_f_anova_nested(3, 1, n_levels, y,
                            IMSLS_ANOVA_TABLE, &aov,
                            IMSLS_Y_MEANS, &ymeans,
                            IMSLS_VARIANCE_COMPONENTS, &varc,
                            IMSLS_EMS, &ems,
                            0);

      printf("pvalue = %f\n", pvalue);
      imsls_f_write_matrix("* * * Analysis of Variance * * *", 15, 1, aov,
                    IMSLS_ROW_LABELS, aov_labels,
                    IMSLS_WRITE_FORMAT, "%10.5f",
                    0);
      imsls_f_write_matrix("* * * Expected Mean Square Coefficients * * *"
                    6, 1, ems,
                    IMSLS_ROW_LABELS, ems_labels,
                    IMSLS_WRITE_FORMAT, "%6.2f",
                    0);
      imsls_f_write_matrix("* * * Means * * *", 17, 1, ymeans,
                    IMSLS_ROW_LABELS, means_labels,
                    IMSLS_WRITE_FORMAT, "%6.2f",
                    0);
      imsls_f_write_matrix("* * Analysis of Variance / Variance Components * *",
                    27, 1, varc,
                    IMSLS_ROW_LABELS, components_labels,
                    IMSLS_WRITE_FORMAT, "%10.5f",
                    0);
}
```

Output

```
pvalue = 0.079854

       * * * Analysis of Variance * * *
degrees of freedom for model                    11.00000
degrees of freedom for error                    12.00000
total (corrected) degrees of freedom            23.00000
sum of squares for model                        10.19054
sum of squares for error                         0.07985
total (corrected) sum of squares                10.27040
model mean square                                0.92641
error mean square                                0.00665
F-statistic                                    139.21599
p-value                                          0.00000
R-squared (in percent)                          99.22248
adjusted R-squared (in percent)                 98.50976
est. standard deviation of within error          0.08158
overall mean of y                                3.01208
coefficient of variation (in percent)            2.70826
```

```
       * * * Expected Mean Square Coefficients * * *
Effect A and Error                      1.00
Effect A and Effect B                   2.00
Effect A and Effect A                   6.00
Effect B and Error                      1.00
Effect B and Effect B                   2.00
Error and Error                         1.00

       * * * Means * * *
Grand mean                 3.01
A means 1                  3.17
A means 2                  2.18
A means 3                  2.95
A means 4                  3.74
AB means 1 1               3.18
AB means 1 2               3.50
AB means 1 3               2.84
AB means 2 1               2.45
AB means 2 2           1.89
AB means 2 3           2.19
AB means 3 1           2.72
AB means 3 2           3.59
AB means 3 3           2.55
AB means 4 1           3.82
AB means 4 2           4.10
AB means 4 3           3.31

       * * Analysis of Variance / Variance Components * *
degrees of freedom for A                                3.00000
sum of squares for A                                    7.56034
mean square of A                                        2.52011
F-statistic for A                                       7.66516
p-value for A                                           0.00973
Estimate of A                                           0.36522
Percent Variation Explained by A                       68.53015
95% Confidence Interval Lower Limit for A               0.03955
95% Confidence Interval Upper Limit for A               5.78674
degrees of freedom for B                                8.00000
sum of squares for B                                    2.63020
mean square of B                                        0.32878
F-statistic for B                                      49.40642
p-value for B                                           0.00000
Estimate of B                                           0.16106
Percent Variation Explained by B                       30.22121
95% Confidence Interval Lower Limit for B               0.06967
95% Confidence Interval Upper Limit for B               0.60042
degrees of freedom for Error                           12.00000
sum of squares for Error                                0.07985
mean square of Error                                    0.00665
F-statistic for Error                               ***********
p-value for Error                                   ***********
Estimate of Error                                       0.00665
Percent Explained by Error                              1.24864
95% Confidence Interval Lower Limit for Error           0.00342
95% Confidence Interval Upper Limit for Error           0.01813
```

anova_balanced

Analyzes a balanced complete experimental design for a fixed, random, or mixed model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_anova_balanced` (*int* `n_factors`, *int* `n_levels[]`, *float* `y[]`, *int* `n_random`, *int* `index_random_factor[]`, *int* `n_model_effects`, *int* `n_factors_per_effect[]`, *int* `index_factor_per_effect[]`, ..., 0)

The type *double* function is `imsls_d_anova_balanced`.

Required Arguments

int `n_factors` (Input)
Number of factors (number of subscripts) in the model, including error.

int `n_levels[]` (Input)
Array of length `n_factors` containing the number of levels for each of the factors.

float `y[]` (Input)
Array of length `n_levels[0] * n_levels[1] *...* n_levels[n_factors-1]` containing the responses. `y[]` must not contain NaN (not a number) for any of its elements, i.e., missing values are not allowed.

int `n_random` (Input)
For positive `n_random`, `|n_random|` is the number of random factors. For negative `n_random`, `|n_random|` is the number of random effects (sources of variation).

int `index_random_factor[]` (Input)
Index array of length `|n_random|` containing either the factor numbers to be considered random (for `n_random` positive) or containing the effect numbers to be considered random (for `n_random` negative). If `n_random` = 0, `index_random_factor` is not referenced.

int `n_model_effects` (Input)
Number of effects (sources of variation) due to the model excluding the overall mean and error.

int `n_factors_per_effect[]` (Input)
Array of length `n_model_effects` containing the number of factors associated with each effect in the model.

int `index_factor_per_effect[]` (Input)
Index vector of length `n_factors_per_efffect[0] +`

n_factors_per_effect[1] + . . . + n_factors_per_effect[n_model_effects-1]. The first n_factors_per_effect[0] elements give the factor numbers in the first effect. The next n_factors_per_effect[1] elements give the factor numbers in the second effect. The last n_factors_per_effect [n_model_effects-1] elements give the factor numbers in the last effect. Main effects must appear before their interactions. In general, an effect *E* cannot appear after an effect *F* if all of the indices for *E* appear also in *F*.

Return Value

The *p*-value for the *F*-statistic.

Synopsis with Optional Arguments

#include <imsls.h>

float *imsls_f_anova_balanced (*int* n_factors, *int* n_levels[], *float* y[], *int* n_random, *int* index_random_factor[], *int* n_model_effects, *int* n_factors_per_effect[], *int* index_factor_per_effect[],
IMSLS_ANOVA_TABLE, *float* **anova_table,
IMSLS_ANOVA_TABLE_USER, *float* anova_table[]
IMSLS_MODEL, *int* model,
IMSLS_CONFIDENCE, *float* confidence,
IMSLS_VARIANCE_COMPONENTS, *float* **variance_components,
IMSLS_VARIANCE_COMPONENTS_USER, *float* variance_components[],
IMSLS_EMS, *float* **ems,
IMSLS_EMS_USER, *float* ems[],
IMSLS_Y_MEANS, *float* **y_means,
IMSLS_Y_MEANS_USER, *float* y_means[],
0)

Optional Arguments

IMSLS_ANOVA_TABLE, *float* **anova_table, (Output)
Address of a pointer to an internally allocated array of size 15 containing the analysis of variance table. The analysis of variance statistics are 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)

`IMSLS_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for array anova_table is provided by the user.
See `IMSLS_ANOVA_TABLE`.

`IMSLS_MODEL`, *int* model, (Input)
Model Option

MODEL	Meaning
0	Searle model
1	Scheffe model

For the Scheffe model, effects corresponding to interactions of fixed and random factors have their sum over the subscripts corresponding to fixed factors equal to zero. Also, the variance of a random interaction effect involving some fixed factors has a multiplier for the associated variance component that involves the number of levels in the fixed factors. The Searle model has no summation restrictions on the random interaction effects and has a multiplier of one for each variance component. The default is model = 0.

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Confidence level for two-sided interval estimates on the variance components, in percent. `confidence` percent confidence intervals are computed, hence, `confidence must be in the interval [0.0, 100.0). confidence often will be 90.0, 95.0, or 99.0. For one-sided intervals with confidence level α, α in the interval [50.0, 100.0), set confidence = 100.0 - 2.0 * 100.0 - α).`
`Default: confidence = 95.0`

`IMSLS_VARIANCE_COMPONENTS`, *float* **`variance_components`, (Output)
Address of a pointer to an array, `variance_components`. `variance_components` is an `(n_model_effects` + 1) by 9 array containing statistics relating to the particular variance components or effects in the model and the error. Rows of `variance_components` correspond to the `n_model_effects` effects plus error.

Element	**Description**
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	*F* -statistic
5	*p*-value for *F* test
6	Variance component estimate
7	Percent of variance of y explained by random effect
8	Lower endpoint for a confidence interval on the variance component
9	Upper endpoint for a confidence interval on the variance component

Elements 6 through 9 contain NaN (not a number) if the effect is fixed, i.e., if there is no variance component to be estimated. If the variance component estimate is negative, columns 8 and 9 contain NaN.

`IMSLS_VARIANCE_COMPONENTS_USER`, *float* `variance_components[]` (Output)
Storage for array variance_components is provided by the user. See `IMSLS_VARIANCE_COMPONENTS`.

`IMSLS_EMS`, *float* `**ems`, (Output)
Address of a pointer to an internally allocated array of length `(n_model_effects + 1) * (n_model_effects + 2)/2` containing expected mean square coefficients. Suppose the effects are *A*, *B*, and *AB*. The ordering of the coefficients in `ems` is as follows:

	Error	*AB*	*B*	*A*
A	`ems`[0]	`ems`[1]	`ems`[2]	`ems`[2
B	`ems`[4]	`ems`[5]	`ems`[6]	

	Error	*AB*	*B*	*A*
AB	`ems`[7]	`ems`[8]		
Error	`ems`[9]			

`IMSLS_EMS_USER`, *float* `ems[]` (Output)
    Storage for `ems` is provided by the user.
    See `IMSLS_EMS`.

`IMSLS_Y_MEANS`, *float* `**y_means` (Output)
    Address of a pointer to an internally allocated array of length (`n_levels`(0) + 1) `*` (`n_levels` (1) + 1) `* . . . *` (`n_levels` (n-1) + 1) containing the subgroup means. Suppose the factors are A, B, and C. The ordering of the means is grand mean, A means, B means, C means, AB means, AC means, BC means, and ABC means.

`IMSLS_Y_MEANS_USER`, *float* `y_means` (Output)
    Storage for `y_means` is provided by the user.
    See `IMSLS_Y_MEANS`.

Description

Function `imsls_f_anova_balanced` analyzes a balanced complete experimental design for a fixed, random, or mixed model. The analysis includes an analysis of variance table, and computation of subgroup means and variance component estimates. A choice of two parameterizations of the variance components for the model can be made.

Scheffé (1959, pages 274–289) discusses the parameterization for `model` = 1. For example, consider the following model equation with fixed factor *A* and random factor *B*:

$$y_{ijk} = \mu + \alpha_i + b_j + c_{ij} + e_{ijk} \quad i = 1, 2, \ldots, a; j = 1, 2, \ldots, b; k = 1, 2, \ldots, n$$

The fixed effects α_i's are subject to the restriction

$$\sum_{i=1}^{a} \alpha_i = 0$$

the b_j's are random effects identically and independently distributed

$$N(0, \sigma_B^2)$$

c_{ij} are interaction effects each distributed

$$N(0, \frac{a-1}{a}\sigma_{AB}^2)$$

and are subject to the restrictions

$$\sum_{i=1}^{a} c_{ij} = 0 \text{ for } j = 1, 2, ..., b$$

and the e_{ijk}'s are errors identically and independently distributed $N(0, \sigma^2)$. In general, interactions of fixed and random factors have sums over subscripts corresponding to fixed factors equal to zero. Also in general, the variance of a random interaction effect is the associated variance component times a product of ratios for each fixed factor in the random interaction term. Each ratio depends on the number of levels in the fixed factor. In the earlier example, the random interaction AB has the ratio $(a-1)/a$ as a multiplier of

$$\sigma_{AB}^2$$

and

$$\operatorname{var}(y_{ijk}) = \sigma_B^2 + \frac{a-1}{a}\sigma_{AB}^2 + \sigma^2$$

In a three-way crossed classification model, an ABC interaction effect with A fixed, B random, and C fixed would have variance

$$\frac{(a-1)(c-1)}{ac}\sigma_{ABC}^2$$

Searle (1971, pages 400–401) discusses the parameterization for `model` = 0. This parameterization does not have the summation restrictions on the effects corresponding to interactions of fixed and random factors. Also, the variance of each random interaction term is the associated variance component, i.e., without the multiplier. This parameterization is also used with unbalanced data, which is one reason for its popularity with balanced data also. In the earlier example,

$$\operatorname{var}\left(y_{ijk}\right) = \tilde{\sigma}_B^2 + \tilde{\sigma}_{AB}^2 + \sigma^2$$

Searle (1971, pages 400–404) compares these two parameterizations. Hocking (1973) considers these different parameterizations and concludes they are equivalent because they yield the same variance-covariance structure for the responses. Differences in covariances for individual terms, differences in expected mean square coefficients and differences in F tests are just a consequence of the definition of the individual terms in the model and are not caused by any fundamental differences in the models. For the earlier two-way model, Hocking states that the relations between the two parameterizations of the variance components are

$$\sigma_B^2 = \tilde{\sigma}_B^2 + \tfrac{1}{a}\tilde{\sigma}_{AB}^2$$
$$\sigma_{AB}^2 = \tilde{\sigma}_{AB}^2$$

where

$$\tilde{\sigma}_B^2 \text{ and } \tilde{\sigma}_{AB}^2$$

are the variance components in the parameterization with `model` = 0.

The computations for degrees of freedom and sums of squares are the same regardless of the option specified by `model`. `imsls_f_anova_balanced` first computes degrees of freedom and sum of squares for a full factorial design. Degrees of freedom for effects in the factorial design that are missing from the specified model are pooled into the model effect containing the fewest subscripts but still containing the factorial effect. If no such model effect exists, the factorial effect is pooled into error. If more than one such effect exists, a terminal error message is issued indicating a misspecified model.

The analysis of variance method is used for estimating the variance components. This method solves a linear system in which the mean squares are set to the expected mean squares. A problem that Hocking (1985, pages 324−330) discusses is that this method can yield a negative variance component estimate. Hocking suggests a diagnostic procedure for locating the cause of the negative estimate. It may be necessary to re-examine the assumptions of the model.

The percentage of variation explained by each random effect is computed (output in `variance_components` element `7`) as the variance of the associated random effect divided by the variance of *y*. The two parameterizations can lead to different values because of the different definitions of the individual terms in the model. For example, the percentage associated with the *AB* interaction term in the earlier two-way mixed model is computed for `model` = 1 using the formula

$$\%\text{ variation(AB|Model=1)} = \frac{\frac{a-1}{a}\sigma_{AB}^2}{\sigma_B^2 + \frac{a-1}{a}\sigma_{AB}^2 + \sigma^2}$$

while for the parameterization `model` = 0, the percentage is computed using the formula

$$\%\text{ variation(AB|Model=0)} = \frac{\tilde{\sigma}_{AB}^2}{\tilde{\sigma}_B^2 + \tilde{\sigma}_{AB}^2 + \sigma^2}$$

In each case, the variance components are replaced by their estimates (stored in `variance_components` element `6`).

Confidence intervals on the variance components are computed using the method discussed by Graybill (1976, Theorem 15.3.5, page 624, and Note 4, page 620).

Example 1

An analysis of a generalized randomized block design is performed using data discussed by Kirk (1982, Table 6.10-1, pages 293−297). The model is

$$y_{ijk} = \mu + \alpha_i + b_j + c_{ij} + e_{ijk} \qquad i = 1, 2, 3, 4; j = 1, 2, 3, 4; k = 1, 2$$

where y_{ijk} is the response for the *k*-th experimental unit in block *j* with treatment *i*; the α_i's are the treatment effects and are subject to the restriction

$$\sum_{i=1}^{2} \alpha_i = 0$$

the b_j's are block effects identically and independently distributed

$$N(0, \sigma_B^2)$$

c_{ij} are interaction effects each distributed

$$N(0, \tfrac{3}{4}\sigma_{AB}^2)$$

and are subject to the restrictions

$$\sum_{i=1}^{4} c_{ij} = 0 \text{ for } j = 1, 2, 3, 4$$

and the e_{ijk}'s are errors, identically and independently distributed $N(0, \sigma^2)$. The interaction effects are assumed to be distributed independently of the errors.

The data are given in the following table:

	Block			
Treatment	**1**	**2**	**3**	**4**
1	3, 6	3, 1	2, 2	3, 2
2	4, 5	4, 2	3, 4	3, 3
3	7, 8	7, 5	6, 5	6, 6
4	7, 8	9, 10	10, 9	8, 11

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

void main()
{
  float pvalue = -99.;
  int n_levels[] = {4, 4, 2};
  int indrf[] = {2, 3};
  int nfef[] = {1, 1, 2};
  int indef[] = {1, 2, 1, 2};
  float y[] = {3.0, 6.0, 3.0, 1.0, 2.0, 2.0, 3.0, 2.0, 4.0, 5.0, 4.0,
             2.0, 3.0, 4.0, 3.0, 3.0, 7.0, 8.0, 7.0, 5.0, 6.0, 5.0,
             6.0, 6.0, 7.0, 8.0, 9.0, 10.0, 10.0, 9.0, 8.0, 11.0};
  float *aov=NULL, *y_means, *variance_components, *ems;

  char    *aov_labels[] = {
                   "degrees of freedom for model",
                   "degrees of freedom for error",
                   "total (corrected) degrees of freedom",
                   "sum of squares for model",
                   "sum of squares for error",
                   "total (corrected) sum of squares",
                   "model mean square",
                   "error mean square",
                   "F-statistic",
                   "p-value",
                   "R-squared (in percent)",
                   "adjusted R-squared (in percent)",
                   "est. standard deviation of within error",
                   "overall mean of y",
                   "coefficient of variation (in percent)"};
  char    *ems_labels[] = {
                     "Effect A and Error",
                     "Effect A and Effect AB",
                     "Effect A and Effect B",
                     "Effect A and Effect A",
                     "Effect B and Error",
                     "Effect B and Effect AB",
                     "Effect B and Effect B",
                     "Effect AB and Error",
                     "Effect AB and Effect AB",
                     "Error and Error"};
  char    *means_labels[] = {
                     "Grand mean",
                     " A means 1",
                     " A means 2",
                     " A means 3",
                     " A means 4",
                     " B means 1",
                     " B means 2",
                     " B means 3",
                     " B means 4",
                     "AB means 1 1",
                     "AB means 1 2",
                     "AB means 1 3",
                     "AB means 1 4",
                     "AB means 2 1",
                     "AB means 2 2",
                     "AB means 2 3",
                     "AB means 2 4",
                     "AB means 3 1",
                     "AB means 3 2",
                     "AB means 3 3",
                     "AB means 3 4",
                     "AB means 4 1",
                     "AB means 4 2",
                     "AB means 4 3",
                     "AB means 4 4",};
  char    *components_labels[] = {
                   "degrees of freedom for A",
                   "sum of squares for A",
                   "mean square of A",
                   "F-statistic for A",
                   "p-value for A",
                   "Estimate of A",
                   "Percent Variation Explained by A",
                   "95% Confidence Interval Lower Limit for A",
                   "95% Confidence Interval Upper Limit for A",
                   "degrees of freedom for B",
                   "sum of squares for B",
                   "mean square of B",
                   "F-statistic for B",
                   "p-value for B",
                   "Estimate of B",
                     "Percent Variation Explained by B",
                     "95% Confidence Interval Lower Limit for B",
                     "95% Confidence Interval Upper Limit for B",
                     "degrees of freedom for AB",
                     "sum of squares for AB",
                     "mean square of AB",
                     "F-statistic for AB",
                     "p-value for AB",
                     "Estimate of AB",
                     "Percent Variation Explained by AB",
                     "95% Confidence Interval Lower Limit for AB",
                     "95% Confidence Interval Upper Limit for AB",
                     "degrees of freedom for Error",
                     "sum of squares for Error",
                     "mean square of Error",
                     "F-statistic for Error",
                     "p-value for Error",
                     "Estimate of Error",
                     "Percent Explained by Error",
                     "95% Confidence Interval Lower Limit for Error",
                     "95% Confidence Interval Upper Limit for Error"};

pvalue = imsls_f_anova_balanced(3, n_levels, y, 2, indrf, 3, nfef, indef,
                             IMSLS_MODEL, 1,
                             IMSLS_EMS, &ems,
                             IMSLS_VARIANCE_COMPONENTS, &variance_components,
                             IMSLS_Y_MEANS, &y_means,
                             IMSLS_ANOVA_TABLE, &aov,
                             0);

printf("p value of F statistic = %f\n", pvalue);
imsls_f_write_matrix("* * * Analysis of Variance * * *", 15, 1, aov,
                            IMSLS_ROW_LABELS, aov_labels,
                            IMSLS_WRITE_FORMAT, "%10.5f",
                            0);
imsls_f_write_matrix("* * * Expected Mean Square Coefficients * * *",
                            10, 1, ems,
                            IMSLS_ROW_LABELS, ems_labels,
                            IMSLS_WRITE_FORMAT, "%6.2f",
                            0);
imsls_f_write_matrix("* * Analysis of Variance / Variance Components * *",
                            36, 1,
              variance_components,
                            IMSLS_ROW_LABELS, components_labels,
                            IMSLS_WRITE_FORMAT, "%10.5f",
                            0);
imsls_f_write_matrix("means", 25, 1, y_means,
                            IMSLS_ROW_LABELS, means_labels,
                            IMSLS_WRITE_FORMAT, "%6.2f",
                            0);

}
```

Output

```
p value of F statistic = 0.000005
          * * * Analysis of Variance * * *

degrees of freedom for model                 15.00000
degrees of freedom for error                 16.00000
```

```
total (corrected) degrees of freedom          31.00000
 sum of squares for model                    216.50000
 sum of squares for error                     19.00000
 total (corrected) sum of squares            235.50000
 model mean square                            14.43333
 error mean square                             1.18750
 F-statistic                                  12.15439
 p-value                                       0.00000
 R-squared (in percent)                       91.93206
 adjusted R-squared (in percent)              84.36836
 est. standard deviation of within error       1.08972
overall mean of y                              5.37500
 coefficient of variation (in percent)  20.27395

            * * * Expected Mean Square Coefficients * * *
Effect A and Error                                 1.00
Effect A and Effect AB                             2.00
Effect A and Effect B                              0.00
Effect A and Effect A                              8.00
Effect B and Error                                 1.00
Effect B and Effect AB                             0.00
Effect B and Effect B                              8.00
Effect AB and Error                                1.00
Effect AB and Effect AB                            2.00
Error and Error                                    1.00

         * * Analysis of Variance / Variance Components * *
 degrees of freedom for A                              3.00000
 sum of squares for A                                194.50000
 mean square of A                                     64.83334
 F-statistic for A                                    32.87324
 p-value for A                                         0.00004
 Estimate of A                                      ..........
 Percent Variation Explained by A                   ..........
 95% Confidence Interval Lower Limit for A          ..........
 95% Confidence Interval Upper Limit for A          ..........
 degrees of freedom for B                              3.00000
 sum of squares for B                                  4.25000
 mean square of B                                      1.41667
 F-statistic for B                                     1.19298
 p-value for B                                         0.34396
 Estimate of B                                         0.02865
 Percent Variation Explained by B                      1.89655
 95% Confidence Interval Lower Limit for B             0.00000
 95% Confidence Interval Upper Limit for B             2.31682
 degrees of freedom for AB                             9.00000
 sum of squares for AB                                17.75000
 mean square of AB                                     1.97222
 F-statistic for AB                                    1.66082
 p-value for AB                                        0.18016
 Estimate of AB                                        0.39236
 Percent Variation Explained by AB                    19.48276
 95% Confidence Interval Lower Limit for AB            0.00000
 95% Confidence Interval Upper Limit for AB            2.75803
 degrees of freedom for Error                         16.00000
 sum of squares for Error                             19.00000
 mean square of Error                                  1.18750
 F-statistic for Error                              ..........
 p-value for Error                                  ..........
```

```
Estimate of Error                                  1.18750
Percent Explained by Error                        78.62069
95% Confidence Interval Lower Limit for Error      0.65868
95% Confidence Interval Upper Limit for Error      2.75057


     means
     Grand mean          5.38
     A means 1           2.75
     A means 2           3.50
     A means 3           6.25
     A means 4           9.00
     B means 1           6.00
     B means 2           5.13
     B means 3           5.13
     B means 4           5.25
     AB means 1 1        4.50
     AB means 1 2        2.00
     AB means 1 3        2.00
     AB means 1 4        2.50
     AB means 2 1        4.50
     AB means 2 2        3.00
     AB means 2 3        3.50
     AB means 2 4        3.00
     AB means 3 1        7.50
     AB means 3 2        6.00
     AB means 3 3        5.50
     AB means 3 4        6.00
     AB means 4 1        7.50
     AB means 4 2        9.50
     AB means 4 3         9.50
     AB means 4 4        9.50
```

crd_factorial

Analyzes data from balanced and unbalanced completely randomized experiments. Funtion `crd_factorial` does permit a factorial treatment structure. However, unlike `anova_factorial`, function `crd_factorial` allows for missing data, unequal replication and one or more locations.

Synopsis

#include `<imsls.h>`

float * `imsls_f_crd_factorial` (*int* `n_obs`, *int* `n_locations`,
int `n_factors`, *int* `n_levels[]`, *int* `model[]`, *float* `y[]`, …, `0`)

The type *double* function is `imsls_d_crd_factorial`.

Required Arguments

int `n_obs` (Input)
Number of missing and non-missing experimental observations.

int `n_locations` (Input)

Number of locations. `n_locations` must be one or greater.

int `n_factors` (Input)

Number of factors in the model.

int `n_levels[]` (Input)

Array of length `n_factors`+1. The `n_levels[0]` through `n_levels`[`n_factors`-1] contain the number of levels for each factor. The last element, `n_levels`[`n_factors`], contains the number of replicates for each treatment combination within a location.

int `model[]` (Input)

A `n_obs` by (`n_factors`+1) array identifying the location and factor levels associated with each observation in `y`. The first column must contain the location identifier and the remaining columns the factor level identifiers in the same order used in `n_levels`. If `n_locations` = 1, the first column is still required, but its contents are ignored.

float `y[]` (Input)

An aray of length `n_obs` containing the experimental observations and any missing values. Missing values are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively.

Return Value

A pointer to the memory location of a two dimensional, `n_anova` by 6 array containing the ANOVA table, where:

$$\texttt{n\_anova} = a + \sum_{i=1}^{m} \binom{\texttt{n\_factors}}{i},$$

where

$$a = \begin{cases} 2 & \text{if } \texttt{n\_locations} = 1 \\ 3 & \text{if } \texttt{n\_locations} > 1 \text{ and treatments are not replicated} \\ 4 & \text{if } \texttt{n\_locations} = 1 \text{ and treatments are replicated at each location} \end{cases}$$

Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table`[i*6], is the source identifier which identifies the type of effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

J	anova_table$_{i,j}$ = anova_table[i*6+j]
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The values for the mean squares, F-statistic and *p*-value are set to NaN for the residual and corrected total effects.

The Source Identifiers in the first column of `anova_table`$_{i,j}$ are the only negative values in `anova_table`. The absolute value of the source identifier is equal to the order of the effect in that row. Main effects, for example, have a source identifier of –1. Two-way interactions use a source identifier of –2, and so on.

Source Identifier	ANOVA Source
-1	Main Efects †
-2	Two-Way Interactions ‡
-3	Three-Way Interactions ‡
.	.
.	.
.	.
`-n_factors`	(`n_factors`)-way Interactions ‡
`-n_factors-1`	Effects Error Term
`-n_factors-2`	Residual ⥣
`-n_factors-3`	Corrected Total

Notes: By default, `model_order` = `n_factors` when treatments are replicated, or `n_locations` >1. However, if treatments are not replicated and `n_locations` =1, `model_order` = `n_factors` -1.

† The number of main effects is equal to `n_factors+1` if `n_locations` >1, and `n_factors` if `n_locations` =1. The first row of values, `anova_table[0]` through `anova_table[5]` contain the location effect if `n_locations` >1. If `n_locations=1`, then these values are the effects for factor 1.

⇑ The residual term is only provided when treatments are replicated, i.e., `n_levels[n_factors]`>1.

‡ The number of interaction effects for the *n*th-way interactions is equal to

$$\binom{\texttt{n\_factors}}{\texttt{n\_way}}.$$

The order of these terms is in ascending order by treatment subscript. The interactions for factor 1 appear first, followed by factor 2, factor 3, and so on.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float * imsls_f_crd_factorial (int n_obs, int n_locations,
        int n_factors, int n_levels[], int model[], float y[],
        IMSLS_RETURN_USER, float anova_table[]
        IMSLS_N_MISSING, int *n_missing,
        IMSLS_CV, float *cv,
        IMSLS_GRAND_MEAN, float *grand_mean,
        IMSLS_FACTOR_MEANS, float **factor_means,
        IMSLS_FACTOR_MEANS_USER, float factor_means[],
        IMSLS_FACTOR_STD_ERRORS, float **factor_std_err,
        IMSLS_FACTOR_STD_ERRORS_USER,
                float factor_std_err[],
        IMSLS_TWO_WAY_MEANS,
                float **two_way_means,
        IMSLS_TWO_WAY_MEANS_USER,
                float two_way_means[],
        IMSLS_TWO_WAY_STD_ERRORS, float **two_way_std_err,
        IMSLS_TWO_WAY_STD_ERRORS_USER, float two_way_std_err[],
        IMSLS_TREATMENT_MEANS, float **treatment_means,
        IMSLS_TREATMENT_MEANS_USER, float treatment_means[],
        IMSLS_TREATMENT_STD_ERROR, float **treatment_std_err,
        IMSLS_TREATMENT_STD_ERROR_USER,
                float treatment_std_err[],
        IMSLS_ANOVA_ROW_LABELS, char ***anova_row_labels
        IMSLS_ANOVA_ROW_LABELS_USER, char *anova_row_labels[], 0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined `n_anova` by 6 array for the `anova_table`.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `*cv` (Output)
Coefficient of Variation computed by:

$$CV = \frac{100 \cdot \sqrt{MS_{residual}}}{\texttt{grand\_mean}}$$

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
Mean of all the data across every location.

`IMSLS_FACTOR_MEANS`, *float* `**factor_means` (Output)
Address of a pointer to an internally allocated array of length `n_levels[0]+n_levels[1]+…+n_levels[n_factors-1]` containing the factor means.

`IMSLS_FACTOR_MEANS_USER`, *float* `factor_means[]` (Output)
Storage for the array `factor_means`, provided by the user.

`IMSLS_FACTOR_STD_ERRORS`, *float* `**factor_std_err` (Output)
Address of a pointer to an internally allocated `n_factors` by 2 array containing factor standard errors and their associated degrees of freedom. The first column contains the standard errors for comparing two factor means and the second its associated degrees of freedom.

`IMSLS_FACTOR_STD_ERRORS_USER`, *float* `factor_std_err[]` (Output)
Storage for the array `factor_std_err`, provided by the user.

`IMSLS_TWO_WAY_MEANS`, *float* `**two_way_means` (Output)
Address of a pointer to an internally allocated one-dimensional array containing the two-way means for all two by two combinations of the factors. The total length of this array when `n_factors > 1` is equal to:

$$\sum_{i=0}^{f} \sum_{j=i+1}^{f+1} \texttt{n\_levels}[i] \times \texttt{n\_levels}[j], \text{ where } f = \texttt{n\_factors}\text{-2}$$

If `n_factors = 1`, `NULL` is returned. If `n_factors>1`, the means would first be produced for all combinations of the first two factors followed by all combinations of the remaining factors using the subscript order suggested by the above formula. For example, if the experiment is a 2x2x2 factorial, the 12 two-way means would appear in the following order: A_1B_1, A_1B_2, A_2B_1, A_2B_2, A_1C_1, A_1C_2, A_2C_1, A_2C_2, B_1C_1, B_1C_2, B_2C_1, and B_2C_2.

IMSLS_TWO_WAY_MEANS_USER, *float* two_way_means[] (Output)
Storage for the array two_way_means, provided by the user.

IMSLS_TWO_WAY_STD_ERRORS, *float* **two_way_std_err (Output)
Address of a pointer to an internally allocated n_two_way by 2 array containing factor standard errors and their associated degrees of freedom., where

$$\texttt{n\_two\_way} = \binom{\texttt{n\_factors}}{2}$$

The first column contains the standard errors for comparing two 2-way interaction means and the second its associated degrees of freedom. The ordering of the rows in this array is similar to that used in IMSLS TWO_WAY_MEANS. For example if n_factors=4, then n_two_way =6 with the order AB, AC, AD, BC, BD, CD.

IMSLS_TWO_WAY_STD_ERRORS_USER, *float* two_way_std_err[] (Output)
Storage for the array two_way_std_err, provided by the user.

IMSLS_TREATMENT_MEANS, *float* **treatment_means (Output)
Address of a pointer to an internally allocated array of size

$$\texttt{n\_levels}[0]\times\texttt{n\_levels}[1]\times\cdots\times\texttt{n\_levels}[\texttt{n\_factors}-1]$$

containing the treatment means. The order of the means is organized in ascending order by the value of the factor identifier. For example, if the experiment is a 2x2x2 factorial, the 8 means would appear in the following order: $A_1B_1C_1$, $A_1B_1C_2$, $A_1B_2C_1$, $A_1B_1C_2$, $A_2B_1C_1$, $A_2B_1C_2$, $A_2B_2C_1$, and $A_2B_2C_2$.

IMSLS_TREATMENT_MEANS_USER, *float* treatment_means[] (Output)
Storage for the array treatment_means, provided by the user.

IMSLS_TREATMENT_STD_ERROR, *float* **treatment_std_err (Output)
The array of length 2 containing standard error for comparing treatments based upon the average number of replicates per treatment and its associated degrees of freedom.

IMSLS_TREATMENT_STD_ERROR_USER, *float* treatment_std_err[] (Output)
Storage for the array treatment_std_err, provided by the user.

IMSLS_ANOVA_ROW_LABELS, *char* ***anova_row_labels (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the n_anova rows of the returned ANOVA table. The label for the *i*-th row of the ANOVA table can be printed with printf("%s", anova_row_labels[i]);

The memory associated with anova_row_labels can be freed with a single call to free(anova_row_labels).

IMSLS_ANOVA_ROW_LABELS_USER, *char* *anova_row_labels[] (Output)
Storage for the anova_row_labels, provided by the user. The amount

of space required will vary depending upon the number of factors and n_anova. An upperbound on the required memory is char *anova_row_labels[n_anova* 60].

Description

The function imsls_f_crd_factorial analyzes factorial experiments replicated in different locations. Unequal replication for each treatment and missing observations are allowed. All factors are regarded as fixed effects in the analysis. However, if multiple locations appear in the data, i.e., n_locations > 1, then all effects involving locations are treated as random effects.

If n_locations = 1, then the residual mean square is used as the error mean square in calculating the F-tests for all other effects. That is

$$F = \frac{\text{MS}_{effect}}{\text{MS}_{residual}}, \text{ when n\_locations} = 1.$$

If n_locations > 1 then the error mean squares for all factor F-tests is the pooled location interaction. For example, if n_factors = 2 then the error sum of squares, degrees of freedom and mean squares are calculated by:

$$\text{SS}_{error} = \text{SS}_{A\times Locations} + \text{SS}_{B\times Locations} + \text{SS}_{A\times B\times Locations}$$

$$\text{df}_{error} = \text{df}_{A\times Locations} + \text{df}_{B\times Locations} + \text{df}_{A\times B\times Locations}$$

$$\text{MS}_{error} = \frac{\text{SS}_{error}}{\text{df}_{error}}$$

Example

The following example is based upon data from a 3x2x2 completely randomized design conducted at one location. For demonstration purposes, observation 9 is set to missing.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "imsls.h"
void ex_crd_doc(){
    int n_obs       = 12;
    int n_locations = 1;
    int n_factors   = 3;
    int n_levels[4] ={3, 2, 2, 1};
    int page_width = 132;
    /*  model information */
    int model[]={
            1, 1, 1, 1,
        1, 1, 1, 2,
        1, 1, 2, 1,
        1, 1, 2, 2,
        1, 2, 1, 1,
        1, 2, 1, 2,
        1, 2, 2, 1,
        1, 2, 2, 2,
        1, 3, 1, 1,
        1, 3, 1, 2,
        1, 3, 2, 1,
        1, 3, 2, 2
};
/* response data */
float y[] ={
        4.42725419998168950,
        2.12795543670654300,
        2.55254390835762020,
        1.21479606628417970,
        2.47588264942169190,
        5.01306104660034180,
        4.73502767086029050,
        4.58392113447189330,
        5.01421167794615030,
        4.11972457170486450,
        6.51671624183654790,
        4.73365202546119690
};

int model_order;
int i, j, k, l, m, n_missing, i2, j2;
int n_factor_levels=0, n_treatments=1;
int n_two_way_means=0, n_two_way_std_err=0;
int n_two_way_interactions=0;
int n_subscripts, n_anova_table=2;
float cv, grand_mean;
float *anova_table;
float *two_way_means, *two_way_std_err;
float *treatment_means, *treatment_std_err;
float *factor_means;
float *factor_std_err;
float aNaN = imsls_f_machine(6);
char  **anova_row_labels;
char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ  ",
    "Mean  \nsquares", "\nF-Test", "\np-Value"};
/*
 * Compute the length of some of the output arrays.
 */
model_order = n_factors-1;
for (i=0; i < n_factors; i++){
    n_factor_levels = n_factor_levels + n_levels[i];
    n_treatments    = n_treatments*n_levels[i];
    for (j=i+1; j < n_factors; j++){
        n_two_way_interactions++;
    }
}
n_two_way_std_err = n_two_way_interactions;
for (i=0; i < n_factors-1; i++){
    for (j=i+1; j < n_factors; j++){
        n_two_way_means = n_two_way_means + n_levels[i]*n_levels[j];
    }
}
n_subscripts = n_factors;
n_anova_table = 2;
for (i=1; i <= model_order; i++){
    n_anova_table += (int)imsls_f_binomial_coefficient(n_subscripts, i);
}

/* Set observation 9 to missing. */
y[8] = aNaN;
anova_table = imsls_f_crd_factorial(n_obs, n_locations, n_factors,
                                    n_levels, model, y,
                                    IMSLS_N_MISSING, &n_missing,
                                    IMSLS_CV, &cv,
                                    IMSLS_GRAND_MEAN, &grand_mean,
                                    IMSLS_FACTOR_MEANS, &factor_means,
                                    IMSLS_FACTOR_STD_ERRORS,
                                     &factor_std_err,
                                    IMSLS_TWO_WAY_MEANS, &two_way_means,
                                    IMSLS_TWO_WAY_STD_ERRORS,
                                     &two_way_std_err,
                                    IMSLS_TREATMENT_MEANS, &treatment_means,
                                    IMSLS_TREATMENT_STD_ERROR,
                                     &treatment_std_err,
                                    IMSLS_ANOVA_ROW_LABELS,
                                     &anova_row_labels,
                                    0) ;
/* Output results. */
```

```
imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
/* Print ANOVA table. */
imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                     n_anova_table, 6, anova_table,
                     IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.3f%8.3f%8.3f%8.3f",
                     IMSLS_ROW_LABELS, anova_row_labels,
                     IMSLS_COL_LABELS, col_labels,
                     0);
printf("\n\nNumber of Missing Values Estimated: %d", n_missing);
printf("\nGrand Mean:                       %7.3f", grand_mean);
printf("\nCoefficient of Variation:         %7.3f", cv);

m=0;
/* Print Factor Means. */
printf("\n\nFactor Means\n");
for(i=0; i < n_factors; i++){
    printf("  Factor %d: ", i+1);
    for(j=0; j < n_levels[i]; j++){
        printf("  %f ", factor_means[m]);
        m++;
    }
    k = (int)factor_std_err[2*i+1];
    printf("\n              std. err.(df):        %f(%d) \n",
           factor_std_err[2*i], k);
}

/* Print Two-Way Means. */
printf("\n\nTwo-Way Means");
m = 0;
l=0;
for(i=0; i < n_factors-1; i++){
    for(j=i+1; j < n_factors; j++){
        printf("\n  Factor %d by Factor %d: \n", i+1, j+1);
        for(i2=0; i2 < n_levels[i]; i2++){
            for(j2=0; j2 < n_levels[j]; j2++){
                printf("  %f ",two_way_means[m]);
                m++;
            }
            printf("\n");
        }
        k = (int)two_way_std_err[l+1];
        printf("  std. err.(df): = %f(%d) \n", two_way_std_err[l], k);
        l+=2;
        }
    }

    /* Print Treatment Means. */
    printf("\n\nTreatment Means\n");
    m = 0;
    for(i=0; i < n_levels[0]; i++){
        for(j=0; j < n_levels[1]; j++){
            for(k=0; k < n_levels[2]; k++){
                printf("  Treatment[%d][%d][%d] Mean: %f \n",
                        i+1, j+1, k+1, treatment_means[m]);
                m++;
            }
        }
    }
    k = (int)treatment_std_err[1];
    printf("\n  Treatment Std. Err (df) %f(%d) \n",
           treatment_std_err[0], k);
}
```

Output

```
             *** ANALYSIS OF VARIANCE TABLE ***
                                  Mean
           ID   DF     SSQ      squares    F-Test   p-Value
[1]        -1    2    13.060     6.530     7.843     0.245
[2]        -1    1     0.107     0.107     0.129     0.780
[3]        -1    1     1.301     1.301     1.563     0.429
[1]x[2]    -2    2     3.768     1.884     2.263     0.425
[1]x[3]    -2    2     5.253     2.626     3.154     0.370
[2]x[3]    -2    1     0.560     0.560     0.672     0.563
Residual   -4    1     1.665     1.665  ........  ........
Total      -5   10    25.715  ........  ........  ........


Number of Missing Values Estimated: 1
Grand Mean:                          3.961
Coefficient of Variation:           32.574

Factor Means
  Factor 1:   2.580637   4.201973   5.101885
```

```
             std. err.(df):        0.912459(1)
  Factor 2:   3.866888   4.056109
             std. err.(df):        0.745020(1)
  Factor 3:   4.290812   3.632185
             std. err.(df):        0.745020(1)


Two-Way Means
  Factor 1 by Factor 2:
  3.277605   1.883670
  3.744472   4.659474
  4.578587   5.625184
  std. err.(df): = 1.290412(1)

  Factor 1 by Factor 3:
  3.489899   1.671376
  3.605455   4.798491
  5.777082   4.426688
  std. err.(df): = 1.290412(1)

  Factor 2 by Factor 3:
  3.980195   3.753580
  4.601429   3.510790
  std. err.(df): = 1.053617(1)


Treatment Means
  Treatment[1][1][1] Mean: 4.427254
  Treatment[1][1][2] Mean: 2.127955
  Treatment[1][2][1] Mean: 2.552544
  Treatment[1][2][2] Mean: 1.214796
  Treatment[2][1][1] Mean: 2.475883
  Treatment[2][1][2] Mean: 5.013061
  Treatment[2][2][1] Mean: 4.735028
  Treatment[2][2][2] Mean: 4.583921
  Treatment[3][1][1] Mean: 5.037448
  Treatment[3][1][2] Mean: 4.119725
  Treatment[3][2][1] Mean: 6.516716
  Treatment[3][2][2] Mean: 4.733652

  Treatment Std. Err (df) 1.824919(1)
```

rcbd_factorial

Analyzes data from balanced and unbalanced randomized complete-block experiments. Unlike `anova_factorial`, function `rcbd_factorial` allows for missing data, unequal replication and one or more locations.

Synopsis

#include `<imsls.h>`

float `* imsls_f_rcbd_factorial` (*int* `n_obs`, *int* `n_locations`, *int* `n_factors`, *int* `n_levels[]`,*int* `model[]`, *float* `y[]`,…, `0)`

The type *double* function is `imsls_d_rcbd_factorial`.

Required Arguments

int `n_obs` (Input)
Number of missing and non-missing experimental observations.

int `n_locations` (Input)
Number of locations. `n_locations` must be one or greater.

int `n_factors` (Input)
Number of factors in the model.

int `n_levels[]` (Input)
Array of length `n_factors`+1. The `n_levels[`0`]` through `n_levels[n_factors-`1`]` contain the number of levels for each factor. The last element, `n_levels[n_factors]`, contains the number of blocks at a location. There must be at least two blocks and two levels for each factor, i.e., `n_levels`[*i*] >2 for *i*=0, 1, …, `n_factors`.

int `model[]` (Input)
A `n_obs` by (`n_factors`+2) array identifying the location, block and factor levels associated with each observation in `y`. The first column must contain the location identifier and the second column must contain the block identifier for the observation associated with that row. The remaining columns, columns 3 through `n_factors`+2, should contain the factor level identifiers in the same order used in `n_levels`. If `n_locations` =1, the first column is still required, but its contents are ignored.

float `y[]` (Input)
An array of length `n_obs` containing the experimental observations and any missing values. Missing values are indicated by placing a NaN (not a number) in `y`. The `NaN` value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively.

Return Value

A pointer to the memory location of a two dimensional, `n_anova` by 6 array containing the ANOVA table, where:

$$\texttt{n\_anova} = a + \sum_{i=1}^{m} \binom{\texttt{n\_factors}}{i},$$

$$a = \begin{cases} 3 & \text{if } \texttt{n\_locations = 1} \\ 5 & \text{if } \texttt{n\_locations > 1} \end{cases},$$

and m= `model_order` = `n_factors` –1.

Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table[i*6]`, is the source identifier which identifies the type of effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

j	$\texttt{anova\_table}_{i,j}$ **= anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The values for the mean squares, F-statistic and *p*-value are set to NaN for the residual and corrected total effects.

The Source Identifiers in the first column of $\texttt{anova\_table}_{i,j}$ are the only negative values in `anova_table[]`. The absolute value of the source identifier is equal to the order of the effect in that row. Main effects, for example, have a source identifier of –1. Two-way interactions use a source identifier of –2, –3 and so on.

Source Identifier	**ANOVA Source**
-1	Main Effects †
-2	Two-Way Interactions ‡
-3	Three-Way Interactions ‡
.	.
.	.

Source Identifier	ANOVA Source
.	.
-n_factors	(n_factors)-way Interactions ‡
-n_factors-1	Error Term for Factors and Interactions
-n_factors-2	Residual *
-n_factors-3	Corrected Total

Notes: The Effects Error Term is equal to the Residual effect if n_locations = 1.

† The number of main effects is equal to n_factors+2 if n_locations > 1, and n_factors +1 if n_locations = 1. The first two rows, anova_table[0] through anova_table[10] are used to represent the location and block effects if n_locations > 1. If n_locations=1, then anova_table[0] through anova_table[5] contain the block effects.

‡ The number of interaction effects for the *n*th-way interactions is equal to

$$\binom{\texttt{n\_factors}}{\texttt{n\_way}}.$$

The order of these terms is in ascending order by treatment subscript. The interactions for factor 1 appear first, followed by factor 2, factor 3, and so on.

* The residual term is only produced when there is replication within blocks.

Synopsis with Optional Arguments

#include <imsls.h>

```
float * imsls_f_rcbd_factorial (int n_obs, int n_locations,
            int n_factors, int n_levels[], int model[],float y[],
            IMSLS_RETURN_USER, float anova_table[],
            IMSLS_N_MISSING, int *n_missing,
            IMSLS_CV, float *cv,
            IMSLS_GRAND_MEAN, float *grand_mean,
            IMSLS_FACTOR_MEANS, float **factor_means,
            IMSLS_FACTOR_MEANS_USER, float factor_means[],
            IMSLS_FACTOR_STD_ERRORS, float **factor_std_err,
            IMSLS_FACTOR_STD_ERRORS_USER, float factor_std_err[],
            IMSLS_TWO_WAY_MEANS, float **two_way_means,
            IMSLS_TWO_WAY_MEANS_USER, float two_way_means[],
            IMSLS_TWO_WAY_STD_ERRORS, float **two_way_std_err,
            IMSLS_TWO_WAY_STD_ERRORS_USER,
                  float two_way_std_err[],
```

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means`,
`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]`,
`IMSLS_TREATMENT_STD_ERROR`, **float* `treatment_std_err`,
`IMSLS_TREATMENT_STD_ERROR_USER`,
float `treatment_std_err[]`
`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels`,
`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined `n_anova` by 6 array for the anova_table.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `*cv` (Output)
Coefficient of Variation computed by:

$$CV = \frac{100 \cdot \sqrt{MS_{residual}}}{\texttt{grand\_mean}}.$$

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
Mean of all the data across every location.

`IMSLS_FACTOR_MEANS`, *float* `**factor_means` (Output)
Address of a pointer to an internally allocated array of length `n_levels[0]+n_levels[1]+…+n_levels[n_factors-1]` containing the factor means.

`IMSLS_FACTOR_MEANS_USER`, *float* `factor_means[]` (Output)
Storage for the array `factor_means`, provided by the user.

`IMSLS_FACTOR_STD_ERRORS`, *float* `**factor_std_err` (Output)
Address of a pointer to an internally allocated `n_factors` by 2 array containing factor standard errors and their associated degrees of freedom. The first column contains the standard errors for comparing two factor means and the second its associated degrees of freedom

`IMSLS_FACTOR_STD_ERRORS_USER`, *float* `factor_std_err[]` (Output)
Storage for the array `factor_std_err`, provided by the user.

`IMSLS_TWO_WAY_MEANS`, *float* `**two_way_means` (Output)
Address of a pointer to an internally allocated one-dimensional array containing the two-way means for all two by two combinations of the factors. The total length of this array when `n_factors` >1 is equal to:

$$\sum_{i=0}^{f} \sum_{j=i+1}^{f+1} \texttt{n\_levels}[i] \times \texttt{n\_levels}[j],$$

where

$$f = \texttt{n\_factors} - 2$$

If `n_factors` = 1, `NULL` is returned. If `n_factors`>1, the means would first be produced for all combinations of the first two factors followed by all combinations of the remaining factors using the subscript order suggested by the above formula. For example, if the experiment is a 2x2x2 factorial, the 12 two-way means would appear in the following order: A_1B_1, A_1B_2, A_2B_1, A_2B_2, A_1C_1, A_1C_2, A_2C_1, A_2C_2, B_1C_1, B_1C_2, B_2C_1, and B_2C_2.

`IMSLS_TWO_WAY_MEANS_USER`, *float* `two_way_means[]` (Output)
Storage for the array `two_way_means`, provided by the user.

`IMSLS_TWO_WAY_STD_ERRORS`, *float* `**two_way_std_err` (Output)
Address of a pointer to an internally allocated `n_two_way` by 2 array containing factor standard errors and their associated degrees of freedom., where

$$\texttt{n\_two\_way} = \binom{\texttt{n\_factors}}{2}$$

The first column contains the standard errors for comparing two 2-way interaction means and the second its associated degrees of freedom. The ordering of the rows in this array is similar to that used in `IMSLS_TWO_WAY_MEANS`. For example if `n_factors=4`, then `n_two_way` = `6` with the order AB, AC, AD, BC, BD, CD.

`IMSLS_TWO_WAY_STD_ERRORS_USER`, *float* `two_way_std_err[]` (Output)
Storage for the array `two_way_std_err`, provided by the user.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size

$$\texttt{n\_levels}[0] \times \texttt{n\_levels}[1] \times \cdots \times \texttt{n\_levels}[\texttt{n\_factors} - 1]$$

containing the treatment means. The order of the means is organized in ascending order by the value of the factor identifier. For example, if the experiment is a 2x2x2 factorial, the 8 means would appear in the following order: $A_1B_1C_1$, $A_1B_1C_2$, $A_1B_2C_1$, $A_1B_1C_2$, $A_2B_1C_1$, $A_2B_1C_2$, $A_2B_2C_1$, and $A_2B_2C_2$.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array `treatment_means`, provided by the user.

`IMSLS_TREATMENT_STD_ERROR`, *float* `*treatment_std_err` (Output)
The array of length 2 containing standard error for comparing treatments

based upon the average number of replicates per treatment and its associated degrees of freedom.

`IMSLS_TREATMENT_STD_ERROR_USER`, *float* `treatment_std_err[]` (Output)
Storage for the array `treatment_std_err`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the *i*th row of the ANOVA table can be printed with `printf("%s", anova_row_labels[i])`.

The memory associated with `anova_row_labels` can be freed with a single call to `free(anova_row_labels)`.

`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels`[100*(`n_anova`+1)].

Description

The function `imsls_f_rcbd_factorial` is capable of analyzing randomized complete block factorial experiments replicated in different locations. Missing observations are estimated using the Yates method. Locations, if used, and blocks are treated as random factors. All treatment factors are regarded as fixed effects in the analysis. If `n_locations` > 1, then blocks are treated as nested within locations and the number of blocks used at each location must be the same.

If `n_locations` = 1, then the residual mean square is used as the error mean square in calculating the F-tests for all other effects. That is

$$F_{effect} = \frac{MS_{effect}}{MS_{residual}}, \text{ when } \texttt{n\_locations} = 1.$$

In this case, the residual mean square is calculating by pooling all interactions between treatments and blocks. For example, if treatments are formed from two factors, A and B, then

$$SS_{residual} = SS_{A\times Blocks} + SS_{B\times Blocks} + SS_{A\times B\times Blocks}$$
$$df_{residual} = df_{A\times Blocks} + df_{B\times Blocks} + df_{A\times B\times Blocks}$$
$$MS_{residual} = \frac{SS_{residual}}{df_{residual}}$$

When `n_locations` = 1, then $MS_{residual}$ is also used to calculate the standard errors between means. For example, in a two factor experiment:

$$\text{Std Err(A)} = \sqrt{\frac{2 \cdot MS_{residual}}{N_A}}$$

$$\text{Std Err(B)} = \sqrt{\frac{2 \cdot MS_{residual}}{N_B}},$$

$$\text{Std Err(A} \times \text{B)} = \sqrt{\frac{2 \cdot MS_{residual}}{N_{A \times B}}}$$

where

$$N_A,\ N_B \text{ and } N_{A \times B}$$

are the number of observations for each level of the effects A, B and their interaction, respectively.

If `n_locations` > 1, then the error mean square is used as the denominator of the F-test for effects:

$$F_{effect} = \frac{MS_{effect}}{MS_{error}}.$$

The error mean square in this calculation is obtained by pooling all interactions between each factor and locations. For example `n_locations` > 1 and `n_factors`=2 then:

$$SS_{error} = SS_{A \times Locations} + SS_{B \times Locations} + SS_{A \times B \times Locations}$$
$$df_{error} = df_{A \times Locations} + df_{B \times Locations} + df_{A \times B \times Locations}$$
$$MS_{error} = \frac{SS_{error}}{df_{error}}$$

In this case, `n_locations` > 1, the standard errors for means are calculated using

$$MS_{error} \text{ instead of } MS_{residual}$$

The F-test for differences between locations is calculated using the mean squares for blocks within locations:

$$F_{locations} = \frac{MS_{locations}}{MS_{blocks(location)}}$$

Example

This example is based upon data from an agricultural trial conducted by DOW Agrosciences. This is a three factor, 3x2x2, experiment replicated in two blocks at one location. For illustration, two observations are set to NaN to simulate missing observations.

```
#include <stdio.h>
#include <math.h>
#include "imsls.h"

void main(){
    int n_obs       = 24;
    int n_locations = 1;
    int n_factors   = 3;
    int n_levels[4] ={3, 2, 2, 2};
    int model[]={
            1, 1, 1, 1, 1,
            1, 2, 1, 1, 1,
            1, 1, 1, 1, 2,
            1, 2, 1, 1, 2,
            1, 1, 1, 2, 1,
            1, 2, 1, 2, 1,
            1, 1, 1, 2, 2,
            1, 2, 1, 2, 2,
            1, 1, 2, 1, 1,
            1, 2, 2, 1, 1,
            1, 1, 2, 1, 2,
            1, 2, 2, 1, 2,
            1, 1, 2, 2, 1,
            1, 2, 2, 2, 1,
            1, 1, 2, 2, 2,
            1, 2, 2, 2, 2,
            1, 1, 3, 1, 1,
            1, 2, 3, 1, 1,
            1, 1, 3, 1, 2,
            1, 2, 3, 1, 2,
            1, 1, 3, 2, 1,
            1, 2, 3, 2, 1,
            1, 1, 3, 2, 2,
            1, 2, 3, 2, 2
    };
    float y[] ={
            4.42725419998168950, 2.98526261840015650,
            2.12795543670654300, 4.36357164382934570,
          2.55254390835762020, 2.78596709668636320,
          1.21479606628417970, 2.68143519759178160,
          2.47588264942169190, 4.69543695449829100,
          5.01306104660034180, 3.01919978857040410,
          4.73502767086029050, 0.00000000000000000,
          0.00000000000000000, 5.05780076980590820,
          5.01421167794615030, 3.61517095565795900,
          4.11972457170486450, 4.71947982907295230,
          6.51671624183654790, 4.22036057710647580,
          4.73365202546119690, 4.68545144796371460
  };

 int page_width = 132;
 int model_order;
 int i, n_subscripts, n_anova_table;
 char **aov_labels;
 char *col_labels[] = {" ", "ID", "df", "SS",
                       "MS", "F-Test", "P-Value"};
 float *anova_table;

 /* Compute number of rows in the anova table. */
 model_order = n_subscripts = n_factors;
 n_anova_table = 3;
 for (i=1; i <= model_order; i++){
     n_anova_table += imsls_d_binomial_coefficient(n_subscripts, i);
 }

 /* Set missing observations. */
 y[13] = imsls_d_machine(6);
 y[14] = imsls_d_machine(6);

 anova_table = imsls_f_rcbd_factorial(n_obs, n_locations, n_factors,
                                      n_levels, model, y,
                                      IMSLS_ANOVA_ROW_LABELS, &aov_labels,
                                      0) ;
 imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
 /*
  * Print ANOVA table.
  */
imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                     10, 6, anova_table,
                     IMSLS_ROW_LABELS, aov_labels,
                     IMSLS_COL_LABELS, col_labels,
                              IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                              0);
}
```

Output

```
              *** ANALYSIS OF VARIANCE TABLE ***
              ID   df        SS       MS   F-Test  P-Value
Blocks        -1    1      0.01     0.01  .......  .......
[1]           -1    2     14.73     7.37     5.15    0.032
[2]           -1    1      0.24     0.24     0.17    0.692
[3]           -1    1      0.15     0.15     0.10    0.756
[1]x[2]       -2    2      5.79     2.89     2.02    0.188
[1]x[3]       -2    2      1.02     0.51     0.36    0.709
[2]x[3]       -2    1      0.20     0.20     0.14    0.719
[1]x[2]x[3]   -3    2      0.13     0.07     0.05    0.956
Error         -4    9     12.88     1.43  .......  .......
Total         -6   21     35.15  .......  .......  .......
```

latin_square

Analyzes data from latin-square experiments. Function `latin_square` also analyzes latin-square experiments replicated at several locations.

Synopsis

#include `<imsls.h>`

float `* imsls_f_latin_square` (*int* `n`, *int* `n_locations`,
    int `n_treatments`, *int* `row[]`, *int* `col[]`, *int* `treatment[]`,
    float `y[]`, …, `0`)

The type *double* function is `imsls_d_latin_square`.

Required Arguments

int `n` (Input)
    Number of missing and non-missing experimental observations. `imsls_f_latin_square` verifies that:

$$n = \texttt{n\_locations} \cdot \texttt{n\_treatments}^2$$

hint `n_locations` (Input)
    Number of locations. `n_locations` must be one or greater. If `n_locations`>1 then the optional array `locations[]` must be included as input to `imsls_f_latin_square`.

int `n_treatments` (Input)

Number of treatments. `n_treatments` must be greater than one. In addition the number of rows and columns must be equal to `n_treatments`.

int `row[]` (Input)

An array of length `n` containing the row identifiers for each observation in `y`. Each row must be assigned values from 1 to `n_treatments`. `imsls_f_latin_square` verifies that the number of unique factor A identifiers is equal to `n_treatments`.

int `col[]` (Input)

An array of length `n` containing the column identifiers for each observation in `y`. Each column must be assigned values from 1 to `n_treatments`. `imsls_f_latin_square` verifies that the number of unique column identifiers is equal to `n_treatments`.

int `treatment[]` (Input)

An array of length `n` containing the treatment identifiers for each observation in `y`. Each treatment must be assigned values from 1 to `n_treatments`. `imsls_f_latin_square` verifies that the number of unique treatment identifiers is equal to `n_treatments`.

float `y[]` (Input)

An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`((6), depending upon whether single or double precision is being used, respectively. The location, row, column, and treatment number for each observation in `y` are identified by the corresponding values in the arguments `locations`, `row`, `col`, and `treatment`.

Return Value

Address of a pointer to the memory location of a two dimensional, 7 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table[`i`*6]`, identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

J	$\texttt{anova\_table}_{i,j}$ = **anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares

J	**anova_table$_{i,j}$ = anova_table[i*6+j]**
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of anova_table$_{i,j}$ are the only negative values in anova_table[]. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	**ANOVA Source**
-1	LOCATIONS †
-2	ROWS
-3	COLUMNS
-4	TREATMENTS
-5	LOCATIONS × TREATMENTS †
-6	ERROR WITHIN LOCATIONS
-7	CORRECTED TOTAL

Notes: † If n_locations=1 rows involving location are set to missing (NaN).

Synopsis with Optional Arguments

#include <imsl.h>

float * imsls_f_latin_square (*int* n, *int* n_locations, *int* n_treatments,
int row[], *int* col[], *int* treatment[], *float* y[],
IMSLS_RETURN_USER, *float* anova_table[],
IMSLS_LOCATIONS, *int* locations[],
IMSLS_N_MISSING, *int* *n_missing,
IMSLS_CV, *float* *cv,
IMSLS_GRAND_MEAN, *float* *grand_mean,
IMSLS_TREATMENT_MEANS, *float* **treatment_means,
IMSLS_TREATMENT_MEANS_USER, *float* treatment_means[],
IMSLS_STD_ERRORS, *float* **std_err,
IMSLS_STD_ERRORS_USER, *float* std_err[],
IMSLS_LOCATION_ANOVA_TABLE *float* **location_anova_table,
IMSLS_LOCATION_ANOVA_TABLE_USER,
float location_anova_table[],
IMSLS_ANOVA_ROW_LABELS, *char* ***anova_row_labels,
IMSLS_ANOVA_ROW_LABELS_USER, *char* *anova_row_labels[],
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined array of length 42 for storage of the 7 by 6 anova table described as the return argument for this routine. For a detailed description of the format for this table, see the previous description of the return arguments for `imsls_f_latin_square`.

`IMSLS_LOCATIONS`, *int* `locations[]` (Input)
An array of length `n` containing the location identifiers for each observation in `y`. Unique integers must be assigned to each location in the study. This argument is required when `n_locations`>1.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `*cv` (Output)
The coefficient of variation computed by using the within location standard deviation.

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
Mean of all the data across every location.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size `n_treatments` containing the treatment means.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array `treatment_means`, provided by the user.

`IMSLS_STD_ERRORS`, *float* **`std_err` (Output)
Address of a pointer to an internally allocated array of length 2 containing the standard error and associated degrees of freedom for comparing two treatment means. `std_err[0]` contains the standard error and its degrees of freedom are returned in `std_err[1]`.

`IMSLS_STD_ERRORS_USER`, *float* `std_err[]` (Output)
Storage for the array `std_err`, provided by the user.

`IMSLS_LOCATION_ANOVA_TABLE`, *float* `**location_anova_table` (Output)
Address of a pointer to an internally allocated 3-dimensional array of size `n_locations` by 7 by 6 containing the anova tables associated with each location. For each location, the 7 by 6 dimensional array corresponds to the anova table for that location. For example, `location_anova_table`[$(i-1)\times 42+(j-1)\times 6+(k-1)$] contains the value in the *k*th column and *j*th row of the anova-table for the *i*th location.

`IMSLS_LOCATION_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for the array `location_anova_table`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the *i*th row of the ANOVA table can be printed with `printf("%s", anova_row_labels[i])`.

The memory associated with `anova_row_labels` can be freed with a single call to `free(anova_row_labels)`.

`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels[600]`.

Description

The function `imsls_f_latin_square` analyzes latin-square experiments, possibly replicated at multiple locations. Latin-square experiments block treatments using two factors: rows and columns. The number of levels associated with rows and columns must equal the number of treatments. Treatments are blocked by rows and columns in a balanced arrangement to ensure that every row contain one replicate of every treatment. The same balance is required for every column, see Table 1. Notice that the four treatments, T1, T2, T3, and T4, appear exactly once in every column and every row.

		Columns			
		C1	**C2**	**C3**	**C4**
Rows	**R1**	T1	T2	T3	T4
	R2	T2	T3	T4	T1
	R3	T3	T4	T1	T2
	R4	T4	T1	T2	T3

Table 1 – A Latin-Square Experiment with Four Treatments

A necessary assumption in Latin-Square experiments is that there are no interactions between treatments and the row and column blocking factors. For data collected at a single location, the Anova table for a Latin-Square experiment is usually organized into five rows, see Table 2.

SOURCE	**DF**	**Sum of Squares**	**Mean Squares**
ROWS	$t-1$	SSR=$t\sum_{i=1}^{t}(\bar{y}_{i.}-\bar{y}_{..})^2$	MSR
COLUMNS	$t-1$	SSC=$t\sum_{j=1}^{t}(\bar{y}_{.j}-\bar{y}_{..})^2$	MSC
TREATMENTS	$t-1$	SST=$t\sum_{k=1}^{t}(\bar{y}_{k}-\bar{y}_{..})^2$	MST
ERROR	$(t-1)(t-2)$	SSE=SSTot-SSR-SSC-SST	MSE
TOTAL	t^2-1	SSTot=$\sum_{i=1}^{t}\sum_{j=1}^{t}\left(y_{ij}-\bar{y}_{..}\right)^2$	

Table 2 – The ANOVA Table for a Latin-Square Experiment at one Location

The statistical model used to represent data is from a single location:

$$y_{ij(k)} = \mu + \rho_i + \gamma_j + \tau_{k(ij)} + \varepsilon_{ij(k)},$$

where

$y_{ij(k)}$ is the observation for the *k*th treatment in the *i*th row and *j*th column of the Latin Square, and, $\tau_{k(ij)}$ is the effect associated with the *k*th treatment. ρ_i and γ_j are the *i*th row and *j*th column effects, respectively, and $\varepsilon_{ij(k)}$ is the noise associated with this observation.

If multiple locations are involved, `imsls_f_latin_square` assumes that treatments are crossed with locations, but that row and column effects are nested within locations, see Table 3. The statistical model used to represent these data is:

$$y_{lij(k)} = \mu + \alpha_l + \rho_{i(l)} + \gamma_{j(l)} + \tau_{k(ij)} + \alpha\tau_{lk(ij)} + \varepsilon_{lij(k)},$$

where

$$\tau_{k(ij)}$$

is the effect associated with the *k*th treatment, and

$$\alpha\tau_{lk(ij)}$$

is the interaction effect between location l and treatment *k*.

SOURCE	**DF**	**Sum of Squares**	**Mean Squares**
LOCATIONS	$r-1$	SSL=$t^2\sum_{l=1}^{r}(\bar{y}_{l..}-\bar{y}_{...})^2$	MSL
ROWS	$r(t-1)$	SSR=$t\sum_{l=1}^{r}\sum_{i=1}^{t}(\bar{y}_{li.}-\bar{y}_{l..})^2$	MSR
COLUMNS	$r(t-1)$	SSC=$t\sum_{l=1}^{r}\sum_{j=1}^{t}(\bar{y}_{l.j}-\bar{y}_{l..})^2$	MSC
TREATMENTS	$t-1$	SST=$r\cdot t\sum_{k=1}^{t}(\bar{y}_{k}-\bar{y}_{...})^2$	MST
LOCATIONS X TREATMENTS	$(r-1)(t-1)$	SSLT by difference	MSLT
ERROR	$(t-1)[r(t-1)-1]$	SSE=$\sum_{l=1}^{r}SSE_l$	MSE
TOTAL	$r\cdot t^2-1$	SSTot=$\sum_{l=1}^{r}\sum_{i=1}^{t}\sum_{j=1}^{t}\left(y_{lij}-\bar{y}_{..}\right)^2$	

Table 3 – The ANOVA Table for a Latin-Square Experiment at Multiple Locations

Example

This example uses 4 treatments organized into a latin square. This example also uses the function `l_print_LSD`(), which is defined in the first example for `imsls_f_lattice`() (page 297).

```
#include <stdio.h>
#include <math.h>
#include "imsls.h"

void l_print_LSD(int n1, int* equalMeans, float *means);

void main()
{
  char **anova_row_labels;
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ  ",
                        "Mean  \nsquares", "\nF-Test", "\np-Value"};
```

```
float alpha = 0.05;
int i, l, page_width = 132;

int n            = 16; /* Total number of observations */
int n_locations  = 1;  /* Number of locations */
int n_treatments = 4;  /* Number of rows, columns and treatments */
int n_aov_rows   = 7;  /* Number of rows in the latin-square anova table */

int col[]={1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4};
int row[]={3, 2, 4, 1, 1, 4, 2, 3, 2, 3, 1, 4, 4, 1, 3, 2};
int treatment[]={1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4};

float y[]={
       1.167,  1.185,  1.655, 1.345, 1.64, 1.29, 1.665, 1.29,
       1.475, 0.71, 1.425, 0.66, 1.565, 1.29, 1.4, 1.18};

float grand_mean;
float cv;
float *aov;
float *treatment_means;
float *std_err;
int    df;
int    *equal_means;

printf("\n\n*** Experimental Design ***");
printf("\n===============================");
printf("\n| COL  |  1  |  2  |  3  |  4  |");
printf("\n===============================");
printf("\n|ROW 1 |  2  |  4  |  3  |  1  |");
printf("\n===============================");
printf("\n|ROW 2 |  3  |  1  |  2  |  4  |");
printf("\n===============================");
printf("\n|ROW 3 |  1  |  3  |  4  |  2  |");
printf("\n===============================");
printf("\n|ROW 4 |  4  |  2  |  1  |  3  |");
printf("\n===============================");

aov = imsls_f_latin_square(n, n_locations, n_treatments, row, col,
                           treatment, y,
                           IMSLS_GRAND_MEAN, &grand_mean,
                           IMSLS_CV, &cv,
                           IMSLS_TREATMENT_MEANS, &treatment_means,
                           IMSLS_STD_ERRORS, &std_err,
                             IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                             0);
  /* Output results. */

  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
  /* Print ANOVA table. */
  imsls_f_write_matrix("\n   *** ANALYSIS OF VARIANCE TABLE ***",
                       7, 6, aov,
                       IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.3f%8.3f%8.3f%8.3f",
                       IMSLS_ROW_LABELS, anova_row_labels,
                       IMSLS_COL_LABELS, col_labels,
                       0);

  printf("\n\nGrand Mean:               %7.3f", grand_mean);
  printf("\n\nCoefficient of Variation: %7.3f\n\n", cv);
  l = 0;
  printf("Treatment Means: \n");
  for (i=0; i < n_treatments; i++){
        printf("treatment[%2d]              %7.4f \n", i+1,
treatment_means[l++]);
  }
  df = (int)std_err[1];
  printf("\n\nStandard Error for Comparing Two Treatment Means: %f \n(df=%d)\n",
      std_err[0], df);
  equal_means = imsls_f_multiple_comparisons(n_treatments, treatment_means, df,
                                             std_err[0]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_print_LSD(n_treatments, equal_means, treatment_means);
}
```

Output

```
*** Experimental Design ***
===============================
| COL  |  1  |  2  |  3  |  4  |
===============================
|ROW 1 |  2  |  4  |  3  |  1  |
===============================
|ROW 2 |  3  |  1  |  2  |  4  |
===============================
|ROW 3 |  1  |  3  |  4  |  2  |
================================
|ROW 4 |  4  |  2  |  1  |  3  |
================================

                       *** ANALYSIS OF VARIANCE TABLE ***
                                                  Mean
                              ID   DF     SSQ     squares    F-Test   p-Value
Locations ................    -1  ...  ........  ........  ........  ........
Rows within Locations .....   -2    3     0.185     0.062     2.064     0.207
Columns within Locations ..   -3    3     0.589     0.196     6.579     0.025
Treatments ................   -4    3     0.352     0.117     3.927     0.073
Locations x Treatments ....   -5  ...  ........  ........  ........  ........
Error within Locations ....   -6    6     0.179     0.030  ........  ........
Corrected Total ...........   -7   15     1.305  ........  ........  ........


Grand Mean:                 1.309

Coefficient of Variation:  13.204

Treatment Means:
treatment[ 1]               1.3380
treatment[ 2]               1.4712
treatment[ 3]               1.0675
treatment[ 4]               1.3587


Standard Error for Comparing Two Treatment Means: 0.122202
(df=6)
[group]          Mean         LSD Grouping
  [3]          1.067500         *
  [1]          1.338000         *       *
  [4]          1.358750         *       *
  [2]          1.471250                 *
```

lattice

Analyzes balanced and partially-balanced lattice experiments. In these experiments, a requirement is that the number of treatments be equal to the square of an integer, such as 9, 16, or 25 treatments. Function `lattice` also analyzes repetitions of lattice experiments.

Synopsis

#include `<imsls.h>`

float `*` `imsls_f_lattice` (*int* `n`, *int* `n_locations`, *int* `n_reps`,
int `n_blocks`, *int* `n_treatments`, *int* `rep[]`, *int* `block[]`,
int `treatment[]`, *float* `y[]`, …, `0`)

The type *double* function is `imsls_d_lattice`.

Required Arguments

int `n` (Input)
Number of missing and non-missing experimental observations. `imsls_f_balanced_lattice` verifies that:

$$\mathtt{n} = \mathtt{n\_locations} \times \mathtt{t} \times \mathtt{r} \quad \textit{where}$$

$$t = \mathtt{n\_treatments} \text{ and } r = \mathtt{n\_reps}.$$

int `n_locations` (Input)
Number of locations or repetitions of the lattice experiments. `n_locations` must be one or greater. If `n_locations`>1 then the optional arguments `IMSLS_LOCATIONS` must be included as input to `imsls_f_lattice`.

int `n_reps` (Input)
Number of replicates per location. Each replicate should consist of t = `n_treatments` organized into $k = \sqrt{t}$ blocks.

int `n_blocks` (Input)
Number of blocks per location. For every location, `n_blocks` must be equal to `n_blocks`= $r{\cdot}k$, where r = `n_reps` and $k = \sqrt{t}$.

int `n_treatments` (Input)
Number of treatments t = `n_treatments` must be equal to k^2.

int `rep[]` (Input)
An array of length `n` containing the replicate identifiers for each observation in `y`. For a balanced-lattice, the number of replicate identifiers must be equal to `n_reps`=(k+1). For a partially-balanced lattice, the number of replicate identifiers depends upon whether the design is a simple lattice, triple lattice, etc. `imsls_f_lattice` verifies that the number of unique replicate identifiers is equal to `n_reps`. If multiple locations or repetitions of the experiment is conducted, i.e., `n_locations`>1, then the replicate and block numbers contained in `rep` and `block` must agree between repetitions.

int `block[]` (Input)
An array of length `n` containing the block identifiers for each observation in `y`. `imsls_f_lattice` verifies that the number of unique block identifiers is equal to `n_blocks`. If multiple locations or

repetitions of the experiment is conducted, i.e., `n_locations`>1, then block numbers must agree between repetitions. That is, the *i*th block in every location or repetition must contain the same treatments.

int `treatment[]` (Input)
An array of length `n` containing the treatment identifiers for each observation in `y`. Each treatment must be assigned values from 1 to `n_treatments`. `imsls_f_lattice` verifies that the number of unique treatment identifiers is equal to `n_treatments`.

float `y[]` (Input)
An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine(6)` or `imsls_d_machine(6)`, depending upon whether single or double precision is being used, respectively. The location, replicate, block, and treatment number for each observation in `y` are identified by the corresponding values in the arguments `locations`, `rep`, `block`, and `treatment`.

Return Value

Address of a pointer to the memory location of a two dimensional, 7 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, `anova_table`$_{i,0}$ = `anova_table[i*6]`, identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

J	**anova_table**$_{i,j}$ **= anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of `anova_table`$_{i,j}$ are the only negative values in `anova_table[]`. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	ANOVA Source
-1	LOCATIONS †
-2	REPLICATES
-3	TREATMENTS(unadjusted)
-4	TREATMENTS(adjusted)
-5	BLOCKS(adjusted)
-6	INTRA-BLOCK ERROR
-7	CORRECTED TOTAL

Notes: † If `n_locations`=1, all entries in this row are set to missing (NaN).

Synopsis with Optional Arguments

#include `<imsl.h>`

```
float * imsls_f_lattice(int n, int n_locations, int n_reps,
        int n_blocks, int n_treatments, int rep[], int block[],
        int treatment[], float y[],
        IMSLS_RETURN_USER, float anova_table[]
        IMSLS_LOCATIONS, int locations[],
        IMSLS_N_MISSING, int *n_missing,
        IMSLS_CV, float *cv,
        IMSLS_GRAND_MEAN, float *grand_mean,
        IMSLS_TREATMENT_MEANS, float **treatment_means,
        IMSLS_TREATMENT_MEANS_USER, float treatment_means[],
        IMSLS_STD_ERRORS, float **std_err,
        IMSLS_STD_ERRORS_USER, float std_err[],
        IMSLS_LOCATION_ANOVA_TABLE float **location_anova_table,
        IMSLS_LOCATION_ANOVA_TABLE_USER,
                float location_anova_table[],
        IMSLS_ANOVA_ROW_LABELS, char ***anova_row_labels,
        IMSLS_ANOVA_ROW_LABELS_USER, char *anova_row_labels[],
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined array of length 42 for storage of the 7 by 6 anova table described as the return argument for `imsls_f_lattice`. For a detailed description of the format for this table, see the previous description of the return arguments for `imsls_d_lattice`.

`IMSLS_LOCATIONS`, *int* `locations[]` (Input)
An array of length `n` containing the location or repetition identifiers for each observation in `y`. Unique integers must be assigned to each location in the study. This argument is required when `n_locations`>1.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `*cv` (Output)
The coefficient of variation computed by using the location standard deviation.

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
The overall adjusted mean averaged over every location.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size `n_treatments` containing the adjusted treatment means.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array `treatment_means`, provided by the user.

`IMSLS_STD_ERRORS`, *float* `**std_err` (Output)
Address of a pointer to an internally allocated array of length 4 containing the standard error and associated degrees of freedom for comparing two treatment means. `std_err[0]` contains the standard error for comparing two treatments that appear in the same block at least once. `std_err[1]` contains the standard error for comparing two treatments that never appear in the same block together. `std_err[2]` contains the standard error for comparing, on average, two treatments from the experiment averaged over cases in which the treatments do or do not appear in the same block. Finally, `std_err[3]` contains the degrees of freedom associated with each of these standard errors, i.e., `std_err[3]`= degrees of freedom for intra-block error.

`IMSLS_STD_ERRORS_USER`, *float* `std_err[]` (Output)
Storage for the array `std_err`, provided by the user.

`IMSLS_LOCATION_ANOVA_TABLE`, *float* `**location_anova_table` (Output)
Address of a pointer to an internally allocated 3-dimensional array of size `n_locations` by 7 by 6 containing the anova tables associated with each location or repetition of the lattice experiment. For each location, the 7 by 6 dimensional array corresponds to the anova table for that location.
For example, `location_anova_table`$[(i\text{-}1)\times 42+(j\text{-}1)\times 6 + (k\text{-}1)]$ contains the value in the kth column and jth row of the anova-table for the ith location.

`IMSLS_LOCATION_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for the array `location_anova_table`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the ith row of the ANOVA table can be printed with `printf("%s", anova_row_labels[i]);`

The memory associated with `anova_row_labels` can be freed with a single call to `free(anova_row_labels)`.

`IMSLS_ANOVA_ROW_LABELS_USER`, *char* *`anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char` *`anova_row_labels`[600];

Description

The function `imsls_f_lattice` analyzes both balanced and partially-balanced lattice experiments, possibly repeated at multiple locations. These designs were originally described by Yates (1936). A defining characteristic of these classes of lattice experiments is that the number of treatments is always the square of an integer, such as t=9, 16, 25, etc. where t is equal to the number of treatments.

Another characteristic of lattice experiments is that blocks are organized into replicates, where each replicate contains one observation for each treatment. This requires the number of blocks in each replicate to be equal to the number of observations per block. That is, the number of blocks per replicate and the number of observations per block are both equal to $k = \sqrt{t}$.

For balanced lattice experiments the number of replicates is always $k+1$. For partially-balanced lattice experiments, the number of replicates is less than $k+1$. Tables of balanced-lattice experiments are tabulated in Cochran & Cox (1950) for t=9, 16, 25, 49, 64 and 81.

The analysis of balanced and partially-balanced experiments is detailed in Cochran & Cox (1950) and Kuehl (2000).

Consider, for example, a 3x3 balanced-lattice, i.e., k=3 and t=9. Notice that the number of replicates is 4 and the number of blocks per replicate is equal to 3. The total number of blocks is equal to

$$\texttt{n\_blocks} = \texttt{n\_locations} \cdot r \cdot (k-1) + 1 \ .$$

For a balanced-lattice,

$$\texttt{n\_blocks} = b = r \cdot k = (k+1) \cdot k = (\sqrt{t}+1) \cdot \sqrt{t} = 4 \cdot 3 = 12 \ .$$

Replicate I	**Replicate II**
Block 1 (T1, T2, T3)	Block 4 (T1, T4, T7)
Block 2 (T4, T5, T6)	Block 5 (T2, T5, T8)
Block 3 (T7, T8, T9)	Block 6 (T3, T6, T9)
Replicate III	**Replicate IV**
Block 7 (T1, T5, T9)	Block 10 (T1, T6, T8)
Block 8 (T2, T6, T7)	Block 11 (T2, T4, T9)
Block 9 (T3, T4, T8)	Block 12 (T3, T5, T7)

Table 1. A 3x3 Balanced-Lattice for 9 Treatments in Four Replicates.

The analysis of variance for data from a balanced-lattice experiment, takes the form familiar to other balanced incomplete block experiments. In these experiments, the error term is divided into two components: the Inter-Block Error and the Intra-Block Error. For single and multiple locations, the general format of the anova tables is illustrated in the Tables 2 and 3.

SOURCE	**DF**	**Sum of Squares**	**Mean Squares**
REPLICATES	$r-1$	SSR	MSR
TREATMENTS(unadj)	$t-1$	SST	MST
TREATMENTS(adj)	$t-1$	SSTa	MSTa
BLOCKS(adj)	$r\cdot(k-1)$	SSBa	MSBa
INTRA-BLOCK ERROR	$(k-1)(r\cdot k-k-1)$	SSI	MSI
TOTAL	$r\cdot t-1$		

Table 2 – The ANOVA Table for a Lattice Experiment at one Location

SOURCE	**DF**	**Sum of Squares**	**Mean Squares**
LOCATIONS	$p-1$	SSL	MSL
REPLICATES WITHIN LOCATIONS	$p(r-1)$	SSR	MSR
TREATMENTS(unadj)	$t-1$	SST	MST
TREATMENTS(adj)	$t-1$	SSTa	MSTa
BLOCKS(adj)	$p\cdot r(k-1)$	SSB	MSB
INTRA-BLOCK ERROR	$p\cdot(k-1)(r\cdot k-k-1)$	SSI	MSI
TOTAL	$p\cdot r\cdot t-1$		

Table 3 – The ANOVA Table for a Lattice Experiment at Multiple Locations

Example 1

This example is a lattice design for 16 treatments conducted at one location. A lattice design with t=k^2=16 treatments is a balanced lattice design with r= k+1=5 replicates and $r\cdot k$=5(4)=20 blocks.

```
#include <stdlib.h>
#include <math.h>
#include "imsls.h"

void l_print_LSD(int n1, int* equalMeans, float *means);

void main()
{
  char **anova_row_labels = NULL;
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ  ",
                       "Mean  \nsquares", "\nF-Test", "\np-Value"};
  float alpha = 0.05;
  int i, l, page_width = 132;
  int n            = 80; /* Total number of observations      */
  int n_locations  = 1;  /* Number of locations               */
  int n_treatments =16;  /* Number of treatments              */
  int n_reps       = 5;  /* Number of replicates              */
  int n_blocks     =20;  /* Total number of blocks            */
  int n_aov_rows   = 7;  /* Number of rows in the anova table */

  int rep[]={
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
      5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5
};

int block[]={
       1,  1,  1,  1,  2,  2,  2,  2,  3,  3,  3,  3,  4,  4,  4,  4,
       5,  5,  5,  5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  8,
       9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12,
      13, 13, 13, 13, 14, 14, 14, 14, 15, 15, 15, 15, 16, 16, 16, 16,
      17, 17, 17, 17, 18, 18, 18, 18, 19, 19, 19, 19, 20, 20, 20, 20
};

int treatment[]={
       1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       1,  5,  9, 13, 10,  2, 14,  6,  7, 15,  3, 11, 16,  8, 12,  4,
       1,  6, 11, 16,  5,  2, 15, 12,  9, 14,  3,  8, 13, 10,  7,  4,
       1, 14,  7, 12, 13,  2, 11,  8,  5, 10,  3, 16,  9,  6, 15,  4,
       1, 10, 15,  8,  9,  2,  7, 16, 13,  6,  3, 12,  5, 14, 11,  4
      };

float y[] ={
      147, 152, 167, 150, 127, 155, 162, 172,
      147, 100, 192, 177, 155, 195, 192, 205,
      140, 165, 182, 152,  97, 155, 192, 142,
      155, 182, 192, 192, 182, 207, 232, 162,
      155, 132, 177, 152, 182, 130, 177, 165,
      137, 185, 152, 152, 185, 122, 182, 192,
      220, 202, 175, 205, 205, 152, 180, 187,
      165, 150, 200, 160, 155, 177, 185, 172,
      147, 112, 177, 147, 180, 205, 190, 167,
      172, 212, 197, 192, 177, 220, 205, 225
};

float grand_mean;
float cv;
float *aov;
float *treatment_means;
float *std_err;
int   *equal_means;
int   df;
```

```
  aov = imsls_f_lattice(n, n_locations, n_reps, n_blocks,
                        n_treatments, rep, block, treatment, y,
                        IMSLS_GRAND_MEAN, &grand_mean,
                        IMSLS_CV, &cv,
                        IMSLS_TREATMENT_MEANS, &treatment_means,
                        IMSLS_STD_ERRORS, &std_err,
                        IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                        0);

  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
  /* Print the ANOVA table. */
  imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                        7, 6, aov,
                        IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                        IMSLS_ROW_LABELS, anova_row_labels,
                        IMSLS_COL_LABELS, col_labels,
                        0);

  printf("\n\nAdjusted Grand Mean:      %7.3f", grand_mean);
  printf("\n\nCoefficient of Variation: %7.3f\n\n", cv);
  l = 0;
  printf("Adjusted Treatment Means: \n");
  for (i=0; i < n_treatments; i++){
        printf("treatment[%2d]             %7.4f \n", i+1,
        treatment_means[l++]);
  }
  df = (int)std_err[3];
  printf("\nStandard Error for Comparing Two Adjusted Treatment Means: %f \n(df=%d)\n",
         std_err[2], df);
  equal_means = imsls_f_multiple_comparisons(n_treatments, treatment_means, df,
                                             std_err[2]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_print_LSD(n_treatments, equal_means, treatment_means);

}

/*
 * Function to display means comparison.
 */
void l_print_LSD(int n, int *equalMeans, float *means){
        float x=0.0;
        int i, j, k;
        int iSwitch;
        int *idx;

        idx = (int *) malloc(n * sizeof (int));

        for (k=0; k < n; k++) {
                idx[k]   =k+1;
        }

        /* Sort means in ascending order*/

        iSwitch=1;
        while (iSwitch != 0){
                iSwitch = 0;
                for (i = 0; i < n-1; i++){
                        if (means[i] > means[i+1]){
                                iSwitch = 1;
                                x = means[i];
                                means[i] = means[i+1];
                                means[i+1] = x;
                                j = idx[i];
                                idx[i] = idx[i+1];
                                idx[i+1] = j;
                        }
                }
        }
        printf("[group] \t  Mean \t\tLSD Grouping \n");
        for (i=0; i < n; i++){
                printf("  [%d] \t\t%f", idx[i], means[i]);
                for (j=1; j < i+1; j++){
                        if(equalMeans[j-1] >= i+2-j){
                                printf("\t  *");
                        }else{
                                if(equalMeans[j-1]>0) printf("\t");
                        }
                }
                if (i < n-1 && equalMeans[i]>0) printf("\t  *");
                printf("\n");
        }
        free(idx);
        idx = NULL;
        return;
}
```

Output

```
                  *** ANALYSIS OF VARIANCE TABLE ***
                                                 Mean
                              ID   DF     SSQ    squares   F-Test  p-Value
Locations ................    -1  ...  ........  .......  .......  .......
Replicates ...............    -2    4   6524.38  1631.10  .......  .......
Treatments (unadjusted) ...   -3   15  27297.13  1819.81     4.12    0.000
Treatments (adjusted) .....   -4   15  21271.29  1418.09     4.21    0.000
Blocks (adjusted) .........   -5   15  11339.28   755.95  .......  .......
Intra-Block Error .........   -6   45  15173.09   337.18  .......  .......
Corrected Total ...........   -7   79  60333.88  .......  .......  .......


Adjusted Grand Mean:      171.450

Coefficient of Variation:  10.710

Adjusted Treatment Means:
treatment[ 1]             166.4533
treatment[ 2]             160.7527
treatment[ 3]             183.6289
treatment[ 4]             175.6298
treatment[ 5]             162.6806
treatment[ 6]             167.6717
treatment[ 7]             168.3821
treatment[ 8]             176.5731
treatment[ 9]             162.6928
treatment[10]             118.5197
treatment[11]             189.0615
treatment[12]             190.4607
treatment[13]             169.4514
treatment[14]             197.0827
treatment[15]             185.3560
treatment[16]             168.8029

Standard Error for Comparing Two Adjusted Treatment Means: 13.221801
(df=45)
[group]           Mean          LSD Grouping
  [10]          118.519737
```

```
[2]           160.752731        *
[5]           162.680649        *       *
[9]           162.692841        *       *
[1]           166.453323        *       *       *
[6]           167.671661        *       *       *
[7]           168.382111        *       *       *
[16]          168.802887        *       *       *
[13]          169.451370        *       *       *
[4]           175.629776        *       *       *       *
[8]           176.573090        *       *       *       *
[3]           183.628906        *       *       *       *
[15]          185.355988        *       *       *       *
[11]          189.061508                *       *       *
[12]          190.460724                        *       *
[14]          197.082703                                *
```

Example 2

This example consists of a 5 × 5 partially-balanced lattice repeated twice. In this case, the number of replicates is not $k+1 = 6$, it is only `n_reps` = 2. Each lattice consists of total of 50 observations which is repeated twice. The first observation in this experiment is missing.

```
#include <stdlib.h>
#include <math.h>
#include "imsls.h"

void l_print_LSD(int n1, int* equalMeans, float *means);

void main()
{
  char **anova_row_labels = NULL;
  char **loc_row_labels   = NULL;
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ  ",
                        "Mean  \nsquares", "\nF-Test", "\np-Value"};
  float alpha = 0.05;
  int i, l, page_width = 132;

  int n = 100;           /* Total number of observations      */
  int n_locations  = 2;  /* Number of locations               */
  int n_treatments =25;  /* Number of treatments              */
  int n_reps       = 2;  /* Number of replicates/location     */
  int n_blocks     =10;  /* Total number of blocks/location   */
  int n_aov_rows   = 7;  /* Number of rows in the anova table */
int rep[]={
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      1, 1, 1, 1, 1,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2,
      2, 2, 2, 2, 2
};

int block[]={
       1,  1,  1,  1,  1,
       2,  2,  2,  2,  2,
       3,  3,  3,  3,  3,
       4,  4,  4,  4,  4,
       5,  5,  5,  5,  5,
       6,  6,  6,  6,  6,
       7,  7,  7,  7,  7,
       8,  8,  8,  8,  8,
       9,  9,  9,  9,  9,
      10, 10, 10, 10, 10,
       1,  1,  1,  1,  1,
       2,  2,  2,  2,  2,
       3,  3,  3,  3,  3,
       4,  4,  4,  4,  4,
       5,  5,  5,  5,  5,
       6,  6,  6,  6,  6,
       7,  7,  7,  7,  7,
       8,  8,  8,  8,  8,
       9,  9,  9,  9,  9,
      10, 10, 10, 10, 10
};

int treatment[]={
       1,  2,  3,  4,  5,
       6,  7,  8,  9, 10,
      11, 12, 13, 14, 15,
      16, 17, 18, 19, 20,
      21, 22, 23, 24, 25,
       1,  6, 11, 16, 21,
       2,  7, 12, 17, 22,
       3,  8, 13, 18, 23,
       4,  9, 14, 19, 24,
       5, 10, 15, 20, 25,
       1,  2,  3,  4,  5,
       6,  7,  8,  9, 10,
      11, 12, 13, 14, 15,
      16, 17, 18, 19, 20,
      21, 22, 23, 24, 25,
       1,  6, 11, 16, 21,
       2,  7, 12, 17, 22,
       3,  8, 13, 18, 23,
       4,  9, 14, 19, 24,
       5, 10, 15, 20, 25
      };
int location[]={
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2
};

float y[] ={
       6,  7,  5,  8,  6,
      16, 12, 12, 13,  8,
      17,  7,  7,  9, 14,
      18, 16, 13, 13, 14,
      14, 15, 11, 14, 14,
      24, 13, 24, 11,  8,
      21, 11, 14, 11, 23,
      16,  4, 12, 12, 12,
      17, 10, 30,  9, 23,
      15, 15, 22, 16, 19,
      13, 26,  9, 13, 11,
      15, 18, 22, 11, 15,
      19, 10, 10, 10, 16,
      21, 16, 17,  4, 17,
      15, 12, 13, 20,  8,
      16,  7, 20, 13, 21,
      15, 10, 11,  7, 14,
       7, 11, 15, 15, 16,
      19, 14, 20,  6, 16,
      17, 18, 20, 15, 14
};

float grand_mean;
float cv;
float *aov;
float *location_anova_table;
float *loc_anova_table;
float *treatment_means;
float *std_err;
int   df;
int   n_missing;
int   *equal_means;

/* Set first observation to missing. */
y[0] = imsls_f_machine(6);

aov = imsls_f_lattice(n, n_locations, n_reps, n_blocks,
                         n_treatments, rep, block, treatment, y,
                         IMSLS_LOCATIONS, location,
                         IMSLS_GRAND_MEAN, &grand_mean,
                         IMSLS_CV, &cv,
                         IMSLS_TREATMENT_MEANS, &treatment_means,
                         IMSLS_STD_ERRORS, &std_err,
                         IMSLS_LOCATION_ANOVA_TABLE, &location_anova_table,
                         IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                         IMSLS_N_MISSING, &n_missing,
                         0);
```

```
  /* Output results. */

  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
  /* Print the ANOVA table. */
  imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                      7, 6, aov,
                      IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                      IMSLS_ROW_LABELS, anova_row_labels,
                      IMSLS_COL_LABELS, col_labels,
                      0);

  /* Print the location ANOVA tables. */
  for (i=0; i < n_locations; i++){
      printf("\n\n\t\t\t\tLOCATION %d", i+1);
      imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                          7, 6, &(location_anova_table[i*42]),
                          IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                          IMSLS_ROW_LABELS, anova_row_labels,
                          IMSLS_COL_LABELS, col_labels,
                          0);
  }

  printf("\n\nAdjusted Grand Mean:      %7.3f", grand_mean);
  printf("\n\nCoefficient of Variation: %7.3f\n\n", cv);
  l = 0;
  printf("Adjusted Treatment Means: \n");
  for (i=0; i < n_treatments; i++){
        printf("treatment[%2d]              %7.4f \n", i+1,
treatment_means[l++]);
  }
  df = std_err[3];
  printf("\nStandard Error for Comparing Two Adjusted Treatment Means: %f \n(df=%d)\n",
        std_err[2], df);
  equal_means = imsls_f_multiple_comparisons(n_treatments, treatment_means, df,
                                             std_err[2]/sqrt(2),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_print_LSD(n_treatments, equal_means, treatment_means);

  printf("\n\nNumber of missing observations: %d\n", n_missing);

}
```

Output

```
                  *** ANALYSIS OF VARIANCE TABLE ***
                                                 Mean
                              ID   DF     SSQ    squares   F-Test  p-Value
Locations ................   -1    1     12.19    12.19     0.25    0.622
Replicates within Locations  -2    2    203.99   101.99     7.44    0.001
Treatments (unadjusted) ...  -3   24    795.46    33.14     0.02    1.000
Treatments (adjusted) .....  -4   24    951.20    39.63     2.89    0.006
Blocks (adjusted) .........  -5   16    770.50    48.16     3.51    0.000
Intra-Block Error .........  -6   55    753.81    13.71  .......  .......
Corrected Total ...........  -7   98   2535.95  .......  .......  .......


                                LOCATION 1
                *** ANALYSIS OF VARIANCE TABLE ***
                                                 Mean
                              ID   DF     SSQ    squares   F-Test  p-Value
Locations ................   -1  ...  ........  .......  .......  .......
Replicates within Locations  -2    1    203.67   203.67  .......  .......
Treatments (unadjusted) ...  -3   24    567.13    23.63     0.78    0.721
Treatments (adjusted) .....  -4   24    661.08    27.54     2.04    0.078
Blocks (adjusted) .........  -5    8    490.51    61.31  .......  .......
Intra-Block Error .........  -6   15    202.93    13.53  .......  .......
Corrected Total ...........  -7   48   1464.24  .......  .......  .......


                                LOCATION 2
                *** ANALYSIS OF VARIANCE TABLE ***
                                                 Mean
                              ID   DF     SSQ    squares   F-Test  p-Value
Locations ................   -1  ...  ........  .......  .......  .......
Replicates within Locations  -2    1      0.32     0.32  .......  .......
Treatments (unadjusted) ...  -3   24    622.52    25.94     1.43    0.196
Treatments (adjusted) .....  -4   24    707.51    29.48     2.83    0.018
Blocks (adjusted) .........  -5    8    269.76    33.72  .......  .......
Intra-Block Error .........  -6   16    166.92    10.43  .......  .......
Corrected Total ...........  -7   49   1059.52  .......  .......  .......


Adjusted Grand Mean:       14.011
```

```
Coefficient of Variation:  26.423

Adjusted Treatment Means:
treatment[ 1]             17.1507
treatment[ 2]             19.2200
treatment[ 3]             11.1261
treatment[ 4]             14.6230
treatment[ 5]             12.6543
treatment[ 6]             11.8133
treatment[ 7]             11.9045
treatment[ 8]             11.3106
treatment[ 9]              9.5576
treatment[10]             11.5889
treatment[11]             22.1321
treatment[12]             12.7233
treatment[13]             13.1293
treatment[14]             17.8763
treatment[15]             18.6576
treatment[16]             14.6568
treatment[17]             11.4980
treatment[18]             13.1540
treatment[19]              5.4010
treatment[20]             12.9323
treatment[21]             15.4108
treatment[22]             17.0020
treatment[23]             13.9081
treatment[24]             17.6550
treatment[25]             13.1864

Standard Error for Comparing Two Adjusted Treatment Means: 4.617277
(df=55)
[group]          Mean          LSD Grouping
  [19]         5.400988         *
  [9]          9.557555         *       *
  [3]          11.126063        *       *       *
  [8]          11.310598        *       *       *
  [17]         11.497972        *       *       *
  [10]         11.588868        *       *       *
  [6]          11.813338        *       *       *
  [7]          11.904538        *       *       *
  [5]          12.654334        *       *       *
  [12]         12.723251        *       *       *
  [20]         12.932302       *       *       *       *
  [13]         13.129311       *       *       *       *
  [18]         13.154031       *       *       *       *
  [25]         13.186358       *       *       *       *
  [23]         13.908089       *       *       *       *
  [4]          14.623020       *       *       *       *
  [16]         14.656771               *       *       *
  [21]         15.410829               *       *       *
  [22]         17.002029               *       *       *
  [1]          17.150679               *       *       *
  [24]         17.655045               *       *       *
  [14]         17.876268               *       *       *
  [15]         18.657581               *       *       *
  [2]          19.220003                       *       *
  [11]         22.132051                               *
Number of missing observations: 1
```

split_plot

Analyzes a wide variety of split-plot experiments with fixed, mixed or random factors. The whole-plots can be assigned to experimental units using either a completely randomized or randomized complete block design. Function `split_plot` also analyzes split-plot experiments replicated at several locations.

Synopsis

#include `<imsls.h>`

float `*` `imsls_f_split_plot` (*int* `n`, *int* `n_locations`, *int* `n_whole`, *int* `n_split`, *int* `rep[]`, *int* `whole[]`, *int* `split[]`, *float* `y[]`,…, `0`)

The type *double* function is `imsls_d_split_plot`.

Required Arguments

int `n` (Input)
Number of missing and non-missing experimental observations. `imsls_f_split_plot` verifies that:

$$n = \sum_{i=1}^{\texttt{n\_locations}} \left(\texttt{n\_whole}\cdot\texttt{n\_split}\cdot\texttt{n\_blocks}_i\right)$$

int `n_locations` (Input)
Number of locations. `n_locations` must be one or greater. If `n_locations`>1, then the optional array `locations[]` must be included as input to `imsls_f_split_plot`.

int `n_whole` (Input)
: Number of levels associated with the whole-plot factor. `n_whole` must be greater than one.

int `n_split` (Input)
: Number of levels associated with the split-plot factor. `n_split` must be greater than one.

int `rep[]` (Input)
: An array of length `n` containing the block, or replicate, identifiers for each observation in `y`. Locations can have different numbers of blocks or replicates. Each block or replicate at a single location must be assigned a different identifier, but different locations can have the same assignments.

int `whole[]` (Input)
: An array of length `n` containing the whole-plot identifiers for each observation in `y`. Each level of the whole-plot factor must be assigned a different integer. `imsls_f_split_plot` verifies that the number of unique whole-plot identifiers is equal to `n_whole`.

int `split[]` (Input)
: An array of length `n` containing the split-plot identifiers for each observation in `y`. Each level of the split-plot factor must be assigned a different integer. `imsls_f_split_plot` verifies that the number of unique split-plot identifiers is equal to `n_split`.

float `y[]` (Input)
: An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively. At a single location, only one missing value per whole-plot is allowed. The location, whole-plot and split-plot for each observation in `y` are identified by the corresponding values in the arguments `locations`, `whole` and `split`.

Return Value

Address of a pointer to the memory location of a two dimensional, 11 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table[i*6]`, identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

j	anova_table$_{i,j}$ = anova_table[I*6+j]
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of `anova_table`$_{i,j}$ are the only negative values in `anova_table[]`. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	ANOVA Source
-1	LOCATION†
-2	BLOCK WITHIN LOCATION‡
-3	WHOLE-PLOT
-4	LOCATION × WHOLE-PLOT†
-5	WHOLE-PLOT ERROR
-6	SPLIT-PLOT
-7	LOCATION × SPLIT-PLOT†
-8	WHOLE-PLOT × SPLIT-PLOT
-9	LOCATION × WHOLE-PLOT × SPLIT-PLOT†
-10	SPLIT-PLOT ERROR⥣
-11	CORRECTED TOTAL

Notes: † If `n_locations`=1 sources involving location are set to missing (NaN).

‡ If `IMSLS_CRD` is set, entries for block within location are set to missing, and its sum of squares and degrees of freedom are pooled into the whole-plot error.
⥣ Split-plot error component calculation varies depending upon the settings for `IMSLS_RCBD`, `IMSLS_LOC_FIXED`, `IMSLS_WHOLE_FIXED`, `IMSLS_SPLIT_FIXED`, and upon whether `n_locations`=1. See the "Description" section below for details.

Synopsis with Optional Arguments

#include `<imsl.h>`

float `*` `imsls_f_split_plot` (*int* `n`, *int* `n_locations`, *int* `n_whole`,
int `n_split`, *int* `rep[]`, *int* `whole[]`, *int* `split[]`, *float* `y[]`,

```
IMSLS_RETURN_USER, float anova_table[]
IMSLS_LOCATIONS, int locations[],
IMSLS_LOC_RANDOM or IMSLS_LOC_FIXED,
IMSLS_RCBD or IMSLS_CRD,
IMSLS_WHOLE_FIXED or IMSLS_WHOLE_RANDOM,
IMSLS_SPLIT_FIXED or IMSLS_SPLIT_RANDOM,
IMSLS_N_MISSING, int  *n_missing,
IMSLS_CV, float **cv,
IMSLS_CV_USER, float cv[],
IMSLS_GRAND_MEAN, float *grand_mean,
IMSLS_WHOLE_PLOT_MEANS, float **whole_plot_means,
IMSLS_WHOLE_PLOT_MEANS_USER, float whole_plot_means[],
IMSLS_SPLIT_PLOT_MEANS, float **split_plot_means,
IMSLS_SPLIT_PLOT_MEANS_USER, float split_plot_means[],
IMSLS_TREATMENT_MEANS, float **treatment_means,
IMSLS_TREATMENT_MEANS_USER, float treatment_means[],
IMSLS_STD_ERRORS, float **std_err,
IMSLS_STD_ERRORS_USER, float std_err[],
IMSLS_N_BLOCKS int **n_blocks,
IMSLS_N_BLOCKS_USER, int n_blocks[],
IMSLS_BLOCK_SS float **block_ss,
IMSLS_BLOCK_SS_USER, float block_ss[],
IMSLS_WHOLE_PLOT_SS  float  **whole_plot_ss,
IMSLS_WHOLE_PLOT_SS_USER,  float  whole_plot_ss[],
IMSLS_SPLIT_PLOT_SS  float  **split_plot_ss,
IMSLS_SPLIT_PLOT_SS_USER,  float  split_plot_ss[],
IMSLS_WHOLEXSPLIT_PLOT_SS  float  **wholexsplit_plot_ss,
IMSLS_WHOLEXSPLIT_PLOT_SS_USER,
        float  wholexsplit_plot_ss[],
IMSLS_WHOLE_PLOT_ERROR_SS  float  **whole_plot_error_ss,
IMSLS_WHOLE_PLOT_ERROR_SS_USER,
        float  whole_plot_error_ss[],
IMSLS_SPLIT_PLOT_ERROR_SS  float  **split_plot_error_ss,
IMSLS_SPLIT_PLOT_ERROR_SS_USER,
        float  split_plot_error_ss[],
IMSLS_TOTAL_SS  float **total_ss,
IMSLS_TOTAL_SS_USER, float  total_ss[],
IMSLS_ANOVA_ROW_LABELS, char ***anova_row_labels,
IMSLS_ANOVA_ROW_LABELS_USER, char  *anova_row_labels[],
0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)

User defined array of length 66 for storage of the 11 by 6 Anova table described as the return argument for `imsls_f_split_plot`. For a detailed description of the format for this table, see the previous description of the return arguments for `imsls_f_split_plot`.

IMSLS_LOCATIONS, *int* locations[] (Input)
An array of length n containing the location identifiers for each observation in y. Unique integers must be assigned to each location in the study. This argument is required when n_locations>1.

IMSLS_LOC_FIXED or IMSLS_LOC_RANDOM (Input)
A characteristic controlling whether the location factor is treated as a fixed or random effect, when n_locations>1. IMSLS_LOC_FIXED and IMSLS_LOC_RANDOM imply that the factor is a fixed effect or random effect, respectively.
Default: IMSLS_LOC_RANDOM

IMSLS_RCBD or IMSLS_CRD (Input)
Whole-plot randomization characteristic: IMSLS_RCBD implies that whole-plots are assigned to whole-plot experimental units using a randomized complete block design. IMSLS_CRD implies that whole-plots are completely randomized to whole-plot experimental units.
Default: IMSLS_RCBD

IMSLS_WHOLE_FIXED or IMSLS_WHOLE_RANDOM (Input)
Whole-plot characteristic. IMSLS_WHOLE_FIXED implies that the whole-plot factor is a fixed effect, and IMSLS_WHOLE_RANDOM implies that it is a random effect.
Default: IMSLS_WHOLE_FIXED

IMSLS_SPLIT_FIXED or IMSLS_SPLIT_RANDOM (Input)
Split-plot characteristic. IMSLS_SPLIT_FIXED implies that the split-plot factor is a fixed effect, and IMSLS_SPLIT_RANDOM implies that it is a random effect.
Default: IMSLS_SPLIT_FIXED.

IMSLS_N_MISSING, *int* *n_missing (Output)
Number of missing values, if any, found in y. Missing values are denoted with a NaN (Not a Number) value.

IMSLS_CV, *float* **cv (Output)
Address of a pointer to an internally allocated array of length 2 containing the whole-plot and split-plot coefficients of variation. cv[0] contains the whole-plot C.V., and cv[1] contains the split-plot C.V.

IMSLS_CV_USER, *float* cv[] (Output)
Storage for the array cv, provided by the user.

IMSLS_GRAND_MEAN, *float* *grand_mean (Output)
Mean of all the data across every location.

IMSLS_WHOLE_PLOT_MEANS, *float* **whole_plot_means (Output)
Address of a pointer to an internally allocated array of length n_whole containing the whole-plot means.

IMSLS_WHOLE_PLOT_MEANS_USER, *float* whole_plot_means[] (Output)
Storage for the array whole_plot_means, provided by the user.

`IMSLS_SPLIT_PLOT_MEANS`, *float* `**split_plot_means` (Output)
Address of a pointer to an internally allocated array of length `n_split` containing the split-plot means.

`IMSLS_SPLIT_PLOT_MEANS_USER`, *float* `split_plot_means[]` (Output)
Storage for the array `split_plot_means`, provided by the user.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size (`n_whole` * `n_split`) containing the treatment means. For

i > 0 and j > 0, $\texttt{treatment\_means}_{i,j}$ = `treatment_means`[(i-1)*`n_split`+j-1]

contains the mean of the observations, averaged over all locations, blocks and replicates, for the *j*th split-plot within the *i*th whole-plot.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array `treatment_means`, provided by the user.

`IMSLS_STD_ERRORS`, *float* `**std_err` (Output)
Address of a pointer to an internally allocated array of length 10 containing five standard errors and their associated degrees of freedom.

Element	**Standard Error for Comparisons Between Two**	**Degrees of Freedom**
`std_err[0]`	Whole-Plot Means	`std_err[5]`
`std_err[1]`	Split-Plot Means	`std_err[6]`
`std_err[2]`	Split-Plots within same Whole-Plot	`std_err[7]`
`std_err[3]`	Whole-Plots within same Split-Plot	`std_err[8]`
`std_err[4]`	Treatment Means (same whole-plot, split-plot and sub-plot)	`std_err[9]`

`IMSLS_STD_ERRORS_USER`, *float* `std_err[]` (Output)
Storage for the array `std_err`, provided by the user.

`IMSLS_N_BLOCKS`, *int* `**n_blocks` (Output)
Address of a pointer to an internally allocated array of length `n_locations` containing the number of blocks, or replicates, at each location.

`IMSLS_N_BLOCKS_USER`, *int* `n_blocks[]` (Output)
Storage for the array `n_blocks`, provided by the user.

`IMSLS_BLOCK_SS`, *float* `**block_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for blocks and their associated degrees of freedom for each location.

`IMSLS_BLOCK_SS_USER`, *float* `block_ss[]` (Output)
Storage for the array `block_ss`, provided by the user. Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for blocks and their associated degrees of freedom for each location.

`IMSLS_WHOLE_PLOT_SS`, *float* `**whole_plot_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for whole-plots and their associated degrees of freedom for each location.

`IMSLS_WHOLE_PLOT_SS_USER`, *float* `whole_plot_ss[]` (Output)
Storage for the array `whole_plot_ss`, provided by the user.

`IMSLS_SPLIT_PLOT_SS`, *float* `**split_plot_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for split-plots and their associated degrees of freedom for each location.

`IMSLS_SPLIT_PLOT_SS_USER`, float `split_plot_ss[]` (Output)
Storage for the array split_plot_ss, provided by the user.

`IMSLS_WHOLEXSPLIT_PLOT_SS`, *float* `**wholexsplit_plot_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for whole-plot by split-plot interaction and their associated degrees of freedom for each location.

`IMSLS_WHOLEXSPLIT_PLOT_SS_USER`, *float* `wholexsplit_plot_ss[]` (Output)
Storage for the array `wholexsplit_plot_ss`, provided by the user.

`IMSLS_WHOLE_PLOT_ERROR_SS`, *float* `**whole_plot_error_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for whole-plots and their associated degrees of freedom for each location.

`IMSLS_WHOLE_PLOT_ERROR_SS_USER`, *float* `whole_plot_error_ss[]` (Output)
Storage for the array `whole_plot_error_ss`, provided by the user.

`IMSLS_SPLIT_PLOT_ERROR_SS`, *float* `**split_plot_error_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the sum of squares for split-plots and their associated degrees of freedom for each location.

`IMSLS_SPLIT_PLOT_ERROR_SS_USER`, *float* `split_plot_error_ss[]` (Output)
Storage for the array `split_plot_error_ss`, provided by the user.

`IMSLS_TOTAL_SS`, *float* `**total_ss` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_locations` by 2 containing the corrected total sum of squares and their associated degrees of freedom for each location.

`IMSLS_TOTAL_SS_USER`, *float* `total_ss[]` (Output)
Storage for the array `total_ss`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the *i*-th row of the ANOVA table can be printed with `printf`("%s", `anova_row_labels`[i]);

The memory associated with `anova_row_labels` can be freed with a single call to `free`(`anova_row_labels`).

`IMSLS_ANOVA_ROW_LABELS_USER`, char `*anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels[600]`.

Description

Function `imsls_f_split_plot` is capable of analyzing a wide variety of split-plot experiments. Whole-plot and split-plot factors can each be designated as either fixed or random, allowing for experiments with fixed, random or mixed treatment effects. By default, `imsls_f_split_plot` assumes that all treatment factors are fixed effects, i.e. `IMSLS_WHOLE_FIXED` and `IMSLS_SPLIT_FIXED` are default settings. Whole-plot or split-plot factors can each be declared as random effects by setting the optional input arguments `IMSLS_WHOLE_RANDOM` and `IMSLS_SPLIT_RANDOM`, respectively.

Split-plot experimental designs can also vary in the assignment of the whole-plot factor to its experimental units. In some cases, this assignment is completely random. For example, in a drug study the experimental unit might be the subject receiving a treatment. The whole-plot factor, possibly different treatments, could be assigned in one of two ways. Each subject could receive only one treatment or each could receive all treatments over an appropriate period of time. If each subject received only a single randomly selected treatment, then this design constitutes a completely randomized design for the whole-plot factor, and the optional input argument `IMSLS_CRD` must be set.

On the other hand, if each subject receives every treatment in random order, then the subject is a blocking factor, and this sampling scheme constitutes a randomized complete block design. In this case, it is necessary to assume that there are no carry-over effects from one treatment to another. This sampling scheme is the default setting, i.e. `IMSLS_RCBD` is the default setting.

A similar randomization choice occurs in agricultural field trials. A trial designed to test different fertilizers and different seed lots can be conducted in one of two ways. The whole-plot factor, fertilizer, can be applied to different fields, or each can be applied to sub-divisions of these fields. In either case, a field is the whole-plot experimental unit. In the first case in which only a single randomly selected fertilizer is applied to a single field, the whole-plot factor is not blocked and this scheme is called as a completely randomized design, and the optional input argument `IMSLS_CRD` must be set. However, if fertilizers are applied to sub-plots within a field, then the whole-plot factor is blocked within fields and this assignment is referred to as a randomized complete block design. By default, this routine assumes that levels of the whole-plot factor are randomly assigned within blocks, i.e. `IMSLS_RCBD` is the default setting for randomizing whole-plots.

The essential distinction between split-plot experiments and completely randomized or randomized complete block experiments is the presence of a second factor that is blocked, or nested, within each level of the whole-plot factor. This second factor is referred to as the split-plot factor, see Figure 1. If levels of this factor were completely randomized, then two or more treatments with the same split-plot level could be assigned to the same whole-plot level, see Figure 2.

Whole Plot Factor			
A2	**A1**	**A4**	**A3**
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

Figure 1 – Split-Plot Experiments – Split-Plot B Nested within Whole-Plot A

CRD			
A3B2	A1B3	A4B1	A4B3
A2B3	A1B1	A3B2	A1B2
A2B2	A3B1	A2B1	A4B2

Figure 2 – Completely Randomized Experiments – Both Factors Randomized

In some studies, a split-plot experiment is replicated at several locations. Function `imsls_f_split_plot` can also analyze split-plot experiments replicated at multiple locations, even when the number of blocks or replicates at each location are different. If only a single replicate or block is used at each location, then location should be treated as a blocking factor, with n_locations set equal to one. If n_locations=1, it is assumed that the experiment was conducted at a single location with more than one block or replicate at that location. In this case, the four entries associated with location in the Anova table will contain missing values.

However, if n_locations>1, it is assumed the experiment was repeated at multiple locations, with replication or blocking occurring at each location. Although the number of blocks, or replicates, at each location can be different, the number of levels for whole-plot and split-plot factors, n_whole and n_split, must be the same at each location. The location associated with y[i] is specified in location[i], which is a required input argument when n_locations>1.

By default, locations are assumed to be random effects. However, they can be specified as fixed effects by setting the optional argument IMSLS_LOC_FIXED. This setting changes the calculations of the F-tests for whole-plot and split-plot factors. If locations are assumed to be fixed effects, then the whole-plot and split-plot errors at each location are pooled to form the whole-plot and split-plot errors. This can dramatically increase the degrees of freedom associated with the F-test for the treatment factors, resulting in smaller *p*-values. However, pooling the error terms from different locations requires experimenters to assume that the errors at each location are approximately the same. This should be verified using a test for homogeneity of variance, such as Bartlett's or Levene's test.

On the other hand, if locations are assumed to be random effects, then tests involving whole-plots use the interaction between whole-plots and locations as the error term for testing whether there are statistically significant differences among whole-plot factor levels. However, this assumes that the interaction of whole-plots and locations is not statistically significant. A test of this assumption uses the pooled whole-plot error. If the interaction between whole-plots and locations is statistically significant, then the nature of that interaction should be explored since it impacts the interpretation of the significance of the whole-plot treatment factor.

Similarly, when locations are assumed to be random effects, tests involving split-plots do not use the split-plot errors pooled across locations. Instead, the error term for split plots is the interaction between locations and split-plots. The split-plot by whole-plot interaction is tested against the location by split-plot by whole-plot interaction.

Suppose, for example, that a researcher wanted to conduct an agricultural experiment comparing the effectiveness of 4 fertilizers with 4 seed lots. One replicate of the experiment is conducted at each of the 3 farms. That is, only a single field at each location is assigned to this experiment.

The field at each farm is divided into 4 whole-plots and the fertilizers are randomly assigned to each of the 4 whole-plots. Each whole-plot is then further divided into 4 split-plots, and the seed lots are randomly assigned to these split-plots.

In this case, each farm is a blocking factor, fertilizers are whole-plots and seed lots are split-plots. The input array `rep` would contain integers from 1 to the number of farms.

However, if each farm allocated more than a single field for this study, then each farm would be treated as a different location with `n_locations` set equal to the number of farms, and fields would be treated as blocking factor. The array `rep` would contain integers from 1 to the number fields used in a farm, and `locations[]` would contain integers from 1 to the number of farms.

In summary this routine can analyze 3x2x2x2=24 different experimental situations, depending upon the settings of:

1. Locations (none, fixed or random): specified by setting `n_locations`, `locations[]` and `IMSLS_LOC_FIXED` or `IMSLS_LOC_RANDOM`.
2. Whole-plot sampling (CRD or RCBD): specified by setting `IMSLS_CRD` or `IMSLS_RCBD`.
3. Whole-plot effect (fixed or random): specified by setting either `IMSLS_WHOLE_FIXED` or `IMSLS_WHOLE_RANDOM`.
4. Split-plot effect (fixed or random): specified by setting either `IMSLS_SPLIT_FIXED` or `IMSLS_SPLIT_RANDOM`.

The default condition depends upon the value for `n_locations`. If `n_locations`>1, locations are assumed to be a random effect. Assignment of experimental units to whole-plots is assumed to use a RCBD design and both whole-plots and split-plots are assumed to be fixed effects.

Example

This example uses data from a split-plot design consisting of 2 whole-plots and 4 split-plots.

```
#include <stdlib.h>
#include <math.h>
#include "imsls.h"

void main()
{
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ",
                        "Mean\nsquares", "\nF", "\np-value"};
  int i, page_width = 132;

  int n = 24;                /* Total number of observations */
  int n_locations = 1;       /* Number of locations */
  int n_whole = 2;           /* Number of Whole-plots within a location */
  int n_split = 4;           /* Number of Split-plots within a location,
Whole_plot */
  int rep[]={
    1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3};
  int whole[]={
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2};
  int split[]={
    1, 2, 3, 4, 1, 2, 3, 4,
    1, 2, 3, 4, 1, 2, 3, 4,
    1, 2, 3, 4, 1, 2, 3, 4};
  float y[] ={
    30.0, 40.0, 38.9, 38.2,
    41.8, 52.2, 54.8, 58.2,
    20.5, 26.9, 21.4, 25.1,
    26.4, 36.7, 28.9, 35.9,
    21.0, 25.4, 24.0, 23.3,
    34.4, 41.0, 33.0, 34.9};
  float grand_mean;
  float *aov;
  float *treatment_means;
  float *whole_plot_means;
  float *split_plot_means;
  int *equal_means;
  char **aov_row_labels;

  aov = imsls_f_split_plot(n, n_locations, n_whole, n_split,
                          rep, whole, split, y,
                          IMSLS_GRAND_MEAN, &grand_mean,
                          IMSLS_TREATMENT_MEANS, &treatment_means,
                          IMSLS_WHOLE_PLOT_MEANS, &whole_plot_means,
                          IMSLS_SPLIT_PLOT_MEANS, &split_plot_means,
                          IMSLS_ANOVA_ROW_LABELS, &aov_row_labels,
                          0);

  /* Output results. */
  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);

  /* Print ANOVA table, without first column. */
  imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                     11, 6, aov,
                     IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                     IMSLS_ROW_LABELS, aov_row_labels,
                     IMSLS_COL_LABELS, col_labels,
                     0);

  /* Print the various means. */
  printf("\n\nGrand mean: %f\n", grand_mean);
  imsls_f_write_matrix("Treatment Means", n_whole, n_split,
                     treatment_means, 0);
  imsls_f_write_matrix("Whole-plot  Means", n_whole, 1,
                     whole_plot_means, 0);
  imsls_f_write_matrix("Split-plot Means", n_split, 1,
                     split_plot_means, 0);

}
```

Output

```
                   *** ANALYSIS OF VARIANCE TABLE ***
                                                   Mean
                          ID   DF       SSQ  squares        F  p-value
Location                  -1  ...  ........  .......  .......  .......
Block Within Location     -2    2   1310.28   655.14    30.82    0.031
Whole-Plot                -3    1    858.01   858.01    40.37    0.024
Location x Whole-Plot     -4  ...  ........  .......  .......  .......
Whole-Plot Error          -5    2     42.51    21.26     2.03    0.173
Split-Plot                -6    3    227.73    75.91     7.26    0.005
Location x Split-Plot     -7  ...  ........  .......  .......  .......
Whole-Plot x Split-Plot   -8    3     13.40     4.47     0.43    0.737
Location x Whole-Plot x   -9  ...  ........  .......  .......  .......
   Split-Plot
Split-Plot Error         -10   12    125.39    10.45  .......  .......
Corrected Total          -11   23   2577.33  .......  .......  .......


Grand mean: 33.870834

                   Treatment Means
             1            2            3            4
1        23.83        30.77        28.10        28.87
2        34.20        43.30        38.90        43.00
```

```
Whole-plot  Means
 1        27.89
 2        39.85

Split-plot Means
 1        29.02
 2        37.03
 3        33.50
 4        35.93
```

split_split_plot

Analyzes data from split-split-plot experiments. The whole-plots can be assigned to experimental units using either a completely randomized or randomized complete block design. Function `split_split_plot` also analyzes split-split-plot experiments replicated at several locations.

Synopsis

#include `<imsls.h>`

float `* imsls_f_split_split_plot` (*int* `n`, *int* `n_locations`, *int* `n_whole`, *int* `n_split`, *int* `n_sub`, *int* `rep[]`, *int* `whole[]`, *int* `split[]`, *int* `sub[]`, *float* `y[]`, …, 0)

The type *double* function is `imsls_d_split_split_plot`.

Required Arguments

int `n` (Input)
Number of missing and non-missing experimental observations. `imsls_f_split_split_plot` verifies that:

$$n = \sum_{i=1}^{\mathtt{n\_locations}} \left(\mathtt{n\_whole} \times \mathtt{n\_split} \times \mathtt{n\_sub} \times \mathtt{n\_blocks}_i\right)$$

where $\mathtt{n\_block}_i$ is equal to the number of blocks or replicates at the *i*th location.

int `n_locations` (Input)
Number of locations. `n_locations` must be one or greater. If `n_locations`>1 then the optional array `locations[]` must be included as input. See optional argument `IMSLS_LOCATIONS`.

int `n_whole` (Input)
  Number of levels associated with the whole-plot factor. `n_whole` must be greater than one.

int `n_split` (Input)
  Number of levels associated with the split-plot factor. `n_split` must be greater than one.

int `n_sub` (Input)
  Number of levels associated with the sub-plot factor. `n_sub` must be greater than one.

int `rep[]` (Input)
  An array of length `n` containing the block, or replicate, identifiers for each observation in `y`. Different locations can have different numbers of blocks or replicates. Each block or replicate at a single location must be assigned a different identifier, but different locations can have the same assignments.

int `whole[]` (Input)
  An array of length `n` containing the whole-plot identifiers for each observation in `y`. Each level of the whole-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique whole-plot identifiers is equal to `n_whole`.

int `split[]` (Input)
  An array of length `n` containing the split-plot identifiers for each observation in `y`. Each level of the split-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique split-plot identifiers is equal to `n_split`.

int `sub[]` (Input)
  An array of length `n` containing the sub-plot identifiers for each observation in `y`. Each level of the sub-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique sub-plot identifiers is equal to `n_sub`.

float `y[]` (Input)
  An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are included by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine(6)` or `imsls_d_machine(6)`, depending upon whether single or double precision is being used, respectively. At a single location, only one missing value per whole-plot is allowed. The location, whole-plot, split-plot and sub-plot for each observation in `y` are identified by the corresponding values in the arguments `locations`, `whole`, `split` and `sub`.

Return Value

Address of a pointer to the memory location of a two dimensional, 20 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, `anova_table`$_{i,0}$ = `anova_table`[i*6], identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

J	**anova_table$_{i,j}$ = anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of `anova_table`$_{i,j}$ are the only negative values in `anova_table[]`. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	ANOVA Source
-1	LOCATION†
-2	BLOCK WITHIN LOCATION‡
-3	WHOLE-PLOT
-4	LOCATION × WHOLE-PLOT†
-5	WHOLE-PLOT ERROR
-6	SPLIT-PLOT
-7	LOCATION × SPLIT-PLOT†
-8	WHOLE-PLOT × SPLIT-PLOT
-9	LOCATION × WHOLE-PLOT × SPLIT-PLOT†
-10	SPLIT-PLOT ERROR⇑
-11	CORRECTED TOTAL
-12	LOCATION × SUB-PLOT†
-13	WHOLE-PLOT × SUB-PLOT
-14	LOCATION × WHOLE-PLOT × SUB-PLOT†
-15	SPLIT-PLOT × SUB-PLOT
-16	LOCATION × SPLIT-PLOT × SUB-PLOT†
-17	WHOLE-PLOT × SPLIT-PLOT × SUB-PLOT
-18	LOCATION × WHOLE-PLOT × SPLIT-PLOT × SUBPLOT†

Source Identifier	ANOVA Source
-19	SUB-PLOT ERROR
-20	CORRECTED TOTAL

Notes: † If `n_locations=1` sources involving location are set to missing (NaN).

‡ If `IMSLS_CRD` is set, entries for blocks within location are set to missing, and its sum of squares and degrees of freedom are pooled into the whole-plot error.
* Split-plot error component calculation varies depending upon `n_locations`. See description below for details.

Synopsis with Optional Arguments

#include `<imsl.h>`

float * `imsls_f_split_split_plot` (*int* `n`, *int* `n_locations`, *int*
`n_whole`, *int* `n_split`, *int* `n_sub`, *int* `rep[]`, *int* `whole[]`,
int `split[]`,*int* `sub[]`, *float* `y[]`,
`IMSLS_RETURN_USER`, *float* `anova_table[]`,
`IMSLS_LOCATIONS`, *int* `locations[]`,
`IMSLS_RCBD` *or* `IMSLS_CRD`,
`IMSLS_N_MISSING`, *int* `*n_missing`,
`IMSLS_CV`, *float* `**cv`,
`IMSLS_CV_USER`, *float* `cv[]`,
`IMSLS_GRAND_MEAN`, *float* `*grand_mean`,
`IMSLS_WHOLE_PLOT_MEANS`, *float* `**whole_plot_means`,
`IMSLS_WHOLE_PLOT_MEANS_USER`, *float* `whole_plot_means[]`,
`IMSLS_SPLIT_PLOT_MEANS`, *float* `**split_plot_means`,
`IMSLS_SPLIT_PLOT_MEANS_USER`, *float* `split_plot_means[]`,
`IMSLS_SUB_PLOT_MEANS`, *float* `**sub_plot_means`,
`IMSLS_SUB_PLOT_MEANS_USER`, *float* `sub_plot_means[]`,
`IMSLS_WHOLE_SPLIT_PLOT_MEANS`,
float `**whole_split_plot_means`,
`IMSLS_WHOLE_SPLIT_PLOT_MEANS_USER`,
float `whole_split_plot_means[]`,
`IMSLS_WHOLE_SUB_PLOT_MEANS`, *float*
`**whole_sub_plot_means`,
`IMSLS_WHOLE_SUB_PLOT_MEANS_USER`
float `whole_sub_plot_means[]`,
`IMSLS_SPLIT_SUB_PLOT_MEANS`, *float* `**split_sub_plot_means`,
`IMSLS_SPLIT_SUB_PLOT_MEANS_USER`,
float `split_sub_plot_means[]`,
`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means`,
`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]`,
`IMSLS_STD_ERRORS`, *float* `**std_err`,
`IMSLS_STD_ERRORS_USER`, *float* `std_err[]`,
`IMSLS_N_BLOCKS` *int* `**n_blocks`,

`IMSLS_N_BLOCKS_USER`, *int* `n_blocks[]`,
`IMSLS_LOCATION_ANOVA_TABLE` *float* `**location_anova_table`,
`IMSLS_LOCATION_ANOVA_TABLE_USER`,
    float `location_anova_table[]`,
`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels`,
`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined array of length 120 for storage of the 20 by 6 anova table described as the return argument for `imsls_f_split_split_plot`. For a detailed description of the format for this table, see the previous description of the return value for `imsls_f_split_split_plot`.

`IMSLS_LOCATIONS`, *int* `locations[]` (Input)
An array of length `n` containing the location identifiers for each observation in `y`. Unique integers must be assigned to each location in the study. This argument is required when `n_locations`>1.

`IMSLS_RCBD` or `IMSLS_CRD` (Input)
Whole-plot randomization characteristic: `IMSLS_RCBD` implies that whole-plots are assigned to whole-plot experimental units using a randomized complete block design. `IMSLS_CRD` implies that whole-plots are completely randomized to whole-plot experimental units.
Default: `IMSLS_RCBD`

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `**cv` (Output)
Address of a pointer to an internally allocated array of length 3 containing the whole-plot, split-plot and sub-plot coefficients of variation. `cv`[0] contains the whole-plot C.V., `cv`[1] contains the split-plot C.V., and `cv`[2] contains the sub-plot C.V.

`IMSLS_CV_USER`, *float* `cv[]` (Output)
Storage for the array `cv`, provided by the user.

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
Mean of all the data across every location.

`IMSLS_WHOLE_PLOT_MEANS`, *float* `**whole_plot_means` (Output)
Address of a pointer to an internally allocated array of length `n_whole` containing the whole-plot means.

`IMSLS_WHOLE_PLOT_MEANS_USER`, *float* `whole_plot_means[]` (Output)
Storage for the array `whole_plot_means`, provided by the user.

IMSLS_SPLIT_PLOT_MEANS, *float* **split_plot_means (Output)
Address of a pointer to an internally allocated array of length n_split containing the split-plot means.

IMSLS_SPLIT_PLOT_MEANS_USER, *float* split_plot_means[] (Output)
Storage for the array split_plot_means, provided by the user.

IMSLS_SUB_PLOT_MEANS, *float* **sub_plot_means (Output)
Address of a pointer to an internally allocated array of length n_sub containing the sub-plot means.

IMSLS_SUB_PLOT_MEANS_USER, *float* sub_plot_means[] (Output)
Storage for the array sub_plot_means, provided by the user.

IMSLS_WHOLE_SPLIT_PLOT_MEANS, *float* **whole_split_plot_means (Output)
Address of a pointer to an internally allocated 2-dimensional array of size n_whole by n_split containing the whole-plot by split-plot means.

IMSLS_WHOLE_SPLIT_PLOT_MEANS_USER, *float* whole_split_plot_means[] (Output)
Storage for the array whole_split_plot_means, provided by the user.

IMSLS_WHOLE_SUB_PLOT_MEANS, *float* **whole_sub_plot_means (Output)
Address of a pointer to an internally allocated 2-dimensional array of size n_whole by n_sub containing the whole-plot by sub-plot means.

IMSLS_WHOLE_SUB_PLOT_MEANS_USER, *float* whole_sub_plot_means[] (Output)
Storage for the array whole_sub_plot_means, provided by the user.

IMSLS_SPLIT_SUB_PLOT_MEANS, *float* **split_sub_plot_means (Output)
Address of a pointer to an internally allocated 2-dimensional array of size n_split by n_sub containing the split-plot by sub-plot means.

IMSLS_SPLIT_SUB_PLOT_MEANS_USER, *float* split_sub_plot_means[] (Output)
Storage for the array split_sub_plot_means, provided by the user.

IMSLS_TREATMENT_MEANS, *float* **treatment_means (Output)
Address of a pointer to an internally allocated array of size (n_whole*n_split*n_sub) containing the treatment means. For $i > 0, j > 0$ and $k > 0$, $\texttt{treatment\_means}_{i,j,k}$ = treatment_means[(i-1)*n_split*n_sub+(j-1)*n_sub + k-1] contains the mean of the observations, averaged over all locations, blocks and replicates, for the kth sub-plot within the jth split-plot within the ith whole-plot.

IMSLS_TREATMENT_MEANS_USER, *float* treatment_means[] (Output)
Storage for the array treatment_means, provided by the user.

IMSLS_STD_ERRORS, *float* **std_err (Output)

Address of a pointer to an internally allocated array of length 8 containing five standard errors and their associated degrees of freedom. The standard errors are in the first five elements and their associated degrees of freedom are reported in std_err[4] through std_err[7].

Element	Standard Error for Comparisons Between Two	Degrees of Freedom
std_err[0]	Whole-Plot Means	std_err[4]
std_err[1]	Split-Plot Means	std_err[5]
std_err[2]	Sub-Plot Means	std_err[6]
std_err[3]	Treatment Means (same whole-plot, split-plot and sub-plot)	std_err[7]

IMSLS_STD_ERRORS_USER, *float* std_err[] (Output)

Storage for the array std_err, provided by the user.

IMSLS_N_BLOCKS, *int* **n_blocks (Output)

Address of a pointer to an internally allocated array of length n_locations containing the number of blocks, or replicates, at each location.

IMSLS_N_BLOCKS_USER, *int* n_blocks[] (Output)

Storage for the array n_blocks, provided by the user.

IMSLS_LOCATION_ANOVA_TABLE, *float* **location_anova_table (Output)

Address of a pointer to an internally allocated 3-dimensional array of size n_locations by 20 by 6 containing the anova tables associated with each location. For each location, the 20 by 6 dimensional array corresponds to the anova table for that location. For example, location_anova_table[(*i*-1)*120+(*j*-1)*6 + (*k*-1)] contains the value in the *k*th column and *j*th row of the returned anova-table for the *i*th location.

IMSLS_LOCATION_ANOVA_TABLE_USER, *float* anova_table[] (Output)

Storage for the array location_anova_table, provided by the user.

IMSLS_ANOVA_ROW_LABELS, *char* ***anova_row_labels (Output)

Address of a pointer to a pointer to an internally allocated array containing the labels for each of the n_anova rows of the returned ANOVA table. The label for the *i*th row of the ANOVA table can be printed with

```
printf("%s", anova_row_labels[i]);
```

The memory associated with anova_row_labels can be freed with a single call to free(anova_row_labels).

IMSLS_ANOVA_ROW_LABELS_USER, *char* *anova_row_labels[] (Output)

Storage for the array anova_row_labels, provided by the user. The amount of space required will vary depending upon the number of factors and n_anova. An upperbound on the required memory is char *anova_row_labels[600].

Description

Function imsls_f_split_split_plot is capable of analyzing a wide variety of split-split-plot experiments.

Split-split-plot experimental designs can vary in the assignment of whole-plot factors to experimental units. In some cases, this assignment is completely random. For example, in a drug study the experimental unit might be the subject receiving a treatment. The whole-plot factor, possibly different treatments, could be assigned in one of two ways. Each subject could receive only one treatment or each could receive all treatments over an appropriate period of time. If each subject received only a single randomly selected treatment, then this design constitutes a completely randomized design for the whole-plot factor, and the optional input argument IMSLS_CRD must be set.

On the other hand, if each subject receives every treatment in random order, then the subject is a blocking factor, and this sampling scheme constitutes a randomized complete block design. In this case, it is necessary to assume that there are no carry-over effects from one treatment to another. This sampling scheme is the default setting, i.e. IMSLS_RCBD is the default setting.

This randomization choice occurs often in agricultural field trials. A trial designed to test different fertilizers and different seed lots can be conducted in one of two ways. The whole-plot factor, fertilizer, can be applied to different fields, or each can be applied to sub-divisions of these fields. In either case, a field, or a sub-division of a field, is the whole-plot experimental unit. In the first case, in which only one randomly selected fertilizer is applied to each field, the whole-plot factor is not blocked and this scheme is called as a completely randomized design, and the optional input argument IMSLS_CRD must be set. However, if fertilizers are applied to sub-divisions within a field, then the whole-plot factor is blocked within fields and this assignment is referred to as a randomized complete block design. By default, imsls_f_split_split_plot assumes that levels of the whole-plot factor are randomly assigned within blocks, i.e. IMSLS_RCBD is the default setting for randomizing whole-plots.

The essential distinction between split-plot and split-split-plot experiments is the presence of a third factor that is blocked, or nested, within each level of the whole-plot and split-plot factors. This third factor is referred to as the sub-plot factor.

Whole Plot Factor			
A2	**A1**	**A4**	**A3**
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

Figure 1 – Split-Plot Experiment – Split-Plot B Nested within Whole-Plot A

Whole Plot Factor A			
A2	**A1**	**A4**	**A3**
A2B3C2 A2B3C1	A1B2C1 A1B2C2	A4B1C2 A4B1C1	A3B3C2 A3B3C1
A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B3C2 A4B3C1	A3B2C2 A3B2C1
A2B2C2 A2B2C1	A1B3C1 A1B3C2	A4B2C1 A4B2C2	A3B1C2 A3B1C1

Figure 2 – Split-Split Plot Experiment – Sub-Plot Factor C Nested Within Split-Plot Factor B, Nested Within Whole-Plot Factor A

Contrast the split-split plot experiment to the same experiment run using a strip-split plot design, see Figure 3. In a strip-split plot experiment factor B is applied in strip across factor A; whereas, in a split-split plot experiment, factor B is randomly assigned to each level of factor A. In a strip-split plot experiment, the level of factor B is constant across a row; whereas in a split-split plot experiment, the levels of factor B change as you go across a row, reflecting the fact that factor B is randomized within each level of factor A.

		Factor A Strip Plots			
		A2	**A1**	**A4**	**A3**
Factor B Strip Plots	**B3**	A2B3C2 A2B3C1	A1B3C1 A1B3C2	A4B3C2 A4B3C1	A3B3C2 A3B3C1
	B1	A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B1C2 A4B1C1	A3B1C2 A3B1C1
	B2	A2B2C2 A2B2C1	A1B2C1 A1B2C2	A4B2C1 A4B2C2	A3B2C2 A3B2C1

Figure 3 – Strip-Split Plot Experiment - Split-Plots Nested Within Strip-Plot Factors A and B

In some studies, a split-split-plot experiment is replicated at several locations. Function `imsls_f_split_split_plot` can analyze these, even when the number of blocks or replicates at each location is different. If only a single replicate or block is used at each location, then location should be treated as a blocking factor, with `n_locations` set equal to one. If `n_locations`=1, it is assumed that the experiment was conducted at a single location with more than one block or replicate at that location. In this case, all entries in the anova table associated with location will contain missing values.

However, if `n_locations`>1, it is assumed the experiment was repeated at multiple locations, with replication or blocking occurring at each location. Although the number of blocks, or replicates, at each location can be different, the number of levels for whole-plot and split-plot factors, `n_whole` and `n_split`, must be the same at each location. The locations associated with each of the observations in `y` are specified in the argument `locations[]`, which is a required input argument when `n_locations`>1.

By default, locations are assumed to be random effects. Tests involving whole-plots use the interaction between whole-plots and locations as the error term for testing whether there are statistically significant differences among whole-plot factor levels. This assumes that the interaction of whole-plots and locations is not statistically significant. A test of this assumption uses the pooled whole-plot error. If the interaction between location and whole-plots, split-plots or sub-plot is statistically significant, then the nature of that interaction should be explored since it impacts the interpretation of the significance of the treatment factors.

When `n_locations` >1 are assumed to be random effects, tests involving split-plots do not use the split-plot errors pooled across locations. Instead, the error term for split plots is the interaction between locations and split-plots. The split-plot by whole-plot interaction is tested against the location by split-plot by whole-plot interaction.

Suppose, for example, that a researcher wanted to conduct an agricultural experiment comparing the effectiveness of 4 fertilizers with 3 rates of application and 2 seed lots. One replicate of the experiment is conducted at each of the 3 farms. That is, only a single field at each location is assigned to this experiment.

Each field is divided into 4 whole-plots and the fertilizers are randomly assigned to each of the 4 whole-plots. Each whole-plot is then further sub-divided into 3 split-plots which are each randomly assigned one of the three fertilizer application rates. Finally, each of these sub-divisions assigned a particular fertilizer and application rate is sub-divided into 2 plots and randomly assigned one of the two seed lots.

In this case, each farm is a blocking factor, fertilizers are whole-plots and fertilizer application rate are split plots, and seed lots are sub-plots. The input array `rep` would contain integers from 1 to the number of farms, with `n_whole`=4, `n_split`=3 and `n_sub`=2.

However, if each farm allocated more than a single field for this study, then each farm would be treated as a different location with `n_locations` set equal to the number of farms, and fields might be treated as blocking factor. The array `rep` would contain integers from 1 to the number fields used in a farm, and `locations[]` would contain integers from 1 to the number of farms.

In summary `imsls_f_split_split_plot` can analyze 3x2=6 different experimental situations, depending upon the settings of:

1. Locations (none, fixed or random): specified by setting `n_locations`, `locations[]` and `IMSLS_LOC_FIXED` or `IMSLS_LOC_RANDOM`.
2. Whole-plot sampling (CRD or RCBD): specified by setting `IMSLS_CRD` or `IMSLS_RCBD`.

The default condition depends upon the value for `n_locations`. If `n_locations`>1, locations are assumed to be a random effect. Assignment of experimental units to whole-plots is assumed to use a RCBD design and whole-plots, split-plots and sub-plots are all assumed to be fixed effects.

Example

This example uses data from a split-split plot design consisting of 2 whole-plots, 2-split-plots and 2 sub-plots.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "imsls.h"

void main()
{
  char **anova_row_labels = NULL;
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ",
                        "Mean\nsquares", "\nF", "\np-value"};
  int i, j, k, l, page_width = 132;

  int n = 24;         /* Total number of observations */
  int n_locations = 1;/* Number of locations */
  int n_whole = 2;    /* Number of Whole-plots within a location */
  int n_split = 2;    /* Number of Split-plots within a location, Whole_plot */
  int n_sub   = 2;
  int rep[]={
    1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3};
  int whole[]={
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2};
  int split[]={
    1, 1, 2, 2, 1, 1, 2, 2,
    1, 1, 2, 2, 1, 1, 2, 2,
    1, 1, 2, 2, 1, 1, 2, 2};
  int sub[]={
    1, 2, 1, 2, 1, 2, 1, 2,
    1, 2, 1, 2, 1, 2, 1, 2,
    1, 2, 1, 2, 1, 2, 1, 2};
  float y[] ={
    30.0, 40.0, 38.9, 38.2,
    41.8, 52.2, 54.8, 58.2,
    20.5, 26.9, 21.4, 25.1,
    26.4, 36.7, 28.9, 35.9,
    21.0, 25.4, 24.0, 23.3,
    34.4, 41.0, 33.0, 34.9};
  float grand_mean;
  float *cv;
  float *aov;
  float *treatment_means;
  float *whole_plot_means;
  float *split_plot_means;
  float *sub_plot_means;
  float *std_err;
  int   *equal_means;

  aov = imsls_f_split_split_plot(n, n_locations, n_whole, n_split, n_sub,
                                 rep, whole, split, sub, y,
                                 IMSLS_GRAND_MEAN, &grand_mean,
                                 IMSLS_CV, &cv,
                                 IMSLS_TREATMENT_MEANS,  &treatment_means,
                                 IMSLS_WHOLE_PLOT_MEANS, &whole_plot_means,
                                 IMSLS_SPLIT_PLOT_MEANS, &split_plot_means,
                                 IMSLS_SUB_PLOT_MEANS,   &sub_plot_means,
                                 IMSLS_STD_ERRORS,       &std_err,
                                 IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                                 0);

  /* Output results. */
  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);

  /* Print ANOVA table. */
  imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                       20, 6, aov,
                       IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                       IMSLS_ROW_LABELS, anova_row_labels,
                       IMSLS_COL_LABELS, col_labels,
                       0);

  printf("\n\nGrand mean:    %7.3f\n", grand_mean);
  printf("Coefficient of Variation ****\n");
  printf("  Whole-Plot: %7.3f\n", cv[0]);
  printf("  Split-Plot: %7.3f\n", cv[1]);
  printf("  Sub-Plot  : %7.3f\n", cv[2]);
  l = 0;
  /*
   * Treatment Means
   */
  printf("\n\n*****************************************************");
  printf("\nTreatment Means: \n");
  for (i=0; i < n_whole; i++){
      for(j=0; j < n_split; j++){
          for(k=0; k < n_sub; k++){
              printf("  treatment[%d][%d][%d] %f \n", i, j, k,
                     treatment_means[l++]);
          }
      }
  }
  printf("\n  Standard Error for Comparing Two Treatment Means: %f \n  (df=%f)\n",
         std_err[3], std_err[7]);
  equal_means = imsls_f_multiple_comparisons(n_whole*n_split*n_sub,
                                             treatment_means, std_err[7],
                                             std_err[3]/sqrt(2),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, .05,
                                             0);
  printf("\n  LSD for Treatment Means (alpha=0.05)");
  imsls_i_write_matrix("  Size of Groups of Means", 1, n_whole*n_split*n_sub-1,
                        equal_means, 0);
  /*
   * Whole-plot Means
   */
  printf("\n\n*****************************************************");
  imsls_f_write_matrix("Whole-plot Means", n_whole, 1,
                       whole_plot_means, 0);
  printf("\nStandard Error for Comparing Two Whole-Plot Means: %f \n(df=%f)\n",
       std_err[0], std_err[4]);
  equal_means = imsls_f_multiple_comparisons(n_whole, whole_plot_means,
                                           std_err[4], std_err[0]/sqrt(2),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, .05,
                                           0);
  printf("\nLSD for Whole-Plot Means (alpha=0.05) \n");
  imsls_i_write_matrix("Size of Groups of Means", 1, n_whole-1,
                      equal_means, 0);
  /*
   * Split-plot Means
   */
  printf("\n\n************************************************************");
  imsls_f_write_matrix("Split-plot Means", n_split, 1,
                      split_plot_means, 0);
  printf("\nStandard Error for Comparing Two Split-Plot Means: %f \n(df=%f)\n",
       std_err[1], std_err[5]);
  equal_means = imsls_f_multiple_comparisons(n_split, split_plot_means,
                                           std_err[5], std_err[1]/sqrt(2),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, .05,
                                           0);
  printf("\nLSD for Split-Plot Means (alpha=0.05) \n");
  imsls_i_write_matrix("Size of Groups of Means", 1, n_split-1,
                       equal_means, 0);
  /*
   * Sub-plot Means
   */
  printf("\n\n************************************************************");
  imsls_f_write_matrix("Sub-plot Means", n_sub, 1,
                      sub_plot_means, 0);
  printf("\nStandard Error for Comparing Two Sub-Plot Means: %f \n(df=%f)\n",
       std_err[2], std_err[6]);
  equal_means = imsls_f_multiple_comparisons(n_sub, sub_plot_means,
                                           std_err[6], std_err[2]/sqrt(2),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, .05,
                                           0);
  printf("\nLSD for Sub-Plot Means (alpha=0.05) \n");
  imsls_i_write_matrix(": Size of Groups of Means", 1, n_sub-1,
      equal_means, 0);
}
```

Output

```
                     *** ANALYSIS OF VARIANCE TABLE ***
                                                           Mean
                                   ID   DF       SSQ   squares        F  p-value
Location                           -1  ...  ........  .......  .......  .......
Block Within Location              -2    2   1310.28   655.14    30.82    0.031
Whole-Plot                         -3    1    858.01   858.01    40.37    0.024
Location x Whole-Plot              -4  ...  ........  .......  .......  .......
Whole-Plot Error                   -5    2     42.51    21.26     0.86    0.490
Split-Plot                         -6    1     17.17    17.17     0.69    0.452
Location x Split-Plot              -7  ...  ........  .......  .......  .......
Whole-Plot x Split-Plot            -8    1      1.55     1.55     0.06    0.815
Location x Whole-Plot x            -9  ...  ........  .......  .......  .......
   Split-Plot
Split-Plot Error                  -10    4     99.32    24.83     7.62    0.008
Sub-Plot                          -11    1    163.80   163.80    50.27    0.000
Location x Sub-Plot               -12  ...  ........  .......  .......  .......
Whole-Plot x Sub-Plot             -13    1     11.34    11.34     3.48    0.099
Location x Whole-Plot x Sub-Plot  -14  ...  ........  .......  .......  .......
Split-plot x Sub-Plot             -15    1     46.76    46.76    14.35    0.005
Location x Split-Plot x Sub-Plot  -16  ...  ........  .......  .......  .......
Whole_plot x Split-Plot           -17    1      0.51     0.51     0.16    0.703
   x Sub-Plot
Location x Whole-Plot x           -18  ...  ........  .......  .......  .......
   Split-Plot x Sub-Plot
Sub-Plot Error                    -19    8     26.07     3.26  .......  .......
Corrected Total                   -20   23   2577.33  .......  .......  .......


Grand mean:     33.871
Coefficient of Variation ****
  Whole-Plot:  13.612
  Split-Plot:  14.712
  Sub-Plot  :   5.329


************************************************************
Treatment Means:
  treatment[0][0][0] 23.833334
  treatment[0][0][1] 30.766668
  treatment[0][1][0] 28.100000
  treatment[0][1][1] 28.866669
  treatment[1][0][0] 34.200001
  treatment[1][0][1] 43.299999
  treatment[1][1][0] 38.899998
  treatment[1][1][1] 43.000000

  Standard Error for Comparing Two Treatment Means: 1.473846
  (df=8.000000)

  LSD for Treatment Means (alpha=0.05)
   Size of Groups of Means
 1   2   3   4   5   6   7
 0   3   0   0   0   0   2


*************************************************************
Whole-plot Means
 1        27.89
 2        39.85

Standard Error for Comparing Two Whole-Plot Means: 2.661792
(df=2.000000)

LSD for Whole-Plot Means (alpha=0.05)

Size of Groups of Means
           0


*************************************************************
Split-plot Means
 1        33.03
 2        34.72

Standard Error for Comparing Two Split-Plot Means: 2.876944
(df=4.000000)

LSD for Split-Plot Means (alpha=0.05)

Size of Groups of Means
           2
```

```
*************************************************************
Sub-plot Means
1        31.26
2        36.48

Standard Error for Comparing Two Sub-Plot Means: 1.473846
(df=8.000000)

LSD for Sub-Plot Means (alpha=0.05)

: Size of Groups of Means
            0
```

strip_plot

Analyzes data from strip-plot experiments. Function `strip_plot` also analyzes strip-plot experiments replicated at several locations.

Synopsis

#include `<imsls.h>`

float `*` `imsls_f_strip_plot` (*int* `n`, *int* `n_locations`, *int* `n_strip_a`,
int `n_strip_b`, *int* `block[]`, *int* `strip_a[]`, *int* `strip_b[]`,
float `y[],…, 0`)

The type *double* function is `imsls_d_strip_plot`.

Required Arguments

int `n` (Input)
Number of missing and non-missing experimental observations. `imsls_f_strip_plot` verifies that:

$$n = \sum_{i=1}^{\texttt{n\_locations}} \left(\texttt{n\_strip\_a} \cdot \texttt{n\_strip} \cdot \texttt{n\_blocks}_i\right)$$

int `n_locations` (Input)
Number of locations. `n_locations` must be one or greater. If `n_locations`>1 then the optional array `locations[]` must be included as input to `imsls_f_strip_plot`. See optional argument `IMSLS_LOCATIONS`.

int `n_strip_a` (Input)
Number of levels associated with the strip factor A. `n_strip_a` must be greater than one.

int `n_strip_b` (Input)

Number of levels associated with the strip factor B. `n_strip_b` must be greater than one.

int `block[]` (Input)

An array of length `n` containing the block identifiers for each observation in `y`. Locations can have different numbers of blocks. Each block at a single location must be assigned a different identifier, but different locations can have the same assignments.

int `strip_a[]` (Input)

An array of length `n` containing the factor A strip-plot identifiers for each observation in `y`. Each level of this factor must be assigned a different integer. This routine verifies that the number of unique factor A strip-plot identifiers is equal to `n_strip_a`.

int `strip_b[]` (Input)

An array of length `n` containing the factor B strip-plot identifiers for each observation in `y`. Each level of this factor must be assigned a different integer. This routine verifies that the number of unique factor B strip-plot identifiers is equal to `n_strip_b`.

float `y[]` (Input)

An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively. The location, strip-plot A, and strip-plot B for each observation in `y` are identified by the corresponding values in the arguments `locations`, `strip_a`, and `strip_b`.

Return Value

Address of a pointer to the memory location of a two dimensional, 12 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table[i*6]`, identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

j	$\texttt{anova\_table}_{i,j}$ **= anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares

j	**anova_table$_{i,j}$ = anova_table[i*6+j]**
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of anova_table$_{i,j}$ are the only negative values in anova_table. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	ANOVA Source
-1	LOCATION†
-2	BLOCK WITHIN LOCATION
-3	STRIP-PLOT A
-4	LOCATION × STRIP-PLOT A†
-5	STRIP-PLOT A ERROR
-6	STRIP-PLOT B
-7	LOCATION × STRIP-PLOT B†
-8	STRIP-PLOT B ERROR
-9	STRIP-PLOT A × STRIP-PLOT B
-10	LOCATION × STRIP-PLOT A × STRIP-PLOT B †
-11	STRIP-PLOT A × STRIP-PLOT B ERROR
-12	CORRECTED TOTAL

Notes: † If n_locations=1 sources involving location are set to missing (NaN).

Synopsis with Optional Arguments

#include <imsl.h>

```
float * imsls_f_strip_plot (int n, int n_locations, int n_strip_a, int
        n_strip_b, int block[], int strip_a[], int strip_b[], float y[],
        IMSLS_RETURN_USER, float anova_table[],
        IMSLS_LOCATIONS, int locations[],
        IMSLS_N_MISSING, int *n_missing,
        IMSLS_CV, float **cv,
        IMSLS_CV_USER, float cv[],
        IMSLS_GRAND_MEAN, float *grand_mean,
        IMSLS_STRIP_PLOT_A_MEANS, float **strip_plot_a_means,
        IMSLS_STRIP_PLOT_A_MEANS_USER,
                float strip_plot_a_means[],
        IMSLS_STRIP_PLOT_B_MEANS, float **strip_plot_b_means,
        IMSLS_STRIP_PLOT_B_MEANS_USER,
                float strip_plot_b_means[],
        IMSLS_TREATMENT_MEANS, float **treatment_means,
```

IMSLS_TREATMENT_MEANS_USER, *float* treatment_means[],
IMSLS_STD_ERRORS, *float* **std_err,
IMSLS_STD_ERRORS_USER, *float* std_err[],
IMSLS_N_BLOCKS *int* **n_blocks,
IMSLS_N_BLOCKS_USER, *int* n_blocks[],
IMSLS_LOCATION_ANOVA_TABLE *float* **location_anova_table,
IMSLS_LOCATION_ANOVA_TABLE_USER,
    float location_anova_table[],
IMSLS_ANOVA_ROW_LABELS, *char* ***anova_row_labels,
IMSLS_ANOVA_ROW_LABELS_USER, *char* *anova_row_labels[],
0)

Optional Arguments

IMSLS_RETURN_USER, *float* anova_table[] (Output)
User defined array of length 72 for storage of the 12 by 6 ANOVA table described as the return argument for imsls_f_strip_plot. For a detailed description of the format for this table, see the previous description of the return arguments for imsls_f_strip_plot.

IMSLS_LOCATIONS, *int* locations[] (Input)
An array of length n containing the location identifiers for each observation in y. Unique integers must be assigned to each location in the study. This argument is required when n_locations>1.

IMSLS_N_MISSING, *int* *n_missing (Output)
Number of missing values, if any, found in y. Missing values are denoted with a NaN (Not a Number) value.

IMSLS_CV, *float* **cv (Output)
Address of a pointer to an internally allocated array of length 3 containing the whole-plot, split-plot and sub-plot coefficients of variation. cv[0] contains the whole-plot C.V., cv[1] contains the split-plot C.V., and cv[2] contains the sub-plot C.V.

IMSLS_CV_USER, *float* cv[] (Output)
Storage for the array cv, provided by the user.

IMSLS_GRAND_MEAN, *float* *grand_mean (Output)
Mean of all the data across every location.

IMSLS_STRIP_PLOT_A_MEANS, *float* **strip_plot_a_means (Output)
Address of a pointer to an internally allocated array of length n_strip_a containing the factor A strip-plot means.

IMSLS_STRIP_PLOT_A_MEANS_USER, *float* strip_plot_a_means []
(Output)
Storage for the array strip_plot_a_means, provided by the user.

IMSLS_STRIP_PLOT_B_MEANS, *float* **strip_plot_b_means (Output)
Address of a pointer to an internally allocated array of length n_strip_b containing the factor B strip-plot means.

`IMSLS_STRIP_PLOT_B_MEANS_USER`, *float* `strip_plot_b_means []` (Output)
Storage for the array `strip_plot_b_means`, provided by the user.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size (`n_split_a`×`n_split_b`) containing the treatment means. For $i > 0$ and $j > 0$, $\texttt{treatment\_means}_{i,j}$ = `treatment_means` [(i-1)×`n_split_a` +(j-1)] contains the mean of the observations, averaged over all locations, blocks and replicates, for the *i*th level of the factor A strip-plot and the *j*th level of the factor B strip-plot.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array `treatment_means`, provided by the user.

`IMSLS_STD_ERRORS`, *float* `**std_err` (Output)
Address of a pointer to an internally allocated array of length 10 containing five standard errors and their associated degrees of freedom. The standard errors are in the first five elements and their associated degrees of freedom are reported in `std_err`[5] through `std_err`[9].

Element	**Standard Error for Comparisons Between Two**	**Degrees of Freedom**
`std_err[0]`	Factor A Strip-Plot Means	`std_err[5]`
`std_err[1]`	Factor B Strip-Plot Means	`std_err[6]`
`std_err[2]`	Factor A Strip-Plot Means at the same level of Factor B	`std_err[7]`
`std_err[3]`	Factor B Strip-Plot Means at the same level of Factor A	`std_err[8]`
`std_err[4]`	Treatment Means (same strip-plot A and strip-plot B)	`std_err[9]`

`IMSLS_STD_ERRORS_USER`, *float* `std_err[]` (Output)
Storage for the array `std_err`, provided by the user.

`IMSLS_N_BLOCKS`, *int* `**n_blocks` (Output)
Address of a pointer to an internally allocated array of length `n_locations` containing the number of blocks, or replicates, at each location.

`IMSLS_N_BLOCKS_USER`, *int* `n_blocks[]` (Output)
Storage for the array `n_blocks`, provided by the user.

`IMSLS_LOCATION_ANOVA_TABLE`, *float* `**location_anova_table` (Output)
Address of a pointer to an internally allocated 3-dimensional array of size `n_locations` by 12 by 6 containing the Anova tables associated with each location. For each location, the 12 by 6 dimensional array

corresponds to the Anova table for that location. For example, `location_anova_table`[(i-1)×72+(j-1)×6 + (k-1)] contains the value in the kth column and jth row of the returned Anova table for the ith location.

`IMSLS_LOCATION_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
Storage for the array `location_anova_table`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the ith row of the ANOVA table can be printed with

```
printf("%s", anova_row_labels[i]);
```

The memory associated with `anova_row_labels` can be freed with a single call to `free`(`anova_row_labels`).

`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels[600]`.

Description

Function `imsls_f_strip_plot` is capable of analyzing a wide variety of strip-plot experiments.

The essential distinction between strip-plot and split-plot experiments is the application of factor B. In a split-plot experiment, levels of Factor B are nested within Factor A, see Figure 2. In strip-plot experiments, Factors A and B are completely crossed, see Figure 1. This occurs, for example, when an agricultural field is used as a block and the levels of factor A are applied in vertical strips across the entire field. Levels of factor B are assigned to horizontal strips across the same block.

		Strip Plot Factor A			
		A2	**A1**	**A4**	**A3**
Strip Plot Factor B	**B3**	A2B3	A1B3	A4B3	A3B3
	B1	A2B1	A1B1	A4B1	A3B1
	B2	A2B2	A1B2	A4B2	A3B2

Figure 1 – Strip-Plot Experiments – Strip-Plots Completely Crossed

Whole Factor Plot			
A2	**A1**	**A4**	**A3**
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

Figure 2 – Split-Plot Experiments – Split-Plot B Nested within Strip-Plot A

In some studies, a strip-plot experiment is replicated at several locations. `imsls_f_strip_plot` can analyze strip-plot experiments replicated at multiple locations, even when the number of blocks or replicates at each location are different. If only a single replicate or block is used at each location, then location should be treated as a blocking factor, with `n_locations` set equal to one. If `n_locations=1`, it is assumed that the experiment was conducted at a single location with more than one block or replicate at that location. In this case, the four entries associated with location in the ANOVA table will contain missing values.

However, if `n_locations>1`, it is assumed the experiment was repeated at multiple locations, with blocking occurring at each location. Although the number of blocks at each location can be different, the number of levels for the factor A and B strip-plots must be the same at each location. The locations associated with each of the observations in `y` are specified in the argument `locations[]`, which is a required input argument when `n_locations`>1.

Locations are assumed to be random effects, then tests involving factor A strip-plots use the interaction between factor A strip-plots and locations as the error term for testing whether there are statistically significant differences among the levels of factor A. However, this assumes that the interaction of factor A and locations is not statistically significant. A test of this assumption is included in the ANOVA table. If the interaction between factor A strip-plots and locations is statistically significant, then the nature of that interaction should be explored since it impacts the interpretation of the significance of the factor A.

Similarly, when locations are assumed to be random effects, tests involving factor B do not use the strip-plot B errors pooled across locations. Instead, the error term for factor B is the interaction between locations and factor B.

Example

This example uses data from a strip-plot design with two levels for the first strip and four for the last strip.

```
#include <stdlib.h>
#include <math.h>
#include "imsls.h"
```

```
void main()
{

  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ",
                        "Mean\nsquares", "\nF", "\np-value"};
  char **anova_row_labels = NULL;
  int i, j, k, l, page_width = 132;
  int n = 24;                /* Total number of observations */
  int n_locations = 1;       /* Number of locations */
  int n_strip_a   = 2;       /* Number of factor A strip-plots within a location */
  int n_strip_b   = 4;       /* Number of factor B strip-plots within a location */

  int block[]={
    1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3};
  int strip_a[]={
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2,
    1, 1, 1, 1, 2, 2, 2, 2};
  int strip_b[]={
    1, 2, 3, 4, 1, 2, 3, 4,
    1, 2, 3, 4, 1, 2, 3, 4,
    1, 2, 3, 4, 1, 2, 3, 4};
  float y[] ={
    30.0, 40.0, 38.9, 38.2,
    41.8, 52.2, 54.8, 58.2,
    20.5, 26.9, 21.4, 25.1,
    26.4, 36.7, 28.9, 35.9,
    21.0, 25.4, 24.0, 23.3,
    34.4, 41.0, 33.0, 34.9};
  float grand_mean=0;
  float *cv;
  float *aov;
  float *treatment_means;
  float *strip_plot_a_means;
  float *strip_plot_b_means;
  float *std_err;
  int n_missing;
  int *equal_means;

  aov = imsls_f_strip_plot(n, n_locations, n_strip_a, n_strip_b,
                           block, strip_a, strip_b, y,
                           IMSLS_GRAND_MEAN, &grand_mean,
                           IMSLS_CV, &cv,
                           IMSLS_N_MISSING,  &n_missing,
                           IMSLS_STRIP_PLOT_A_MEANS, &strip_plot_a_means,
                           IMSLS_STRIP_PLOT_B_MEANS, &strip_plot_b_means,
                           IMSLS_TREATMENT_MEANS, &treatment_means,
                           IMSLS_STD_ERRORS,  &std_err,
                           IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                           0);

  /* Output results. */
  imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);

  /* Print ANOVA table. */
  imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                       12, 6, aov,
                       IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                       IMSLS_ROW_LABELS, anova_row_labels,
                       IMSLS_COL_LABELS, col_labels,
                       0);

  printf("\nGrand mean: %f\n", grand_mean);

  /* Print treatment means */
  imsls_f_write_matrix("Treatment Means", n_strip_a, n_strip_b,
                       treatment_means, 0);
  printf("\n\nStandard Error for Comparing Two Treatment Means: \n");
  printf("  Same Level of Factor B          %f (df=%f)\n",
         std_err[2], std_err[7]);
  printf("  Same Level of Factor A          %f (df=%f)\n",
         std_err[3], std_err[8]);
  printf("  Different Factor A and B Levels %f (df=%f)\n\n\n\n",
         std_err[4], std_err[9]);


  /* Print factor A means */
  imsls_f_write_matrix("Factor A Means", n_strip_a, 1,
                       strip_plot_a_means, 0);
  printf("\nStandard Error for Comparing Two Factor A Means: \n  %f (df=%f)\n",
         std_err[0], std_err[5]);
  equal_means = imsls_f_multiple_comparisons(n_strip_a, strip_plot_a_means,
                                             std_err[5],
                                             std_err[0]/sqrt(2),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, .05,
                                             0);
  /* Print multiple comparison results */
```

```
  imsls_i_write_matrix("LSD Comparison : Size of Groups of Means", 1, n_strip_a-1,
      equal_means, 0);


  /* Print factor B means */
  imsls_f_write_matrix("\n\nFactor B Means", n_strip_b, 1,
                       strip_plot_b_means, 0);
  printf("\nStandard Error for Comparing Two Factor B Means: \n  %f (df=%f)\n",
         std_err[1], std_err[6]);
  equal_means = imsls_f_multiple_comparisons(n_strip_b, strip_plot_b_means,
std_err[6],
                                             std_err[1]/sqrt(2),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, .05,
                                             0);
  /* Multiple comparison results */
  imsls_i_write_matrix("LSD Comparison : Size of Groups of Means",
                       1, n_strip_b-1, equal_means, 0);
}
```

Output

```
                         *** ANALYSIS OF VARIANCE TABLE ***
                                                         Mean
                                  ID   DF       SSQ   squares        F  p-value
Location                          -1  ...  ........  .......  .......  .......
Block Within Location             -2    2   1310.28   655.14    19.89    0.009
Strip-Plot A                      -3    1    858.01   858.01    40.37    0.024
Location x Strip-Plot A           -4  ...  ........  .......  .......  .......
Strip-Plot A Error                -5    2     42.51    21.26     4.62    0.061
Strip-Plot B                      -6    3    227.73    75.91     4.66    0.052
Location x Strip-Plot B           -7  ...  ........  .......  .......  .......
Strip-Plot B Error                -8    6     97.76    16.29     3.54    0.075
Strip-Plot A x Strip-Plot B       -9    3     13.40     4.47     0.97    0.466
Location x Strip-Plot A          -10  ...  ........  .......  .......  .......
   x Strip-Plot B
Strip-Plot A x Strip-Plot B Error  -11    6     27.63     4.60  .......  .......
Corrected Total                  -12   23   2577.33  .......  .......  .......


Grand mean: 33.870834
```

```
                   Treatment Means
             1            2            3            4
1        23.83        30.77        28.10        28.87
2        34.20        43.30        38.90        43.00


Standard Error for Comparing Two Treatment Means:
  Same Level of Factor B          2.417643 (df=4.772558)
  Same Level of Factor A          2.639322 (df=9.140633)
  Different Factor A and B Levels 3.121075 (df=8.405353)



Factor A Means
1        27.89
2        39.85

Standard Error for Comparing Two Factor A Means:
  1.882171 (df=2.000000)

LSD Comparison : Size of Groups of Means
                    0

Factor B Means
1        29.02
2        37.03
3        33.50
4        35.93

Standard Error for Comparing Two Factor B Means:
  2.330465 (df=6.000000)

LSD Comparison : Size of Groups of Means
                1   2   3
                2   3   0
```

strip_split_plot

Analyzes data from strip-split-plot experiments. Function `strip_split_plot` also analyzes strip-split-plot experiments replicated at several locations.

Synopsis

#include `<imsls.h>`

float * `imsls_f_strip_split_plot` (*int* `n`, *int* `n_locations`, *int* `n_strip_a`, *int* `n_strip_b`, *int* `n_split`, *int* `block[]`, *int* `strip_a[]`, *int* `strip_b[]`, *int* `split[]`, *float* `y[]`,…, 0)

The type *double* function is `imsls_d_strip_split_plot`.

Required Arguments

int `n` (Input)
Number of missing and non-missing experimental observations. `imsls_f_strip_split_plot` verifies that:

$$n = \sum_{i=1}^{\texttt{n\_locations}} \left(\texttt{n\_strip\_a} \times \texttt{n\_strip\_b} \times \texttt{n\_split} \times \texttt{n\_block}_i\right)$$

where $\texttt{n\_blocks}_i$ is the number of blocks at location *i*.

int `n_locations` (Input)
Number of locations. `n_locations` must be one or greater. If `n_locations`>1 then the optional array `locations[]` must be included as input to `imsls_f_strip_split_plot`.

int `n_strip_a` (Input)
Number of levels associated with the strip-plot A factor. `n_strip_a` must be greater than one.

int `n_strip_b` (Input)
Number of levels associated with the strip-plots B factor. `n_strip_b` must be greater than one.

int `n_split` (Input)
Number of levels associated with the split factor. `n_split` must be greater than one.

int `block[]` (Input)
An array of length n containing the block identifiers for each observation in `y`. Locations can have different numbers of blocks. Each block at a single location must be assigned a different identifier, but different locations can have the same assignments.

int `strip_a[]` (Input)
An array of length n containing the strip-plot A level identifiers for each observation in `y`. Each level of this factor must be assigned a different integer. `imsls_f_strip_split_plot` verifies that the number of unique strip-plot identifiers is equal to `n_strip_a`.

int `strip_b[]` (Input)
An array of length `n` containing the strip-plot B identifiers for each observation in `y`. Each level of this factor must be assigned a different integer. `imsls_f_strip_split_plot` verifies that the number of unique strip-plot identifiers is equal to `n_strip_b`.

int `split[]` (Input)
An array of length n containing the split-plot level identifiers for each

observation in y. Each level of this factor must be assigned a different integer. `imsls_f_strip_split_plot` verifies that the number of unique split-plot identifiers is equal to `n_split`.

float `y[]` (Input)
An array of length n containing the experimental observations and any missing values. Missing values cannot be omitted. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively. The location, strip-plot A, strip-plot B and split-plot for each observation in y are identified by the corresponding values in the argument's locations, `strip_a`, `strip_b`, and `split`.

Return Value

Address of a pointer to the memory location of a two dimensional, 22 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, $\texttt{anova\_table}_{i,0}$ = `anova_table`[i*6], identifies the source for the effect associated with values in that row. The remaining values in a row contain the ANOVA table values using the following convention:

J	**$\texttt{anova\_table}_{i,j}$ = anova_table[i*6+j]**
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	F-statistic
5	*p*-value for this F-statistic

The Source Identifiers in the first column of $\texttt{anova\_table}_{i,j}$ are the only negative values in `anova_table[]`. Assignments of identifiers to ANOVA sources use the following coding:

Source Identifier	**ANOVA Source**
-1	LOCATION†
-2	BLOCKs WITHIN LOCATION
-3	STRIP-PLOT A
-4	LOCATION × STRIP-PLOT A †
-5	STRIP-PLOT A ERROR

Source Identifier	ANOVA Source
-6	SPLIT-PLOT
-7	SPLIT-PLOT × STRIP-PLOT A
-8	LOCATION × SPLIT-PLOT †
-9	SPLIT-PLOT ERROR
-10	LOCATION × SPLIT-PLOT × STRIP-PLOT A †
-11	STRIP-PLOT B
-12	LOCATION × STRIP-PLOT B †
-13	STRIP_PLOT B ERROR
-14	STRIP-PLOT A × STRIP-PLOT B
-15	LOCATION × STRIP-PLOT A × STRIP-PLOT B
-16	STRIP-PLOT A × STRIP-PLOT B ERROR
-17	SPLIT-PLOT × STRIP-PLOT B
-18	STRIP-PLOT A × STRIP-PLOT B × SPLIT-PLOT
-19	LOCATION × SPLIT-PLOT × STRIP-PLOT B †
-20	LOCATION × STRIP-PLOT A × STRIP-PLOT B × SPLIT-PLOT †
-21	STRIP-PLOT A × STRIP-PLOT B × SPLIT-PLOT ERROR
-22	CORRECTED TOTAL

Notes: † If `n_locations=1` sources involving location are set to missing (NaN).

Synopsis with Optional Arugments

#include `<imsl.h>`

float `*` `imsls_f_strip_split_plot` (*int* `n`, *int* `n_locations`,
int `n_strip_a`, *int* `n_strip_b`, *int* `n_split`, *int* `block[]`,
int `strip_a[]`, *int* `strip_b[]`,*int* `split[]`, *float* `y[]`,
`IMSLS_RETURN_USER`, *float* `anova_table[]`
`IMSLS_LOCATIONS`, *int* `locations[]`,
`IMSLS_N_MISSING`, *int* `*n_missing`,
`IMSLS_CV`, *float* `**cv`,
`IMSLS_CV_USER`, *float* `cv[]`,
`IMSLS_GRAND_MEAN`, *float* `*grand_mean`,
`IMSLS_STRIP_PLOT_A_MEANS`, *float* `**strip_plot_a_means`,
`IMSLS_STRIP_PLOT_A_MEANS_USER`,
float `strip_plot_a_means[]`,

IMSLS_STRIP_PLOT_B_MEANS, *float* `**strip_plot_b_means`,
IMSLS_STRIP_PLOT_B_MEANS_USER,
        float `strip_plot_b_means[]`,
IMSLS_SPLIT_PLOT_MEANS, *float* `**split_plot_means`,
IMSLS_SPLIT_PLOT_MEANS_USER, *float* `split_plot_means[]`,
IMSLS_STRIP_PLOT_AB_MEANS, *float* `**strip_plot_ab_means`,
IMSLS_STRIP_PLOT_AB_MEANS_USER,
        float `strip_plot_ab_means[]`,
IMSLS_STRIP_PLOT_A_SPLIT_PLOT_MEANS,
        float `**strip_plot_a_split_plot_means`,
IMSLS_STRIP_PLOT_A_SPLIT_PLOT_MEANS_USER,
        float `strip_plot_a_split_plot_means[]`,
IMSLS_STRIP_PLOT_B_SPLIT_PLOT_MEANS,
        float `**strip_plot_b_split_plot_means`,
IMSLS_STRIP_PLOT_B_SPLIT_PLOT_MEANS_USER,
        float `strip_plot_b_split_plot_means[]`,
IMSLS_TREATMENT_MEANS, *float* `**treatment_means`,
IMSLS_TREATMENT_MEANS_USER, *float* `treatment_means[]`,
IMSLS_STD_ERRORS, *float* `**std_err`,
IMSLS_STD_ERRORS_USER, *float* `std_err[]`,
IMSLS_N_BLOCKS *int* `**n_blocks`,
IMSLS_N_BLOCKS_USER, *int* `n_blocks[]`,
IMSLS_LOCATION_ANOVA_TABLE *float* `**location_anova_table`,
IMSLS_LOCATION_ANOVA_TABLE_USER,
        float `location_anova_table[]`,
IMSLS_ANOVA_ROW_LABELS, *char* `***anova_row_labels`,
IMSLS_ANOVA_ROW_LABELS_USER, *char* `*anova_row_labels[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `anova_table[]` (Output)
User defined array of length 132 for storage of the 22 by 6 anova table described as the return argument for `imsls_f_strip_split_plot`. For a detailed description of the format for this table, see the previous description of the return arguments for `imsls_f_strip_split_plot`.

`IMSLS_LOCATIONS`, *int* `locations[]` (Input)
An array of length `n` containing the location identifiers for each observation in `y`. Unique integers must be assigned to each location in the study. This argument is required when `n_locations`>1.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV`, *float* `**cv` (Output)
Address of a pointer to an internally allocated array of length 3 containing the strip-plots and split-plot coefficients of variation. `cv[0]`

contains the strip-plot A C.V., `cv[1]` contains the strip-plot B C.V., and `cv[2]` contains the split-plot C.V.

`IMSLS_CV_USER`, *float* `cv[]` (Output)
Storage for the array `cv`, provided by the user.

`IMSLS_GRAND_MEAN`, *float* `*grand_mean` (Output)
Mean of all the data across every location.

`IMSLS_STRIP_PLOT_A_MEANS`, *float* `**strip_plot_a_means` (Output)
Address of a pointer to an internally allocated array of length `n_strip_a` containing the factor A strip-plot means.

`IMSLS_STRIP_PLOT_A_MEANS_USER`, *float* `strip_plot_a_means[]` (Output)
Storage for the array `strip_plot_a_means`, provided by the user.

`IMSLS_STRIP_PLOT_B_MEANS`, *float* `**split_plot_b_means` (Output)
Address of a pointer to an internally allocated array of length `n_split_b` containing the strip-plot B means.

`IMSLS_STRIP_PLOT_B_MEANS_USER`, *float* `strip_plot_b_means[]` (Output)
Storage for the array `split_plot_b_means`, provided by the user.

`IMSLS_SPLIT_PLOT_MEANS`, *float* `**split_plot_means` (Output)
Address of a pointer to an internally allocated array of length `n_split` containing the strip-plot B means.

`IMSLS_SPLIT_PLOT_MEANS_USER`, *float* `split_plot_means[]` (Output)
Storage for the array `split_plot_means`, provided by the user.

`IMSLS_STRIP_PLOT_A_SPLIT_PLOT_MEANS`, *float* `**strip_plot_a_split_plot_means` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_strip_a` by `n_split` containing the means for all combinations of the factor A strip-plot and split-plots.

`IMSLS_STRIP_PLOT_A_SPLIT_PLOT_MEANS_USER`, *float* `strip_plot_a_split_plot_means []` (Output)
Storage for the array `strip_a_split_plot_means`, provided by the user.

`IMSLS_STRIP_PLOT_B_SPLIT_PLOT_MEANS`, *float* `**split_plot_b_split_plot_means` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_split_b` by `n_split` containing the means for all combinations of strip-plot B and split-plots.

`IMSLS_STRIP_B_PLOT_SPLIT_PLOT_MEANS_USER`, *float* `strip_plot_b_split_plot_means[]` (Output)
Storage for the array `strip_b_split_plot_means`, provided by the user.

`IMSLS_STRIP_PLOT_AB_MEANS`, *float* `**strip_plot_ab_means` (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_strip_a` by `n_strip_b` containing the means for all combinations of strip-plots.

`IMSLS_STRIP_PLOT_AB_MEANS_USER`, *float* `strip_plot_ab_means[]` (Output)
Storage for the array `strip_plot_ab_means`, provided by the user.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)
Address of a pointer to an internally allocated array of size (`n_strip_a*n_strip_b*n_split`) containing the treatment means. For $i > 0$ and $j > 0$, $\texttt{treatment\_means}_{i,j}$ = `treatment_means` [(i-1)`*n_split` +(j-1)] contains the mean of the observations, averaged over all locations, blocks and replicates, for the *i*th level of the strip-plot and the *j*th level of the split-plot.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)
Storage for the array treatment_means, provided by the user.

`IMSLS_STD_ERRORS`, *float* `**std_err` (Output)
Address of a pointer to an internally allocated array of length 20 containing ten standard errors and their associated degrees of freedom. The standard errors are in the first 10 elements and their associated degrees of freedom are reported in `std_err[10]` through `std_err[19]`.

Element	Standard Error for Comparisons Between Two	Degrees of Freedom
`std_err[0]`	Strip-Plot A Means	`std_err[10]`
`std_err[1]`	Strip-Plot B Means	`std_err[11]`
`std_err[2]`	Split-Plot Means	`std_err[12]`
`std_err[3]`	Strip-Plot A Means at the same level of split-plots	`std_err[13]`
`std_err[4]`	Strip-Plot A Means at the same level of strip-plot B	`std_err[14]`
`std_err[5]`	Strip-Plot B Means at the same level of split-plots	`std_err[15]`
`std_err[6]`	Strip-Plot B Means at the same level of strip-plot A	`std_err[16]`
`std_err[7]`	Split-Plot Means at the same level of split-plot A	`std_err[17]`
`std_err[8]`	Split-Plot Means at the same level of strip-plot B	`std_err[18]`
`std_err[9]`	Treatment Means (same strip-plot A, strip-plot B and split-plot)	`std_err[19]`

`IMSLS_STD_ERRORS_USER`, *float* `std_err[]` (Output)
Storage for the array `std_err`, provided by the user.

`IMSLS_N_BLOCKS`, *int* `**n_blocks` (Output)
Address of a pointer to an internally allocated array of length `n_locations` containing the number of blocks, or replicates, at each location. This value must be greater than one, `n_blocks` > 1.

`IMSLS_N_BLOCKS_USER`, *int* `n_blocks[]` (Output)
User provided storage for the array `n_blocks`.

`IMSLS_LOCATION_ANOVA_TABLE`, *float* `**location_anova_table` (Output)
Address of a pointer to an internally allocated 3-dimensional array of size `n_locations` by 22 by 6 containing the anova tables associated with each location. For each location, the 22 by 6 dimensional array corresponds to the anova table for that location. For example, `location_anova_table[(i-1)*132+(j-1)*6 +(k-1)]` contains the value in the *k*th column and *j*th row of the returned anova-table for the *i*th location.

`IMSLS_LOCATION_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)
User provided storage for the array `location_anova_table`.

`IMSLS_ANOVA_ROW_LABELS`, *char* `***anova_row_labels` (Output)
Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the *i*th row of the ANOVA table can be printed with

```
printf("%s", anova_row_labels[i]);
```

The memory associated with `anova_row_labels` can be freed with a single call to `free`(`anova_row_labels`).

`IMSLS_ANOVA_ROW_LABELS_USER`, *char* `*anova_row_labels[]` (Output)
Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels[800]`.

Description

Function `imsls_f_strip_split_plot` is capable of analyzing a wide variety of strip-split plot experiments, also referred to as strip-strip plot experiments. By default, `imsls_f_strip_split_plot` assumes that both strip-plot factors, and split-plots are fixed effects, and the location effects, if any, are random effects. The nature of randomization used in an experiment determines analysis of the data. Two popular forms of randomization in strip-plot and split-plot experiments are illustrated in the following two figures. In both experiments, the strip-plot factor, factor A, has 4 levels that are randomly assigned to a block or field in four strips.

		Factor A Strip-Plots			
		A2	**A1**	**A4**	**A3**
Factor B Strip Plots	**B3**	A2B3	A1B3	A4B3	A3B3
	B1	A2B1	A1B1	A4B1	A3B1
	B2	A2B2	A1B2	A4B2	A3B2

Figure 1 – Strip-Plot Experiment - Strip-Plots Completely Crossed

In the strip-plot experiment, factor B, has 3 levels that are randomly assigned as strips across each of the four levels of factor A. In this case, factors A and B are completely crossed. The randomization applied to factor B is independent of the application of the strip-plots, factor A.

Contrast this to the randomization depicted in Figure 2. In this split-plot experiment, the levels of factor B are nested within each level of factor A whole-plots. Factor B is randomized independently within each level of factor A. Unlike the strip-plot experiment, in the split-plot experiment different levels of factor B appear in the same row.

Whole-Plot Factor			
A2	**A1**	**A4**	**A3**
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

Figure 2 – Split-Plot Experiment – Factor B Split-Plots Nested within Factor A Whole-Plots

A strip-split plot experiment is a strip-plot experiment with a third factor randomized within each level of strip-plot factor A, see Figure 3. The third factor, referred to as the split-plot factor, is randomly assigned to experimental units within each level of strip-plot factor A, see Figure 3. `imsls_f_strip_split_plot` analyzes strip-split plot experiments consisting of two strip-plot factors and one split-plot factor nested within strip-plot factors A and B.

		Factor A Strip Plots			
		A2	**A1**	**A4**	**A3**
Factor B Strip Plots	**B3**	A2B3C2 A2B3C1	A1B3C1 A1B3C2	A4B3C2 A4B3C1	A3B3C2 A3B3C1
	B1	A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B1C2 A4B1C1	A3B1C2 A3B1C1
	B2	A2B2C2 A2B2C1	A1B2C1 A1B2C2	A4B2C1 A4B2C2	A3B2C2 A3B2C1

Figure 3 – Strip-Split Plot Experiment - Split-Plots Nested Within Strip-Plot Factors A

Strip-split plot experiments are closely related to split-split plot experiments, see Figure 4. The main difference between the two is that in strip-split plot experiments, the order of the levels for factor B are not applied randomly across factor A. Each level of factor B is constant across any row. In this example, the entire first row is assigned to the third level of factor B. In the equivalent split-split plot experiment, the levels of factor B are not constant across any row. The levels are randomized within each level of factor A.

Whole Plot Factor A			
A2	**A1**	**A4**	**A3**
A2B3C2 A2B3C1	A1B2C1 A1B2C2	A4B1C2 A4B1C1	A3B3C2 A3B3C1
A2B1C1 A2B1C2	A1B1C1 A1B1C2	A4B3C2 A4B3C1	A3B2C2 A3B2C1
A2B2C2 A2B2C1	A1B3C1 A1B3C2	A4B2C1 A4B2C2	A3B1C2 A3B1C1

Figure 4 – Split-Split Plot Experiment – Sub-Plot Factor C Nested Within Split-Plot Factor B, Nested Within Whole-Plot Factor A

In some studies, a strip-split-plot experiment is replicated at several locations. Function `imsls_f_strip_split_plot` can analyze strip-split plot experiments replicated at multiple locations, even when the number of blocks or replicates at each location might be different different. If only a single replicate or block is used at each location, then location should be treated as a blocking factor, with `n_locations`=1. If `n_locations`=1, it is assumed that either the experiment was conducted at multiple locations, each with a single block, or at a single location with more than one block or replicate at that location. When `n_locations`=1, all entries associated with location in the anova table will contain missing values.

However, if `n_locations`>1, it is assumed the experiment was repeated at multiple locations, with blocking occurring at each location. Although the number of blocks at each location can be different, the number of levels for the strip-plot and split-plot factors strip-plots must be the same at each location. The locations associated with each of the observations in `y` are specified in the argument `locations[]`, which is a required input argument when `n_locations`>1.

By default, locations are assumed to be random effects. Tests involving strip-plots use the interaction between strip-plots and locations as the error term for testing whether there are statistically significant differences among strip-plots. However, this assumes that the interaction of strip-plots and locations is not statistically significant. A test of this assumption is included in the anova table. If any interactions between locations and strip-plot or split-plot factors are statistically significant, then the nature of these interactions should be explored since this impacts the interpretation of the significance of the treatment factors.

Similarly, when locations are assumed to be random effects, tests involving split-plots do not use the split-plot errors pooled across locations. Instead, the error term for split-plots is the interaction between locations and split-plots.

Suppose, for example, that a researcher wanted to conduct an agricultural experiment comparing the effectiveness of 4 fertilizers with 3 seed lots and 3 rates of application. One replicate of the experiment is conducted at each of the 3 farms. That is, only a single field at each location is assigned to this experiment.

Each field is divided into 4 vertical strips and 3 horizontal strips. The vertical strips are randomly assigned to fertilizers and the rows are randomly assigned to application rates. Fertilizers and application rates represent strip-plot factors A and B respectively. Seed lots are randomly assigned to three sub-divisions within each combination of strip-plots.

		Fertilizer Strip Plots			
		F2	**F1**	**F4**	**F3**
Application Rate Strip Plot	**R3**	F2R3S1 F2R3S2 F2R3S3	F1R3S3 F1R3S2 F1R3S1	F4R3S3 F4R3S2 F4R3S1	F3R3S2 F3R3S1 F3R3S3
	R2	F2R1S3 F2R1S1 F2R1S2	F1R1S2 F1R1S3 F1R1S1	F4R1S3 F4R1S1 F4R1S2	F3R1S1 F3R1S2 F3R1S3
	R1	F2R2S1 F2R2S2 F2R2S3	F1R2S1 F1R2S3 F1R2S2	F4R2S2 F4R2S3 F4R2S1	F3R2S3 F3R2S1 F3R2S2

Figure 4 – Strip-Split Plot Experiment – Fertilizer Strip-Plots, Application Rate Strip-Plots, and Seed Lot Split-Plots

In this case, each farm is a blocking factor, fertilizers are factor A strip-plots, fertilizer application rates are factor B strip-plots, and seed lots are split-plots. The input array `rep` would contain integers from 1 to the number of farms.

In summary, `imsls_f_strip_split_plot` can analyze 2x2x2x2=16 different experimental situations, depending upon the settings of:

Example

The experiment was conducted using a 2 x 2 strip_split plot arrangement with each of the four plots divided into 2 sub-divisions that were randomly assigned one of two split-plot levels. This was replicated 3 times producing an experiment with *n* = 2x2x2x3 = 24 observations.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "imsls.h"

void l_printLSD(int n1, int *equalMeans, float *means);
void l_printLSD2Table(int n1, int n2, int* equalMeans, float *means);
void l_printLSD3Table(int n1, int n2, int n3, int* equalMeans, float *means);

void main()
{
  char **anova_row_labels;
  char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ",
                        "Mean\nsquares", "\nF", "\np-value"};
  int i, j, k, l, page_width = 132;

  int n = 24;            /* Total number of observations */
  int n_locations = 1;   /* Number of locations */
  int n_strip_a = 2;     /* Number of Factor A strip-plots within a location */
  int n_strip_b = 2;     /* Number of Factor B strip-plots within a location */
  int n_split   = 2;     /* Number of split-plots within each Factor A strip-plot */
  int block[]={
        1, 1, 1, 1, 1, 1, 1, 1,
        2, 2, 2, 2, 2, 2, 2, 2,
        3, 3, 3, 3, 3, 3, 3, 3};
  int strip_a[]={
        1, 1, 1, 1, 2, 2, 2, 2,
        1, 1, 1, 1, 2, 2, 2, 2,
        1, 1, 1, 1, 2, 2, 2, 2};
  int strip_b[]={
        1, 1, 2, 2, 1, 1, 2, 2,
      1, 1, 2, 2, 1, 1, 2, 2,
      1, 1, 2, 2, 1, 1, 2, 2};
int split[]={
      1, 2, 1, 2, 1, 2, 1, 2,
      1, 2, 1, 2, 1, 2, 1, 2,
      1, 2, 1, 2, 1, 2, 1, 2};
float y[] ={
      30.0, 40.0, 38.9, 38.2,
      41.8, 52.2, 54.8, 58.2,
      20.5, 26.9, 21.4, 25.1,
      26.4, 36.7, 28.9, 35.9,
      21.0, 25.4, 24.0, 23.3,
      34.4, 41.0, 33.0, 34.9};
float alpha = 0.05;
float grand_mean = 0;
float *cv;
float *aov;
float *treatment_means;
float *strip_plot_a_means;
float *strip_plot_b_means;
float *split_plot_means;
float *strip_a_split_plot_means;
float *strip_b_split_plot_means;
float *strip_plot_ab_means;
float *std_err;
int   *equal_means;

aov = imsls_f_strip_split_plot(n, n_locations, n_strip_a, n_strip_b, n_split,
                         block, strip_a, strip_b, split, y,
                         IMSLS_GRAND_MEAN, &grand_mean,
                         IMSLS_CV, &cv,
                         IMSLS_TREATMENT_MEANS,  &treatment_means,
                         IMSLS_STRIP_PLOT_A_MEANS, &strip_plot_a_means,
                         IMSLS_STRIP_PLOT_B_MEANS, &strip_plot_b_means,
                         IMSLS_SPLIT_PLOT_MEANS, &split_plot_means,
                         IMSLS_STRIP_PLOT_A_SPLIT_PLOT_MEANS,
                         &strip_a_split_plot_means,
                         IMSLS_STRIP_PLOT_B_SPLIT_PLOT_MEANS,
                         &strip_b_split_plot_means,
                         IMSLS_STRIP_PLOT_AB_MEANS, &strip_plot_ab_means,
                         IMSLS_STD_ERRORS, &std_err,
                         IMSLS_ANOVA_ROW_LABELS, &anova_row_labels,
                         0);
```

```
/* Output results. */
imsls_page(IMSLS_SET_PAGE_WIDTH, &page_width);
/* Print ANOVA table, without first column. */
imsls_f_write_matrix("   *** ANALYSIS OF VARIANCE TABLE ***",
                     22, 6, aov,
                     IMSLS_WRITE_FORMAT, "%3.0f%3.0f%8.2f%7.2f%7.2f%7.3f",
                     IMSLS_ROW_LABELS, anova_row_labels,
                     IMSLS_COL_LABELS, col_labels,
                     0);

/*
 * Print the various means.
 */
printf("\nGrand mean: %f\n\n", grand_mean);
printf("Coefficient of Variation\n");
printf("  Strip-Plot A:      %9.4f\n", cv[0]);
printf("  Strip-Plot B:      %9.4f\n", cv[1]);
printf("  Split-Plot:        %9.4f\n\n", cv[2]);
l = 0;

/*
 * Print the Treatment Means.
 */
printf("\n\n**********************************************************");
printf("\nTreatment Means\n");
for (i=0; i < n_strip_a; i++){
   for(j=0; j < n_strip_b; j++){
      for(k=0; k < n_split; k++){
         printf("treatment[%d][%d][%d]   %9.4f \n",
                i+1, j+1, k+1, treatment_means[l++]);
      }
   }
}
printf("\nStandard Error for Comparing Two Treatment Means: %f \n(df=%f)\n",
       std_err[9], std_err[19]);
equal_means = imsls_f_multiple_comparisons(n_strip_a*n_strip_b*n_split,
                                           treatment_means, std_err[19],
                                           std_err[9]/sqrt(2.0),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, alpha,
                                           0);
l_printLSD3Table(n_strip_a, n_strip_b, n_split, equal_means, treatment_means);
```

```
  /*
   * Print the Strip-plot A Means.
   */
  printf("\n\n***********************************************************");
  imsls_f_write_matrix("Strip-plot A Means", n_strip_a, 1,
                       strip_plot_a_means, 0);
  printf("\nStandard Error for Comparing Two Strip-Plot A Means: %f
\n(df=%f)\n",
         std_err[0], std_err[10]);
  equal_means = imsls_f_multiple_comparisons(n_strip_a, strip_plot_a_means,
                                             std_err[10], std_err[0]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_printLSD(n_strip_a, equal_means, strip_plot_a_means);

  /*
   * Print Strip-plot B Means.
   */
  printf("\n\n***********************************************************");
  imsls_f_write_matrix("Strip-plot B Means", n_strip_b, 1,
                       strip_plot_b_means, 0);
  printf("\nStandard Error for Comparing Two Strip-Plot B Means: %f \n(df=%f)\n",
         std_err[1], std_err[11]);
  equal_means = imsls_f_multiple_comparisons(n_strip_b, strip_plot_b_means,
                                             std_err[11], std_err[1]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_printLSD(n_strip_b, equal_means, strip_plot_b_means);

  /*
   * Print the Split-plot Means.
   */
  printf("\n\n***********************************************************");
  imsls_f_write_matrix("Split-plot Means", n_split, 1,
                       split_plot_means, 0);
  printf("\nStandard Error for Comparing Two Split-Plot Means: %f \n(df=%f)\n",
         std_err[2], std_err[12]);
  equal_means = imsls_f_multiple_comparisons(n_split, split_plot_means,
                                             std_err[12], std_err[2]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
```

```
l_printLSD(n_split, equal_means, split_plot_means);

/*
 * Print the Strip-plot A by Split-plot Means.
 */
printf("\n\n*************************************************************");
imsls_f_write_matrix("Strip-plot A by Split-plot Means", n_strip_a, n_split,
                     strip_a_split_plot_means, 0);
printf("\nStandard Error for Comparing Two Means: %f \n(df=%f)\n",
       std_err[3], std_err[13]);
equal_means = imsls_f_multiple_comparisons(n_strip_a*n_split,
                                           strip_a_split_plot_means,
                                           std_err[13],
                                           std_err[3]/sqrt(2.0),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, alpha,
                                           0);
l_printLSD2Table(n_strip_a, n_split, equal_means, strip_a_split_plot_means);

/*
 * Print the Strip-plot A by Strip-plot B Means.
 */
printf("\n\n*************************************************************");
imsls_f_write_matrix("Strip-plot A by Strip-plot B Means", n_strip_a,
                     n_strip_b, strip_plot_ab_means, 0);
printf("\nStandard Error for Comparing Two Means: %f \n(df=%f)\n",
       std_err[4], std_err[14]);
equal_means = imsls_f_multiple_comparisons(n_strip_a*n_strip_b,
                                           strip_plot_ab_means, std_err[14],
                                           std_err[4]/sqrt(2.0),
                                           IMSLS_LSD,
                                           IMSLS_ALPHA, alpha,
                                           0);
l_printLSD2Table(n_strip_a, n_strip_b, equal_means, strip_plot_ab_means);

/*
 * Print the Strip-Plot B by Split-Plot Means.
 */
printf("\n\n*************************************************************");
imsls_f_write_matrix("Strip-Plot B by Split-Plot Means", n_strip_b, n_split,
                     strip_b_split_plot_means, 0);
printf("\nStandard Error for Comparing Two Means: %f \n(df=%f)\n",
       std_err[5], std_err[15]);
```

```
  equal_means = imsls_f_multiple_comparisons(n_strip_b*n_split,
                                             strip_b_split_plot_means,
                                             std_err[15], std_err[5]/sqrt(2.0),
                                             IMSLS_LSD,
                                             IMSLS_ALPHA, alpha,
                                             0);
  l_printLSD2Table(n_strip_b, n_split, equal_means, strip_b_split_plot_means);

}
/*
 * Local functions to output  results of means comparisons.
 */
void l_printLSD(int n, int *equalMeans, float *means){
        float x=0.0;
        int i, j, k;
        int iSwitch;
        int *idx;

        idx = (int *) malloc(n * sizeof (int));

        for (k=0; k < n; k++) {
                idx[k] =k+1;
        }
        /* Sort means in ascending order*/
        iSwitch=1;
        while (iSwitch != 0){
                iSwitch = 0;
                for (i = 0; i < n-1; i++){
                        if (means[i] > means[i+1]){
                                iSwitch = 1;
                                x = means[i];
                                means[i] = means[i+1];
                                means[i+1] = x;
                                j = idx[i];
                                idx[i] = idx[i+1];
                                idx[i+1] = j;
                        }
                }
        }
        printf("[group] \t  Mean \t\tLSD Grouping \n");
        for (i=0; i < n; i++){
                printf("  [%d] \t\t%f", idx[i], means[i]);
                for (j=1; j < i+1; j++){
                        if(equalMeans[j-1] >= i+2-j){
                                printf("\t  *");
                        }else{
                                if(equalMeans[j-1]>=0) printf("\t");
                        }
                }
                if (i < n-1 && equalMeans[i]>0) printf("\t  *");
                printf("\n");
        }
        free(idx);
        return;

}
void l_printLSD2Table(int n1, int n2, int *equalMeans, float *means){
        float x=0.0;
        int i, j, k, n;
        int iSwitch;
        int *idx;
        n = n1*n2;

        idx = (int *) malloc(2*n * sizeof (int));
        i = 1;
        j = 1;
        for (k=0; k < n; k++) {
                idx[2*k]   = i;
                idx[2*k+1] = j++;
                if (j > n2){
                        j = 1;
                        i++;
                }
        }

        /* sort means in ascending order*/

        iSwitch=1;
        while (iSwitch != 0){
                iSwitch = 0;
                for (i = 0; i < n-1; i++){
                        if (means[i] > means[i+1]){
                                iSwitch = 1;
                                x = means[i];
                                means[i] = means[i+1];
                        means[i+1] = x;
                        j = idx[2*i];
                        idx[2*i] = idx[2*(i+1)];
                        idx[2*(i+1)] = j;
                        j = idx[2*i+1];
                        idx[2*i+1] = idx[2*(i+1)+1];
                        idx[2*(i+1)+1] = j;
                    }
                }
        }
        printf("[A][B] \tMean \t\tLSD Grouping \n");
        for (i=0; i < n; i++){
                printf("[%d][%d] \t%f", idx[2*i], idx[2*i+1],means[i]);

                for (j=1; j < i+1; j++){
                        if(equalMeans[j-1] >= i+2-j){
                                printf("\t*");
                        }else{
                                if(equalMeans[j-1]>0) printf("\t");
                        }
                }
                if (i < n-1 && equalMeans[i]>0) printf("\t*");
                printf("\n");
        }
        free(idx);
        idx = NULL;
        return;

}
void l_printLSD3Table(int n1, int n2, int n3, int *equalMeans, float *means){
        float x=0.0;
        int i, j, k, m, n;
        int iSwitch;
        int *idx;
        n = n1*n2*n3;

        idx = (int *) malloc(3*n * sizeof (int));
        i = 1;
        j = 1;
        k = 1;
        for (m=0; m < n; m++) {
                idx[3*m]   = i;
                idx[3*m+1] = j;
                idx[3*m+2] = k++;
                if (k > n3){
                        k = 1;
                        j++;
                        if (j > n2){
                                j = 1;
                                i++;
                        }
                }
        }

        /* sort means in ascending order*/

        iSwitch=1;
        while (iSwitch != 0){
                iSwitch = 0;
                for (i = 0; i < n-1; i++){
                        if (means[i] > means[i+1]){
                                iSwitch = 1;
                                x = means[i];
                                means[i] = means[i+1];
                                means[i+1] = x;
                                j = idx[3*i];
                                idx[3*i] = idx[3*(i+1)];
                                idx[3*(i+1)] = j;
                                j = idx[3*i+1];
                                idx[3*i+1] = idx[3*(i+1)+1];
                                idx[3*(i+1)+1] = j;
                                j = idx[3*i+2];
                                idx[3*i+2] = idx[3*(i+1)+2];
                                idx[3*(i+1)+2] = j;
                        }
                }
        }
        printf("[A][B][Split] \t  Mean \t\t  LSD Grouping \n");
        for (i=0; i < n; i++){
                printf("[%d][%d]  [%d] \t%f", idx[3*i], idx[3*i+1], idx[3*i+2],
means[i]);

                for (j=1; j < i+1; j++){
                        if(equalMeans[j-1] >= i+2-j){
                                printf("\t*");
                        }else{
                                if(equalMeans[j-1]>0) printf("\t");
```

```
                    }
                }
                if (i < n-1 && equalMeans[i]>0) printf("\t*");
                printf("\n");
        }
        free(idx);
        return;

}
```

Output

```
                    *** ANALYSIS OF VARIANCE TABLE ***
                                                       Mean
                               ID   DF       SSQ  squares         F  p-value
Location ...................   -1  ...  ........  .......  .......  .......
Blocks .....................   -2    2   1310.28   655.14    14.53    0.061
Strip-Plot A ...............   -3    1    858.01   858.01    40.37    0.024
Location x A ...............   -4  ...  ........  .......  .......  .......
Strip-Plot A Error .........   -5    2     42.51    21.26     1.48    0.385
Split-Plot .................   -6    1    163.80   163.80    41.22    0.003
Split-Plot x A .............   -7    1     11.34    11.34     2.85    0.166
Location x Split-Plot ......   -8  ...  ........  .......  .......  .......
Split-Plot Error ...........   -9    4     15.90     3.97     1.56    0.338
Location x Split-Plot x A ...  -10  ...  ........  .......  .......  .......
Strip-Plot B ...............  -11    1     17.17    17.17     0.47    0.565
Location x B ...............  -12  ...  ........  .......  .......  .......
Strip-Plot B Error .........  -13    2     73.51    36.75     2.85    0.260
A x B ......................  -14    1      1.55     1.55     0.12    0.762
Location x A x B ...........  -15  ...  ........  .......  .......  .......
A x B Error ................  -16    2     25.82    12.91     5.08    0.080
Split-Plot x B .............  -17    1     46.76    46.76    18.39    0.013
Split-Plot x A x B .........  -18    1      0.51     0.51     0.20    0.677
Location x Split-Plot x B ...  -19  ...  ........  .......  .......  .......
Location x Split-Plot x A x B  -20  ...  ........  .......  .......  .......
Split-Plot x A x B Error ....  -21    4     10.17     2.54  .......  .......
Corrected Total ............  -22   23   2577.33  .......  .......  .......

Grand mean: 33.870834

Coefficient of Variation
```

```
  Strip-Plot A:        13.6116
  Strip-Plot B:        17.8986
  Split-Plot:           5.8854


**************************************************************
Treatment Means
treatment[1][1][1]     23.8333
treatment[1][1][2]     30.7667
treatment[1][2][1]     28.1000
treatment[1][2][2]     28.8667
treatment[2][1][1]     34.2000
treatment[2][1][2]     43.3000
treatment[2][2][1]     38.9000
treatment[2][2][2]     43.0000

Standard Error for Comparing Two Treatment Means: 1.302029
(df=4.000000)
[A][B][Split]     Mean            LSD Grouping
[1][1]  [1]     23.833334
[1][2]  [1]     28.100000       *
[1][2]  [2]     28.866669       *
[1][1]  [2]     30.766668       *       *
[2][1]  [1]     34.200001               *
[2][2]  [1]     38.899998
[2][2]  [2]     43.000000                       *
[2][1]  [2]     43.299999                       *


**************************************************************
Strip-plot A Means
  1        27.89
  2        39.85

Standard Error for Comparing Two Strip-Plot A Means: 1.882171
(df=2.000000)
[group]           Mean          LSD Grouping
  [1]           27.891665
  [2]           39.849998


**************************************************************
Strip-plot B Means
```

```
  1        33.03
  2        34.72

Standard Error for Comparing Two Strip-Plot B Means: 2.474972
(df=2.000000)
[group]           Mean          LSD Grouping
  [1]           33.025002         *
  [2]           34.716667         *


*************************************************************
Split-plot Means
 1        31.26
 2        36.48

Standard Error for Comparing Two Split-Plot Means: 0.813813
(df=4.000000)
[group]           Mean          LSD Grouping
  [1]           31.258331
  [2]           36.483334


*************************************************************
Strip-plot A by Split-plot Means
                1            2
   1        25.97        29.82
   2        36.55        43.15

Standard Error for Comparing Two Means: 1.150906
(df=4.000000)
[A][B]  Mean            LSD Grouping
[1][1]  25.966667
[1][2]  29.816668
[2][1]  36.549999
[2][2]  43.149998


*************************************************************
Strip-plot A by Strip-plot B Means
                 1            2
    1        27.30        28.48
    2        38.75        40.95
```

```
Standard Error for Comparing Two Means: 2.074280
(df=2.000000)
[A][B]  Mean            LSD Grouping
[1][1]  27.299997       *
[1][2]  28.483335       *
[2][1]  38.750000               *
[2][2]  40.949997               *


*************************************************************
Strip-Plot B by Split-Plot Means
                1           2
   1        29.02        37.03
   2        33.50        35.93

Standard Error for Comparing Two Means: 0.920673
(df=4.000000)
[A][B]  Mean            LSD Grouping
[1][1]  29.016668
[2][1]  33.500000       *
[2][2]  35.933334       *       *
[1][2]  37.033333               *
```

homogeneity

Conducts Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.

Synopsis

#include `<imsls.h>`

float `* imsls_f_homogeneity` (*int* `n`, *int* `n_treatment`, *int* `treatment[]`, *float* `y[]`,…, 0)

The type *double* is `imsls_d_homogeneity`.

Required Arguments

int `n` (Input)
Number of experimental observations.

int `n_treatment` (Input)
Number of treatments. `n_treatment` must be greater than one.

int `treatment[]` (Input)
An array of length `n` containing the treatment identifiers for each observation in `y`. Each level of the treatment must be assigned a

different integer. `imsls_f_homogeneity` verifies that the number of unique treatment identifiers is equal to `n_treatment`.

float `y[]` (Input)
An array of length `n` containing the experimental observations and any missing values. Missing values can be included in this array, although they are ignored in the analysis. They are indicated by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine`(6) or `imsls_d_machine`(6), depending upon whether single or double precision is being used, respectively.

Return Value

Address of a pointer to the memory location of an array of length 2 containing the *p*-values for Bartletts and Levene's tests.

Synopsis with Optional Arugments

```
#include <imsl.h>

float *imsls_f_homogeneity (int n, int n_treatment,
        int n_treatment[], float y[],
        IMSLS_RETURN_USER, float p_value[]
        IMSLS_LEVENES_MEAN or IMSLS_LEVENES_MEDIAN,
        IMSLS_N_MISSING, int *n_missing,
        IMSLS_CV, float *cv,
        IMSLS_GRAND_MEAN, float *grand_mean,
        IMSLS_TREATMENT_MEANS, float **treatment_means,
        IMSLS_TREATMENT_MEANS_USER, float treatment_means[],
        IMSLS_RESIDUALS, float **residuals,
        IMSLS_RESIDUALS_USER, float residuals[],
        IMSLS_STUDENTIZED_RESIDUALS,
                float **studentized_residuals,
        IMSLS_STUDENTIZED_RESIDUALS_USER,
                float studentized_residuals[],
        IMSLS_STD_DEVS, float **std_devs,
        IMSLS_STD_DEVS_USER, float std_devs[],
        IMSLS_BARTLETTS, float *bartletts,
        IMSLS_LEVENES, float *levenes,
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `p_value[]` (Output)
User defined array of length 2 for storage of the *p*-values from Bartlett's and Levene's tests for homogeneity of variance. The first value returned contains the *p*-value for Bartlett's test and the second value contains the *p*-value for Levene's test.

`IMSLS_LEVENES_MEAN` or `IMSLS_LEVENES_MEDIAN` (Input)

Calculates Levene's test using either the treatment means or medians. `IMSLS_LEVENES_MEAN` indicates that Levene's test is calculated using the mean, and `IMSLS_LEVENES_MEDIAN` indicates that it is calculated using the median.
Default: `IMSLS_LEVENES_MEAN`

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)

Number of missing values, if any, found in `y`. Missing values are denoted with a `NaN` (Not a Number) value in `y`. In these analyses, any missing values are ignored.

`IMSLS_CV`, *float* `*cv` (Output)

The coefficient of variation computed using the grand mean and pooled within treatment standard deviation.

`IMSLS_GRAND_MEAN`, *float* `grand_mean` (Output)

Mean of all the data across every location.

`IMSLS_TREATMENT_MEANS`, *float* `**treatment_means` (Output)

Address of a pointer to an internally allocated array of size `n_treatment` containing the treatment means.

`IMSLS_TREATMENT_MEANS_USER`, *float* `treatment_means[]` (Output)

Storage for the array `treatment_means`, provided by the user.

`IMSLS_RESIDUALS`, *float* `**residuals` (Output)

Address of a pointer to an internally allocated array of length `n` containing the residuals for non-missing observations. The ordering of the values in this array corresponds to the ordering of values in `y` and identified by the values in `treatments`.

`IMSLS_RESIDUALS_USER`, *float* `residuals[]` (Output)

Storage for the array `residuals`, provided by the user.

`IMSLS_STUDENTIZED_RESIDUALS`, *float* `**studentized_residuals` (Output)

Address of a pointer to an internally allocated array of length `n` containing the studentized residuals for non-missing observations. The ordering of the values in this array corresponds to the ordering of values in `y` and identified by the values in `treatments`.

`IMSLS_STUDENTIZED_RESIDUALS_USER`, *float* `studentized_residuals[]` (Output)

Storage for the array `studentized_residuals`, provided by the user.

`IMSLS_STD_DEVS`, *float* `**std_devs` (Output)

Address of a pointer to an internally allocated array of length `n_treatment` containing the treatment standard deviations.

`IMSLS_STD_DEVS_USER`, *float* `std_devs[]` (Output)

Storage for the array `std_devs`, provided by the user.

`IMSLS_BARTLETTS`, *float* `*bartletts` (Output)
Test statistic for Bartlett's test.

`IMSLS_LEVENES`, *float* `*levenes` (Output)
Test statistic for Levene's test.

Description

Traditional analysis of variance assumes that variances within treatments are equal. This is referred to as homogeneity of variance. The function `imsls_f_homogeneity` conducts both the Bartlett's and Levene's tests for this assumption:

$$H_o : \sigma_1 = \sigma_2 = \cdots = \sigma_t$$

versus

$$H_a : \sigma_i \neq \sigma_j$$

for at least one pair (i≠j), where *t*=`n_treatments`.

Bartlett's test, Bartlett (1937), uses the test statistic:

$$\chi^2 = \frac{M}{C}$$

where

$$M = N \cdot \ln(S_p^2) - \sum n_i \ln(S_i^2),\ N = \sum_{i=1}^{t} n_i,\ S_p^2 = \frac{\sum_{i=1}^{t}(n_i - 1)S_i^2}{\sum_{i=1}^{t}(n_i - 1)}$$

$$C = 1 + \frac{1}{3(t-1)}\left[\sum \frac{1}{n_i} - \frac{1}{N}\right]$$

and S_i^2 is the variance of the n_i non-missing observations in the *i*th treatment.

S_p^2 is referred to as the pooled variance, and it is also known as the error mean squares from a 1-way analysis of variance.

If the usual assumptions associated with the analysis of variance are valid, then Bartlett's test statistic is a chi-squared random variable with degrees of freedom equal to *t*-1.

The original Levene's test, Levene (1960) and Snedecor & Cochran (1967), uses a different test statistic, F_0, equal to:

$$F_0 = \frac{\sum_{i=1}^{t} n_i \left(\bar{z}_{i.} - \bar{z}_{..}\right)^2 /(t-1)}{\sum_{i=1}^{t}\sum_{j=1}^{n_i}\left(z_{ij} - \bar{z}_{i.}\right)^2 /(N-t)},$$

where

$$z_{ij} = | x_{ij} - \bar{x}_{i.} |,$$

x_{ij} is the *j*th observation from the ith treatment and $\bar{x}_{i.}$ is the mean for the *i*th treatment. Conover, Johnson, and Johnson (1981) compared over 50 similar tests for homogeneity and concluded that one of the best tests was Levene's test when the treatment mean, $\bar{x}_{i.}$ is replaced with the treatment median, $\tilde{x}_{i.}$. This version of Levene's test can be requested by setting `IMSLS_LEVENES_MEDIAN`. In either case, Levene's test statistic is treated as a F random variable with numerator degrees of freedom equal to (*t*-1) and denominator degrees of freedom (N-*t*).

The residual for the jth observation within the ith treatment, e_{ij}, returned from `IMSLS_RESIDUALS` is unstandarized, i.e. $e_{ij} = x_{ij} - \bar{x}_i$. For investigating problems of homogeneity of variance, the studentized residuals returned by `IMSLS_STUDENTIZED_RESIDUALS` are recommended since they are standarzied by the standard deviation of the residual. The formula for calculating the studentized residual is:

$$\tilde{e}_{ij} = \frac{e_{ij}}{\sqrt{S_p^2(1-\frac{1}{n_i})}},$$

where the coefficient of variation, returned from `IMSLS_CV`, is also calculated using the pooled variance and the grand mean $\bar{x}_{..} = \sum_i \sum_j x_{ij}$:

$$CV = \frac{100 \bullet \sqrt{S_p^2}}{\bar{x}..}$$

Example

This example applies Bartlett's and Levene's test to verify the homogeneity assumption for a one-way analysis of variance. There are eight treatments, each with 3 replicates for a total of 24 observations. The estimated treatment standard deviations range from 5.35 to 13.17.

In this case, Bartlett's test is not statistically significant for a stated significance level of .05; whereas Levene's test is significant with $p = 0.006$.

```
#include "imsls.h"
```

```
void ex_homog_b()
{
  int i, page_width = 132;

  int n = 24;
  int n_treatment = 8;
  int treatment[]={
    1, 2, 3, 4, 5, 6, 7, 8,
    1, 2, 3, 4, 5, 6, 7, 8,
    1, 2, 3, 4, 5, 6, 7, 8};
  float y[] ={
    30.0, 40.0, 38.9, 38.2,
    41.8, 52.2, 54.8, 58.2,
    20.5, 26.9, 21.4, 25.1,
    26.4, 36.7, 28.9, 35.9,
    21.0, 25.4, 24.0, 23.3,
    34.4, 41.0, 33.0, 34.9};

  float bartletts;
  float levenes;
  float grand_mean;
  float cv;
  float *treatment_means=NULL;
  float *residuals=NULL;
  float *studentized_residuals=NULL;
  float *std_devs=NULL;
  int n_missing = 0;
  float *p;

  p = imsls_f_homogeneity(n, n_treatment, treatment, y,
                      IMSLS_BARTLETTS, &bartletts,
                      IMSLS_LEVENES, &levenes,
                      IMSLS_LEVENES_MEDIAN,
                      IMSLS_N_MISSING, &n_missing,
                      IMSLS_GRAND_MEAN, &grand_mean,
                      IMSLS_CV, &cv,
                      IMSLS_TREATMENT_MEANS, &treatment_means,
                      IMSLS_STD_DEVS, &std_devs,
                      0);

  printf("\n\n\n *** Bartlett\'s Test ***\n\n");
  printf("Bartlett\'s p-value        = %10.3f\n", p[0]);
  printf("Bartlett\'s test statistic = %10.3f\n", bartletts);
  printf("\n\n\n *** Levene\'s Test ***\n\n");
  printf("Levene\'s p-value        = %10.3f\n", p[1]);
  printf("Levene\'s test statistic = %10.3f\n", levenes);

  imsls_f_write_matrix("Treatment means", n_treatment, 1, treatment_means, 0);
  imsls_f_write_matrix("Treatment std devs", n_treatment, 1, std_devs, 0);
  printf("\ngrand_mean = %10.3f\n", grand_mean);
  printf("cv         = %10.3f\n", cv);
  printf("n_missing  = %d\n", n_missing);

}
```

Output

```
 *** Bartlett's Test ***

Bartlett's p-value        =      0.056
Bartlett's test statistic =      2.257



 *** Levene's Test ***

Levene's p-value        =      0.006
Levene's test statistic =      0.135

Treatment means
1        23.83
2        30.77
3        28.10
4        28.87
5        34.20
6        43.30
7        38.90
8        43.00

Treatment std devs
  1         5.35
  2         8.03
  3         9.44
  4         8.13
  5         7.70
  6         8.00
  7        13.92
  8        13.17

grand_mean =      33.871
cv         =      28.378
n_missing  = 0
```

multiple_comparisons

Performs multiple comparisons of means using one of Student-Newman-Keuls, LSD, Bonferroni, Tukey's, or Duncan's MRT procedures.

Synopsis

#include `<imsls.h>`

int `*imsls_f_multiple_comparisons` (*int* `n_groups`, *float* `means[]`,
int `df`, *float* `std_error`, ..., 0)

The type *double* function is `imsls_d_multiple_comparisons`.

Required Arguments

int `n_groups` (Input)
Number of groups i.e., means, being compared.

float `means[]` (Input)
Array of length `n_groups` containing the means.

int `df` (Input)
Degrees of freedom associated with `std_error`.

float `std_error` (Input)
Effective estimated standard error of a mean. In fixed effects models, `std_error` equals the estimated standard error of a mean. For example, in a one-way model

$$\texttt{std\_error} = \sqrt{\frac{s^2}{n}}$$

where s^2 is the estimate of σ^2 and n is the number of responses in a sample mean. In models with random components, use

$$\texttt{std\_error} = \frac{sedif}{\sqrt{2}}$$

where *sedif* is the estimated standard error of the difference of two means.

Return Value

Pointer to the array of length `n_groups` – 1 indicating the size of the groups of means declared to be equal. Value `equal_means` [`I`] = `J` indicates the `I`-th smallest mean and the next `J` – 1 larger means are declared equal. Value `equal_means` [`I`] = 0 indicates no group of means starts with the `I`-th smallest mean.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_multiple_comparisons` (*int* `n_groups`, *float* `means []`,
    int `df`, *float* `std_error`,
    `IMSLS_ALPHA`, *float* `alpha`,
    `IMSLS_SNK`, *or*
    `IMSLS_LSD`, *or*
    `IMSLS_TUKEY`, *or*
    `IMSLS_BONFERRONI`,
    `IMSLS_RETURN_USER`, *int* `*equal_means`,
    0)

Optional Arguments

`IMSLS_ALPHA`, *float* `alpha` (Input)
Significance level of test. Argument alpha must be in the interval [0.01, 0.10].
Default: `alpha` = 0.01

`IMSLS_RETURN_USER`, *int* `*equal_means` (Output)
If specified, `equal_means` is an array of length `n_groups` – 1 specified by the user. On return, `equal_means` contains the size of the groups of means declared to be equal. Value `equal_means` [`I`] = `J` indicates the *i*th smallest mean and the next `J` – 1 larger means are declared equal. Value `equal_means` [`I`] = 0 indicates no group of means starts with the *i*th smallest mean.

`IMSLS_SNK`, *or*

`IMSLS_LSD`, *or*

`IMSLS_TUKEY`, *or*

`IMSLS_BONFERRONI`, *or*

Argument	Method
`IMSLS_SNK`	Student-Newman-Keuls (default)
`IMSLS_LSD`	Least significant difference
`IMSLS_TUKEY`	Tukey's *w*-procedure, also called the honestly significant difference procedure.
`IMSLS_BONFERRONI`	Bonferroni *t* statistic

Description

Function `imsls_f_multiple_comparisons` performs a multiple comparison analysis of means using one of Student-Newman-Keuls, LSD, Bonferroni, or Tukey's procedures. The null hypothesis is equality of all possible ordered subsets of a set of means. The methods are discussed in many elementary statistics texts, e.g., Kirk (1982, pp. 123–125).

The output consists of an array of `n_groups` –1 integers that describe grouping of means that are considered not statistically significantly different.

For example, if `n_groups`=4 and the returned array is equal to {0, 2, 2} then we conclude that:

1. The smallest mean is significantly different from the others,
2. The second and third smallest means are not significantly different from one another,
3. The second and fourth means are significantly different
4. The third and fourth means are not significantly different from one another.

These relationships can be depicted graphically as three groups of means:

Smallest Mean	Group 1	Group 2	Group 3
1	x		
2		x	
3		x	x
4			x

Examples

Example 1

A multiple-comparisons analysis is performed using data discussed by Kirk (1982, pp. 123–125). The results show that there are three groups of means with three separate sets of values: (36.7, 40.3, 43.4), (40.3, 43.4, 47.2), and (43.4, 47.2, 48.7).

In this case, the ordered means are {36.7, 40.3, 43.4, 47.2, 48.7} corresponding to treatments {1, 5, 3, 4, 2}. Since the output table is:

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

we can say that within each of these three groups, means are not significantly different from one another.

Treatment	Mean	Group 1	Group 2	Group 3
1	36.7	x		
5	40.3	x	x	
3	43.4	x	x	x
4	47.2		x	x
2	48.7			x

```
#include <imsls.h>

void main ()
{
    int n_groups       =  5;
    int df             = 45;
    float std_error    = 1.6970563;
    float means[5]     = {36.7, 48.7, 43.4, 47.2, 40.3};
    int *equal_means;
                       /* Perform multiple comparisons tests */
    equal_means = imsls_f_multiple_comparisons(n_groups, means, df,
        std_error, 0);
                       /* Print results */
    imsls_i_write_matrix("Size of Groups of Means", 1, n_groups-1,
        equal_means, 0);

}
```

Output

```
Size of Groups of Means
     1   2   3   4
     3   3   3   0
```

Example 2

This example uses the same data as the previous example but also uses additional methods by specifying optional arguments.

Example 2 uses the same data as Example 1: Ordered treatment means correspond to treatment order {1,5,3,4,2}.

The table produced for Bonferroni is:

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

Thus, these are two groups of similar means.

Treatment	Mean	Group 1	Group 2
1	36.7	x	
5	40.3	x	x
3	43.4	x	x
4	47.2		x
2	48.7		x

```
#include <imsls.h>
void main()
{
    int n_groups       =  5;
    int df             = 45;
    float std_error    = 1.6970563;
    float means[5]     = {36.7, 48.7, 43.4, 47.2, 40.3};
    int equal_means[4];

  /* Student-Newman-Keuls */
       imsls_f_multiple_comparisons(n_groups, means, df, std_error,
       IMSLS_RETURN_USER, equal_means, 0);
       imsls_i_write_matrix("SNK         ", 1, n_groups-1, equal_means, 0);

      /* Bonferroni */
       imsls_f_multiple_comparisons(n_groups, means, df, std_error,
       IMSLS_BONFERRONI,
       IMSLS_RETURN_USER, equal_means,
       0);
       imsls_i_write_matrix("Bonferonni  ", 1, n_groups-1, equal_means, 0);
```

```
        /* Least Significant Difference */
         imsls_f_multiple_comparisons(n_groups, means, df, std_error,
         IMSLS_LSD,
         IMSLS_RETURN_USER, equal_means,
         0);
         imsls_i_write_matrix("LSD         ", 1, n_groups-1, equal_means, 0);

        /* Tukey's */
         imsls_f_multiple_comparisons(n_groups, means, df, std_error,
         IMSLS_TUKEY,
         IMSLS_RETURN_USER, equal_means,
         0);
         imsls_i_write_matrix("Tukey       ", 1, n_groups-1, equal_means, 0);


}
```

Output

```
SNK
1   2   3   4
3   3   3   0

Bonferonni
1   2   3   4
3   4   0   0

LSD
1   2   3   4
2   2   3   0

Tukey
1   2   3   4
3   3   3   0
```

yates

Estimates missing observations in designed experiments using Yate's method.

Synopsis

#include `<imsls.h>`

int `imsls_f_yates`(*int* `n`, *int* `n_independent`, *float* `x[]`,…, 0)

The type *double* function is `imsls_d_yates`.

Required Arguments

int `n` (Input)
 Number of observations.

int n_independent (Input)
 Number of independent variables.

float `x[]` (Input/Output)

A `n` by (`n_independent+1`) 2-dimensional array containing the experimental observations and missing values. The first `n_independent` columns contain values for the independent variables and the last column contains the corresponding observations for the dependent variable or response. The columns assigned to the independent variables should not contain any missing values. Missing values are included in this array by placing a `NaN` (not a number) in the last column of `x`. The `NaN` value can be set using either the function `imsls_f_machine(6)` or `imsls_d_machine(6)`, depending upon whether single or double precision is being used, respectively. Upon successful completion, missing values are replaced with estimates calculated using Yates' method.

Return Value

The number of missing values replaced with estimates using the Yates procedure. A negative return value indicates that the routine was unable to successfully estimate all missing values. Typically this occurs when all of the observations for a particular treatment combination are missing. In this case, Yate's missing value method does not produce a unique set of missing value estimates.

Synopsis with Optional Arugments

#include `<imsls.h>`

```
int  imsls_f_yates (int n, int n_independent, float x[],
        IMSLS_DESIGN, int design,
        IMSLS_INITIAL_ESTIMATES, int n_missing,
               float initial_estimates[],
        IMSLS_GET_SS, float get_ss (int n, int n_independent,
        int n_levels[], float dataMatrix[]),
        IMSLS_GRAD_TOL, float grad_tol,
        IMSLS_STEP_TOL, float step_tol,
        IMSLS_MAX_ITN, int **itmax,
        IMSLS_MISSING_INDEX, int **missing_index[],
        IMSLS_MISSING_INDEX_USER, int missing_index[],
        IMSLS_ERROR_SS, float *error_ss,
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *int* n_missing (Output)

The number of missing values replaced with Yate's estimates. A negative return value indicates that the routine was unable to successfully estimate all missing values.

`IMSLS_DESIGN`, *int* `design` (Input)

An integer indicating whether a custom or standard design is being used.

The association of values for this variable and standard designs is described in the following table:

Design	Description
0	*CRD* – Completely Randomized Design. The input matrix, x, is assumed to have only two columns. The first is used to contain integers identifying the treatments. The second column should contain corresponding observations for the dependent variable. In this case, `n_independent`=1. Default value when `n_independent`=1.
1	*RCBD* – Randomized Complete Block Design. The input matrix is assumed to have only three columns. The first is used to contain the treatment identifiers and the second the block identifiers. The last column contains the corresponding observations for the dependent variable. In this case, `n_independent`=2. This is the default value when `n_independent`=2.
2	Another design. In this case, the function `get_ss` is a required input. The design matrix is passed to that routine. Initial values for missing observations are set to the grand mean of the data, unless initial values are specified using `IMSLS_INITIAL_ESTIMATES`.

Default: `design`=0 or `design`=1, depending upon whether `n_independent`=1 or 2 respectively. If `n_independent`>2, then `design` must be set to 2, and `get_ss` must be provided as input to `imsls_f_yates`.

`IMSLS_INITIAL_ESTIMATES`, *int* `n_missing`, *float* `initial_estimates[]` (Input)
Initial estimates for the missing values. Argument `n_missing` is the number of missing values. Argument `initial_estimates` is an array of length `n_missing` containing the initial estimates.
Default: For `design`=0 and `design`=1, the initial estimates are calculated using the Yates formula for those designs. For `design`=2, the mean of the non-missing observations is used as the initial estimate for all missing values.

`IMSLS_MAX_ITN`, *int* `itmax` (Input)
Maximum number of iterations in the optimization routine for finding the missing value estimates that minimize the error sum of squares in the analysis of variance.
Default: `itmax` = 500.

`IMSLS_GET_SS`, *float* `get_ss`(*int* `n`, *int* `n_independent`, *int* `n_levels[]`, *float* `dataMatrix[]`) (Input/Output)
A user-supplied function that returns the error sum of squares calculated

using the n by (`n_independent`+1) matrix `dataMatrix`. `imsls_f_yates` calculates the error sum of squares assuming that `dataMatrix` contains no missing observations. In general, `dataMatrix` should be equal to the input matrix `x` with missing values replaced by estimates. `imsls_f_yates` is required input when `design`=2. The array `n_levels` should be of length `n_independent` and contain the number of levels associated with each of the first `n_independent` columns in the `dataMatrix` and `x` arrays.

`IMSLS_GRAD_TOL`, *float* `grad_tol` (Input)
Scaled gradient tolerance used to determine whether the difference between the error sum of squares is small enough to stop the search for missing value estimates.

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

`IMSLS_STEP_TOL`, *float* `step_tol` (Input)
Scaled step tolerance used to determine whether the difference between missing value estimates is small enough to stop the search for missing value estimates.
Default: `step_tol` = $\varepsilon^{2/3}$, where ε is the machine precision.

`IMSLS_MISSING_INDEX`, *int* `*missing_index` (Output)
An array of length `n_missing` containing the indices for the missing values in `x`. The number of missing values, `n_missing`, is the return value of `imsls_f_yates`.

`IMSLS_MISSING_INDEX_USER`, *int* `missing_index[]` (Output)
Storage for the array `missing_index`, provided by the user.

`IMSLS_ERROR_SS`, *float* `*errr_ss` (Output)
The value of the error sum of squares calculated using the missing value estimates. If `design`=2 then this is equal to the value returned from `get_ss` using the Yates missing value estimates.

Description

Several functions for analysis of variance require balanced experimental data, i.e. data containing no missing values within a block and an equal number of replicates for each treatment. If the number of missing observations in an experiment is smaller than the Yates method as described in Yates (1933) and Steel and Torrie (1960), can be used to estimate the missing values. Once the missing values are replaced with these estimates, the data can be passed to an analysis of variance that requires balanced data.

The basic principle behind the Yates method for estimating missing observations is to replace the missing values with values that minimize the error sum of squares in the analysis of variance. Since the error sum of squares depends upon the underlying model for the analysis of variance, the Yates formulas for estimating missing values vary from anova to anova.

Consider, for example, the model underlying experiments conducted using a completely randomized design. If y_{ij} is the Ith observation for the ith treatment then the error sum of squares for a CRD is calculated using the following formula:

$$SSE = \sum_{i=1}^{t} \sum_{j=1}^{r} \left(y_{ij} - \overline{y}_{i.} \right)^2 \; where \, \overline{y}_{i.} \text{ is the } i\text{th treatment mean}.$$

If an observation y_{ij} is missing then SSE is minimized by replacing that missing observation with the estimate

$$\hat{x}_{ij} = \overline{y}_{i.} .$$

For a randomized complete block design (RCBD), the calculation for estimating a single missing observation can be derived from the RCBD error sum of squares:

$$SSE = \sum_{i=1}^{t} \sum_{j=1}^{r} \left(y_{ij} - \overline{y}_{i.} - \overline{y}_{.j} + \overline{\overline{y}}_{..} \right)^2$$

If only a single observation, y_{ij}, is missing from the *j*th block and *i*th treatment, the estimate for this missing observation can be derived by solving the equation:

$$\hat{x}_{ij} = \overline{y}_{i.} + \overline{y}_{.j} - \overline{\overline{y}}_{..} .$$

The solution is referred to as the Yates formula for a RCBD:

$$\hat{x}_{ij} = \frac{t \cdot y_{.j} + r \cdot y_{i.} - y_{..}}{(r-1)(t-1)}, \text{ where}$$

r=`n_blocks`, *t*=`n_treatments`, y_i=total of all non-missing observations from the ith treatment, $y_{.j}$ =total of all non-missing observations from the *j*th block, and *y*=total of all non-missing observations.

If more than one observation is missing, `imsls_f_yates` minimization procedure is used to estimate missing values. For a CRD, all missing observations are set equal to their corresponding treatment means calculated using the non-missing observations. That is, $\hat{x}_{ij} = \overline{y}_{i.}$.

For RCBD designs with more than one missing value, Yate's formula for estimating a single missing observation is used to obtain initial estimates for all missing values. These are passed to a function minimization routine to obtain the values that minimize SSE.

For other designs, specify `design`=2 and `IMSLS_GET_SS`. The function `get_ss` is used to obtain the Yates missing value estimates by selecting the estimates that minimize sum of squares returned by `get_ss`. When called, `get_ss` calculates the error sum of squares at each iteration assuming that the data matrix it receives is balanced and contains no missing values.

Example

Missing values can occur in any experiment. Estimating missing values via the Yates method is usually done by minimizing the error sum of squares for that experiment. If only a single observation is missing and there is an analytical formula for calculating the error sum of squares then a formula for estimating the missing value is fairly easily derived. Consider for example a split-plot experiment with a single missing value.

Suppose, for example, that x_{ijk}, the observation for the *i*th whole-plot, *j*th split plot and *k*th block is missing. Then the estimate for a single missing observation in the *i*th whole plot is equal to:

$$Y = \frac{r \cdot W + s \cdot x_{ij.} - x_{i..}}{(r-1)(s-1)}, \text{ where}$$

r = number of blocks, s = number of split-plots, W = total of all non-missing values in same block as the missing observation, $x_{ij.}$ = total of the non-missing observations across blocks of observations from *i*th whole-plot factor level and the *j*th split-plot level, and $x_{i..}$ = the total of all observations, across split-plots and blocks of the non-missing observations for the *i*th whole plot.

If more than a single observation is missing, then an iterative solution is required to obtain missing value estimates that minimize the error sum of squares.

Function `imsls_f_yates` simplifies this procedure. Consider, for example, a split-plot experiment conducted at a single location using fixed-effects whole and split plots. If there are no missing values, then the error sum of squares can be calculated from a 3-way analysis of variance using whole-plot, split-plot and blocks as the 3 factors. For balanced data without missing values, the errors sum of squares would be equal to the sum of the 3-way interaction between these factors and the split-plot by block interaction.

Calculating the error sum of squares using this 3-way analysis of variance is achieved using the `anova_factorial` routine.

```
float get_ss(int n, int n_independent, int *n_levels, float *x)
{
  /* This routine assumes that the first three columns of dataMatrix   */
  /* contain the whole-plot,split-plot and block identifiers in that   */
  /* order.  The last column of this matrix, the fourth column, must   */
  /* contain the observations from the experiment.  It is assumed that */
  /* dataMatrix is balanced and does not contain any missing           */
  /* observations.                                                     */

  int i;
  float errorSS, pValue;
  float *test_effects = NULL;
  float *anova_table = NULL;
  float responses[24];
  /* Copy responses from the last column of x into a 1-D array        */
  /* as expected by imsls_f_anova_factorial.                           */

  for (i=0;i<n;i++) {
    responses[i] = x[i*(n_independent+1)+n_independent];
  }
  /* Compute the error sum of squares.                                 */
  pValue = imsls_f_anova_factorial(n_independent, n_levels, responses,
                           IMSLS_TEST_EFFECTS, &test_effects,
                           IMSLS_ANOVA_TABLE, &anova_table,
                           IMSLS_POOL_INTERACTIONS, 0);
  errorSS = anova_table[4] + test_effects[21];

  /* Free memory returned by imsls_f_anova_factorial.                 */
  if (test_effects != NULL) free(test_effects);
  if (anova_table != NULL) free(anova_table);
  return errorSS;
}
```

The above function is passed to the `imsls_f_yates` as an argument, together with a matrix containing the data for the split-plot experiment. For this example, the following data matrix obtained from an agricultural experiment will be used. In this experiment, 4 whole plots were randomly assigned to two 2 blocks. Whole-plots were subdivided into 2 split-plots. The whole-plot factor consisted of 4 different seed lots, and the split-plot factor consisted of 2 seed protectants. The data matrix of this example is a n=24 by 4 matrix with two missing observations.

$$X = \left[\begin{array}{ccc|c}
1 & 1 & 1 & NaN \\
1 & 2 & 1 & 53.8 \\
1 & 3 & 1 & 49.5 \\
1 & 1 & 2 & 41.6 \\
1 & 2 & 2 & NaN \\
1 & 3 & 2 & 53.8 \\
2 & 1 & 1 & 53.3 \\
2 & 2 & 1 & 57.6 \\
2 & 3 & 1 & 59.8 \\
2 & 1 & 2 & 69.6 \\
2 & 2 & 2 & 69.6 \\
2 & 3 & 2 & 65.8 \\
3 & 1 & 1 & 62.3 \\
3 & 2 & 1 & 63.4 \\
3 & 3 & 1 & 64.5 \\
3 & 1 & 2 & 58.5 \\
3 & 2 & 2 & 50.4 \\
3 & 3 & 2 & 46.1 \\
4 & 1 & 1 & 75.4 \\
4 & 2 & 1 & 70.3 \\
4 & 3 & 1 & 68.8 \\
4 & 1 & 2 & 65.6 \\
4 & 2 & 2 & 67.3 \\
4 & 3 & 2 & 65.3
\end{array}\right]$$

The following program uses these data with `imsls_f_yates` to replace the two missing values with Yates estimates.

```
#include <stdlib.h>
#include "imsls.h"

float get_ss(int n, int n_independent, int *n_levels, float *x);

#define N 24
```

```
#define N_INDEPENDENT 3

void main()
{
  char *col_labels[] = {" ", "Whole", "Split", "Block", " "};
  int i;
  int n = N;
  int n_independent = N_INDEPENDENT;
  int whole[N]={1,1,1,1,1,1,
             2,2,2,2,2,2,
             3,3,3,3,3,3,
             4,4,4,4,4,4};
  int split[N]={1,2,3,1,2,3,
             1,2,3,1,2,3,
             1,2,3,1,2,3,
             1,2,3,1,2,3};
  int block[N]={1,1,1,2,2,2,
             1,1,1,2,2,2,
             1,1,1,2,2,2,
             1,1,1,2,2,2};
  float y[N] ={0.0,  53.8, 49.5, 41.6, 0.0,  53.8,
             53.3, 57.6, 59.8, 69.6, 69.6, 65.8,
             62.3, 63.4, 64.5, 58.5, 50.4, 46.1,
             75.4, 70.3, 68.8, 65.6, 67.3, 65.3};

  float x[N][N_INDEPENDENT+1];
  float error_ss;
  int *missing_idx;
  int n_missing;

  /* Set the first and fifth observations to missing values. */
  y[0] = imsls_f_machine(6);
  y[4] = imsls_f_machine(6);

  /* Fill the array x with the classification variables and observations. */
  for (i=0;i<n; i++) {
    x[i][0] = (float)whole[i];
    x[i][1] = (float)split[i];
    x[i][2] = (float)block[i];
    x[i][3] = y[i];
  }
  /* Sort the data since imsls_f_anova_factorial expects sorted data. */
  imsls_f_sort_data(n, n_independent+1, (float*)x, 3, 0);

  n_missing = imsls_f_yates(n, n_independent, (float *)&(x[0][0]),
                        IMSLS_DESIGN, 2,
                        IMSLS_GET_SS, get_ss,
                         IMSLS_ERROR_SS, &error_ss,
                         IMSLS_MISSING_INDEX, &missing_idx,
                         0);
  printf("Returned error sum of squares = %f\n\n", error_ss);
  printf("Missing values replaced: %d\n", n_missing);
  printf("Whole     Split    Block    Estimate\n");
  for (i=0;i<n_missing;i++) {
    printf("%3d        %3d      %3d      %7.3f\n",
           (int)x[missing_idx[i]][0],
           (int)x[missing_idx[i]][1],
           (int)x[missing_idx[i]][2],
           x[missing_idx[i]][n_independent]);
  }
  imsls_f_write_matrix("Sorted x, with estimates", n, n_independent+1,
                       (float*)x,
                       IMSLS_WRITE_FORMAT, "%-4.0f%-4.0f%-4.0f%5.2f",
                       IMSLS_COL_LABELS, col_labels,
                       IMSLS_NO_ROW_LABELS, 0);

}

float get_ss(int n, int n_independent, int *n_levels, float *x)
{
  int i;
  float errorSS, pValue;
  float *test_effects = NULL;
  float *anova_table = NULL;
  float responses[24];
  /*
   * Copy responses from the last column of x into a 1-D array
   * as expected by imsls_f_anova_factorial.
   */
  for (i=0;i<n;i++) {
    responses[i] = x[i*(n_independent+1)+n_independent];
  }
  /*
   * Compute the error sum of squares.
   */
  pValue = imsls_f_anova_factorial(n_independent, n_levels, responses,
                              IMSLS_TEST_EFFECTS, &test_effects,
                              IMSLS_ANOVA_TABLE, &anova_table,
                              IMSLS_POOL_INTERACTIONS, 0);
  errorSS = anova_table[4] + test_effects[21];

  /* Free memory returned by imsls_f_anova_factorial. */
  if (test_effects != NULL) free(test_effects);
  if (anova_table != NULL) free(anova_table);
  return errorSS;
}
```

After running this code to replace missing values with Yates estimates, it would be followed by a call to the split-plot analysis of variance:

```
float *aov_table, y[24];
int expunit[24], whole[24], split[24];
for(int i=0; i < 24; i++){whole[i]  = x[i];    split[i] = x[i+24];
                          expunit[i]= x[i+48]; y[i]     = x[i+72];}
float aov_table = imsls_f_split_plot (24, 1, 4, 3, expunit, whole,
                                     split, y[], 0);
```

Output

```
Returned error sum of squares = 95.620010

Missing values replaced: 2
Whole     Split    Block    Estimate
  1          1        1       37.300
  1          2        2       58.100

  Sorted x, with estimates
   Whole  Split  Block
    1      1      1     37.30
    1      1      2     41.60
    1      2      1     53.80
    1      2      2     58.10
    1      3      1     49.50
    1      3      2     53.80
    2      1      1     53.30
    2      1      2     69.60
    2      2      1     57.60
    2      2      2     69.60
    2      3      1     59.80
    2      3      2     65.80
    3      1      1     62.30
    3      1      2     58.50
    3      2      1     63.40
    3      2      2     50.40
    3      3      1     64.50
    3      3      2     46.10
    4      1      1     75.40
    4      1      2     65.60
    4      2      1     70.30
```

```
        4       2       2      67.30
        4       3       1      68.80
        4       3       2      65.30
```

Chapter 5: Categorical and Discrete Data Analysis

Routines

Usage Notes

Routine `imsls_f_contingency_table` (page 404) computes many statistics of interest in a two-way table. Statistics computed by this routine includes the usual chi-squared statistics, measures of association, Kappa, and many others. Exact probabilities for two-way tables can be computed by `imsls_f_exact_enumeration` (page 417), but this routine uses the total enumeration algorithm and, thus, often uses orders of magnitude more computer time than `imsls_f_exact_network` (page 419), which computes the same probabilities by use of the network algorithm (but can still be quite expensive).

The routine `imsls_f_categorical_glm` (page 425) in the second section is concerned with generalized linear models (see McCullagh and Nelder 1983) in discrete data. This routine can be used to compute estimates and associated statistics in probit, logistic, minimum extreme value, Poisson, negative binomial (with known number of successes), and logarithmic models. Classification variables as well as weights, frequencies and additive constants may be used so that general linear models can be fit. Residuals, a measure of influence, the coefficient estimates, and other statistics are returned for each model fit. When infinite parameter estimates are required, extended maximum likelihood estimation may be used. Log-linear models can be fit in `imsls_f_categorical_glm` through the use of Poisson regression models.

Results from Poisson regression models involving structural and sampling zeros will be identical to the results obtained from the log-linear model routines but will be fit by a quasi-Newton algorithm rather than through iterative proportional fitting.

contingency_table

Performs a chi-squared analysis of a two-way contingency table.

Synopsis

#include `<imsls.h>`

float `imsls_f_contingency_table` (*int* `n_rows`, *int* `n_columns`,
float `table[]`, ..., 0)

The type *double* function is `imsls_d_contingency_table`.

Required Arguments

int `n_rows` (Input)
Number of rows in the table.

int `n_columns` (Input)
Number of columns in the table.

float `table[]` (Input)
Array of length `n_rows` × `n_columns` containing the observed counts in the contingency table.

Return Value

Pearson chi-squared *p*-value for independence of rows and columns.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_contingency_table` (*int* `n_rows`, *int* `n_columns`,
float `table[]`,
`IMSLS_CHI_SQUARED`, *int* `*df`, *float* `*chi_squared`,
float `*p_value`,
`IMSLS_LRT`, *int* `*df`, *float* `*g_squared`, *float* `*p_value`,
`IMSLS_EXPECTED`, *float* `**expected`,
`IMSLS_EXPECTED_USER`, *float* `expected[]`,
`IMSLS_CONTRIBUTIONS`, *float* `**chi_squared_contributions`,
`IMSLS_CONTRIBUTIONS_USER`,
float `chi_squared_contributions[]`,
`IMSLS_CHI_SQUARED_STATS`, *float* `**chi_squared_stats`,
`IMSLS_CHI_SQUARED_STATS_USER`,
float `chi_squared_stats[]`,

IMSLS_STATISTICS, *float* **statistics,
IMSLS_STATISTICS_USER, *float* statistics[],
0)

Optional Arguments

IMSLS_CHI_SQUARED, *int* *df, *float* *chi_squared, *float* *p_value
(Output)
Argument df is the degrees of freedom for the chi-squared tests associated with the table, chi_squared is the Pearson chi-squared test statistic, and argument p_value is the probability of a larger Pearson chi-squared.

IMSLS_LRT, *int* *df, *float* *g_squared, *float* *p_value (Output)
Argument df is the degrees of freedom for the chi-squared tests associated with the table, argument g_squared is the likelihood ratio G^2 (chi-squared), and argument p_value is the probability of a larger G^2.

IMSLS_EXPECTED, *float* **expected (Output)
Address of a pointer to the internally allocated array of size (n_rows + 1) × (n_columns + 1) containing the expected values of each cell in the table, under the null hypothesis, in the first n_rows rows and n_columns columns. The marginal totals are in the last row and column.

IMSLS_EXPECTED_USER, *float* expected[] (Output)
Storage for array expected is provided by the user. See IMSLS_EXPECTED.

IMSLS_CONTRIBUTIONS, *float* **chi_squared_contributions (Output)
Address of a pointer to an internally allocated array of size (n_rows + 1) × (n_columns + 1) containing the contributions to chi-squared for each cell in the table in the first n_rows rows and n_columns columns. The last row and column contain the total contribution to chi-squared for that row or column.

IMSLS_CONTRIBUTIONS_USER, *float* chi_squared_contributions[]
(Output)
Storage for array chi_squared_contributions is provided by the user. See IMSLS_CONTRIBUTIONS.

IMSLS_CHI_SQUARED_STATS, *float* **chi_squared_stats (Output)
Address of a pointer to an internally allocated array of length 5 containing chi-squared statistics associated with this contingency table. The last three elements are based on Pearson's chi-square statistic (see IMSLS_CHI_SQUARED).

The chi-squared statistics are given as follows:

Element	Chi-squared Statistics
0	exact mean
1	exact standard deviation
2	Phi
3	contingency coefficient
4	Cramer's V

`IMSLS_CHI_SQUARED_STATS_USER`, *float* `chi_squared_stats[]` (Output)
Storage for array `chi_squared_stat` is provided by the user. See `IMSLS_CHI_SQUARED_STATS`.

`IMSLS_STATISTICS`, *float* `**statistics` (Output)
Address of a pointer to an internally allocated array of size 23 × 5 containing statistics associated with this table. Each row corresponds to a statistic.

Row	Statistic
0	Gamma
1	Kendall's τ_b
2	Stuart's τ_c
3	Somers' D for rows (given columns)
4	Somers' D for columns (given rows)
5	product moment correlation
6	Spearman rank correlation
7	Goodman and Kruskal τ for rows (given columns)
8	Goodman and Kruskal τ for columns (given rows)
9	uncertainty coefficient U (symmetric)
10	uncertainty $U_{r\mid c}$ (rows)
11	uncertainty $U_{c\mid r}$ (columns)
12	optimal prediction λ (symmetric)
13	optimal prediction $\lambda_{r\mid c}$ (rows)
14	optimal prediction $\lambda_{c\mid r}$ (columns)
15	optimal prediction $\lambda_{r\mid c}$ (rows)
16	optimal prediction $\lambda_{c\mid r}$ (columns)
17	test for linear trend in row probabilities if `n_rows` = 2 If `n_rows` is not 2, a test for linear trend in column probabilities if `n_columns` = 2.
18	Kruskal-Wallis test for no row effect

Row	Statistic
19	Kruskal-Wallis test for no column effect
20	kappa (square tables only)
21	McNemar test of symmetry (square tables only)
22	McNemar one degree of freedom test of symmetry (square tables only)

If a statistic cannot be computed, or if some value is not relevant for the computed statistic, the entry is NaN (Not a Number). The columns are as follows:

Column	Value
0	estimated statistic
1	standard error for any parameter value
2	standard error under the null hypothesis
3	*t* value for testing the null hypothesis
4	*p*-value of the test in column 3

In the McNemar tests, column 0 contains the statistic, column 1 contains the chi-squared degrees of freedom, column 3 contains the exact *p*-value (1 degree of freedom only), and column 4 contains the chi-squared asymptotic *p*-value. The Kruskal-Wallis test is the same except no exact *p*-value is computed.

`IMSLS_STATISTICS_USER`, *float* `statistics[]` (Output)
Storage for array `statistics` provided by the user. See `IMSLS_STATISTICS`.

Description

Function `imsls_f_contingency_table` computes statistics associated with an $r \times c$ (`n_rows` × `n_columns`) contingency table. The function computes the chi-squared test of independence, expected values, contributions to chi-squared, row and column marginal totals, some measures of association, correlation, prediction, uncertainty, the McNemar test for symmetry, a test for linear trend, the odds and the log odds ratio, and the kappa statistic (if the appropriate optional arguments are selected).

Notation

Let x_{ij} denote the observed cell frequency in the *ij* cell of the table and *n* denote the total count in the table. Let $p_{ij} = p_{i\bullet}p_{j\bullet}$ denote the predicted cell probabilities under the null hypothesis of independence, where $p_{i\bullet}$ and $p_{j\bullet}$ are the row and column marginal relative frequencies. Next, compute the expected cell counts as $e_{ij} = np_{ij}$.

Also required in the following are a_{uv} and b_{uv} for u, $v = 1, \ldots, n$. Let (r_s, c_s) denote the row and column response of observation s. Then, $a_{uv} = 1$, 0, or -1, depending on whether $r_u < r_v$, $r_u = r_v$, or $r_u > r_v$, respectively. The b_{uv} are similarly defined in terms of the c_s variables.

Chi-squared Statistic

For each cell in the table, the contribution to χ^2 is given as $(x_{ij} - e_{ij})^2/e_{ij}$. The Pearson chi-squared statistic (denoted χ^2) is computed as the sum of the cell contributions to chi-squared. It has $(r - 1)(c - 1)$ degrees of freedom and tests the null hypothesis of independence, i.e., $H_0{:}p_{ij} = p_{i\bullet}p_{j\bullet}$. The null hypothesis is rejected if the computed value of χ^2 is too large.

The maximum likelihood equivalent of χ^2, G^2 is computed as follows:

$$G^2 = -2\sum_{i,j} x_{ij} \ln\left(x_{ij} / np_{ij}\right)$$

G^2 is asymptotically equivalent to χ^2 and tests the same hypothesis with the same degrees of freedom.

Measures Related to Chi-squared (Phi, Contingency Coefficient, and Cramer's *V*)

There are three measures related to chi-squared that do not depend on sample size:

$$\text{phi, } \phi = \sqrt{\chi^2/n}$$

$$\text{contingency coefficient, } P = \sqrt{\chi^2 / \left(n + \chi^2\right)}$$

$$\text{Cramer's } V\text{, } V = \sqrt{\chi^2 / \left(n \min\left(r, c\right)\right)}$$

Since these statistics do not depend on sample size and are large when the hypothesis of independence is rejected, they can be thought of as measures of association and can be compared across tables with different sized samples. While both P and V have a range between 0.0 and 1.0, the upper bound of P is actually somewhat less than 1.0 for any given table (see Kendall and Stuart 1979, p. 587). The significance of all three statistics is the same as that of the χ^2 statistic, `chi_squared`.

The distribution of the χ^2 statistic in finite samples approximates a chi-squared distribution. To compute the exact mean and standard deviation of the χ^2 statistic, Haldane (1939) uses the multinomial distribution with fixed table marginals. The exact mean and standard deviation generally differ little from the mean and standard deviation of the associated chi-squared distribution.

Standard Errors and *p*-values for Some Measures of Association

In Columns 1 through 4 of statistics, estimated standard errors and asymptotic *p*-values are reported. Estimates of the standard errors are computed in two ways.

The first estimate, in Column 1 of the array `statistics`, is asymptotically valid for any value of the statistic. The second estimate, in Column 2 of the array, is only correct under the null hypothesis of no association. The z-scores in Column 3 of statistics are computed using this second estimate of the standard errors. The p-values in Column 4 are computed from this z-score. See Brown and Benedetti (1977) for a discussion and formulas for the standard errors in Column 2.

Measures of Association for Ranked Rows and Columns

The measures of association, ϕ, P, and V, do not require any ordering of the row and column categories. Function `imsls_f_contingency_table` also computes several measures of association for tables in which the rows and column categories correspond to ranked observations. Two of these tests, the product-moment correlation and the Spearman correlation, are correlation coefficients computed using assigned scores for the row and column categories. The cell indices are used for the product-moment correlation, while the average of the tied ranks of the row and column marginals is used for the Spearman rank correlation. Other scores are possible.

Gamma, Kendall's τ_b, Stuart's τ_c, and Somers' D are measures of association that are computed like a correlation coefficient in the numerator. In all these measures, the numerator is computed as the "covariance" between the a_{uv} variables and b_{uv} variables defined above, i.e., as follows:

$$\sum_u \sum_v a_{uv} b_{uv}$$

Recall that a_{uv} and b_{uv} can take values −1, 0, or 1. Since the product $a_{uv}b_{uv} = 1$ only if a_{uv} and b_{uv} are both 1 or are both −1, it is easy to show that this ''covariance'' is twice the total number of agreements minus the number of disagreements, where a disagreement occurs when $a_{uv}b_{uv} = -1$.

Kendall's τ_b is computed as the correlation between the a_{uv} variables and the b_{uv} variables (see Kendall and Stuart 1979, p. 593). In a rectangular table ($r \neq c$), Kendall's τ_b cannot be 1.0 (if all marginal totals are positive). For this reason, Stuart suggested a modification to the denominator of τ in which the denominator becomes the largest possible value of the "covariance." This maximizing value is approximately $n^2m/(m - 1)$, where $m = \min(r, c)$. Stuart's τ_c uses this approximate value in its denominator. For large n, $\tau_c \approx m\tau_b/(m - 1)$.

Gamma can be motivated in a slightly different manner. Because the "covariance" of the a_{uv} variables and the b_{uv} variables can be thought of as twice the number of agreements minus the disagreements, $2(A - D)$, where A is the number of agreements and D is the number of disagreements, Gamma is motivated as the probability of agreement minus the probability of disagreement, given that either agreement or disagreement occurred. This is shown as $\gamma = (A - D)/(A + D)$.

Two definitions of Somers' D are possible, one for rows and a second for columns. Somers' D for rows can be thought of as the regression coefficient for predicting a_{uv} from b_{uv}. Moreover, Somer's D for rows is the probability of agreement minus the probability of disagreement, given that the column variable, b_{uv}, is not 0. Somers' D for columns is defined in a similar manner.

A discussion of all of the measures of association in this section can be found in Kendall and Stuart (1979, p. 592).

Measures of Prediction and Uncertainty

Optimal Prediction Coefficients: The measures in this section do not require any ordering of the row or column variables. They are based entirely upon probabilities. Most are discussed in Bishop et al. (1975, p. 385).

Consider predicting (or classifying) the column for a given row in the table. Under the null hypothesis of independence, choose the column with the highest column marginal probability for all rows. In this case, the probability of misclassification for any row is 1 minus this marginal probability. If independence is not assumed within each row, choose the column with the highest row conditional probability. The probability of misclassification for the row becomes 1 minus this conditional probability.

Define the optimal prediction coefficient $\lambda_{c\,|\,r}$ for predicting columns from rows as the proportion of the probability of misclassification that is eliminated because the random variables are not independent. It is estimated by

$$\lambda_{c|r} = \frac{(1 - p_{\bullet m}) - (1 - \sum_i p_{im})}{1 - p_{\bullet m}}$$

where m is the index of the maximum estimated probability in the row (p_{im}) or row margin ($p_{\bullet m}$). A similar coefficient is defined for predicting the rows from the columns. The symmetric version of the optimal prediction λ is obtained by summing the numerators and denominators of $\lambda_{r\,|\,c}$ and $\lambda_{c\,|\,r}$, then dividing. Standard errors for these coefficients are given in Bishop et al. (1975, p. 388).

A problem with the optimal prediction coefficients λ is that they vary with the marginal probabilities. One way to correct this is to use row conditional probabilities. The optimal prediction $\lambda*$ coefficients are defined as the corresponding λ coefficients in which first the row (or column) marginals are adjusted to the same number of observations. This yields

$$\lambda^*_{c|r} = \frac{\sum_i \max_j p_{j|i} - \max_j (\sum_i p_{j|i})}{R - \max_j (\sum_i p_{j|i})}$$

where i indexes the rows, j indexes the columns, and $p_{j|i}$ is the (estimated) probability of column j given row i.

$$\lambda^*_{r|c}$$

is similarly defined.

Goodman and Kruskal τ: A second kind of prediction measure attempts to explain the proportion of the explained variation of the row (column) measure given the column (row) measure. Define the total variation in the rows as follows:

$$n/2-(\sum_i x_{i\bullet}^2)/(2n)$$

Note that this is 1/(2n) times the sums of squares of the a_{uv} variables.

With this definition of variation, the Goodman and Kruskal τ coefficient for rows is computed as the reduction of the total variation for rows accounted for by the columns, divided by the total variation for the rows. To compute the reduction in the total variation of the rows accounted for by the columns, note that the total variation for the rows within column j is defined as follows:

$$q_j = x_{\bullet j}/2-(\sum_i x_{ij}^2)/(2x_{i\bullet})$$

The total variation for rows within columns is the sum of the q_j variables. Consistent with the usual methods in the analysis of variance, the reduction in the total variation is given as the difference between the total variation for rows and the total variation for rows within the columns.

Goodman and Kruskal's τ for columns is similarly defined. See Bishop et al. (1975, p. 391) for the standard errors.

Uncertainty Coefficients: The uncertainty coefficient for rows is the increase in the log-likelihood that is achieved by the most general model over the independence model, divided by the marginal log-likelihood for the rows. This is given by the following equation:

$$U_{r|c} = \frac{\sum_{i,j} x_{ij}\log\left(x_{i\bullet}x_{\bullet j}/nx_{ij}\right)}{\sum_i x_{i\bullet}\log\left(x_{i\bullet}/n\right)}$$

The uncertainty coefficient for columns is similarly defined. The symmetric uncertainty coefficient contains the same numerator as $U_{r\,|\,c}$ and $U_{c\,|\,r}$ but averages the denominators of these two statistics. Standard errors for U are given in Brown (1983).

Kruskal-Wallis: The Kruskal-Wallis statistic for rows is a one-way analysis-of-variance-type test that assumes the column variable is monotonically ordered. It tests the null hypothesis that no row populations are identical, using average ranks for the column variable. The Kruskal-Wallis statistic for columns is similarly defined. Conover (1980) discusses the Kruskal-Wallis test.

Test for Linear Trend: When there are two rows, it is possible to test for a linear trend in the row probabilities if it is assumed that the column variable is monotonically ordered. In this test, the probabilities for row 1 are predicted by the column index using weighted simple linear regression. This slope is given by

$$\hat{\beta} = \frac{\sum_j x_{\bullet j}\left(x_{1j}/x_{\bullet j} - x_{1\bullet}/n\right)\left(j - \bar{j}\right)}{\sum_j x_{\bullet j}\left(j - \bar{j}\right)^2}$$

where

$$\bar{j} = \sum_j x_{\bullet j}\, j/n$$

is the average column index. An asymptotic test that the slope is 0 may then be obtained (in large samples) as the usual regression test of zero slope.

In two-column data, a similar test for a linear trend in the column probabilities is computed. This test assumes that the rows are monotonically ordered.

Kappa: Kappa is a measure of agreement computed on square tables only. In the kappa statistic, the rows and columns correspond to the responses of two judges. The judges agree along the diagonal and disagree off the diagonal. Let

$$p_0 = \sum_i x_{ii}/n$$

denote the probability that the two judges agree, and let

$$p_c = \sum_i e_{ii}/n$$

denote the expected probability of agreement under the independence model. Kappa is then given by $(p_0 - p_c)/(1 - p_c)$.

McNemar Tests: The McNemar test is a test of symmetry in a square contingency table. In other words, it is a test of the null hypothesis $H_0{:}\theta_{ij} = \theta_{ji}$. The multiple degrees-of-freedom version of the McNemar test with $r\,(r - 1)/2$ degrees of freedom is computed as follows:

$$\sum_{i<j} \frac{\left(x_{ij} - x_{ji}\right)^2}{\left(x_{ij} + x_{ji}\right)}$$

The single degree-of-freedom test assumes that the differences, $x_{ij} - x_{ji}$, are all in one direction. The single degree-of-freedom test will be more powerful than the multiple degrees-of-freedom test when this is the case. The test statistic is given as follows:

$$\frac{\left(\sum_{i<j}\left(x_{ij} - x_{ji}\right)\right)^2}{\sum_{i<j}\left(x_{ij} + x_{ji}\right)}$$

The exact probability can be computed by the binomial distribution.

Examples

Example 1

The following example is taken from Kendall and Stuart (1979) and involves the distance vision in the right and left eyes. Output contains only the *p*-value.

```
#include <imsls.h>

void main()
{
    int n_rows     = 4;
    int n_columns  = 4;
    float table[4][4]    = {821, 112, 85, 35,
                            116, 494, 145, 27,
                            72, 151, 583, 87,
                            43, 34, 106, 331};
    float p_value;

    p_value = imsls_f_contingency_table(n_rows, n_columns,
                                        &table[0][0], 0);
    printf ("P-value = %10.6f.\n", p_value);

}
```

Output

```
P-value =   0.000000.
```

Example 2

The following example, which illustrates the use of Kappa and McNemar tests, uses the same distance vision data as the previous example. The available statistics are output using optional arguments.

```
#include <imsls.h>

void main()
{
    int      n_rows = 4;
    int      n_columns = 4;
    int      df1, df2;
    float    table[16]  =  {821.0, 112.0, 85.0, 35.0,
                            116.0, 494.0, 145.0, 27.0,
                            72.0, 151.0, 583.0, 87.0,
                            43.0, 34.0, 106.0, 331.0};
    float    p_value1, p_value2, chi_squared, g_squared;
    float    *expected, *chi_squared_contributions;
    float    *chi_squared_stats, *statistics;
    char     *labels[] = {
             "Exact mean",
             "Exact standard deviation",
             "Phi",
             "P",
             "Cramer’s V"};
    char     *stat_row_labels[] = {"Gamma", "Tau B", "Tau C",
             "D-Row", "D-Column", "Correlation", "Spearman",
             "GK tau rows", "GK tau cols.", "U - sym.", "U - rows",
             "U - cols.", "Lambda-sym.", "Lambda-row", "Lambda-col.",
             "l-star-rows", "l-star-col.", "Lin. trend",
             "Kruskal row", "Kruskal col.", "Kappa", "McNemar",
             "McNemar df=1"};
    char     *stat_col_labels[] = {"","statistic", "standard error",
             "std. error under Ho", "t-value testing Ho",
             "p-value"};

    imsls_f_contingency_table (n_rows, n_columns, table,
             IMSLS_CHI_SQUARED, &df1, &chi_squared, &p_value1,
             IMSLS_LRT, &df2, &g_squared, &p_value2,
             IMSLS_EXPECTED, &expected,
             IMSLS_CONTRIBUTIONS,
                        &chi_squared_contributions,
             IMSLS_CHI_SQUARED_STATS, &chi_squared_stats,
             IMSLS_STATISTICS, &statistics,
             0);

    printf("Pearson chi-squared statistic     %11.4f\n", chi_squared);
    printf("p-value for Pearson chi-squared   %11.4f\n", p_value1);
    printf("degrees of freedom                %11d\n", df1);
    printf("G-squared statistic               %11.4f\n", g_squared);
    printf("p-value for G-squared             %11.4f\n", p_value2);
    printf("degrees of freedom                %11d\n", df2);

    imsls_f_write_matrix("* * * Table Values * * *\n", 4, 4,
             table,
             IMSLS_WRITE_FORMAT, "%11.1f",
             0);

    imsls_f_write_matrix("* * * Expected Values * * *\n", 5, 5,
             expected,
             IMSLS_WRITE_FORMAT, "%11.2f",
             0);
    imsls_f_write_matrix("* * * Contributions to Chi-squared* * *\n",
             5, 5,
             chi_squared_contributions,
             IMSLS_WRITE_FORMAT, "%11.2f",
             0);
    imsls_f_write_matrix("* * * Chi-square Statistics * * *\n",
             5, 1,
             chi_squared_stats,
             IMSLS_ROW_LABELS, labels,
             IMSLS_WRITE_FORMAT, "%11.4f",
             0);
    imsls_f_write_matrix("* * * Table Statistics * * *\n",
             23, 5,
             statistics,
             IMSLS_ROW_LABELS, stat_row_labels,
             IMSLS_COL_LABELS, stat_col_labels,
             IMSLS_WRITE_FORMAT, "%9.4f",
             0);
}
```

Output

```
Pearson chi-squared statistic       3304.3682
p-value for Pearson chi-squared        0.0000
degrees of freedom                          9
G-squared statistic                 2781.0188
p-value for G-squared                  0.0000
degrees of freedom                          9

              * * * Table Values * * *

             1            2            3            4
1        821.0        112.0         85.0         35.0
2        116.0        494.0        145.0         27.0
3         72.0        151.0        583.0         87.0
4         43.0         34.0        106.0        331.0

                   * * * Expected Values * * *

             1            2            3            4            5
1       341.69       256.92       298.49       155.90      1053.00
2       253.75       190.80       221.67       115.78       782.00
3       289.77       217.88       253.14       132.21       893.00
4       166.79       125.41       145.70        76.10       514.00
5      1052.00       791.00       919.00       480.00      3242.00

             * * * Contributions to Chi-squared* * *

             1            2            3            4            5
1       672.36        81.74       152.70        93.76      1000.56
2        74.78       481.84        26.52        68.08       651.21
3       163.66        20.53       429.85        15.46       629.50
4        91.87        66.63        10.82       853.78      1023.10
5      1002.68       650.73       619.88      1031.08      3304.37

 * * * Chi-square Statistics * * *

Exact mean                     9.0028
Exact standard deviation       4.2402
Phi                            1.0096
P                              0.7105
Cramer’s V                     0.5829

                    * * * Table Statistics * * *

              statistic  standard error  std. error  t-value testing
                                           under Ho               Ho
Gamma            0.7757          0.0123      0.0149          52.1897
Tau B            0.6429          0.0122      0.0123          52.1897
Tau C            0.6293          0.0121   .........          52.1897
D-Row            0.6418          0.0122      0.0123          52.1897
D-Column         0.6439          0.0122      0.0123          52.1897
Correlation      0.6926          0.0128      0.0172          40.2669
Spearman         0.6939          0.0127      0.0127          54.6614
GK tau rows      0.3420          0.0123   .........        .........
GK tau cols.     0.3430          0.0122   .........        .........
U - sym.         0.3171          0.0110   .........        .........
U - rows         0.3178          0.0110   .........        .........
U - cols.        0.3164          0.0110   .........        .........
```

```
Lambda-sym.       0.5373          0.0124   .........        .........
Lambda-row        0.5374          0.0126   .........        .........
Lambda-col.       0.5372          0.0126   .........        .........
l-star-rows       0.5506          0.0136   .........        .........
l-star-col.       0.5636          0.0127   .........        .........
Lin. trend     .........       .........   .........        .........
Kruskal row    1561.4861          3.0000   .........        .........
Kruskal col.   1563.0300          3.0000   .........        .........
Kappa             0.5744          0.0111      0.0106          54.3583
McNemar           4.7625          6.0000   .........        .........
McNemar df=1      0.9487          1.0000   .........           0.3459

                 p-value
Gamma             0.0000
Tau B             0.0000
Tau C             0.0000
D-Row             0.0000
D-Column          0.0000
Correlation       0.0000
Spearman          0.0000
GK tau rows    .........
GK tau cols.   .........
U - sym.       .........
U - rows       .........
U - cols.      .........
Lambda-sym.    .........
Lambda-row     .........
Lambda-col.    .........
l-star-rows    .........
l-star-col.    .........
Lin. trend     .........
Kruskal row       0.0000
Kruskal col.      0.0000
Kappa             0.0000
McNemar           0.5746
McNemar df=1      0.3301
```

Warning Errors

`IMSLS_DF_GT_30`	The degrees of freedom for “IMSLS_CHI_SQUARED” are greater than 30. The exact mean, standard deviation, and the normal distribution function should be used.
`IMSLS_EXP_VALUES_TOO_SMALL`	Some expected values are less than #. Some asymptotic *p*-values may not be good.
`IMSLS_PERCENT_EXP_VALUES_LT_5`	Twenty percent of the expected values are calculated less than 5.

exact_enumeration

Computes exact probabilities in a two-way contingency table using the total enumeration method.

Synopsis

#include `<imsls.h>`

float `imsls_f_exact_enumeration` (*int* `n_rows`, *int* `n_columns`,
    float `table[]`, ..., 0)

The type *double* function is `imsls_d_exact_enumeration`.

Required Arguments

int `n_rows` (Input)
    Number of rows in the table.

int `n_columns` (Input)
    Number of columns in the table.

float `table[]` (Input)
    Array of length `n_rows` × `n_columns` containing the observed counts in the contingency table.

Return Value

The *p*-value for independence of rows and columns. The *p*-value represents the probability of a more extreme table where "extreme" is taken in the Neyman-Pearson sense. The *p*-value is "two-sided".

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_exact_enumeration` (*int* `n_rows`, *int* `n_columns`, *float*
    `table[]`,
    `IMSLS_PROB_TABLE`, *float* `*prt`,
    `IMSLS_P_VALUE`, *float* `*p_value`,
    `IMSLS_CHECK_NUMERICAL_ERROR`, *float* `*check`,
    0)

Optional Arguments

`IMSLS_PROB_TABLE`, *float* `*prt` (Output)
    Probablitity of the observed table occuring, given that the null hypothesis of independent rows and columns is true.

`IMSLS_P_VALUE`, *float* `*p_value` (Output)
    The *p*-value for independence of rows and columns. The *p*-value

represents the probability of a more extreme table where “extreme” is taken in the Neyman-Pearson sense. The *p*-value is “two-sided”.

The *p*-value is also returned in functional form (see “Return Value”).

A table is more extreme if its probability (for fixed marginals) is less than or equal to `prt`.

`IMSLS_CHECK_NUMERICAL_ERROR`, *float* `*check` (Output)
Sum of the probabilities of all tables with the same marginal totals. Parameter check should have a value of 1.0. Deviation from 1.0 indicates numerical error.

Description

Function `imsls_f_exact_enumeration` computes exact probabilities for an $r \times c$ contingency table for fixed row and column marginals (a marginal is the number of counts in a row or column), where r = `n_rows` and c = `n_columns`. Let f_{ij} denote the count in row i and column j of a table, and let $f_{i\bullet}$ and $f_{\bullet j}$ denote the row and column marginals. Under the hypothesis of independence, the (conditional) probability of the fixed marginals of the observed table is given by

$$P_f = \frac{\prod_{i=1}^{r} f_{i\bullet}! \prod_{j=1}^{c} f_{\bullet j}!}{f_{\bullet\bullet}! \prod_{i=1}^{r} \prod_{j=1}^{c} f_{ij}!}$$

where $f_{\bullet\bullet}$ is the total number of counts in the table. P_f corresponds to output argument `prt`.

A “more extreme” table X is defined in the probablistic sense as more extreme than the observed table if the conditional probability computed for table X (for the same marginal sums) is less than the conditional probability computed for the observed table. The user should note that this definition can be considered “two-sided” in the cell counts.

Because `imsls_f_exact_enumeration` used total enumeration in computing the probability of a more extreme table, the amount of computer time required increases very rapidly with the size of the table. Tables with a large total count $f_{\bullet\bullet}$ or a large value of $r \times c$ should not be analyzed using `imsls_f_exact_enumeration`. In such cases, try using `imsls_f_exact_network`.

Example

In this example, the exact conditional probability for the 2×2 contingency table

$$\begin{bmatrix} 8 & 12 \\ 8 & 2 \end{bmatrix}$$

is computed.

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

void main()
{
    float p;
    float table[4] = {8, 12,
                      8,  2};

    p = imsls_f_exact_enumeration(2, 2, table, 0);
    printf("p-value = %9.4f\n", p);
}
```

Output

```
p-value =    0.0577
```

exact_network

Computes Fisher exact probabilities and a hybrid approximation of the Fisher exact method for a two-way contingency table using the network algorithm.

Synopsis

#include `<imsls.h>`

float `imsls_f_exact_network` (*int* `n_rows`, *int* `n_columns`,
    float `table[]`, ..., 0)

The type *double* function is `imsls_d_exact_network`.

Required Arguments

int `n_rows` (Input)
    Number of rows in the table.

int `n_columns` (Input)
    Number of columns in the table.

float `table[]` (Input)
    Array of length `n_rows` × `n_columns` containing the observed counts in the contingency table.

Return Value

The *p*-value for independence of rows and columns. The *p*-value represents the probability of a more extreme table where “extreme” is taken in the Neyman-Pearson sense. The *p*-value is “two-sided”.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float imsls_f_exact_network (int n_rows, int n_columns,
        float table[],
        IMSLS_PROB_TABLE, float *prt,
        IMSLS_P_VALUE, float *p_value,
        IMSLS_APPROXIMATION_PARAMETERS, float expect,
                float percent, float expected_minimum,
        IMSLS_NO_APPROXIMATION,
        IMSLS_WORKSPACE, int factor1, int factor2,
                int max_attempts, int *n_attempts,
        0)
```

Optional Arguments

`IMSLS_PROB_TABLE`, *float* `*prt` (Output)
: Probability of the observed table occurring given that the null hypothesis of independent rows and columns is true.

`IMSLS_P_VALUE`, *float* `*p_value` (Output)
: The *p*-value for independence of rows and columns. The *p*-value represents the probability of a more extreme table where “extreme” is in the Neyman-Pearson sense. The `p_value` is “two-sided”. The *p*-value is also returned in functional form (see “Return Value”).

  A table is more extreme if its probability (for fixed marginals) is less than or equal to `prt`.

`IMSLS_APPROXIMATION_PARAMETERS`, *float* `expect`, *float* `percent`, *float* `expected_minimum`. (Input)
: Parameter `expect` is the expected value used in the hybrid approximation to Fisher’s exact test algorithm for deciding when to use asymptotic probabilities when computing path lengths. Parameter `percent` is the percentage of remaining cells that must have estimated expected values greater than `expect` before asymptotic probabilities can be used in computing path lengths. Parameter `expected_minimum` is the minimum cell estimated value allowed for asymptotic chi-squared probabilities to be used.

  Asymptotic probabilities are used in computing path lengths whenever `percent` or more of the cells in the table have estimated expected values of `expect` or more, with no cell having expected value less than `expected_minimum`. See the “Description” section for details.

  Defaults: `expect` = 5.0, `percent` = 80.0, `expected_minimum` = 1.0
  Note that these defaults correspond to the “Cochran” condition.

`IMSLS_NO_APPROXIMATION`,
: The Fisher exact test is used. Arguments `expect`, `percent`, and `expected_minimum` are ignored.

IMSLS_WORKSPACE, *int* factor1, *int* factor2,

int max_attempts, (Input)

int *n_attempts (Output)

The network algorithm requires a large amount of workspace. Some of the workspace requirements are well-defined, while most of the workspace requirements can only be estimated. The estimate is based primarily on table size.

Function imsls_f_exact_enumeration allocates a default amount of workspace suitable for small problems. If the algorithm determines that this initial allocation of workspace is inadaquate, the memory is freed, a larger amount of memory allocated (twice as much as the previous allocation), and the network algorithm is re-started. The algorithm allows for up to max_attempts attempts to complete the algorithm.

Because each attempt requires computer time, it is suggested that factor1 and factor2 be set to some large numbers (like 1,000 and 30,000) if the problem to be solved is large. It is suggested that factor2 be 30 times larger than factor1. Although imsls_f_exact_enumeration will eventually work its way up to a large enough memory allocation, it is quicker to allocate enough memory initially.

The known (well-defined) workspace requirements are as follows: Define $f_{\bullet\bullet} = \Sigma\Sigma f_{ij}$ equal to the sum of all cell frequencies in the observed table, $nt = f_{\bullet\bullet} + 1$, $mx = \max$ (n_rows, n_columns), $mn = \min$ (n_rows, n_columns), $t1 = \max(800 + 7mx, (5 + 2mx)$ (n_rows + n_columns + 1)), and $t2 = \max(400 + mx, + 1,$ n_rows + n_columns + 1).

The following amount of integer workspace is allocated: $3mx + 2mn + t1$.

The following amount of *float* (or *double*, if using imsls_d_exact_network) workspace is allocated: $nt + t2$.

The remainder of the workspace that is required must be estimated and allocated based on factor1 and factor2. The amount of integer workspace allocated is $6n$ (factor1 + factor2). The amount of real workspace allocated is n (6factor1 + 2factor2). Variable n is the index for the attempt, $1 < n \leq$ max_attempts.

Defaults: factor1 = 100, factor2 = 3000, max_attempts = 10

Description

Function imsls_f_exact_network computes Fisher exact probabilities or a hybrid algorithm approximation to Fisher exact probabilities for an $r \times c$ contingency table with fixed row and column marginals (a marginal is the number of counts in a row or column), where r = n_rows and c = n_columns. Let f_{ij} denote the count in row i and column j of a table, and let f_i and $f_{\bullet j}$ denote the row and column marginals. Under the hypothesis of independence, the (conditional) probability of the fixed marginals of the observed table is given by

$$P_f = \frac{\prod_{i=1}^{r} f_{i\bullet}! \prod_{j=1}^{c} f_{\bullet j}!}{f_{\bullet\bullet}! \prod_{i=1}^{r} \prod_{j=1}^{c} f_{ij}!}$$

where $f_{\bullet\bullet}$ is the total number of counts in the table. P_f corresponds to output argument `prt`.

A "more extreme" table *X* is defined in the probablistic sense as more extreme than the observed table if the conditional probability computed for table *X* (for the same marginal sums) is less than the conditional probability computed for the observed table. The user should note that this definition can be considered "two-sided" in the cell counts.

See Example 1 for a comparison of execution times for the various algorithms. Note that the Fisher exact probability and the usual asymptotic chi-squared probability will usually be different. (The network approximation is often 10 times faster than the Fisher exact test, and even faster when compared to the total enumeration method.)

Examples

Example 1

The following example demonstrates and compares the various methods of computing the chi-squared *p*-value with respect to accuracy and execution time. As seen in the output of this example, the Fisher exact probability and the usual asymptotic chi-squared probability (generated using function `imsls_f_contingency_table`) can be different. Also, note that the network algorithm *with* approximation can be up to 10 times faster than the network algorithm *without* approximation, and up to 100 times faster than the total enumeration method.

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

void main()
{
    int n_rows = 3;
    int n_columns = 5;
    float p;
    float table[15] = {20, 20, 0, 0, 0,
                       10, 10, 2, 2, 1,
                       20, 20, 0, 0, 0};
    double a, b;

    printf("Asymptotic Chi-Squared p-value\n");
    p = imsls_f_contingency_table(n_rows, n_columns, table, 0);
    printf("p-value = %9.4f\n", p);
    printf("\nNetwork Algorithm with Approximation\n");
    a = imsls_ctime();
    p = imsls_f_exact_network(n_rows, n_columns, table, 0);
    b = imsls_ctime();
    printf("p-value = %9.4f\n", p);
    printf("Execution time = %10.4f\n", b-a);

    printf("\nNetwork Algoritm without Approximation\n");
    a = imsls_ctime();
    p = imsls_f_exact_network(n_rows, n_columns, table,
        IMSLS_NO_APPROXIMATION, 0);
    b = imsls_ctime();
    printf("p-value = %9.4f\n", p);
    printf("Execution time = %10.4f\n", b-a);

    printf("\nTotal Enumeration Method\n");
    a = imsls_ctime();
    p = imsls_f_exact_enumeration(n_rows, n_columns, table, 0);
    b = imsls_ctime();
    printf("p-value = %9.4f\n", p);
    printf("Execution time = %10.4f\n", b-a);


}
```

Output

```
Asymptotic Chi-Squared p-value
p-value =    0.0323

Network Algorithm with Approximation
p-value =    0.0601
Execution time =     0.0400

Network Algoritm without Approximation
p-value =    0.0598
Execution time =     0.4300

Total Enumeration Method
p-value =    0.0597
Execution time =     3.1400
```

Example 2

This document example demonstrates the optional keyword `IMSLS_WORKSPACE` and how different workspace settings affect execution time. Setting the workspace available too low results in poor performance since the algorithm will fail, re-allocate a larger amount of workspace (a factor of 10 larger) and re-start the calculations (See Test #3, for which `n_attempts` is returned with a value of 2). Setting the workspace available very large will provide no improvement in performance.

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

void main()
{
    int n_rows = 3;
    int n_columns = 5;
    float p;
    float table[15] = {20, 20, 0, 0, 0,
                       10, 10, 2, 2, 1,
                       20, 20, 0, 0, 0};
    double a, b;
    int i, n_attempts, simulation_size = 10;

    printf("Test #1, factor1 = 1000, factor2 = 30000\n");
    a = imsls_ctime();
    for (i=0; i<simulation_size; i++) {
        p = imsls_f_exact_network(n_rows, n_columns, table,
            IMSLS_NO_APPROXIMATION,
            IMSLS_WORKSPACE, 1000, 30000, 10, &n_attempts, 0);
    }
    b = imsls_ctime();
    printf("n_attempts = %2d\n", n_attempts);
    printf("Execution time = %10.4f\n", b-a);

    printf("\nTest #2, factor1 = 100, factor2 = 3000\n");
    a = imsls_ctime();
    for (i=0; i<simulation_size; i++) {
        p = imsls_f_exact_network(n_rows, n_columns, table,
            IMSLS_NO_APPROXIMATION,
            IMSLS_WORKSPACE, 100, 3000, 10, &n_attempts, 0);
    }
    b = imsls_ctime();
    printf("n_attempts = %2d\n", n_attempts);
    printf("Execution time = %10.4f\n", b-a);

    printf("\nTest #3, factor1 = 10, factor2 = 300\n");
    a = imsls_ctime();
    for (i=0; i<simulation_size; i++) {
        p = imsls_f_exact_network(n_rows, n_columns, table,
            IMSLS_NO_APPROXIMATION,
            IMSLS_WORKSPACE, 10, 300, 10, &n_attempts, 0);
    }
    b = imsls_ctime();
    printf("n_attempts = %2d\n", n_attempts);
    printf("Execution time = %10.4f\n", b-a);
}
```

Output

```
Test #1, factor1 = 1000, factor2 = 30000
n_attempts =  1
Execution time =     4.3700

Test #2, factor1 = 100, factor2 = 3000
n_attempts =  1
Execution time =     4.2900

Test #3, factor1 = 10, factor2 = 300
n_attempts =  2
Execution time =     8.3700
```

Warning Errors

`IMSLS_HASH_TABLE_ERROR_2`	The value “ldkey” = # is too small. “ldkey” is calculated as “factor1”*pow(10,”n_attempt”−1) ending this execution attempt.
`IMSLS_HASH_TABLE_ERROR_3`	The value “ldstp” = # is too small. “ldstp” is calculated as “factor2”*pow(10,”n_attempt”−1) ending this execution attempt.

Fatal Errors

`IMSLS_HASH_TABLE_ERROR_1`	The hash table key cannot be computed because the largest key is larger than the largest representable integer. The algorithm cannot proceed.

categorical_glm

Analyzes categorical data using logistic, Probit, Poisson, and other generalized linear models.

Synopsis

#include `<imsls.h>`

int `imsls_f_categorical_glm` (*int* `n_observations`, *int* `n_class`,
    int `n_continuous`, *int* `model`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_categorical_glm`.

Required Arguments

int `n_observations` (Input)
    Number of observations.

int `n_class` (Input)
    Number of classification variables.

int `n_continuous` (Input)
    Number of continuous variables.

int `model` (Input)
    Argument `model` specifies the model used to analyze the data. The six models are as follows:

Model	Relationship[*]	PDF of Response Variable
0	Exponential	Poisson
1	Logistic	Negative Binomial
2	Logistic	Logarithmic
3	Logistic	Binomial
4	Probit	Binomial
5	Log-log	Binomial

Note that the lower bound of the response variable is 1 for `model` = 3 and is 0 for all other models. See the "Description" section for more information about these models.

float `x[]` (Input)
Array of size `n_observations` by (`n_class` + `n_continuous`) + m containing data for the independent variables, dependent variable, and optional parameters.

The columns must be ordered such that the first `n_class` columns contain data for the class variables, the next `n_continuous` columns contain data for the continuous variables, and the next column contains the response variable. The final (and optional) $m - 1$ columns contain the optional parameters.

Return Value

An integer value indicating the number of estimated coefficients (`n_coefficients`) in the model.

Synopsis with Optional Arguments

```
#include <imsls.h>

int imsls_f_categorical_glm (int n_observations, int n_class,
        int n_continuous, int model, float x[],
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_X_COL_FREQUENCIES, int ifrq,
        IMSLS_X_COL_FIXED_PARAMETER, int ifix,
        IMSLS_X_COL_DIST_PARAMETER, int ipar,
        IMSLS_X_COL_VARIABLES, int iclass[], int icontinuous[],
                int iy,
        IMSLS_EPS, float eps,
        IMSLS_MAX_ITERATIONS, int max_iterations,
        IMSLS_INTERCEPT,
        IMSLS_NO_INTERCEPT,
```

[*]Relationship between the parameter, θ or λ, and a linear model of the explanatory variables, $X\beta$.

IMSLS_EFFECTS, *int* n_effects, *int* n_var_effects[],
    int indices_effects,
IMSLS_INITIAL_EST_INTERNAL,
IMSLS_INITIAL_EST_INPUT, *int* n_coef_input,
    float estimates[],
IMSLS_MAX_CLASS, *int* max_class,
IMSLS_CLASS_INFO, *int* **n_class_values,
    float **class_values,
IMSLS_CLASS_INFO_USER, *int* n_class_values[],
    float class_values[],
IMSLS_COEF_STAT, *float* **coef_statistics,
IMSLS_COEF_STAT_USER, *float* coef_statistics[],
IMSLS_CRITERION, *float* *criterion,
IMSLS_COV, *float* **cov,
IMSLS_COV_USER, *float* cov[],
IMSLS_MEANS, *float* **means,
IMSLS_MEANS_USER, *float* means[],
IMSLS_CASE_ANALYSIS, *float* **case_analysis,
IMSLS_CASE_ANALYSIS_USER, *float* case_analysis[],
IMSLS_LAST_STEP, *float* **last_step,
IMSLS_LAST_STEP_USER, *float* last_step[],
IMSLS_OBS_STATUS, *int* **obs_status,
IMSLS_OBS_STATUS_USER, *int* obs_status[],
IMSLS_ITERATIONS, *int* *n, *float* **iterations,
IMSLS_ITERATIONS_USER, *int* *n, *float* iterations[],
IMSLS_N_ROWS_MISSING, *int* *n_rows_missing,
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of input array x.
Default: x_col_dim = n_class + n_continuous +1

IMSLS_X_COL_FREQUENCIES, *int* ifrq (Input)
Column number ifrg of x containing the frequency of response for each observation.

IMSLS_X_COL_FIXED_PARAMETER, *int* ifix (Input)
Column number ifix in x containing a fixed parameter for each observation that is added to the linear response prior to computing the model parameter. The 'fixed' parameter allows one to test hypothesis about the parameters via the log-likelihoods.

IMSLS_X_COL_DIST_PARAMETER, *int* ipar (Input)
Column number ipar in x containing the value of the known distribution parameter for each observation, where x[i][ipar] is the known distribution parameter associated with the *i*-th observation. The meaning of the distributional parameter depends upon model as follows:

model	Parameter	Meaning of `x [i] [ipar]`
0	E	ln (E) is a fixed intercept to be included in the linear predictor (i.e., the *offset*).
1	S	Number of successes required for the negative binomial distribution.
2	-	Not used for this model.
3-5	N	Number of trials required for the binomial distribution.

Default: When `model` ≠ 2, each observation is assumed to have a parameter value of 1. When `model` = 2, this parameter is not referenced.

`IMSLS_X_COL_VARAIBLES`, *int* `iclass[]`, *int* `icontinuous[]`, *int* `iy` (Input)

This keyword allows specification of the variables to be used in the analysis and overrides the default ordering of variables described for input argument `x`. Columns are numbered 0 to `x_col_dim_1`. To avoid errors, always specify the keyword `IMSLS_X_COL_DIM` when using this keyword.

Argument `iclass` is an index vector of length `n_class` containing the column numbers of `x` that correspond to classification variables.

Argument `icontinuous` is an index vector of length `n_continuous` containing the column numbers of `x` that correspond to continuous variables.

Argument `iy` indicates the column of `x` which contains the independent variable.

`IMSLS_EPS`, *float* `eps` (Input)

Argument `eps` is the convergence criterion. Convergence is assumed when the maximum relative change in any coefficient estimate is less than `eps` from one iteration to the next or when the relative change in the log-likelihood, criterion, from one iteration to the next is less than eps / 100.0.
Default: `eps` = 0.001

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)

Maximum number of iterations. Use `max_iterations` = 0 to compute the Hessian, stored in `cov`, and the Newton step, stored in `last_step`, at the initial estimates (The initial estimates must be input. Use keyword `IMSLS_INITIAL_EST_INPUT`).
Default: `max_iterations` = 30

`IMSLS_INTERCEPT`, *or*
`IMSLS_NO_INTERCEPT`,

By default, or if `IMSLS_INTERCEPT` is specified, the intercept is automatically included in the model. If `IMSLS_NO_INTERCEPT` is specified, there is no intercept in the model (unless otherwise provided for by the user).

IMSLS_EFFECTS, *int* n_effects, *int* n_var_effects[],
int indices_effects[] (Input)
Variable n_effects is the number of effects (sources of variation) in the model. Variable n_var_effects is an array of length n_effects containing the number of variables associated with each effect in the model. Argument indices_effects is an index array of length n_var_effects [0] + n_var_effects [1] + …
+ n_var_effects [n_effects – 1]. The first n_var_effects [0] elements give the column numbers of x for each variable in the first effect. The next n_var_effects [1] elements give the column numbers for each variable in the second effect. The last n_var_effects [n_effects – 1] elements give the column numbers for each variable in the last effect.

IMSLS_INITIAL_EST_INTERNAL, *or*
IMSLS_INITIAL_EST_INPUT, *int* n_coef_input, *float* estimates[]
(Input)
By default, or if IMSLS_INIT_INTERNAL is specified, then unweighted linear regression is used to obtain initial estimates. If IMSLS_INITIAL_EST_INPUT is specified, then the n_coef_input elements of estimates contain initial estimates of the parameters (which requires that the user know the number of coefficients in the model prior to the call to imsls_f_categorical_glm which can be obtained by calling imsls_f_regressors_for_glm.

IMSLS_MAX_CLASS, *int* max_class (Input)
An upper bound on the sum of the number of distinct values taken on by each classification variable.
Default: max_class = n_observations × n_class

IMSLS_CLASS_INFO, *int* **n_class_values, *float* **class_values
(Output)
Argument n_class_values the address of a pointer to the internally allocated array of length n_class containing the number of values taken by each classification variable; the *i*-th classification variable has n_class_values [*i*] distinct values. Argument class_values is the address of a pointer to the internally allocated array of length

$$\sum_{i=0}^{\text{n\_class}-1} \text{n\_class\_values}[i]$$

containing the distinct values of the classification variables in ascending order. The first n_class_values [0] elements of class_values contain the values for the first classification variables, the next n_class_values [1] elements contain the values for the second classification variable, etc.

IMSLS_CLASS_INFO_USER, *int* n_class_values[],
float class_values[] (Output)
Storage for arrays n_class_values and class_values is provided by the user. See IMSLS_CLASS_INFO.

IMSLS_COEF_STAT, *float* **coef_statistics (Output)
Address of a pointer to an internally allocated array of size n_coefficients × 4 containing the parameter estimates and associated statistics, where n_coefficients can be computed by calling imsls_regressors_for_glm.

Column	Statistic
0	Coefficient Estimate.
1	Estimated standard deviation of the estimated coefficient.
2	Asymptotic normal score for testing that the coefficient is zero.
3	The *p*-value associated with the normal score in column 2.

IMSLS_COEF_STAT_USER, *float* coef_statistics[] (Output)
Storage for array coef_statistics is provided by the user. See IMSLS_COEF_STAT.

IMSLS_CRITERION, *float* *criterion (Output)
Optimized criterion. The criterion to be maximized is a constant plus the log-likelihood.

IMSLS_COV, *float* **cov (Output)
Address of a pointer to the internally allocated array of size n_coefficients × n_coefficients containing the estimated asymptotic covariance matrix of the coefficients. For max_iterations = 0, this is the Hessian computed at the initial parameter estimates, where n_coefficients can be computed by calling imsls_regressors_for_glm.

IMSLS_COV_USER, *float* cov[] (Ouput)
Storage for array cov is provided by the user. See IMSLS_COV above.

IMSLS_MEANS, *float* **means (Output)
Address of a pointer to the internally allocated array containing the means of the design variables. The array is of length n_coefficients if IMSLS_NO_INTERCEPT is specified, and of length n_coefficients – 1 otherwise, where n_coefficients can be computed by calling imsls_regressors_for_glm.

IMSLS_MEANS_USER, *float* means[] (Output)
Storage for array means is provided by the user. See IMSLS_MEANS.

`IMSLS_CASE_ANALYSIS`, *float* `**case_analysis` (Output)
Address of a pointer to the internally allocated array of size `n_observations` × 5 containing the case analysis.

Column	Statistic
0	Predicted mean for the observation if `model` = 0. Otherwise, contains the probability of success on a single trial.
1	The residual.
2	The estimated standard error of the residual.
3	The estimated influence of the observation.
4	The standardized residual.

Case statistics are computed for all observations except where missing values prevent their computation.

`IMSLS_CASE_ANALYSIS_USER`, *float* `case_analysis[]` (Output)
Storage for array `case_analysis` is provided by the user. See `IMSLS_CASE_ANALYSIS`.

`IMSLS_LAST_STEP`, *float* `**last_step` (Output)
Address of a pointer to the internally allocated array of length `n_coefficients` containing the last parameter updates (excluding step halvings). For `max_iterations` = 0, `last_step` contains the inverse of the Hessian times the gradient vector, all computed at the initial parameter estimates.

`IMSLS_LAST_STEP_USER`, *float* `last_step[]` (Output)
Storage for array `last_step` is provided by the user. See `IMSLS_LAST_STEP`.

`IMSLS_OBS_STATUS`, *int* `**obs_status` (Output)
Address of a pointer to the internally allocated array of length `n_observations` indicating which observations are included in the extended likelihood.

`obs_status` **[*i*]**	Status of observation
0	Observation *i* is in the likelihood
1	Observation *i* cannot be in the likelihood because it contains at least one missing value in `x`.
2	Observation *i* is not in the likelihood. Its estimated parameter is infinite.

`IMSLS_OBS_STATUS_USER`, *int* `obs_status[]` (Output)
Storage for array `obs_status` is provided by the user. See `IMSLS_OBS_STATUS`.

`IMSLS_N_ROWS_MISSING`, *int* `*n_rows_missing` (Output)
Number of rows of data that contain missing values in one or more of the following arrays or columns of `x`; `ipar`, `iy`, `ifrq`, `ifix`, `iclass`, `icontinuous`, or `indices_effects`.

Remarks

1. Dummy variables are generated for the classification variables as follows: An ascending list of all distinct values of each classification variable is obtained and stored in `class_values`. Dummy variables are then generated for each but the last of these distinct values. Each dummy variable is zero unless the classification variable equals the list value corresponding to the dummy variable, in which case the dummy variable is one. See keyword `IMSLS_LEAVE_OUT_LAST` for optional argument `IMSLS_DUMMY` in routine `imsls_f_regressors_for_glm` (Chapter 2, "Regression").
2. The "product" of a classification variable with a covariate yields dummy variables equal to the product of the covariate with each of the dummy variables associated with the classification variable.
3. The "product" of two classification variables yields dummy variables in the usual manner. Each dummy variable associated with the first classification variable multiplies each dummy variable associated with the second classification variable. The resulting dummy variables are such that the index of the second classification variable varies fastest.

Description

Function `imsls_f_categorical_glm` uses iteratively reweighted least squares to compute (extended) maximum likelihood estimates in some generalized linear models involving categorized data. One of several models, including the probit, logistic, Poisson, logarithmic, and negative binomial models, may be fit.

Note that each row vector in the data matrix can represent a single observation; or, through the use of optional argument `IMSLS_X_COL_FREQUENCIES`, each row can represent several observations. Also note that classification variables and their products are easily incorporated into the models via the usual regression-type specifications.

The models available in `imsls_f_categorical_glm` are:

Model	PDF of the Response Variable	Parameterization
0	$f(y) = (\lambda_y \exp(-\lambda)) / y!$	$\lambda = N \times \exp(\omega + \eta)$
1	$f(y) = \binom{S+y-1}{y-1} \theta^S (1-\theta)^y$	$\theta = \frac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$

Model	PDF of the Response Variable	Parameterization
2	$f(y) = (1-\theta)^y / (y \ln \theta)$	$\theta = \dfrac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$
3	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$	$\theta = \dfrac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$
4	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$	$\theta = \Phi(\omega+\eta)$
5	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$	$\theta = 1 - \exp(-\exp(\omega+\eta))$

Here, Φ denotes the cumulative normal distribution, N and S are known distribution parameters specified for each observation via the optional argument `IMSLS_X_COL_DIST_PARAMETER`, and ω is an optional fixed parameter of the linear response, γ_i, specified for each observation. (If `IMSLS_X_COL_FIXED_PARAMETER` is not specified, then ω is taken to be 0.) Since the log-log model (`model` = 5) probabilities are not symmetric with respect to 0.5, quantitatively, as well as qualitatively, different models result when the definitions of "success" and "failure" are interchanged in this distribution. In this model and all other models involving θ, θ is taken to be the probability of a"success".

Computational Details

The computations proceed as follows:

1. The input parameters are checked for consistency and validity.
2. Estimates of the means of the "independent" or design variables are computed. The frequency or the observation in all but binomial distribution models is taken from vector frequencies. In binomial distribution models, the frequency is taken as the product of n = `parameter` [i] and `frequencies` [i]. Means are computed as

$$\bar{x} = \frac{\sum f_i x_i}{\sum f_i}$$

3. By default, and when `IMSLS_INITIAL_EST_INTERNAL` is specified, initial estimates of the coefficients are obtained (based upon the observation intervals) as multiple regression estimates relating transformed observation probabilities to the observation design vector. For example, in the binomial distribution models, θ may be estimated as

$$\hat{\theta} = \texttt{y}[i] / \texttt{parameter}[i]$$

and, when `model` = 3, the linear relationship is given by

$$\ln\left(\hat{\theta}/\left(1-\hat{\theta}\right)\right) \approx X\beta$$

while if `model` = 4, Φ^{-1} (θ) = Xβ. When computing initial estimates, standard modifications are made to prevent illegal operations such as division by zero. Regression estimates are obtained at this point, as well as later, by use of function `imsls_f_regression` (Chapter 2, "Regression").

4. Newton-Raphson iteration for the maximum likelihood estimates is implemented via iteratively re-weighted least squares. Let

$$\Psi\left(x_i^T\beta\right)$$

denote the log of the probability of the *i*-th observation for coefficients β. In the least-squares model, the weight of the *i*-th observation is taken as the absolute value of the second derivative of

$$\Psi\left(x_i^T\beta\right)$$

with respect to

$$\gamma_i = x_i^T\beta$$

(times the frequency of the observation), and the dependent variable is taken as the first derivative Ψ with respect to γ_i, divided by the square root of the weight times the frequency. The Newton step is given by

$$\Delta\beta = (\sum\left|\Psi''\left(\gamma_i\right)\right|x_i x_i^T)^{-1}\sum\Psi'\left(\gamma_i\right)x_i$$

where all derivatives are evaluated at the current estimate of γ and $\beta_{n+1} = \beta - \Delta\beta$. This step is computed as the estimated regression coefficients in the least-squares model. Step halving is used when necessary to ensure a decrease in the criterion.

5. Convergence is assumed when the maximum relative change in any coefficient update from one iteration to the next is less than `eps` or when the relative change in the log-likelihood from one iteration to the next is less than `eps` / 100. Convergence is also assumed after `maxit` iterations or when step halving leads to a step size of less than 0.0001 with no increase in the log-likelihood.

6. Residuals are computed according to methods discussed by Pregibon (1981). Let l_i (γ_i) denote the log-likelihood of the *i*-th observation evaluated at γ_i. Then, the standardized residual is computed as

$$r_i = \frac{l_i'\left(\hat{\gamma}_i\right)}{\sqrt{l_i'\left(\hat{\gamma}_i\right)}}$$

where

$$\hat{\gamma}_i$$

is the value of γ_i when evaluated at the optimal

$$\hat{\beta}$$

The denominator of this expression is used as the "standard error of the residual" while the numerator is "raw" residual. Following Cook and Weisberg (1982), the influence of the *i*-th observation is assumed to be

$$l_i'(\hat{\gamma}_i)^T l''(\hat{\gamma})^{-1} l_i'(\hat{\gamma}_i)$$

This quantity is a one-step approximation to the change in the estimates when the *i*-th observation is deleted. Here, the partial derivatives are with respect to β.

Programming Notes

1. Indicator (dummy) variables are created for the classification variables using function `imsls_f_regressors_for_glm` (see Chapter 2, "Regression") using keyword `IMSLS_LEAVE_OUT_LAST` as the argument to the `IMSLS_DUMMY` optional argument.

2. To enhance precision, "centering" of covariates is performed if the model has an intercept and `n_observations - n_rows_missing` > 1. In doing so, the sample means of the design variables are subracted from each observation prior to its inclusion in the model. On convergence, the intercept, its variance, and its covariance with the remaining estimates are transformed to the uncentered estimate values.

3. Two methods for specifying a binomial distribution model are possible. In the first method, frequencies contains the frequency of the observation while `y` is 0 or 1 depending upon whether the observation is a success or failure. In this case, N = parameter [*i*] is always 1. The model is treated as repeated Bernoulli trials, and interval observations are not possible. A second method for specifying binomial models is to use `y` to represent the number of successes in parameter [*i*] trials. In this case, frequencies will usually be 1.

Examples

Example 1

The first example is from Prentice (1976) and involves the mortality of beetles after five hours exposure to eight different concentrations of carbon disulphide. The table below lists the number of beetles exposed (N) to each concentration level of carbon disulphide (x, given as log dosage) and the number of deaths which result (y). The data is given as follows:

Log Dosage	Number of Beetles Exposed	Number of Deaths
1.690	59	6
1.724	60	13
1.755	62	18
1.784	56	28
1.811	63	52
1.836	59	53
1.861	62	61
1.883	60	60

The number of deaths at each concentration level are fitted as a binomial response using logit (`model` = 3), probit (`model` = 4), and log-log (`model` = 5) models. Note that the log-log model yields a smaller absolute log likelihood (14.81) than the logit model (18.78) or the probit model (18.23). This is to be expected since the response curve of the log-log model has an asymmetric appearance, but both the logit and probit models are symmetric about θ = 0.5.

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

main ()

{

    static float x[8][3] = {  1.69,  6, 59,
                             1.724, 13, 60,
                             1.755, 18, 62,
                             1.784, 28, 56,
                             1.811, 52, 63,
                             1.836, 53, 59,
                             1.861, 61, 62,
                             1.883, 60, 60};

    float *coef_statistics, criterion;
    int  n_obs=8, n_class=0, n_continuous=1;
    int n_coef, model=3, ipar=2;
    char *fmt = "%12.4f";
    static char *clabels[] = {"", "coefficients", "s.e", "z", "p"};

    n_coef = imsls_f_categorical_glm (n_obs, n_class, n_continuous,
              model, &x[0][0],
              IMSLS_X_COL_DIST_PARAMETER, ipar,
              IMSLS_COEF_STAT, &coef_statistics,
              IMSLS_CRITERION, &criterion, 0);

    imsls_f_write_matrix ("Coefficient statistics for model 3", n_coef, 4,
                           coef_statistics,
              IMSLS_WRITE_FORMAT, fmt, IMSLS_NO_ROW_LABELS, IMSLS_COL_LABELS,
                           clabels,0);
```

```
     printf ("\nLog likelihood    %f \n", criterion);

    model=4;

    n_coef = imsls_f_categorical_glm (n_obs, n_class, n_continuous,
             model, &x[0][0],
             IMSLS_X_COL_DIST_PARAMETER, ipar,
             IMSLS_COEF_STAT, &coef_statistics,
             IMSLS_CRITERION, &criterion, 0);


    imsls_f_write_matrix ("Coefficient statistics for model 4", n_coef, 4,
                          coef_statistics,
             IMSLS_WRITE_FORMAT, fmt, IMSLS_NO_ROW_LABELS, IMSLS_COL_LABELS,
                          clabels,0);
     printf ("\nLog likelihood    %f \n", criterion);

    model=5;

    n_coef = imsls_f_categorical_glm (n_obs, n_class, n_continuous,
             model, &x[0][0],
             IMSLS_X_COL_DIST_PARAMETER, ipar,
             IMSLS_COEF_STAT, &coef_statistics,
             IMSLS_CRITERION, &criterion, 0);


    imsls_f_write_matrix ("Coefficient statistics for model 5", n_coef, 4,
                          coef_statistics,
             IMSLS_WRITE_FORMAT, fmt, IMSLS_NO_ROW_LABELS, IMSLS_COL_LABELS,
                          clabels,0);
     printf ("\nLog likelihood    %f \n", criterion);

}
```

Output

```
          Coefficient statistics for model 3
coefficients           s.e           z            p
    -60.7568        5.2093      -11.6632       0.0000
     34.2985        2.9164       11.7607       0.0000

Log likelihood    -18.778187

          Coefficient statistics for model 4
coefficients           s.e           z            p
    -34.9441        2.6527      -13.1732       0.0000
     19.7367        1.4852       13.2888       0.0000

Log likelihood    -18.232355

          Coefficient statistics for model 5
coefficients           s.e           z            p
    -39.6133        3.2428      -12.2156       0.0000
     22.0685        1.8047       12.2284       0.0000

Log likelihood    -14.807850
```

Example 2

Consider the use of a loglinear model to analyze survival-time data. Laird and Oliver (1981) investigate patient survival post heart valve replacement surgery. Surveilance after surgery of the 109 patients included in the study ranged from 3 to 97 months. All patients were classified by heart valve type (aortic or mitral) and by age (less than 55 years or at least 55 years). The data could be considered as a three-way contingency table where patients are classified by valve type, age, and survival (yes or no). However, it would be inappropriate to analyze this data using the standard methodology associated with contingency tables; since, this methodology ignores survival *time*.

Consider a variable, say exposure time (E_{ij}), that is defined as the sum of the length of times patients of each cross-classification are at risk. The length of time for a patient that dies is the number of months from surgery until death and for a survivor, the length of time is the number of months from surgery until the study ends or the patient withdraws from the study. Now we can model the effect of A = age and V = valve type on the expected number of deaths conditional on exposure time. Thus, for the data (shown in the table below), assume the number of deaths are independent Poisson random variables with means m_{ij} and fit the following model,

$$\log\left(\frac{m_{ij}}{E_{ij}}\right) = u + \lambda_i^A + \lambda_j^V$$

where u is the overall mean,

$$\lambda_i^A$$

is the effect of age, and

$$\lambda_j^V$$

is the effect of the valve type.

		Heart Valve Type	
Age		**Aortic (0)**	**Mitral (1)**
< 55 years (Age = 0)	Deaths	4	1
	Exposure	1259	2082
≥ 55 years (Age = 1)	Deaths	7	9
	Exposure	1417	1647

From the coefficient statistics table of the output, note that the risk is estimated to be $e^{1.22} = 3.39$ times higher for older patients in the study. This increase in risk is significant ($p = 0.02$). However, the decrease in risk for the mitral valve patients is estimated to be $e^{-0.33} = 0.72$ times that of the aortic valve patients and this risk is not significant ($p = 0.45$).

```
#include <imsls.h>

main ()
{
    int   nobs = 4;
    int   n_class = 2;
    int   n_cont = 0;
    int   model = 0;
    float x[16] = {
        4, 1259, 0, 0,
        1, 2082, 0, 1,
        7, 1417, 1, 0,
        9, 1647, 1, 1
    };
    int iclass[2] = {2, 3};
    int icont[1] = {-1};
    int   n_coef;
    float *coef;

    char  *clabels[5] = {"", "coefficient", "std error", "z-statistic", "p-
                         value"};
    char  *fmt = "%10.6W";

    n_coef = imsls_f_categorical_glm(nobs, n_class, n_cont, model, x,
       IMSLS_COEF_STAT, &coef,
       IMSLS_X_COL_VARIABLES, iclass, icont, 0,
       IMSLS_X_COL_DIST_PARAMETER, 1,
       0);

        imsls_f_write_matrix("Coefficient Statistics", n_coef, 4, coef,
        IMSLS_COL_LABELS, clabels, IMSLS_ROW_NUMBER_ZERO,
        IMSLS_WRITE_FORMAT, fmt, 0);
}
```

Output

```
              Coefficient Statistics
   coefficient   std error  z-statistic     p-value
0      -5.4210      0.3921     -13.8246      0.0000
1      -1.2209      0.5138      -2.3763      0.0177
2       0.3299      0.4382       0.7528      0.4517
```

Warning Errors

`IMSLS_TOO_MANY_HALVINGS`	Too many step halvings. Convergence is assumed.
`IMSLS_TOO_MANY_ITERATIONS`	Too many iterations. Convergence is assumed.

Fatal Errors

`IMSLS_TOO_FEW_COEF`	`IMSLS_INITIAL_EST_INPUT` is specified and "`n_coef_input`" = #. The model specified requires # coefficients.
`IMSLS_MAX_CLASS_TOO_SMALL`	The number of distinct values of the classification variables exceeds "`max_class`" = #.
`IMSLS_INVALID_DATA_8`	"`n_class_values`[#]" = #. The number of distinct values for each classification variable must be greater than one.
`IMSLS_NMAX_EXCEEDED`	The number of observations to be deleted has exceeded "`lp_max`" = #. Rerun with a different model or increase the workspace.

Chapter 6: Nonparametric Statistics

Routines

Usage Notes

Much of what is considered nonparametric statistics is included in other chapters. Topics of possible interest in other chapters are: nonparametric measures of location and scale (Chapter 1, "Basic Statistics"), nonparametric measures in a contingency table (Chapter 5, "Categorical and Discrete Data Analysis"), measures of correlation in a contingency table (Chapter 3, "Correlation and Covariance"), and tests of goodness of fit and randomness (Chapter 7, "Tests of Goodness of Fit and Randomness").

Missing Values

Most routines described in this chapter automatically handle missing values (NaN, "Not a Number"; see the introduction of this manual).

Tied Observations

Many of the routines described in this chapter contain an argument `IMSLS_FUZZ` in the input. Observations that are within `fuzz` of each other in absolute value are said to be tied. Moreover, in some routines, an observation within `fuzz` of some value is said to be equal to that value. In routine `imsls_f_wilcoxon_sign_rank` (page 445), for example, such observations are eliminated from the analysis. If `fuzz` = 0.0, observations must be identically equal before they are considered to be tied. Other positive values of `fuzz` allow for numerical imprecision or roundoff error.

sign_test

Performs a sign test.

Synopsis

#include `<imsls.h>`

float `imsls_f_sign_test` (*int* `n_observations`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_sign_test`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `x[]` (Input)
Array of length `n_observations` containing the input data.

Return Value

Binomial probability of `n_positive_deviations` or more positive differences in `n_observations` – `n_zero_deviation` trials. Call this value *probability*. If no option is chosen, the null hypothesis is that the median equals 0.0.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float imsls_f_sign_test (int n_observations, float x[],
        IMSLS_PERCENTAGE, float percentage,
        IMSLS_PERCENTILE, float percentile,
        IMSLS_N_POSITIVE_DEVIATIONS,
                int *n_positive_deviations,
        IMSLS_N_ZERO_DEVIATIONS, int *n_zero_deviations,
        0)
```

Optional Arguments

`IMSLS_PERCENTAGE`, *float* `percentage` (Input)
: Value in the range (0, 1). Argument `percentile` is the 100 × `percentage` percentile of the population.
: Default: `percentage` = 0.5

`IMSLS_PERCENTILE`, *float* `percentile` (Input)
: Hypothesized percentile of the population from which `x` was drawn.
: Default: `percentile` = 0.0

`IMSLS_N_POSITIVE_DEVIATIONS`, *int* `*n_positive_deviations` (Output)
: Number of positive differences `x`[j – 1] – `percentile` for j = 1, 2, …, `n_observations`.

`IMSLS_N_ZERO_DEVIATIONS`, *int* `*n_zero_deviations` (Output)
: Number of zero differences (ties) `x`[j – 1] – `percentile` for j = 1, 2, …, `n_observations`.

Description

Function `imsls_f_sign_test` tests hypotheses about the proportion p of a population that lies below a value q, where p corresponds to argument `percentage` and q corresponds to argument `percentile`. In continuous distributions, this can be a test that q is the 100 p-th percentile of the population from which `x` was obtained. To carry out testing, `imsls_f_sign_test` tallies the number of values above q in `n_positive_deviations`. The binomial probability of `n_positive_deviations` or more values above q is then computed using the proportion p and the sample size `n_observations` (adjusted for the missing observations and ties).

Hypothesis testing is performed as follows for the usual null and alternative hypotheses:

- H_0: $Pr(\mathrm{x} \le q) \ge p$ (the p-th quantile is at least q)
  H_1: $Pr(\mathrm{x} \le q) < p$
  Reject H_0 if *probability* is less than or equal to the significance level
- H_0: $Pr(\mathrm{x} \le q) \le p$ (the p-th quantile is at least q)
  H_1: $Pr(\mathrm{x} \le q) > p$
  Reject H_0 if *probability* is greater than or equal to 1 minus the significance level
- H_0: $Pr\ (\mathrm{x} = q) = p$ (the p-th quantile is q)
  H_1: $Pr((\mathrm{x} \le q) < p)$ or $Pr((\mathrm{x} \le q) > p)$
  Reject H_0 if *probability* is less than or equal to half the significance level or greater than or equal to 1 minus half the significance level

The assumptions are as follows:

1. They are independent and identically distributed.
2. Measurement scale is at least ordinal; i.e., an ordering less than, greater than, and equal to exists in the observations.

Many uses for the sign test are possible with various values of *p* and *q*. For example, to perform a matched sample test that the difference of the medians of *y* and *z* is 0.0, let $p = 0.5$, $q = 0.0$, and $x_i = y_i - z_i$ in matched observations *y* and *z*. To test that the median difference is *c*, let $q = c$.

Examples

Example 1

This example tests the hypothesis that at least 50 percent of a population is negative. Because 0.18 < 0.95, the null hypothesis at the 5-percent level of significance is not rejected.

```
#include <imsls.h>

void main ()
{
    int         n_observations = 19;
    float       probability;
    float       x[19] = {92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0,
          -25.0, -4.0, 22.0, 2.0, 41.0, 13.0, 8.0, 33.0,
          45.0, -33.0, -45.0, -12.0};

    probability = imsls_f_sign_test(n_observations, x, 0);

    printf("probability = %10.6f\n", probability);
}
```

Output

```
probability =   0.179642
```

Example 2

This example tests the null hypothesis that at least 75 percent of a population is negative. Because 0.923 < 0.95, the null hypothesis at the 5-percent level of significance is rejected.

```
#include <imsls.h>

void main ()
{
    int         n_observations = 19;
    int         n_positive_deviations, n_zero_deviations;
    float       probability;
    float       percentage = 0.75;
    float       percentile = 0.0;
    float       x[19] = {92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0,
          -25.0, -4.0, 22.0, 2.0, 41.0, 13.0, 8.0, 33.0,
        45.0, -33.0, -45.0, -12.0};


    probability = imsls_f_sign_test(n_observations, x, IMSLS_PERCENTAGE,
            percentage, IMSLS_PERCENTILE, percentile,
            IMSLS_N_POSITIVE_DEVIATIONS, &n_positive_deviations,
            IMSLS_N_ZERO_DEVIATIONS, &n_zero_deviations, 0);

    printf("probability = %10.6f.\n", probability);
    printf("Number of positive deviations is %d.\n",
           n_positive_deviations);
    printf("Number of ties is %d.\n", n_zero_deviations);
}
```

Output

```
probability =   0.922543.
Number of positive deviations is 12.
Number of ties is 0.
```

wilcoxon_sign_rank

Performs a Wilcoxon signed rank test.

Synopsis

#include `<imsls.h>`

float `*imsls_f_wilcoxon_sign_rank` (*int* `n_observations`,
    float `x[]`, ..., 0)

The type *double* function is `imsls_d_wilcoxon_sign_rank`.

Required Arguments

int `n_observations` (Input)
    Number of observations in `x`.

float `x[]` (Input)
    Array of length `n_observations` containing the data.

Return Value

Pointer to an array of length two containing the values described below.

The asymptotic probability of not exceeding the standardized (to an asymptotic variance of 1.0) minimum of (W+, W-) using method 1 under the null hypothesis that the distribution is symmetric about 0.0.

And, the asymptotic probability of not exceeding the standardized (to an asymptotic variance of 1.0) minimum of (W+, W-) using method 2 under the null hypothesis that the distribution is symmetric about 0.0.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_wilcoxon_sign_rank` (*int* `n_observations`,
    float `x[]`,
    `IMSLS_FUZZ`, *float* `fuzz`,
    `IMSLS_STAT`, *float* `**stat`,
    `IMSLS_STAT_USER`, *float* `stat[]`,
    `IMSLS_N_MISSING`, *float* `*n_missing`,
    `IMSLS_RETURN_USER`, *float* `prob[]`,
    0)

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz` (Input)
Nonnegative constant used to determine ties in computing ranks in the combined samples. A tie is declared when two observations in the combined sample are within `fuzz` of each other.
Default value for `fuzz` is 0.0.

`IMSLS_STAT`, *float* `**stat` (Output)
Address of a pointer to an internally allocated array of length 10 containing the following statistics:

Row	Statistics
0	The positive rank sum, W+, using method
1	The absolute value of the negative rank sum, W-, using method 1.
2	The standardized (to anasymptotic variance of 1.0) minimum of (W+, W-) using method
3	The asymptotic probability of not exceeding `stat(2)` under the null hypothesis that the distribution is symmetric about 0.0.
4	The positive rank sum, W+, using method 2.
5	The absolute value of the negative rank sum, W-, using method 2.
6	The standardized (to an asymptotic variance of 1.0) minimum of (W+, W-) using method 2.
7	The asymptotic probability of not exceeding `stat(6)` under the null hypothesis that the distribution is symmetric about 0.0.
8	The number of zero observations.
9	The total number of observations that are tied, and that are not within fuzz of zero.

`IMSLS_STAT_USER`, *float* `stat[]` (Output)
Storage for array `stat` is provided by the user.
See `IMSLS_STAT`.

`IMSLS_N_MISSING`, *float* `*n_missing`, (Output)
Number of missing values in y.

`IMSLS_RETURN_USER`, *float* `prob[]`, (Output)
User allocated storage for return values.
See Return Value.

Description

Function `imsls_f_wilcoxon_sign_rank` performs a Wilcoxon signed rank test of symmetry about zero. In one sample, this test can be viewed as a test that the population median is zero. In matched samples, a test that the medians of the two populations are equal can be computed by first computing difference scores. These difference scores would then be used as input to `imsls_f_wilcoxon_sign_rank`. A general reference for the methods used is Conover (1980).

Routine `imsls_f_wilcoxon_sign_rank` computes statistics for two methods for handling zero and tied observations. In the first method, observations within `fuzz` of zero are not counted, and the average rank of tied observations is used. (Observations within `fuzz` of each other are said to be tied.) In the second method, observations within `fuzz` of zero are randomly assigned a positive or negative sign, and the ranks of tied observations are randomly permuted.

The *W*+ and *W*– statistics are computed as the sums of the ranks of the positive observations and the sum of the ranks of the negative observations, respectively. Asymptotic probabilities are computed using standard methods (see, e.g., Conover 1980, page 282).

The *W*+ and *W*– statistics may be used to test the following hypotheses about the median, *M*. In deciding whether to reject the null hypothesis, use the bracketed statistic if method 2 for handling ties is preferred. Possible null hypotheses and alternatives are given as follows:

- $H_0 : M \le 0 \qquad H_1 : M > 0$
  Reject if `stat[0]` [or `stat[4]`] is too large.
- $H_0 : M \ge 0 \qquad H_1 : M < 0$
  Reject if `stat[1]` [or `stat[5]`] is too large.
- $H_0 : M = 0 \qquad H_1 : M \ne 0$
  Reject if `stat[2]`[or `stat[6]`] is too small. Alternatively, if an asymptotic test is desired, reject if 2 `*` `stat[3]` [or 2 `*` `stat[7]`] is less than the significance level.

Tabled values of the test statistic can be found in the references. If possible, tabled values should be used. If the number of nonzero observations is too large, then the asymptotic probabilities computed by `imsls_f_wilcoxon_sign_rank` can be used.

The assumptions required for the hypothesis tests are as follows:

1. The distribution of each X_i is symmetric.
2. The X_i are mutually independent.
3. All X_i's have the same median.
4. An ordering of the observations exists (i.e., $X_1 > X_2$ and $X_2 > X_3$ implies that $X_1 > X_3$).

If other assumptions are made, related hypotheses that are more (or less) restrictive can be tested.

Example

This example illustrates the application of the Wilcoxon signed rank test to a test on a difference of two matched samples (matched pairs) {X1 = 223, 216, 211, 212, 209, 205, 201; and X2 = 208, 205, 202, 207, 206, 204, 203}. A test that the median difference is 10.0 (rather than 0.0) is performed by subtracting 10.0 from each of the differences prior to calling `wilcoxon_sign_rank`. As can be seen from the output, the null hypothesis is rejected. The warning error will always be printed when the number of observations is 50 or less unless printing is turned off for warning errors.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
   float *stat=NULL, *result=NULL;
   int nobs = 7, nmiss;
   float fuzz = .0001;
   float x[] = {-25., -21., -19., -15., -13., -11., -8.};
   result = imsls_f_wilcoxon_sign_rank(nobs, x,
                                    IMSLS_N_MISSING, &nmiss,
                                    IMSLS_FUZZ, fuzz,
                                    IMSLS_STAT, &stat,
                                    0);
   printf("Statistic\t\t\tMethod 1\tMethod 2\n");
   printf("W+\t\t\t\t %3.0f\t\t %3.0f\n", stat[0], stat[4]);
   printf("W-\t\t\t\t %3.0f\t\t %3.0f\n", stat[1], stat[5]);
   printf("Standardized Minimum\t\t%6.4f\t\t%6.4f\n", stat[2], stat[6]);
   printf("p-value\t\t\t\t %6.4f\t\t %6.4f\n\n", stat[3], stat[7]);
   printf("Number of zeros\t\t\t%3.0f\n", stat[8]);
   printf("Number of ties\t\t\t%3.0f\n", stat[9]);
   printf("Number of missing\t\t  %d\n", nmiss);
}
```

Output

```
*** WARNING  ERROR 4 from imsls_f_wilcoxon_sign_rank.  NOBS = 7.  The number
***          of observations, NOBS, is less than 50, and exact
***          tables should be referenced for probabilities.

Statistic                   Method 1     Method 2
W+......................       0            0
W-......................      28           28
Standardized Minimum.....  -2.3664      -2.3664
p-value..................   0.0090       0.0090

Number of zeros..........       0
Number of ties...........       0
Number of missing........       0
```

noether_cyclical_trend

Performs the Noether test for cyclical trend.

Synopsis

#include `<imsls.h>`

float `*imsls_f_noether_cyclical_trend` (*int* `n_observations`,
float `x[]`, ..., 0)

The type *double* function is `imsls_d_noether_cyclical_trend`.

Required Arguments

int `n_observations` (Input)
Number of observations in `x`. `n_observations` must be greater than or equal to 3.

float `x[]` (Input)
Array of length `n_observations` containing the data in chronological order.

Return Value

Array, `p`, of length 3 containing the probabilities of `stat[1]` or more, `stat[2]` or more, or `stat[3]` or more monotonic sequences.

If `stat[0]` is less than 1, `p[0]` is set to NaN (not a number).

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_noether_cyclical_trend` ((*int* `n_observations`,
float `x[]`,
`IMSLS_FUZZ, float fuzz,`
`IMSLS_STAT, int **stat,`
`IMSLS_STAT_USER, int stat[],`
`IMSLS_N_MISSING, int *n_missing,`
`IMSLS_RETURN_USER, float p[],`
0)

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz` (Input)
Nonnegative constant used to determine ties in computing ranks in the combined samples. A tie is declared when two observations in the combined sample are within fuzz of each other.
Default value for `fuzz` is 0.0.

`IMSLS_STAT`, *int* `**stat` (Output)
Address of a pointer to an internally allocated array of length 6 containing the following statistics:

Row	Statistics
`stat[0]`	The number of consecutive sequences of length three used to detect cyclical trend when tying middle elements are eliminated from the sequence, and the next consecutive observation is used.
`stat[1]`	The number of monotonic sequences of length three in the set defined by `stat[0]`.
`stat[2]`	The number of nonmonotonic sequences where tied threesomes are counted as nonmonotonic.
`stat[3]`	The number of monotonic sequences where tied threesomes are counted as monotonic.
`stat[4]`	The number of middle observations eliminated because they were tied in forming the `stat[0]` sequences.
`stat[5]`	The number of tied sequences found in forming the `stat[2]` and `stat[3]` sequences. A sequence is called a tied sequence if the middle element is tied with either of the two other elements.

`IMSLS_STAT_USER`, *int* `stat[]` (Output)
Storage for array stat is provided by the user.
See `IMSLS_STAT`.

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values in `x`.

`IMSLS_RETURN_USER`, *float* `p[]` (Input)
User allocated array of length 3 containing the return values.

Description

Routine `imsls_f_noether_cyclical_trend` performs the Noether test for cyclical trend (Noether 1956) for a sequence of measurements. In this test, the observations are first divided into sets of three consecutive observations. Each set is then inspected, and if the set is monotonically increasing or decreasing, the count variable is incremented.

The count variables, `stat[1]`, `stat[2]`, and `stat[3]`, differ in the manner in which ties are handled. A tie can occur in a set (of size three) only if the middle element is tied with either of the two ending elements. Tied ending elements are not considered. In `stat[1]`, tied middle observations are eliminated, and a new set of size 3 is obtained by using the next observation in the sample. In `stat[2]`, the original set of size three is used, and tied middle observations are counted as nonmonotonic. In `stat[3]`, tied middle observations are counted as monotonic.

The probabilities of occurrence of the counts are obtained from the binomial distribution with $p = 1/3$, where p is the probability that a random sample of size three from a continuous distribution is monotonic. The binomial sample size is, of course, the number of sequences of size three found (adjusted for ties).

Hypothesis test:

$H_0 : q = \Pr(X_i > X_{i-1} > X_{i-2}) + \Pr(X_i < X_{i-1} < X_{i-2}) \leq 1/3$ $H_1 : q > 1/3$
Reject if `p[0]` (or `p[1]` or `p[2]` depending on the method used for handling ties) is less than the significance level of the test.

Assumption: The observations are independent and are from a continuous distribution.

Example

A test for cyclical trend in a sequence of 1000 randomly generated observations is performed. Because of the sample used, there are no ties and all three test statistics yield the same result.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float *pvalue=NULL;
        int nobs = 1000, nmiss, *stat = NULL;
        float *x = NULL;
        imsls_random_seed_set(123457);
        x = imsls_f_random_uniform(nobs, 0);
        pvalue = imsls_f_noether_cyclical_trend(nobs, x,
                                  IMSLS_STAT, &stat,
                                  IMSLS_N_MISSING, &nmiss,
                                  0);
        imsls_f_write_matrix("P", 0, 2, pvalue, 0);
        imsls_i_write_matrix("STAT", 0, 5, stat, 0);
        printf("\n n missing = %d\n", nmiss);
}
```

Output

```
P
 0        1        2
0.6979   0.6979   0.6979
STAT
 0     1     2     3     4     5
333   107   107   107     0     0
n missing = 0
```

cox_stuart_trends_test

Performs the Cox and Stuart sign test for trends in location and dispersion.

Synopsis

#include `<imsls.h>`

float `*imsls_f_cox_stuart_trends_test` (*int* `n_observations`,
    float `x[]`, …, 0)

The type *double* function is `imsls_d_ cox_stuart_trends_test`.

Required Arguments

int `n_observations` (Input)
    Number of observations in `x`. `n_observations` must be greater than or equal to 3.

float `x[]` (Input)
    Array of length `n_observations` containing the data in chronological order.

Return Value

Array, `pstat`, of length 8 containing the probabilities. **The first four elements of `pstat` are computed from two groups of observations.**

I	`pstat[I]`
0	Probability of `nstat[0] + nstat[2]` or more negative signs (ties are considered negative).
1	Probability of obtaining `nstat[1]` or more positive signs (ties are considered negative).
2	Probability of `nstat[0] + nstat[2]` or more negative signs (ties are considered positive).
3	Probability of obtaining `nstat[1]` or more positive signs (ties are considered positive).

The last four elements of `pstat` are computed from three groups of observations.

4	Probability of `nstat[0] + nstat[2]` or more negative signs (ties are considered negative).
5	Probability of obtaining `nstat[1]` or more positive signs (ties are considered negative).
6	Probability of `nstat[0] + nstat[2]` or more negative signs (ties are considered positive).
7	Probability of obtaining `nstat[1]` or more positive signs (ties are considered positive).

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_cox_stuart_trends_test` (*int* `n_observations`,
    float `x[]`,
    `IMSLS_DISPERSION`, *int* `k`, `int ids`,
    `IMSLS_FUZZ`, *float* `fuzz`,
    `IMSLS_STAT`, *int* `**nstat`,
    `IMSLS_STAT_USER`, *int* `nstat[]`,
    `IMSLS_N_MISSING`, *int* `*n_missing`,
    `IMSLS_RETURN_USER`, *float* `pstat[]`,
    0)

Optional Arguments

`IMSLS_DISPERSION`, *int* `k`, *int* `ids`, (Input)
If `IMSLS_DISPERSION` is called, the Cox and Stuart tests for trends in dispersion are computed. Otherwise, as default, the Cox and Stuart tests for trends in location are computed. *`k`* is the number of consecutive `x` elements to be used to measure dispersion.

If `ids` is zero, the range is used as a measure of dispersion. Otherwise, the centered sum of squares is used.

`IMSLS_FUZZ`, *float* `fuzz` (Input)
Value used to determine when elements in `x` are tied.
If `|x[i] - x[j]|` is less than or equal to `fuzz, x[i]` and `x[j]` are said to be tied. `fuzz` must be nonnegative. Default value for `fuzz` is 0.0.

`IMSLS_STAT`, *int* `**nstat` (Output)
Address of a pointer to an internally allocated array of length 8 containing the following statistics:

I	**nstat[I]**
0	Number of negative differences (two groups)
1	Number of positive differences (two groups)
2	Number of zero differences (two groups)
3	Number of differences used to calculate pstat[0] through pstat[3] (two groups).
4	Number of negative differences (three groups)
5	Number of positive differences (three groups)
6	Number of zero differences (three groups)
7	Number of differences used to calculate pstat [4] through pstat[7] (three groups).

`IMSLS_STAT_USER`, i*nt* `nstat[]` (Output)
Storage for array `nstat` is provided by the user.
See `IMSLS_STAT.`

`IMSLS_N_MISSING`, *int* `*n_missing` (Output)
Number of missing values in `X.`

`IMSLS_RETURN_USER`, *float* `pstat[]` (Input)
User allocated array of length 8 containing the return values.

Description

Function `imsls_f_cox_stuart_trends_test` tests for trends in dispersion or location in a sequence of random variables depending upon the call of `IMSLS_DISPERSION`. A derivative of the sign test is used
(see Cox and Stuart 1955).

Location Test

For the location test (`Default`) with two groups, the observations are first divided into two groups with the middle observation thrown out if there are an odd number of observations. Each observation in group one is then compared with the observation in group two that has the same lexicographical order. A count is made of the number of times a group-one observation is less than (`nstat[0]`), greater than (`nstat[1]`), or equal to (`nstat[2]`), its counterpart in group two. Two observations are counted as equal if they are within `fuzz` of one another.

In the three-group test, the observations are divided into three groups, with the center group losing observations if the division is not exact. The first and third groups are then compared as in the two-group case, and the counts are stored in `nstat[4]` through `nstat[6]`.

Probabilities in `pstat` are computed using the binomial distribution with sample size equal to the number of observations in the first group (`nstat[3]` or `nstat[7]`), and binomial probability $p = 0.5$.

Dispersion Test

The dispersion tests (when optional argument `IMSLS_DISPERSION` is called) proceed exactly as with the tests for location, but using one of two derived dispersion measures. The input value `k` is used to define `n_observations/k` groups of consecutive observations starting with observation 1. The first `k` observations define the first group, the next `k` observations define the second group, etc., with the last observations omitted if `n_observations` is not evenly divisible by `k`. A dispersion score is then computed for each group as either the range (`ids` = 0), or a multiple of the variance (`ids` ≠ 0) of the observations in the group. The dispersion scores form a derived sample. The tests proceed on the derived sample as above.

Ties

Ties are defined as occurring when a group one observation is within `fuzz` of its last group counterpart. Ties imply that the probability distribution of `x` is not strictly continuous, which means that $\Pr(x_1 > x_2) \neq 0.5$ under the null hypothesis of no trend (and the assumption of independent identically distributed observations). When ties are present, the computed binomial probabilities are not exact, and the hypothesis tests will be conservative.

Hypothesis tests

In the following, *i* indexes an observation from group 1, while *j* indexes the corresponding observation in group 2 (two groups) or group 3 (three groups).

- $H_0 : \Pr(X_i > X_j) = \Pr(X_i < X_j) = 0.5$
  $H_1 : \Pr(X_i > X_j) < \Pr(X_i < X_j)$

Hypothesis of upward trend. Reject if `pstat[2]` (or `pstat[6]`) is less than the significance level.

- $H_0 : \Pr(X_i > X_j) = \Pr(X_i < X_j) = 0.5$
  $H_1 : \Pr(X_i > X_j) > \Pr(X_i < X_j)$
  Hypothesis of downward trend. Reject if `pstat[1]` (or `pstat[5]`) is less than the significance level.
- $H_0 : \Pr(X_i > X_j) = \Pr(X_i < X_j) = 0.5$
  $H_1 : \Pr(X_i > X_j) \neq \Pr(X_i < X_j)$
  Two tailed test. Reject if 2 max(`pstat[1]`, `pstat[2]`) (or 2 max(`pstat[5]`, `pstat[6]`) is less than the significance level.

Assumptions

1. The observations are a random sample; i.e., the observations are independently and identically distributed.
2. The distribution is continuous.

Example

This example illustrates both the location and dispersion tests. The data, which are taken from Bradley (1968), page 176, give the closing price of AT&T on the New York stock exchange for 36 days in 1965. Tests for trends in location (`Default`), and for trends in dispersion (`IMSLS_DISPERSION`) are performed. Trends in location are found.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
float *pstat=NULL;
int nobs = 36, ids = 0, k = 2, nmiss, *stat = NULL;
float fuzz = 0.001;
float x[] = {9.5, 9.875, 9.25, 9.5, 9.375, 9.0, 8.75, 8.625, 8.0, 8.25,
8.25, 8.375, 8.125, 7.875, 7.5, 7.875, 7.875, 7.75,7.75, 7.75, 8.0, 7.5,
7.5, 7.125, 7.25, 7.25, 7.125, 6.75,6.5, 7.0, 7.0, 6.75, 6.625, 6.625,
7.125, 7.75};
       pstat = imsls_f_cox_stuart_trends_test(nobs, x,
                                           IMSLS_FUZZ, fuzz,
                                           IMSLS_STAT, &stat,
                                           IMSLS_N_MISSING, &nmiss,
                                            0);
       imsls_i_write_matrix("nstat", 1, 8, stat, 0);
       imsls_f_write_matrix("pstat", 1, 8, pstat,
                         IMSLS_WRITE_FORMAT, "%10.5f", 0);
```

```
        printf("n missing = %d\n", nmiss);
        pstat = imsls_f_cox_stuart_trends_test(nobs, x,
                                    IMSLS_DISPERSION, k, ids,
                                    IMSLS_FUZZ, fuzz,
                                    IMSLS_STAT, &stat,
                                    IMSLS_N_MISSING, &nmiss,
                                    0);
        imsls_i_write_matrix("nstat", 0, 7, stat, 0);
        imsls_f_write_matrix("pstat", 0, 7, pstat, 0);
        printf("n missing = %d\n", nmiss);
}
```

Output

```
*** WARNING  Error from imsls_cox_stuart_trends_test.  At least one tie is
detected in X.

            NSTAT
0    1    2    3    4    5    6    7
0   17    1   18    0   12    0   12

            PSTAT
      0             1             2             3             4
1.00000       0.00007       1.00000       0.00000       1.00000


      5             6             7
0.00024       1.00000       0.00024
 n missing = 0

*** WARNING  Error from imsls_cox_stuart_trends_test.  At least one tie is
detected in X.

            NSTAT
0   1   2   3   4   5   6   7
4   3   2   9   4   2   0   6

                      PSTAT
       0             1             2             3             4
0.253906      0.910156      0.746094      0.500000      0.343750
       5             6             7
0.890625      0.343750      0.890625
 n missing = 0
```

tie_statistics

Compute tie statistics for a sample of observations.

Synopsis

#include `<imsls.h>`

float `*imsls_f_tie_statistics` (*int* `n_oservations`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_tie_statistics`.

Required Arguments

int `n_observations` (Input)
Number of observations in `x`.

float `x[]` (Input)
Array of length `n_observations` containing the observations.

`x` must be ordered monotonically increasing with all missing values removed.

Return Value

Array of length 4 containing the tie statistics.

$$\text{ties}[0] = \sum_{j=1}^{\tau} \left[t_j \left(t_j - 1 \right) \right] / 2$$

$$\text{ties}[1] = \sum_{j=1}^{\tau} \left[t_j \left(t_j - 1 \right) \left(t_j + 1 \right) \right] / 12$$

$$\text{ties}[2] = \sum_{j=1}^{\tau} t_j \left(t_j - 1 \right) \left(2t_j + 5 \right)$$

$$\text{ties}[3] = \sum_{j=1}^{\tau} t_j \left(t_j - 1 \right) \left(t_j - 2 \right)$$

where t_j is the number of ties in the j-th group (rank) of ties, and τ is the number of tie groups in the sample.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float * imsls_f_tie_statistics (int n_oservations, float x[],
        IMSLS_FUZZ, float fuzz,
        IMSLS_RETURN_USER, float ties[],
        0)
```

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz`, (Input)
Value used to determine ties.
Observations *i* and *j* are tied if the successive differences `x[k + 1] - x[k]` between observations *i* and *j*, inclusive, are all less than `fuzz`. `fuzz` must be nonnegative. Default: `fuzz = 0.0`

`IMSLS_RETURN_USER`, *float* `ties[]`, (Output)
If specified `ties[]` returns the tie statistics. Storage for `ties[]` is provided by the user. See **Return Value**.

Description

Function `imsls_f_tie_statistics` computes tie statistics for a monotonically increasing sample of observations. "Tie statistics" are statistics that may be used to correct a continuous distribution theory nonparametric test for tied observations in the data. Observations *i* and *j* are tied if the successive differences X($k + 1$) − X(k), inclusive, are all less than `fuzz`. Note that if each of the monotonically increasing observations is equal to its predecessor plus a constant, if that constant is less than `fuzz`, then all observations are contained in one tie group. For example, if `fuzz` = 0.11, then the following observations are all in one tie group.

```
0.0, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 1.00
```

Example

We want to compute tie statistics for a sample of length 7.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float *ties=NULL;
        int nobs = 7;
        float fuzz = .001;
        float x[] = {1.0, 1.0001, 1.0002, 2., 3., 3., 4.};
        ties = imsls_f_tie_statistics(nobs, x,
                              IMSLS_FUZZ, fuzz,
                              0);
       imsls_f_write_matrix("TIES\n", 0, 3, ties,
                    IMSLS_WRITE_FORMAT, "%5.2f",
                    0);
       }
```

Output

```
TIES
0       1       2       3
4.00    2.50   84.00    6.00
```

wilcoxon_rank_sum

Performs a Wilcoxon rank sum test.

Synopsis

#include `<imsls.h>`

float `imsls_f_wilcoxon_rank_sum` (*int* `n1_observations`, *float* `x1[]`,
int `n2_observations`, *float* `x2[]`, ..., 0)

The type *double* function is `imsls_d_wilcoxon_rank_sum`.

Required Arguments

int `n1_observations` (Input)
Number of observations in the first sample.

float `x1[]` (Input)
Array of length `n1_observations` containing the first sample.

int `n2_observations` (Input)
Number of observations in the second sample.

float `x2[]` (Input)
Array of length `n2_observations` containing the second sample.

Return Value

The two-sided *p*-value for the Wilcoxon rank sum statistic that is computed with average ranks used in the case of ties.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_wilcoxon_rank_sum` (*int* `n1_observations`, *float* `x1[]`,
int `n2_observations`, *float* `x2[]`,
`IMSLS_FUZZ`, *float* `fuzz`,
`IMSLS_STAT`, *float* `**stat`,
`IMSLS_STAT_USER`, *float* `stat[]`,
`0)`

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz` (Input)
Nonnegative constant used to determine ties in computing ranks in the combined samples. A tie is declared when two observations in the combined sample are within `fuzz` of each other.
Default: `fuzz` = 100 × `imsls_f_machine(4)` × max $\{|x_{i1}|, |x_{j2}|\}$

`IMSLS_STAT`, *float* `**stat` (Output)
Address of a pointer to an internally allocated array of length 10 containing the following statistics:

Row	Statistics
0	Wilcoxon W statistic (the sum of the ranks of the x observations) adjusted for ties in such a manner that W is as small as possible
1	$2 \times E(W) - W$, where $E(W)$ is the expected value of W
2	probability of obtaining a statistic less than or equal to $\min\{W, 2 \times E(W) - W\}$
3	W statistic adjusted for ties in such a manner that W is as large as possible
4	$2 \times E(W) - W$, where $E(W)$ is the expected value of W, adjusted for ties in such a manner that W is as large as possible
5	probability of obtaining a statistic less than or equal to $\min\{W, 2 \times E(W) - W\}$, adjusted for ties in such a manner that W is as large as possible
6	W statistic with average ranks used in case of ties
7	estimated standard error of `stat [6]` under the null hypothesis of no difference
8	standard normal score associated with `stat [6]`
9	two-sided p-value associated with `stat[8]`

`IMSLS_STAT_USER`, *float* `stat[]` (Output)
Storage for array `stat` is provided by the user. See `IMSLS_STAT`.

Description

Function `imsls_f_wilcoxon_rank_sum` performs the Wilcoxon rank sum test for identical population distribution functions. The Wilcoxon test is a linear transformation of the Mann-Whitney U test. If the difference between the two populations can be attributed solely to a difference in location, then the Wilcoxon test becomes a test of equality of the population means (or medians) and is the nonparametric equivalent of the two-sample *t*-test. Function `imsls_f_wilcoxon_rank_sum` obtains ranks in the combined sample after first eliminating missing values from the data. The rank sum statistic is then computed as the sum of the ranks in the `x1` sample. Three methods for handling ties are used. (A tie is counted when two observations are within `fuzz` of each other.) Method 1 uses the largest possible rank for tied observations in the smallest sample, while Method 2 uses the smallest possible rank for these observations. Thus, the range of possible rank sums is obtained.

Method 3 for handling tied observations between samples uses the average rank of the tied observations. Asymptotic standard normal scores are computed for the *W* score (based on a variance that has been adjusted for ties) when average ranks are used (see Conover 1980, p. 217), and the probability associated with the two-sided alternative is computed.

Hypothesis Tests

In each of the following tests, the first line gives the hypothesis (and its alternative) under the assumptions 1 to 3 below, while the second line gives the hypothesis when assumption 4 is also true. The rejection region is the same for both hypotheses and is given in terms of Method 3 for handling ties. Another output statistic should be used, (`stat`[0] or `stat`[3]), if another method for handling ties is desired.

Test	Null Hypothesis	Alternative Hypothesis	Action
1	H_0:Pr(`x1` < `x2`) = 0.5	H_1:Pr(`x1` < `x2`) ≠ 0.5	Reject if `stat` [9] is less than the significance level of the test. Alternatively,
	H_0:E(`x1`) = E(`x2`)	H_1:E(`x1`) ≠ E(`x2`)	reject the null hypothesis if `stat` [6] is too large or too small.
2	H_0:Pr(`x1` < `x2`) ≤ 0.5	H_1:Pr(`x1` < `x2`) > 0.5	Reject if `stat` [6] is too small
	H_0:E(`x1`) ≥ E(`x2`)	H_1:E(`x1`) < E(`x2`)	
3	H_0:Pr(`x1` < `x2`) ≥ 0.5	H_1:Pr(`x1` < `x2`) < 0.5	Reject if `stat` [6] is too large
	H_o:E(`x1`) ≤ E(`x2`))	H_1:E(`x1`) > E(`x2`)	

Assumptions

1. Arguments `x1` and `x2` contain random samples from their respective populations.
2. All observations are mutually independent.
3. The measurement scale is at least ordinal (i.e., an ordering less than, greater than, or equal to exists among the observations).

4. If *f*(*x*) and *g*(*y*) are the distribution functions of *x* and *y*, then *g*(*y*) = *f*(*x* + *c*) for some constant *c*(i.e., the distribution of *y* is, at worst, a translation of the distribution of *x*).

Tables of critical values of the *W* statistic are given in the references for small samples.

Examples

Example 1

The following example is taken from Conover (1980, p. 224). It involves the mixing time of two mixing machines using a total of 10 batches of a certain kind of batter, five batches for each machine. The null hypothesis is not rejected at the 5-percent level of significance. The warning error is always printed when one or more ties are detected, unless printing for warning errors is turned off. See function `imsls_error_options` (Chapter 14, "Utilties").

```
#include <imsls.h>

void main()
{
    int    n1_observations = 5;
    int    n2_observations = 5;
    float  x1[5] = {7.3, 6.9, 7.2, 7.8, 7.2};
    float  x2[5] = {7.4, 6.8, 6.9, 6.7, 7.1};
    float  p_value;

    p_value = imsls_f_wilcoxon_rank_sum(n1_observations, x1,
                   n2_observations, x2, 0);
    printf("p-value = %11.4f\n", p_value);

}
```

Output

```
*** WARNING Error IMSLS_AT_LEAST_ONE_TIE from imsls_f_wilcoxon_rank_sum.
***         At least one tie is detected between the samples.

p-value =      0.1412
```

Example 2

The following example uses the same data as the previous example. Now, all the statistics are output in the array `stat`.

```
#include <imsls.h>

void main()
{
    int    n1_observations = 5;
    int    n2_observations = 5;
    float  x1[5] = {7.3, 6.9, 7.2, 7.8, 7.2};
    float  x2[5] = {7.4, 6.8, 6.9, 6.7, 7.1};
    float  *stat;
    char   *labels[10] = {"Wilcoxon W statistic .....................",
                     "2*E(W) - W ..............................",
                     "p-value .................................",
                     "Adjusted Wilcoxon statistic ..............",
                     "Adjusted 2*E(W) - W ......................",
                     "Adjusted p-value .........................",
                     "W statistics for averaged ranks...........",
                     "Standard error of W (averaged ranks) ......",
                     "Standard normal score of W (averaged ranks)",
                     "Two-sided p-value of W (averaged ranks ...."};
    imsls_f_wilcoxon_rank_sum(n1_observations, x1,
                   n2_observations, x2,
                   IMSLS_STAT, &stat,
                   0);
    imsls_f_write_matrix("statistics", 10, 1, stat,
                   IMSLS_ROW_LABELS, labels,
                   IMSLS_WRITE_FORMAT, "%7.3f",
                   0);
}
```

Output

```
*** WARNING Error IMSLS_AT_LEAST_ONE_TIE from imsls_f_wilcoxon_rank_sum.
***         At least one tie is detected between the samples.

                     statistics
Wilcoxon W statistic .....................   34.000
2*E(W) - W ...............................   21.000
p-value ..................................    0.110
Adjusted Wilcoxon statistic ..............   35.000
Adjusted 2*E(W) - W ......................   20.000
Adjusted p-value .........................    0.075
W statistics for averaged ranks...........   34.500
Standard error of W (averaged ranks) ......    4.758
Standard normal score of W (averaged ranks)    1.471
Two-sided p-value of W (averaged ranks ....    0.141
```

Warning Errors

`IMSLS_NOBSX_NOBSY_TOO_SMALL`	“n1_observations” = # and “n2_observations” = #. Both sample sizes, “n1_observations” and “n2_observations”, are less than 25. Significance levels should be obtained from tabled values.
`IMSLS_AT_LEAST_ONE_TIE`	At least one tie is detected between the samples.

Fatal Errors

`IMSLS_ALL_X_Y_MISSING`	Each element of “x1” and/or “x2” is a missing (NaN, Not a Number) value.

kruskal_wallis_test

Performs a Kruskal-Wallis test for identical population medians.

Synopsis

#include `<imsls.h>`

float `*imsls_f_kruskal_wallis_test` (*int* `n_groups`, *int* `ni[]`,
    float `y[]`, ..., 0)

The type *double* function is `imsls_d_kruskal_wallis_test`.

Required Arguments

int `n_groups` *(Input)*
    Number of groups.

int `ni[]` *(Input)*
    Array of length `n_groups` containing the number of responses for each of the `n_groups` groups.

float `y[]` (Input)
    Array of length `ni[0] + ... + ni[n_groups-1]` that contains the responses for each of the n_groups groups. `y` must be sorted by group, with the `ni[0]` observations in group 1 coming first, the `ni[1]` observations in group two coming second, and so on.

Return Value

Array of length 4 containing the Kruskal-Wallis statistics.

I	**`stat[I]`**
0	Kruskal-Wallis H statistic.
1	Asymptotic probability of a larger H under the null hypothesis of identical population medians.
2	H corrected for ties.
3	Asymptotic probability of a larger H (corrected for ties) under the null hypothesis of identical populations

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_kruskal_wallis_test` (*int* `n_groups`, *int* `ni`, *float* `y[]`,
    `IMSLS_FUZZ`, *float* fuzz,
    `IMSLS_RETURN_USER`, *float* `stat[]`,
    0)

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz` (Input)
Constant used to determine ties in `y`. If (after sorting) `|y[i] - y[i + 1]|` is less than or equal to `fuzz`, then a tie is counted. `fuzz` must be nonnegative.

`IMSLS_RETURN_USER`, *float* `stat[]` (Output)
User defined array for storage of Kruskal-Wallis statistics.

Description

The function `imsls_f_kruskal_wallis_test` generalizes the Wilcoxon two-sample test computed by routine `imsls_f_wilcoxon_rank_sum` (page 460) to more than two populations. It computes a test statistic for testing that the population distribution functions in each of K populations are identical. Under appropriate assumptions, this is a nonparametric analogue of the one-way analysis of variance. Since more than two samples are involved, the alternative is taken as the analogue of the usual analysis of variance alternative, namely that the populations are not identical.

The calculations proceed as follows: All observations are ranked regardless of the population to which they belong. Average ranks are used for tied observations (observations within `fuzz` of each other). Missing observations (observations equal to NaN, not a number) are not included in the ranking. Let R_i denote the sum of the ranks in the i-th population. The test statistic H is defined as:

$$H = \frac{1}{S^2} \sum_{i=1}^{K} \left(\frac{R_i^2}{n_i} - \frac{N(N+1)^2}{4} \right)$$

where N is the total of the sample sizes, n_i is the number of observations in the i-th sample, and S^2 is computed as the (bias corrected) sample variance of the R_i.

The null hypothesis is rejected when `stat[3]` (or `stat[1]`) is less than the significance level of the test. If the null hypothesis is rejected, then the procedures given in Conover (1980, page 231) may be used for multiple comparisons. The routine `imsls_f_kruskal_wallis_test` (page 465) computes asymptotic probabilities using the chi-squared distribution when the number of groups is 6 or greater, and a Beta approximation (see Wallace 1959) when the number of groups is 5 or less. Tables yielding exact probabilities in small samples may be obtained from Owen (1962).

Example

The following example is taken from Conover (1980, page 231). The data represents the yields per acre of four different methods for raising corn. Since $H = 25.5$, the four methods are clearly different. The warning error is always printed when the Beta approximation is used, unless printing for warning errors is turned off.

```
#include <imsls.h>
void main()
{
       int ngroup = 4, ni[] = {9, 10, 7, 8};
       float y[] = {83., 91., 94., 89., 89., 96., 91., 92., 90., 91., 90.,
                    81., 83., 84., 83., 88., 91., 89., 84., 101., 100., 91.,
                    93., 96., 95., 94., 78., 82., 81., 77., 79., 81., 80.,
                    81.};
       float fuzz = .001, stat[4];
       char *rlabel[] = {"H (no ties)    =",
                         "Prob (no ties) =",
                         "H (ties)       =",
                         "Prob (ties)    ="};
       imsls_f_kruskal_wallis_test(ngroup, ni, y,
                              IMSLS_FUZZ, fuzz,
                              IMSLS_RETURN_USER, stat,
                              0);
       imsls_f_write_matrix(" ", 4, 1, stat,
                       IMSLS_ROW_LABELS, rlabel,
                       0);
}
```

Output

```
*** WARNING  ERROR  from imsls_kruskal_wallis_test.  The chi-squared degrees
***   of freedom are less than 5, so the Beta approximation is used.

H (no ties)    =     25.46
Prob (no ties) =      0.00
H (ties)       =     25.63
Prob (ties)    =      0.00
```

friedmans_test

Performs Friedman's test for a randomized complete block design.

Synopsis

#include `<imsls.h>`

float `imsls_f_friedmans_test` (*int* `n_blocks`, *int* `n_treatments`,
  float `y[]`, ..., 0)

The type *double* function is `imsls_d_friedmans_test`.

Required Arguments

int `n_blocks` (Input)
  Number of blocks.

int `n_treatments` (Input)
: Number of treatments.

float `y[]` *(Input)*
: Array of size `n_blocks * n_treatments` containing the observations. The first `n_treatments` positions of `y[]` contain the observations on treatments 1, 2, …, `n_treatments` in the first block. The second `n_treatments` positions contain the observations in the second block, etc., and so on.

Return Value

The Chi-squared approximation of the asymptotic p-value for Friedman's two-sided test statistic.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_friedmans_test` (*int* `n_blocks`, *int* `n_treatments`,
  float `y[]`,
  `IMSLS_FUZZ`, *float* `fuzz`,
  `IMSLS_ALPHA`, *float* `alpha`,
  `IMSLS_STAT`, *float* `**stat`,
  `IMSLS_STAT_USER`, *float* `stat[]`,
  `IMSLS_SUM_RANK`, *int* `**sum_ranks`,
  `IMSLS_SUM_RANK_USER`, *int* `sum_rank[]`
  `IMSLS_DIFFERENCE`, *float* `*difference`,
  0)

Optional Arguments

`IMSLS_FUZZ`, *float* `fuzz` (Input)
: Constant used to determine ties. In the ordered observations, if `|y[i] -y[i + 1]|` is less than or equal to `fuzz`, then `y[i] and y[i + 1]` are said to be tied. Default value is 0.0.

`IMSLS_ALPHA`, *float* `alpha` (Input)
: Critical level for multiple comparisons. `alpha` should be between 0 and 1 exclusive. Default value is 0.05.

`IMSLS_STAT`, *float* `**stat` (Output)
: Address of a pointer to an array of length 6 containing the Friedman statistics. Probabilities reported are computed under the appropriate null hypothesis.

I	**stat(I)**
0	Friedman two-sided test statistic.
1	Approximate *F* value for `stat[0]`.

2	Page test statistic for testing the ordered alternative that the median of treatment *i* is less than or equal to the median of treatment *i* + 1, with strict inequality holding for some *i*.
3	Asymptotic *p*-value for `stat[0]`. Chi-squared approximation.
4.	Asymptotic *p*-value for `stat[1]`. F approximation.
5.	Asymptotic *p*-value for `stat[2]`. Normal approximation.

`IMSLS_STAT_USER`, *float* `stat[]` (Output)
Storage for array `stat` is provided by the user. See `IMSLS_STAT`.

`IMSLS_SUM_RANK`, *float* `**sum_rank`, (Output)
Address of a pointer to an array of length `n_treatments` containing the sum of the ranks of each treatment.

`IMSLS_SUM_RANK_USER`, *float* `sum_rank[]`, (Output)
Storage for array `sum_rank` is provided by the user.
See `IMSLS_SUM_RANK`.

`IMSLS_DIFFERENCE`, *float* `*difference`, (Output
Minimum absolute difference in two elements of `sum_rank` to infer at the alpha level of significance that the medians of the corresponding treatments are different.

Description

Function `imsls_f_friedmans_test` may be used to test the hypothesis of equality of treatment effects within each block in a randomized block design. No missing values are allowed. Ties are handled by using the average ranks. The test statistic is the nonparametric analogue of an analysis of variance *F* test statistic.

The test proceeds by first ranking the observations within each block. Let *A* denote the sum of the squared ranks, i.e., let

$$A = \sum_{i=1}^{k} \sum_{j=1}^{b} \text{Rank}\left(Y_{ij}\right)^2$$

where Rank(Y_{ij}) is the rank of the *i*-th observation within the *j*-th block, b = `NB` is the number of blocks, and k = `NT` is the number of treatments. Let

$$B = \frac{1}{b} \sum_{i=1}^{k} R_i^2$$

where

$$R_i = \sum_{j=1}^{b} \text{Rank}\left(Y_{ij}\right)$$

The Friedman test statistic (`stat[0]`) is given by:

$$T = \frac{(k-1)\left(bB - b^2k(k+1)^2/4\right)}{A - bk(k+1)^2/4}$$

that, under the null hypothesis, has an approximate chi-squared distribution with $k - 1$ degrees of freedom. The asymptotic probability of obtaining a larger chi-squared random variable is returned in `stat[3]`.

If the F distribution is used in place of the chi-squared distribution, then the usual oneway analysis of variance F-statistic computed on the ranks is used. This statistic, reported in `stat[1]`, is given by

$$F = \frac{(b-1)T}{b(k-1)-T}$$

and asymptotically follows an F distribution with $(k - 1)$ and $(b - 1)(k - 1)$ degrees of freedom under the null hypothesis. `stat[4]` is the asymptotic probability of obtaining a larger F random variable. (If $A = B$, `stat[0]` and `stat[1]` are set to machine infinity, and the significance levels are reported as $k!/(k!)^b$, unless this computation would cause underflow, in which case the significance levels are reported as zero.) Iman and Davenport (1980) discuss the relative advantages of the chi-squared and F approximations. In general, the F approximation is considered best.

The Friedman T statistic is related both to the Kendall coefficient of concordance and to the Spearman rank correlation coefficient. See Conover (1980) for a discussion of the relationships.

If, at the α = `alpha` level of significance, the Friedman test results in rejection of the null hypothesis, then an asymptotic test that treatments i and j are different is given by: reject H_0 if $|R_i - R_j| >$ `D`, where

$$\mathrm{D} = t_{1-\alpha/2}\sqrt{2b(A-B)/\left((b-1)(k-1)\right)}$$

where t has $(b - 1)(k - 1)$ degrees of freedom. Page's statistic (`stat[2]`) is used to test the same null hypothesis as the Friedman test but is sensitive to a monotonic increasing alternative. The Page test statistic is given by

$$Q = \sum_{i=1}^{k} jR_i$$

It is largest (and thus most likely to reject) when the R_i are monotonically increasing.

Assumptions

The assumptions in the Friedman test are as follows:

1. The *k*-vectors of responses within each of the *b* blocks are mutually independent (i.e., the results within one block have no effect on the results within another block).
2. Within each block, the observations may be ranked.

The hypothesis tested is that each ranking of the random variables within each block is equally likely. The alternative is that at least one of the treatments tends to have larger values than one or more of the other treatments. The Friedman test is a test for the equality of treatment means or medians.

Example

The following example is taken from Bradley (1968), page 127, and tests the hypothesis that 4 drugs have the same effects upon a person's visual acuity. Five subjects were used.

```
#include <imsls.h>
void main()
{
   int n_blocks = 5, n_treatments = 4;
   float y[20] = {.39,.55,.33,.41,.21,.28,.19,.16,.73,.69,.64,
                  .62,.41,.57,.28,.35,.65,.57,.53,.60};
   float fuzz = .001,
   alpha = .05;
   float pvalue, *sum_rank, stat[6], difference;
   pvalue = imsls_f_friedmans_test(n_blocks,
                              n_treatments, y,
                              IMSLS_SUM_RANK, &sum_rank,
                              IMSLS_STAT_USER, stat,
                              IMSLS_DIFFERENCE, &difference,
                              0);
   printf("\np value for Friedman's T = %f\n\n", pvalue);
   printf("Friedman's T = ...........  %4.2f\n", stat[0]);
   printf("Friedman's F = ...........  %4.2f\n", stat[1]);
   printf("Page Test = ..............%5.2f\n", stat[2]);
   printf("Prob Friedman's T = .......  %7.5f\n", stat[3]);
   printf("Prob Friedman's F = .......  %7.5f\n", stat[4]);
   printf("Prob Page Test = ..........  %7.5f\n", stat[5]);
   printf("Sum of Ranks = ...........  %4.2f %4.2f %4.2 %4.2f\n"
           sum_rank[0], sum_rank[1], sum_rank[2], sum_rank[3]);
   printf("difference = .............  %7.5f\n", difference);
}
```

Output

```
P value for Friedman's T = 0.040566

Friedman T.........    8.28
Friedman F.........    4.93
Page test.......... 111.00
Prob Friedman T....    0.04057
Prob Friedman F....    0.01859
Prob Page test.....    0.98495
Sum of Ranks.......   16.00   17.00    7.00   10.00
D..................    6.65638
```

The Friedman null hypothesis is rejected at the α = .05 while the Page null hypothesis is not. (A Page test with a monotonic decreasing alternative would be rejected, however.) Using `sum_rank` and `difference`, one can conclude that treatment 3 is different from treatments 1 and 2, and that treatment 4 is different from treatment 2, all at the α = .05 level of significance.

cochran_q_test

Performs a Cochran *Q* test for related observations.

Synopsis

#include `<imsls.h>`

float `imsls_f_`cochran_q_test (*int* `n_observations`, *int* `n_variables`,
    float `*x`, ..., 0)

The type *double* function is `imsls_d_cochran_q_test`.

Required Arguments

int `n_observations` (Input)
    Number of blocks for each treatment.

int `n_variables` (Input)
    Number of treatments.

float `*x` (Input)
    Array of size `n_observations` × `n_variables` containing the matrix of dichotomized data. There are `n_observations` readings of zero or one on each of the `n_variables` treatments.

Return Value

The *p*-value, `p_value`, for the Cochran *Q* statistic.

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_cochran_q_test (int n_observations,
        int n_variables, float *x,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_Q_STATISTIC, float *q,
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Number of columns in `x`.
Default: `x_col_dim` = `n_variables`

`IMSLS_Q_STATISTIC`, *float* `*q` (Output)
Cochran's Q statistic.

Description

Function `imsls_f_cochran_q_test` computes the Cochran Q test statistic that may be used to determine whether or not M matched sets of responses differ significantly among themselves. The data may be thought of as arising out of a randomized block design in which the outcome variable must be success or failure, coded as 1.0 and 0.0, respectively. Within each block, a multivariate vector of 1's of 0's is observed. The hypothesis is that the probability of success within a block does not depend upon the treatment.

Assumptions

1. The blocks are a random sample from the population of all possible blocks.
2. The outcome of each treatment is dichotomous.

Hypothesis

The hypothesis being tested may be stated in at least two ways.

1. H_0 : All treatments have the same effect.
   H_1 : The treatments do not all have the same effect.
2. Let p_{ij} denote the probability of outcome 1.0 in block i, treatment j.
   H_0:$p_{i1} = p_{i2} = \ldots = p_{ic}$ for each i.
   H_1:$p_{ij} \neq p_{ik}$ for some i, and some $j \neq k$.
   where c (equal to `n_variables`) is the number of treatments.

The null hypothesis is rejected if Cochrans's Q statistic is too large.

Remarks

1. The input data must consist of zeros and ones only. For example, the data may be pass-fail information on `n_variables` questions asked of `n_observations` people or the test responses of `n_observations` individuals to `n_variables` different conditions.

2. The resulting statistic is distributed approximately as chi-squared with `n_variables` − 1 degrees of freedom if `n_observations` is not too small. `n_observations` greater than or equal to 5 × `n_variables` is a conservative recommendation.

Example

The following example is taken from Siegal (1956, p. 164). It measures the responses of 18 women to 3 types of interviews.

```
#include <imsls.h>
main()
{
    float pq;
    float x[54] = {
        0.0, 0.0, 0.0,
        1.0, 1.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 0.0,
        1.0, 0.0, 0.0,
        1.0, 1.0, 0.0,
        1.0, 1.0, 0.0,
        0.0, 1.0, 0.0,
        1.0, 0.0, 0.0,
        0.0, 0.0, 0.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, 0.0,
        1.0, 1.0, 0.0,
        1.0, 1.0, 0.0,
        1.0, 1.0, 1.0,
        1.0, 1.0, 0.0,
        1.0, 1.0, 0.0};

    pq = imsls_f_cochran_q_test(18, 3, x, 0);
    printf("pq = %9.5f\n", pq);
    return;
}
```

Output

```
pq =   0.00024
```

Warning Errors

`IMSLS_ALL_0_OR_1`	"x" consists of either all ones or all zeros. "q" is set to NaN (not a number). "pq" is set to 1.0.

Fatal Errors

`IMSLS_INVALID_X_VALUES`	"x[#][#]" = #. "x" must consist of zeros and ones only.

k_trends_test

Performs a k-sample trends test against ordered alternatives.

Synopsis

#include `<imsls.h>`

float `*imsls_f_ k_trends_test` (*int* `n_groups`, *int* `ni[]`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_ k_trends_test`.

Required Arguments

int `n_groups` *(Input)*
Number of groups. Must be greater than or equal to 3.

int `ni[]` *(Input)*
Array of length `n_groups` containing the number of responses for each of the `n_groups` groups.

float `y[]` (Input)
Array of length `ni[0] + ... + ni[n_groups-1]` that contains the responses for each of the n_groups groups. `y` must be sorted by group, with the `ni[0]` observations in group 1 coming first, the `ni[1]` observations in group two coming second, and so on.

Return Value

Array of length 17 containing the test results.

I	**stat[I]**
0	Test statistic (ties are randomized).
1	Conservative test statistic with ties counted in favor of the null hypothesis.
2	*p*-value associated with `stat[0]`.

3	p-value associated with `stat[1]`.
4	Continuity corrected `stat[2]`.
5	Continuity corrected `stat [3]`.
6	Expected mean of the statistic.
7	Expected kurtosis of the statistic. (The expected skewness is zero.)
8	Total sample size.
9	Coefficient of rank correlation based upon `stat[0]`.
10	Coefficient of rank correlation based upon `stat[1]`.
11	Total number of ties between samples.
12	The t-statistic associated with `stat [2]`.
13	The t-statistic associated with `stat[3]`.
14	The t-statistic associated with `stat [4]`.
15	The t-statistic associated with `stat[5]`.
16	Degrees of freedom for each t-statistic.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_k_trends_test` (*int* `n_groups`, *int* `ni`, *float* `y[]`,
    `IMSLS_RETURN_USER`, *float* `stat[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `stat[]` (Output)
    User defined array for storage of test results.

Description

Function `imsls_f_k_trends_test` performs a k-sample trends test against ordered alternatives. The alternative to the null hypothesis of equality is that $F_1(\mathrm{X}) < F_2(\mathrm{X}) < \ldots F_k(\mathrm{X})$, where F_1, F_2, etc., are cumulative distribution functions, and the operator < implies that the less than relationship holds for all values of `X`. While the trends test used in `k_trends_test` requires that the background populations be continuous, ties occurring within a sample have no effect on the test statistic or associated probabilities. Ties between samples are important, however. Two methods for handling ties between samples are used. These are:

1. Ties are randomly split (`stat[0]`).

2. Ties are counted in a manner that is unfavorable to the alternative hypothesis (`stat[1]`).

Computational Procedure

Consider the matrices

$$M^{km} = \left(m_{ij}^{km}\right) = \begin{pmatrix} 2 \text{ if } X_{ki} < X_{mj} \\ 0 \text{ otherwise} \end{pmatrix}$$

where X_{ki} is the i-th observation in the k-th population, X_{mj} is the j-th observation in the m-th population, and each matrix M^{km} is n_k by n_m where n_i = `ni`(i). Let S_{km} denote the sum of all elements in M^{km}. Then, `stat[1]` is computed as the sum over all elements in S_{km}, minus the expected value of this sum (computed as

$$\sum_{k<m} n_k n_m$$

when there are no ties and the distributions in all populations are equal). In `stat[0]`, ties are broken randomly, and the element in the summation is taken as 2.0 or 0.0 depending upon the result of breaking the tie.

`stat[2]` and `stat[3]` are computed using the t distribution. The probabilities reported are asymptotic approximations based upon the t statistics in `stat[12]` and `stat[13]`, which are computed as in Jonckheere (1954, page 141). Similarly, `stat[4]` and `stat[5]` give the probabilities for `stat[14]` and `stat[15]`, the continuity corrected versions of `stat[2]` and `stat[3]`. The degrees of freedom for each t statistic (`stat[16]`) are computed so as to make the t distribution selected as close as possible to the actual distribution of the statistic (see Jonckheere 1954, page 141).

`stat[6]`, the variance of the test statistic `stat[0]`, and `stat[7]`, the kurtosis of the test statistic, are computed as in Jonckheere (1954, page 138). The coefficients of rank correlation in `stat[8]` and `stat[9]` reduce to the Kendall τ statistic when there are just two groups.

Exact probabilities in small samples can be obtained from tables in Jonckheere (1954). Note, however, that the t approximation appears to be a good one.

Assumptions

1. The X_{mi} for each sample are independently and identically distributed according to a single continuous distribution.
2. The samples are independent.

Hypothesis tests

$H_0 : F_1(\mathrm{x}) \geq F_2(\mathrm{x}) \geq \ldots \geq F_k(\mathrm{x})$
$H_1 : F_1(\mathrm{x}) < F_2(\mathrm{x}) < \ldots < F_k(\mathrm{x})$

Reject if `stat[2]` (or `stat[3]`, or `stat[4]` or `stat[5]`, depending upon the method used) is too large.

Example

The following example is taken from Jonckheere (1954, page 135). It involves four observations in four independent samples.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
 float *stat;
 int n_groups = 4;
 int ni[] = {4, 4, 4, 4};
char *fmt = "%9.5f";
char *rlabel[] = {
"stat[0] - Test Statistic  (random) ............",
"stat[1] - Test Statistic  (null hypothesis) ...",
 "stat[2] - p-value for stat[0] ................",
"stat[3] - p-value for stat[1] ................",
"stat[4] - Continuity corrected for stat[2] ....",
"stat[5] - Continuity corrected for stat[3] ....",
"stat[6] - Expected mean ......................",
"stat[7] - Expected kurtosis ..................",
"stat[8] - Total sample size ..................",
"stat[9] - Rank corr. coef. based on stat[0] ...",
"stat[10]- Rank corr. coef. based on stat[1] ...",
"stat[11]- Total number of ties ...............",
"stat[12]- t-statistic associated w/stat[2] ....",
"stat[13]- t-statistic asscoiated w/stat[3] ....",
"stat[14]- t-statistic associated w/stat[4] ....",
"stat[15]- t-statistic asscoiated w/stat[5] ....",
"stat[16]- Degrees of freedom ................."};

 float y[] = {19., 20., 60., 130., 21., 61., 80., 129.,
           40., 99., 100., 149., 49., 110., 151., 160.};

 stat = imsls_f_k_trends_test(n_groups, ni, y, 0);

 imsls_f_write_matrix("stat", 17, 1, stat,
                  IMSLS_WRITE_FORMAT, fmt,
                        IMSLS_ROW_LABELS, rlabel,
                        0);
}
```

Output

```
stat(0) - Test statistic (random) ..........    46.00000
stat(1) - Test statistic (null hypothesis) ..    46.00000
stat(2) - p-value for stat(0) ..............     0.01483
stat(3) - p-value for stat(1) ..............     0.01483
stat(4) - Continuity corrected stat(2) ......     0.01683
stat(5) - Continuity corrected stat(3) ......     0.01683
stat(6) - Expected mean ....................   458.66666
stat(7) - Expected kurtosis ................    -0.15365
stat(8) - Total sample size ................    16.00000
stat(9)- Rank corr. coef. based on stat(0) .     0.47917
stat(10)- Rank corr. coef. based on stat(1) .     0.47917
stat(11)- Total number of ties .............     0.00000
stat(12)- t-statistic associated w/stat(2) ..     2.26435
stat(13)- t-statistic associated w/stat(3) ..     2.26435
stat(14)- t-statistic associated w/stat(4) ..     2.20838
stat(15)- t-statistic associated w/stat(5) ..     2.20838
stat(16)- Degrees of freedom ...............    36.04963
```

Chapter 7: Tests of Goodness of Fit

Routines

Usage Notes

The routines in this chapter are used to test for goodness of fit and randomness. The goodness-of-fit tests are described in Conover (1980). There are two goodness-of-fit tests for general distributions, a Kolmogorov-Smirnov test and a chi-squared test. The user supplies the hypothesized cumulative distribution function for these two tests. There are three routines that can be used to test specifically for the normal or exponential distributions.

The tests for randomness are often used to evaluate the adequacy of pseudorandom number generators. These tests are discussed in Knuth (1981).

The Kolmogorov-Smirnov routines in this chapter compute exact probabilities in small to moderate sample sizes. The chi-squared goodness-of-fit test may be used with discrete as well as continuous distributions.

The Kolmogorov-Smirnov and chi-squared goodness-of-fit test routines allow for missing values (NaN, not a number) in the input data. The routines that test for randomness do not allow for missing values.

chi_squared_test

Performs a chi-squared goodness-of-fit test.

Synopsis

#include `<imsls.h>`

float `imsls_f_chi_squared_test` (*float* `user_proc_cdf()`,
    int `n_observations`, *int* `n_categories`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_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)
    Number of data elements input in `x`.

int `n_categories` (Input)
    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 `<imsls.h>`

float `imsls_f_chi_squared_test` (*float* `user_proc_cdf()`,
    int `n_observations`, *int* `n_categories`, *float* `x[]`,
    `IMSLS_N_PARAMETERS_ESTIMATED`, *int* `n_parameters`,
    `IMSLS_CUTPOINTS`, *float* `**cutpoints`,
    `IMSLS_CUTPOINTS_USER`, *float* `cutpoints[]`,
    `IMSLS_CUTPOINTS_EQUAL`,
    `IMSLS_CHI_SQUARED`, *float* `*chi_squared`,
    `IMSLS_DEGREES_OF_FREEDOM`, *float* `*df`,
    `IMSLS_FREQUENCIES`, *float* `frequencies[]`,
    `IMSLS_BOUNDS`, *float* `lower_bound`, *float* `upper_bound`,
    `IMSLS_CELL_COUNTS`, *float* `**cell_counts`,
    `IMSLS_CELL_COUNTS_USER`, *float* `cell_counts[]`,
    `IMSLS_CELL_EXPECTED`, *float* `**cell_expected`,
    `IMSLS_CELL_EXPECTED_USER`, *float* `cell_expected[]`,

`IMSLS_CELL_CHI_SQUARED`, *float* `**cell_chi_squared`,
`IMSLS_CELL_CHI_SQUARED_USER`, *float* `cell_chi_squared[]`,
`IMSLS_FCN_W_DATA`, *float* `fcn()`, *void* `*data`,
0)

Optional Arguments

`IMSLS_N_PARAMETERS_ESTIMATED`, *int* `n_parameters` (Input)
Number of parameters estimated in computing the cumulative distribution function.

`IMSLS_CUTPOINTS`, *float* `**cutpoints` (Output)
Address of a pointer to an internally allocated array of length `n_categories` – 1 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 `IMSLS_CUTPOINTS_EQUAL` is specified, equal probability cutpoints are computed and returned in `cutpoints`.

`IMSLS_CUTPOINTS_USER`, *float* `cutpoints []` (Input/Output)
Storage for array `cutpoints` is provided by the user. See `IMSLS_CUTPOINTS`.

`IMSLS_CUTPOINTS_EQUAL`
If `IMSLS_CUTPOINTS_USER` is specified, then equal probability cutpoints can still be used if, in addition, the `IMSLS_CUTPOINTS_EQUAL` option is specified. If `IMSLS_CUTPOINTS_USER` is not specified, equal probability cutpoints are used by default.

`IMSLS_CHI_SQUARED`, *float* `*chi_squared` (Output)
If specified, the chi-squared test statistic is returned in `*chi_squared`.

`IMSLS_DEGREES_OF_FREEDOM`, *float* `*df` (Output)
If specified, the degrees of freedom for the chi-squared goodness-of-fit test is returned in `*df`.

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array with `n_observations` components containing the vector frequencies for the observations stored in `x`.

`IMSLS_BOUNDS`, *float* `lower_bound`, *float* `upper_bound` (Input)
If `IMSLS_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 `IMSLS_BOUNDS` is used. By convention, `lower_bound` is excluded from the first interval, but `upper_bound` is included in the last interval.

IMSLS_CELL_COUNTS, *float* **cell_counts (Output)
Address of a pointer to an internally allocated array of length n_categories containing the cell counts. The cell counts are the observed frequencies in each of the n_categories cells.

IMSLS_CELL_COUNTS_USER, *float* cell_counts[] (Output)
Storage for array cell_counts is provided by the user. See IMSLS_CELL_COUNTS.

IMSLS_CELL_EXPECTED, *float* **cell_expected (Output)
Address of a pointer to an internally allocated array of length n_categories containing the cell expected values. The expected value of a cell is the expected count in the cell given that the hypothesized distribution is correct.

IMSLS_CELL_EXPECTED_USER, *float* cell_expected[] (Output)
Storage for array cell_expected is provided by the user. See IMSLS_CELL_EXPECTED.

IMSLS_CELL_CHI_SQUARED, *float* **cell_chi_squared (Output)
Address of a pointer to an internally allocated array of length n_categories containing the cell contributions to chi-squared.

IMSLS_CELL_CHI_SQUARED_USER, *float* cell_chi_squared[] (Output)
Storage for array cell_chi_squared is provided by the user. See IMSLS_CELL_CHI_SQUARED.

IMSLS_FCN_W_DATA, *float* user_proc_cdf (*float* y), *void* *data, (Input)
User-supplied function that returns the hypothesized, cumulative distribution function, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Function imsls_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 can be continuous, discrete, or a mixture of discrete and continuous distributions, is specified by the user-defined function user_proc_cdf. Because the user is allowed to give a range for the observations, a test that is conditional on 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 imsls_f_chi_squared_test, but intervals that are not equiprobable can be specified through the use of optional argument IMSLS_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 `imsls_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 `IMSLS_BOUNDS` is specified, the endpoints are user-defined by 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 user-specified endpoints.
- If the cutpoints are determined by `imsls_f_chi_squared_test`, then the cumulative probability at x_i, $F(x_i)$, is computed by the function `user_proc_cdf`.

The tally for x_i is made in interval number $\lfloor mF(x_i) + 1 \rfloor$, where m = `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 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.

Examples

Example 1

This example illustrates the use of `imsls_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 `imsls_f_normal_cdf` (Chapter 11, Probability Distribution Functions and Inverses) as the hypothesized distribution function. In this example, the null hypothesis is not rejected.

```
#include <imsls.h>

#define SEED                    123457
#define N_CATEGORIES                10
#define N_OBSERVATIONS            1000

main()
{
    float       *x, p_value;
    imsls_random_seed_set(SEED);
                                /* Generate Normal deviates */
    x = imsls_f_random_normal (N_OBSERVATIONS, 0);
                                /* Perform chi squared test */
    p_value = imsls_f_chi_squared_test (imsls_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, optional arguments are used for the data in the initial example.

```
#include <imsls.h>

#define SEED                    123457
#define N_CATEGORIES                10
#define N_OBSERVATIONS            1000

main()
{
    float       *cell_counts, *cutpoints, *cell_chi_squared;
    float       chi_squared_statistics[3], *x;
    char        *stat_row_labels[] = {"chi-squared",
                                  "degrees of freedom","p-value"};
    imsls_random_seed_set(SEED);
                                /* Generate normal deviates */
    x = imsls_f_random_normal (N_OBSERVATIONS, 0);
                                /* Perform chi squared test */
    chi_squared_statistics[2] =
        imsls_f_chi_squared_test (imsls_f_normal_cdf,
                                 N_OBSERVATIONS,  N_CATEGORIES, x,
                  IMSLS_CUTPOINTS,         &cutpoints,
                  IMSLS_CELL_COUNTS,        &cell_counts,
                  IMSLS_CELL_CHI_SQUARED,   &cell_chi_squared,
                  IMSLS_CHI_SQUARED,        &chi_squared_statistics[0],
                  IMSLS_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
                  0);
                                /* Print results */
    imsls_f_write_matrix ("\nChi Squared Statistics\n", 3, 1,
        chi_squared_statistics,
        IMSLS_ROW_LABELS, stat_row_labels,
        0);
    imsls_f_write_matrix ("Cut Points", 1, N_CATEGORIES-1,
        cutpoints, 0);
    imsls_f_write_matrix ("Cell Counts", 1, N_CATEGORIES,
        cell_counts, 0);
    imsls_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 1,000 with parameter θ = 5.0 is generated by function `imsls_f_random_poisson` (Chapter 12, Random Number Generation”). In the call to `imsls_f_chi_squared_test`, function `imsls_f_poisson_cdf` (Chapter 11, “Probability Distribution Functions and Inverses) is used as function `user_proc_cdf`.

```
#include <imsls.h>

#define SEED                     123457
#define N_CATEGORIES             10
#define N_PARAMETERS_ESTIMATED   0
#define N_NUMBERS                1000
#define THETA                    5.0

float           user_proc_cdf(float);

main()
{
    int         i, *poisson;
    float       cell_statistics[3][N_CATEGORIES];
    float       chi_squared_statistics[3], x[N_NUMBERS];
    float       cutpoints[]       = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5,
                                     7.5, 8.5, 9.5};
    char        *cell_row_labels[] = {"count", "expected count",
                                     "cell chi-squared"};
    char        *cell_col_labels[] = {"Poisson value", "0", "1", "2",
                                  "3", "4", "5", "6", "7",
                                  "8", "9"};
    char        *stat_row_labels[] = {"chi-squared",
                                  "degrees of freedom","p-value"};

    imsls_random_seed_set(SEED);
                            /* Generate the data */
    poisson = imsls_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] =
        imsls_f_chi_squared_test(user_proc_cdf, N_NUMBERS,
            N_CATEGORIES, x,
                IMSLS_CUTPOINTS_USER,        cutpoints,
                IMSLS_CELL_COUNTS_USER,      &cell_statistics[0][0],
                IMSLS_CELL_EXPECTED_USER,    &cell_statistics[1][0],
                IMSLS_CELL_CHI_SQUARED_USER, &cell_statistics[2][0],
                IMSLS_CHI_SQUARED,           &chi_squared_statistics[0],
                IMSLS_DEGREES_OF_FREEDOM,    &chi_squared_statistics[1],
                0);
                            /* Print results */
    imsls_f_write_matrix("\nChi-squared Statistics\n", 3, 1,
                                        &chi_squared_statistics[0],
                        IMSLS_ROW_LABELS,     stat_row_labels,
                        0);
    imsls_f_write_matrix("\nCell Statistics\n", 3, N_CATEGORIES,
                                        &cell_statistics[0][0],
                        IMSLS_ROW_LABELS,     cell_row_labels,
                        IMSLS_COL_LABELS,     cell_col_labels,
                        IMSLS_WRITE_FORMAT,   "%9.1f",
                        0);
}


float user_proc_cdf(float k)
{
    float           cdf_v;

    cdf_v = imsls_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
```

Programming Notes

Function `user_proc_cdf` must be supplied with calling sequence `user_proc_cdf(y)`, which returns the value of the cumulative distribution function at any point `y` in the (optionally) specified range. Many of the cumulative distribution functions in Chapter 11, "Probability Distribution Functions and Inverses," 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.

Warning Errors

`IMSLS_EXPECTED_VAL_LESS_THAN_1`	An expected value is less than 1.
`IMSLS_EXPECTED_VAL_LESS_THAN_5`	An expected value is less than 5.

Fatal Errors

`IMSLS_ALL_OBSERVATIONS_MISSING`	All observations contain missing values.
`IMSLS_INCORRECT_CDF_1`	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 1.
`IMSLS_INCORRECT_CDF_2`	Function `user_proc_cdf` is not a cumulative distribution function. The probability of the range of the distribution is not positive.
`IMSLS_INCORRECT_CDF_3`	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.
`IMSLS_INCORRECT_CDF_4`	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.
`IMSLS_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.

normality_test

Performs a test for normality.

Synopsis

#include `<imsls.h>`

float `imsls_f_normality_test` (*int* `n_observations`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_normality_test`.

Required Arguments

int `n_observations` (Input)
Number of observations. Argument `n_observations` must be in the range from 3 to 2,000, inclusive, for the Shapiro-Wilk *W* test and must be greater than 4 for the Lilliefors test.

float `x[]` (Input)
Array of size `n_observations` containing the observations.

Return Value

The *p*-value for the Shapiro-Wilk *W* test or the Lilliefors test for normality. The Shapiro-Wilk test is the default. If the Lilliefors test is used, probabilities less than 0.01 are reported as 0.01, and probabilities greater than 0.10 for the normal distribution are reported as 0.5. Otherwise, an approximate probability is computed.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float imsls_f_normality_test (int n_observations, float x[],
        IMSLS_SHAPIRO_WILK_W, float *shapiro_wilk_w,
        IMSLS_LILLIEFORS, float *max_difference,
        IMSLS_CHI_SQUARED, int n_categories, float *df,
                float *chi_squared,
        0)
```

Optional Arguments

`IMSLS_SHAPIRO_WILK_W`, *float* `*shapiro_wilk_w` (Output)
Indicates the Shapiro-Wilk *W* test is to be performed. The Shapiro-Wilk *W* statistic is returned in `shapiro_wilk_w`. Argument `IMSLS_SHAPIRO_WILK_W` is the default test.

`IMSLS_LILLIEFORS`, *float* `*max_difference` (Output)
Indicates the Lilliefors test is to be performed. The maximum absolute difference between the empirical and the theoretical distributions is returned in `max_difference`.

`IMSLS_CHI_SQUARED`, *int* `n_categories` (Input),
float `*df`, *float* `*chi_squared` (Output)
Indicates the chi-squared goodness-of-fit test is to be performed. Argument `n_categories` is the number of cells into which the observations are to be tallied. The degrees of freedom for the test are returned in argument df, and the chi-square statistic is returned in argument `chi_squared`.

Description

Three methods are provided for testing normality: the Shapiro-Wilk *W* test, the Lilliefors test, and the chi-squared test.

Shapiro-Wilk W Test

The Shapiro-Wilk *W* test is thought by D'Agostino and Stevens (1986, p. 406) to be one of the best omnibus tests of normality. The function is based on the approximations and code given by Royston (1982a, b, c). It can be used in samples as large as 2,000 or as small as 3. In the Shapiro and Wilk test, *W* is given by

$$W = \left(\sum a_i x_{(i)}\right)^2 / \left(\sum \left(x_i - \bar{x}\right)^2\right)$$

where $x_{(i)}$ is the *i*-th largest order statistic and x is the sample mean. Royston (1982) gives approximations and tabled values that can be used to compute the coefficients a_i, $i = 1, \ldots, n$, and obtains the significance level of the *W* statistic.

Lilliefors Test

This function computes Lilliefors test and its *p*-values for a normal distribution in which both the mean and variance are estimated. The one-sample, two-sided Kolmogorov-Smirnov statistic *D* is first computed. The *p*-values are then computed using an analytic approximation given by Dallal and Wilkinson (1986). Because Dallal and Wilkinson give approximations in the range (0.01, 0.10) if the computed probability of a greater *D* is less than 0.01, an `IMSLS_NOTE` is issued and the *p*-value is set to 0.50. Note that because parameters are estimated, *p*-values in Lilliefors test are not the same as in the Kolmogorov-Smirnov Test.

Observations should not be tied. If tied observations are found, an informational message is printed. A general reference for the Lilliefors test is Conover (1980). The original reference for the test for normality is Lilliefors (1967).

Chi-Squared Test

This function computes the chi-squared statistic, its p-value, and the degrees of freedom of the test. Argument `n_categories` finds the number of intervals into which the observations are to be divided. The intervals are equiprobable except for the first and last interval which are infinite in length.

If more flexibility is desired for the specification of intervals, the same test can be performed with a call to function `imsls_f_chi_squared_test` (page 482) using the optional arguments described for that function.

Examples

Example 1

The following example is taken from Conover (1980, pp. 195, 364). The data consists of 50 two-digit numbers taken from a telephone book. The *W* test fails to reject the null hypothesis of normality at the .05 level of significance.

```
#include <imsls.h>

void main()
{

  int    n_observations = 50;
  float  x[] = {23.0, 36.0, 54.0, 61.0, 73.0, 23.0,
                37.0, 54.0, 61.0, 73.0, 24.0, 40.0,
                56.0, 62.0, 74.0, 27.0, 42.0, 57.0,
                63.0, 75.0, 29.0, 43.0, 57.0, 64.0,
                77.0, 31.0, 43.0, 58.0, 65.0, 81.0,
                32.0, 44.0, 58.0, 66.0, 87.0, 33.0,
                45.0, 58.0, 68.0, 89.0, 33.0, 48.0,
                58.0, 68.0, 93.0, 35.0, 48.0, 59.0,
                70.0, 97.0};
  float  p_value;
```

```
                                   /* Shapiro-Wilk test */
  p_value = imsls_f_normality_test (n_observations, x,
                                    0);
  printf ("p-value = %11.4f.\n", p_value);

}
```

Output

```
p-value =      0.2309
```

Example 2

The following example uses the same data as the previous example. Here, the Shapiro-Wilk *W* statistic is output.

```
#include <imsls.h>

void main()
{

  int    n_observations = 50;
  float  x[] = {23.0, 36.0, 54.0, 61.0, 73.0, 23.0,
                37.0, 54.0, 61.0, 73.0, 24.0, 40.0,
                56.0, 62.0, 74.0, 27.0, 42.0, 57.0,
                63.0, 75.0, 29.0, 43.0, 57.0, 64.0,
                77.0, 31.0, 43.0, 58.0, 65.0, 81.0,
                32.0, 44.0, 58.0, 66.0, 87.0, 33.0,
                45.0, 58.0, 68.0, 89.0, 33.0, 48.0,
                58.0, 68.0, 93.0, 35.0, 48.0, 59.0,
                70.0, 97.0};
  float  p_value, shapiro_wilk_w;

                                   /* Shapiro-Wilk test */
  p_value = imsls_f_normality_test (n_observations, x,
                                    IMSLS_SHAPIRO_WILK_W,
                                    &shapiro_wilk_w,
                                    0);
  printf ("p-value = %11.4f.\n", p_value);
  printf ("Shapiro Wilk W statistic = %11.4f.\n",
          shapiro_wilk_w);

}
```

Output

```
p-value =      0.2309.
Shapiro Wilk W statistic =      0.9642
```

Warning Errors

`IMSLS_ALL_OBS_TIED`	All observations in “x” are tied.

Fatal Errors

`IMSLS_NEED_AT_LEAST_5`	All but # elements of “x” are missing. At least five nonmissing observations are necessary to continue.
`IMSLS_NEG_IN_EXPONENTIAL`	In testing the exponential distribution, an invalid element in “x” is found (“x[]” = #). Negative values are not possible in exponential distributions.
`IMSLS_NO_VARIATION_INPUT`	There is no variation in the input data. All nonmissing observations are tied.

kolmogorov_one

Performs a Kolmogorov-Smirnov one-sample test for continuous distributions.

Synopsis

#include `<imsls.h>`

float `*imsls_f_kolmogorov_one` (*float* `cdf()`, *int* `n_observations`,
    float `x[]`, ..., 0)

`The type double function is imsls_d_kolmogorov_one.`

Required Arguments

float `cdf` (*float* `x`) (Input)
User-supplied function to compute the cumulative distribution function (`CDF`) at a given value. The form is `CDF`(`x`), where
`x` is the value at which `cdf` is to be evaluated (Input)
and `cdf` is the value of `CDF` at `x`. (Output)

int `n_observations` (Input)
Number of observations.

float `x[]` (Input)
Array of size `n_observations` containing the observations.

Return Value

Pointer to an array of length 3 containing Z, p_1, and p_2 .

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_kolmogorov_one` (*float* `cdf()`, *int* `n_observations`,
    float `x[]`,
    `IMSLS_DIFFERENCES`, *int* `**differences`,

```
IMSLS_DIFFERENCES_USER, int differences[]
IMSLS_N_MISSING, int *n_missing,
IMSLS_RETURN_USER, , float test_statistic[],
IMSLS_FCN_W_DATA, float cdf (), void *data,
0)
```

Optional Arguments

`IMSLS_DIFFERENCES`, *int* `**differences` (Output)
Address of a pointer to the internally allocated array containing D_n, D_n^+, D_n^-.

`IMSLS_DIFFERENCES_USER`, *int* `differences[]`
Storage for the array `differences` is provided by the user. See `IMSLS_DIFFERENCES.`

`IMSLS_N_MISSING`, *int* `*n_missing` (Ouput)
Number of missing values is returned in `*n_missing.`

`IMSLS_RETURN_USER`, *float* `test_statistics[]` (Output)
If specified, the *Z*-score and the *p*-values for hypothesis test against both one-sided and two-sided alternatives is stored in array `test_statistics` provided by the user.

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

Description

The routine `imsls_f_kolmogorov_one` performs a Kolmogorov-Smirnov goodness-of-fit test in one sample. The hypotheses tested follow:

- $H_0 : F(x) = F^*(x) \quad H_1 : F(x) \neq F^*(x)$
- $H_0 : F(x) \geq F^*(x) \quad H_1 : F(x) < F^*(x)$
- $H_0 : F(x) \leq F^*(x) \quad H_1 : F(x) > F^*(x)$

where F is the cumulative distribution function (`CDF`) of the random variable, and the theoretical `cdf`, F^*, is specified via the user-supplied function `cdf`. Let n = `n_observations - n_missing`. The test statistics for both one-sided alternatives

$$D_n^+ = differences[1]$$

and

$$D_n^- = differences[2]$$

and the two-sided (D_n = `differences[0]`) alternative are computed as well as an asymptotic *z*-score (`test_statistics[0]`) and *p*-values associated with the one-sided (`test_statistics[1]`) and two-sided (`test_statistics[2]`) hypotheses. For $n > 80$, asymptotic *p*-values are used (see Gibbons 1971). For $n \leq 80$, exact one-sided *p*-values are computed according to a method given by Conover (1980, page 350). An approximate two-sided test *p*-value is obtained as twice the one-sided *p*-value. The approximation is very close for one-sided *p*-values less than 0.10 and becomes very bad as the one-sided *p*-values get larger.

Programming Notes

1. The theoretical `CDF` is assumed to be continuous. If the `CDF` is not continuous, the statistics

$$D_n^*$$

will not be computed correctly.

2. Estimation of parameters in the theoretical `CDF` from the sample data will tend to make the *p*-values associated with the test statistics too liberal. The empirical `CDF` will tend to be closer to the theoretical `CDF` than it should be.

3. No attempt is made to check that all points in the sample are in the support of the theoretical `CDF`. If all sample points are not in the support of the `CDF`, the null hypothesis must be rejected.

Example

In this example, a random sample of size 100 is generated via routine `imsls_f_random_uniform` (Chapter 12, Random Number Generation") for the uniform (0, 1) distribution. We want to test the null hypothesis that the `cdf` is the standard normal distribution with a mean of 0.5 and a variance equal to the uniform (0, 1) variance (1/12).

```
#include <imsls.h>
#include <stdio.h>
float cdf(float);
void main()
{
  float *statistics=NULL, *diffs = NULL, *x=NULL;
  int nobs = 100, nmiss;
  imsls_random_seed_set(123457);
  x = imsls_f_random_uniform(nobs, 0);
  statistics = imsls_f_kolmogorov_one(cdf, nobs, x,
                                    IMSLS_N_MISSING, &nmiss,
                                    IMSLS_DIFFERENCES, &diffs,
                                    0);
  printf("D      = %8.4f\n", diffs[0]);
  printf("D+     = %8.4f\n", diffs[1]);
  printf("D-     = %8.4f\n", diffs[2]);
  printf("Z      = %8.4f\n", statistics[0]);
  printf("Prob greater D one sided  = %8.4f\n", statistics[1]);
  printf("Prob greater D two sided  = %8.4f\n", statistics[2]);
  printf("N missing = %d\n", nmiss);
}
float cdf(float x)
{
  float mean = .5, std = .2886751, z;
  z = (x-mean)/std;
  return(imsls_f_normal_cdf(z));
}
```

Output

```
D     =   0.1471
D+    =   0.0810
D-    =   0.1471
Z     =   1.4708
Prob greater D one-sided =   0.0132
Prob greater D two-sided =   0.0264
N missing =    0
```

kolmogorov_two

Performs a Kolmogorov-Smirnov two-sample test.

Synopsis

#include `<imsls.h>`

float `*imsls_f_kolmogorov_two` (*int* `n_observations_x`, *float* `x[]`, *int* `n_observations_y`, *float* `y[]`, ..., 0)

The type *double* function is `imsls_d_kolmogorov_two`.

Required Arguments

int `n_observations_x` (Input)
Number of observations in sample one.

float `x[]` (Input)

Array of size `n_observations_x` containing the observations from sample one.

int `n_observations_y` (Input)

Number of observations in sample two.

float `y[]` (Input)

Array of size `n_observations_y` containing the observations from sample two.

Return Value

Pointer to an array of length 3 containing Z, p_1, and p_2 .

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_kolmogorov_two` (*int* `n_observations_x`, *float* `x[]`, *int* `n_observations_y`, *float* `y[]`, …
`IMSLS_DIFFERENCES`, *int* `**differences`,
`IMSLS_DIFFERENCES_USER`, *int* `differences[]`,
`IMSLS_N_MISSING_X`, *int* `*xmissing`,
`IMSLS_N_MISSING_Y`, *int* `*ymissing`,
`IMSLS_RETURN_USER`, *float* `test_statistic[]`,
0)

Optional Arguments

`IMSLS_DIFFERENCES`, *int* `**differences` (Output)

Address of a pointer to the internally allocated array containing D_n, D_n^+, D_n^-.

`IMSLS_DIFFERENCES_USER`, *int* `differences[]` (Output)

Storage for array `differences` is provided by the user. See `IMSLS_DIFFERENCES`.

`IMSLS_N_MISSING_X`, *int* `*xmissing` (Ouput)

Number of missing values in the `x` sample is returned in `*xmissing`.

`IMSLS_N_MISSING_Y`, *int* `*ymissing` (Ouput)

Number of missing values in the `y` sample is returned in `*ymissing`.

`IMSLS_RETURN_USER`, *float* `test_statistics[]` (Output)

If specified, the *Z*-score and the *p*-values for hypothesis test against both one-sided and two-sided alternatives is stored in array `test_statistics` provided by the user.

Description

Function `imsls_f_kolmogorov_two` computes Kolmogorov-Smirnov two-sample test statistics for testing that two continuous cumulative distribution functions (CDF's) are identical based upon two random samples. One- or two-sided alternatives are allowed. Exact *p*-values are computed for the two-sided test when `n_observations_x * n_observations_y` is less than 104.

Let $F_n(x)$ denote the empirical CDF in the *X* sample, let $G_m(y)$ denote the empirical CDF in the *Y* sample, where n = `n_observations_x` − `n_missing_x` and m = `n_observations_y` − `n_missing_y`, and let the corresponding population distribution functions be denoted by $F(x)$ and $G(y)$, respectively. Then, the hypotheses tested by `imsls_f_kolmogorov_two` are as follows:

$$\begin{aligned}
&\bullet H_0 : F(x) = G(x) \quad H_1 : F(x) \neq G(x) \\
&\bullet H_0 : F(x) \leq G(x) \quad H_1 : F(x) > G(x) \\
&\bullet H_0 : F(x) \geq G(x) \quad H_1 : F(x) < G(x)
\end{aligned}$$

The test statistics are given as follows:

$$\begin{aligned}
D_{mn} &= \max\left(D_{mn}^{+}, D_{mn}^{-}\right) && \text{(diffs[0])} \\
D_{mn}^{+} &= \max_x (F_n(x) - G_m(x)) && \text{(diffs[1])} \\
D_{mn}^{-} &= \max_x (\mathrm{G}_m(x) - F_n(x)) && \text{(diffs[2])}
\end{aligned}$$

Asymptotically, the distribution of the statistic

$$Z = D_{mn}\sqrt{(m+n)/(m*n)}$$

(returned in `test_statistics[0]`) converges to a distribution given by Smirnov (1939).

Exact probabilities for the two-sided test are computed when $n*m$ is less than or equal to 10^4, according to an algorithm given by Kim and Jennrich (1973). When $n*m$ is greater than 10^4, the very good approximations given by Kim and Jennrich are used to obtain the two-sided *p*-values. The one-sided probability is taken as one half the two-sided probability. This is a very good approximation when the *p*-value is small (say, less than 0.10) and not very good for large *p*-values.

Example

The following example illustrates the `imsls_f_kolmogorov_two` routine with two randomly generated samples from a uniform(0,1) distribution. Since the two theoretical distributions are identical, we would not expect to reject the null hypothesis.

```
#include <imsls.h>

#include <stdio.h>
```

```
void main()
{
        float *statistics=NULL, *diffs = NULL, *x=NULL, *y=NULL;
        int nobsx = 100,  nobsy = 60, nmissx, nmissy;
        imsls_random_seed_set(123457);
        x = imsls_f_random_uniform(nobsx, 0);
        y = imsls_f_random_uniform(nobsy, 0);
        statistics = imsls_f_kolmogorov_two(nobsx, x, nobsy, y,
                                IMSLS_N_MISSING_X, &nmissx,
                                IMSLS_N_MISSING_Y, &nmissy,
                                IMSLS_DIFFERENCES, &diffs,
                                0);
        printf("D      = %8.4f\n", diffs[0]);
        printf("D+     = %8.4f\n", diffs[1]);
        printf("D-     = %8.4f\n", diffs[2]);
        printf("Z      = %8.4f\n", statistics[0]);
        printf("Prob greater D one sided  = %8.4f\n", statistics[1]);
        printf("Prob greater D two sided  = %8.4f\n", statistics[2]);
        printf("Missing X = %d\n", nmissx);
        printf("Missing Y = %d\n", nmissy);
}
```

Output

```
D     =   0.1800
D+    =   0.1800
D-    =   0.0100
Z     =   1.1023
Prob greater D one sided  =   0.0720
Prob greater D two sided  =   0.1440
Missing X =   0
Missing Y =   0
```

multivar_normality_test

Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.

Synopsis

#include `<imsls.h>`

float `*imsls_f_multivar_normality_test` (*int* `n_observations`,
    int `n_variables`, *float* `x[]`, ..., 0)

The type *double* function is `imsls_d_multivar_normality_test`.

Required Arguments

int `n_observations` (Input)
    Number of observations (number of rows of data) `x`.

int `n_variables` (Input)
    Dimenionality of the multivariate space for which the skewness and kurtosis are to be computed. Number of variables in `x`.

float `x[]` (Input)
    Array of size `n_observations` by `n_variables` containing the data.

Return Value

A pointer to an array of dimension 13 containing output statistics

I	**stat[I]**
0	estimated skewness
1	expected skewness assuming a multivariate normal distribution
2	asymptotic chi-squared statistic assuming a multivariate normal distribution
3	probability of a greater chi-squared
4	Mardia and Foster's standard normal score for skewness
5	estimated kurtosis
6	expected kurtosis assuming a multivariate normal distribution
7	asymptotic standard error of the estimated kurtosis
8	standard normal score obtained from `stat[5]` through `stat[7]`
9	*p*-value corresponding to `stat[8]`
10	Mardia and Foster's standard normal score for kurtosis
11	Mardia's S_W statistic based upon `stat[4]` and `stat[10]`

12 *p*-value for `stat[11]`

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_multivar_normality_test` (*int* `n_observations_x`, *int* `n_variables`, *float* `x[]`, ...
`IMSLS_FREQUENCIES`, *float* `frequencies[]`,
`IMSLS_WEIGHTS`, *float* `weights[]`,
`IMSLS_SUM_FREQ`, *int* `*sum_frequencies`,
`IMSLS_SUM_WEIGHTS`, *float* `*sum_weights`,
`IMSLS_N_ROWS_MISSING`, *int* `*nrmiss`,
`IMSLS_MEANS`, *float* `**means`,
`IMSLS_MEANS_USER`, *float* `means[]`,
`IMSLS_R`, *float* `**R_matrix`,
`IMSLS_R_USER`, *float* `R_matrix[]`,
`IMSLS_RETURN_USER`, *float* `test_statistics[]`,
0)

Optional Arguments

`IMSLS_FREQUENCIES`, *float* `frequencies[]` (Input)
Array of size `n_rows` containing the frequencies. Frequencies must be integer valued. Default assumes all frequencies equal one.

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of size `n_rows` containing the weights. Weights must be greater than non-negative. Default assumes all weights equal one.

`IMSLS_SUM_FREQ`, *int* `*sum_frequencies` (Output)
The sum of the frequencies of all observations used in the computations.

`IMSLS_SUM_WEIGHTS`, *float* *`weights[]` (Output)
The sum of the weights times the frequencies for all observations used in the computations.

`IMSLS_N_ROWS_MISSING`, *int* **`nrmiss` (Output)
Number of rows of data in `x[]` containing any missing values (`NaN`).

`IMSLS_MEANS`, *float* **`means` (Output)
The address of a pointer to an array of length `n_variables` containing the sample means.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
Storage for array `means` is provided by user. See `IMSLS_MEANS`.

`IMSLS_R`, *float* **`R_matrix` (Output)
The address of a pointer to an `n_variables` by `n_variables` upper

triangular matrix containing the Cholesky R^TR factorization of the covariance matrix.

`IMSLS_R_USER`, *float* `R_matrix[]` (Output)
Storage for array `R_matrix` is provided by user. See `IMSLS_R`.

`IMSLS_RETURN_USER`, *float* `stat[]` (Output)
User supplied array of dimension 13 containing the estimates and their associated test statistics.

Description

Function `imsls_f_multivar_normality_test` computes Mardia's (1970) measures $b_{1,p}$ and $b_{2,p}$ of multivariate skewness and kurtosis, respectfully, for p = `n_variables`. These measures are then used in computing tests for multivariate normality. Three test statistics, one based upon $b_{1,p}$ alone, one based upon $b_{2,p}$ alone, and an omnibus test statistic formed by combining normal scores obtained from $b_{1,p}$ and $b_{2,p}$ are computed. On the order of np^3, operations are required in computing $b_{1,p}$ when the method of Isogai (1983) is used, where n = `n_observations`. On the order of np^2, operations are required in computing $b_{2,p}$.

Let

$$d_{ij} = \sqrt{w_i w_j}\,(x_i - \bar{x})^T S^{-1}(x_j - \bar{x})$$

where

$$S = \frac{\sum_{i=1}^{n} w_i f_i (x_i - \bar{x})(x_i - \bar{x})^T}{\sum_{i=1}^{n} f_i}$$

$$\bar{x} = \frac{1}{\sum_{i=1}^{n} w_i f_i} \sum_{i=1}^{n} w_i f_i x_i$$

f_i is the frequency of the i-th observation, and w_i is the weight for this observation. (Weights w_i are defined such that x_i is distributed according to a multivariate normal, $N(\mu, \Sigma/w_i)$ distribution, where Σ is the covariance matrix.) Mardia's multivariate skewness statistic is defined as:

$$b_{1,p} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} f_i f_j d_{ij}^3$$

while Mardia's kurtosis is given as:

$$b_{2,p} = \frac{1}{n}\sum_{i=1}^{n} f_i d_{ii}^2$$

Both measures are invariant under the affine (matrix) transformation $AX + D$, and reduce to the univariate measures when p = `n_variables` = 1. Using formulas given in Mardia and Foster (1983), the approximate expected value, asymptotic standard error, and asymptotic p-value for $b_{2,p}$, and the approximate expected value, an asymptotic chi-squared statistic, and p-value for the $b_{1,p}$ statistic are computed. These statistics are all computed under the null hypothesis of a multivariate normal distribution. In addition, standard normal scores $W_1(b_{1,p})$ and $W_2(b_{2,p})$ (different from but similar to the asymptotic normal and chi-squared statistics above) are computed. These scores are combined into an asymptotic chi-squared statistic with two degrees of freedom:

$$S_W = W_1^2\left(b_{1,p}\right) + W_2^2\left(b_{2,p}\right)$$

This chi-squared statistic may be used to test for multivariate normality. A p-value for the chi-squared statistic is also computed.

Example

In the following example, 150 observations from a 5 dimensional standard normal distribution are generated via routine `imsls_f_random_normal` (Chapter 12, "Random Number Generation"). The skewness and kurtosis statistics are then computed for these observations.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
  float *x, swt, *xmean, *r,  *stats;
  int nobs = 150, ncol = 5, nvar = 5, izero = 0, ni, nrmiss;
  imsls_random_seed_set(123457);
  x = imsls_f_random_normal(nobs*nvar,  0);
  stats = imsls_f_multivar_normality_test(nobs, nvar, x,
                               IMSLS_SUM_FREQ, &ni,
                               IMSLS_SUM_WEIGHTS, &swt,
                               IMSLS_N_ROWS_MISSING, &nrmiss,
                               IMSLS_R, &r,IMSLS_MEANS, &xmean,
                              0);
  printf("Sum of frequencies  = %d\nSum of the weights =%8.3f\nNumber
                               rows missing = %3d\n", ni, swt, nrmiss);
  imsls_f_write_matrix("stat", 13, 1, stats,
                       IMSLS_ROW_NUMBER_ZERO,
                       0)
}
```

Output

```
Sum of frequencies  = 150
Sum of the weights  = 150.000
```

```
Number rows missing =   0

   stat
0       0.73
1       1.36
2      18.62
3       0.99
4      -2.37
5      32.67
6      34.54
7       1.27
8      -1.48
9       0.14
10      1.62
11      8.24
12      0.02

                  means
     1         2         3         4         5
0.02623   0.09238   0.06536   0.09819   0.05639

                    R
        1        2        3        4        5
1   1.033   -0.084   -0.065    0.108   -0.067
2   0.000    1.049   -0.097   -0.042   -0.021
3   0.000    0.000    1.063    0.006   -0.145
4   0.000    0.000    0.000    0.942   -0.084
5   0.000    0.000    0.000    0.000    0.949
```

randomness_test

Performs a test for randomness.

Synopsis

#include `<imsls.h>`

float `imsls_f_randomness_test` (*int* `n_observations`, *float* `x[]`, *int* `n_run`..., 0)

The type *double* function is `imsls_d_randomness_test`.

Required Arguments

int `n_observations` (Input)
Number of observations in `x`.

float `x[]` (Input)
Array of size `n_observations` containing the data.

int `n_run` (Input)
Length of longest run for which tabulation is desired. For optional arguments `IMSLS_PAIRS`, `IMSLS_DSQUARE`, and `IMSLS_DCUBE`,

`n_run` stands for the number of equiprobable cells into which the statistics are to be tabulated.

Return Value

The probability of a larger chi-squared statistic for testing the null hypothesis of a uniform distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `imsls_f_randomness_test` (*int* `n_observations_x`, *float* `x[]`, *int* `n_run`, ...

`IMSLS_RUNS`, *float* `**runs_count`, *float* `**covariances`,

`IMSLS_RUNS_USER`, *float* `runs_count[]`, *float* `covariances[]`,
`IMSLS_PAIRS`, *int* `pairs_lag`, *float* `**pairs_count`,

`IMSLS_PAIRS_USER`, *int* `pairs_lag`, *float* `pairs_count[]`,

`IMSLS_DSQUARE`, *float* `**dsquare_count`,

`IMSLS_DSQUARE_USER`, *float* `dsquare_count[]`,

`IMSLS_DCUBE`, *float* `**dcube_count`,

`IMSLS_DCUBE_USER`, *float* `dcube_count[]`,

`IMSLS_RUNS_EXPECT`, *float* `**runs_expect`,

`IMSLS_RUNS_EXPECT_USER`, *float* `runs_expect[]`,

`IMSLS_EXPECT`, *float* `*expect`,

`IMSLS_CHI_SQUARED`, *float* `*chi_squared`,

`IMSLS_DF`, *float* `*df`,

`IMSLS_RETURN USER`, *float* `*pvalue`,

0)

Optional Arguments

`IMSLS_RUNS`, *float* `**runs_count`, *float* `**covariances`, (Output) or

`IMSLS_PAIRS`, *int* `pairs_lag` (Input), *float* `**pairs_count`, (Output) or

`IMSLS_DSQUARE`, *float* `**dsquare_count`, (Output) or

`IMSLS_DCUBE`, *float* `**dcube_count`, (Output)

`IMSLS_RUNS` indicates the runs test is to be performed. Array of length `n_run` containing the counts of the number of runs up of each length is returned in `*runs_counts`. `n_run` by `n_observations` matrix containing the variances and covariances of the counts is returned in `*covariances`. `IMSLS_RUNS` is the default test, however, to return the counts and covariances `IMSLS_RUNS` argument must be used.

IMSLS_PAIRS indicates the pairs test is to be performed. The lag to be used in computing the pairs statistic is stored in pairs_lag. Pairs (X[i], X[i + pairs_lag]) for i = 0,…, N - pairs_lag -1 are tabulated, where N is the total sample size. n_run by n_run matrix containing the count of the number of pairs in each cell is returned in pairs_user.

IMSLS_DSQUARE indicates the d^2 test is to be performed. **dsquare_counts is an address of a pointer to an internally allocated array of length n_run containing the tabulations for the d^2 test.

IMSLS_DCUBE indicates the triplets test is to be performed. **dcube_counts is an address of a pointer to an internally allocated array of length n_run by n_run by n_run containing the tabulations for the triplets test.

IMSLS_RUNS_USER, *float* runs_counts[], *float* covariances[] (Output)
Storage for runs_counts and covariances is provided by the user. See IMSLS_RUNS.

IMSLS_PAIRS_USER, *int* pairs_lag, *float* pairs_counts[] (Output)
Storage for pairs_lag and pairs_counts is provided by the user. See IMSLS_PAIRS.

IMSLS_DSQUARE_USER, *float* dsquare_count[] (Output)
Storage for dsquare_count is provided by the user. See IMSLS_DSQUARE.

IMSLS_DCUBE_USER, *float* dcube_count[] (Output)
Storage for dcube_count is provided by the user. See IMSLS_DCUBE.

IMSLS_CHI_SQUARED, *float* *chi_squared (Output)
Chi-squared statistic for testing the null hypothesis of a uniform distribution.

IMSLS_DF, *float* *df (Output)
Degrees of freedom for chi-squared.

IMSLS_RETURN_USER, *float* *pvalue (Output)
If specified, pvalue returns the probability of a larger chi-squared statistic for testing the null hypothesis of a uniform distribution.

If IMSLS_RUNS is specified:

IMSLS_RUNS_EXPECT, *float* **runs_expect (Output)
The address of a pointer to an internally allocated array of length n_run containing the expected number of runs of each length.

IMSLS_RUNS_EXPECT_USER, *float* runs_expect[] (Output)
Storage for runs_expect is provided by the user. See IMSLS_RUNS_EXPECT.

If IMSLS_PAIRS, IMSLS_DSQUARE, or IMSLS_DCUBE is specified:

`IMSLS_EXPECT`, *float* `**expect` (Output)

Expected number of counts for each cell. This argument is optional only if one of `IMSLS_PAIRS`, `IMSLS_DSQUARE`, or `IMSLS_DCUBE` is used.

Description

Runs Up Test

Function `imsls_f_randomness_test` performs one of four different tests for randomness. Optional argument `IMSLS_RUNS` computes statistics for the runs up test. Runs tests are used to test for cyclical trend in sequences of random numbers. If the runs down test is desired, each observation should first be multiplied by −1 to change its sign, and `IMSLS_RUNS` called with the modified vector of observations.

`IMSLS_RUNS` first tallies the number of runs up (increasing sequences) of each desired length. For $i = 1, \ldots, r - 1$, where $r =$ `n_run`, `runs_count`[i] contains the number of runs of length i. `runs_count[n_run]` contains the number of runs of length `n_run` or greater. As an example of how runs are counted, the sequence (1, 2, 3, 1) contains 1 run up of length 3, and one run up of length 1.

After tallying the number of runs up of each length, `IMSLS_RUNS` computes the expected values and the covariances of the counts according to methods given by Knuth (1981, pages 65–67). Let R denote a vector of length `n_run` containing the number of runs of each length so that the i-th element of R, r_i, contains the count of the runs of length i. Let Σ_R denote the covariance matrix of R under the null hypothesis of randomness, and let μ_R denote the vector of expected values for R under this null hypothesis, then an approximate chi-squared statistic with `n_run` degrees of freedom is given as

$$\chi^2 = (R - \mu_R)^T \Sigma_R^{-1} (R - \mu_R)$$

In general, the larger the value of each element of μ_R, the better the chi-squared approximation.

Pairs Test

`IMSLS_PAIRS` computes the pairs test (or the Good's serial test) on a hypothesized sequence of uniform (0,1) pseudorandom numbers. The test proceeds as follows. Subsequent pairs (`X`(i), `X`(i + `pairs_lag`)) are tallied into a $k \times k$ matrix, where $k =$ `n_run`. In this tally, element (j, m) of the matrix is incremented, where

$$j = \lfloor kX(i) \rfloor + 1$$
$$m = \lfloor kX(i+l) \rfloor + 1$$

where $l =$ `pairs_lag`, and the notation $\lfloor\ \rfloor$ represents the greatest integer function, $\lfloor Y \rfloor$ is the greatest integer less than or equal to Y, where Y is a real number. If $l = 1$, then $i = 1, 3, 5, \ldots, n - 1$. If $l > 1$, then $i = 1, 2, 3, \ldots, n - l$, where n is the total number of pseudorandom numbers input on the current invocation of IMSLS_PAIRS (*i*.e., n = n_observations).

Given the tally matrix in pairs_count, chi-squared is computed as

$$\chi^2 = \sum_{i,j=0}^{k-1} \frac{(o_{ij} - e)^2}{e}$$

where $e = \sum o_{ij}/k^2$, and o_{ij} is the observed count in cell (i, j) (o_{ij} = pairs_count(i, j)).

Because pair statistics for the trailing observations are not tallied on any call, the user should call IMSLS_PAIRS with n_observations as large as possible. For pairs_lag < 20 and n_observations = 2000, little power is lost.

d^2 Test

IMSLS_DSQAR computes the d^2 test for succeeding quadruples of hypothesized pseudorandom uniform (0, 1) deviates. The d^2 test is performed as follows. Let X_1, X_2, X_3, and X_4 denote four pseudorandom uniform deviates, and consider

$$D^2 = (X_3 - X_1)^2 + (X_4 - X_2)^2$$

The probability distribution of D^2 is given as

$$\Pr(D^2 \le d^2) = d^2\pi - \frac{8d^3}{3} + \frac{d^4}{2}$$

when $D^2 \le 1$, where π denotes the value of pi. If $D^2 > 1$, this probability is given as

$$\Pr(D^2 \le d^2) = \frac{1}{3} + (\pi - 2)d^2 + 4\sqrt{d^2 - 1}$$
$$+ 8\frac{(d^2-1)^{\frac{3}{2}}}{3} - \frac{d^4}{2} - 4d^2 \arctan\left(\frac{\sqrt{1 - \frac{1}{d^2}}}{\frac{1}{d}}\right)$$

See Gruenberger and Mark (1951) for a derivation of this distribution.

For each succeeding set of 4 pseudorandom uniform numbers input in X, d^2 and the cumulative probability of d^2 ($\Pr(D^2 \le d^2)$) are computed. The resulting probability is tallied into one of k = n_run equally spaced intervals.

Let n denote the number of sets of four random numbers input (n = the total number of observations/4). Then, under the null hypothesis that the numbers input are random uniform (0, 1) numbers, the expected value for each element in dsquare_count is $e = n/k$. An approximate chi-squared statistic is computed as

$$\chi^2 = \sum_{i=0}^{k-1} \frac{(o_i - e)^2}{e}$$

where o_i = `dsquare_count`(i) is the observed count. Thus, χ^2 has $k - 1$ degrees of freedom, and the null hypothesis of pseudorandom uniform (0, 1) deviates is rejected if χ^2 is too large. As n increases, the chi-squared approximation becomes better. A useful generalization is that $e > 5$ yields a good chi-squared approximation.

Triplets Test

`IMSLS_DCUBE` computes the triplets test on a sequence of hypothesized pseudorandom uniform(0, 1) deviates. The triplets test is computed as follows:

Each set of three successive deviates, X_1, X_2, and X_3, is tallied into one of m^3 equal sized cubes, where m = `n_run`. Let $i = [mX_1] + 1$, $j = [mX_2] + 1$, and $k = [mX_3] + 1$. For the triplet (X_1, X_2, X_3), `dcube_count`(i, j, k) is incremented.

Under the null hypothesis of pseudorandom uniform(0, 1) deviates, the m^3 cells are equally probable and each has expected value $e = n/m^3$, where n is the number of triplets tallied. An approximate chi-squared statistic is computed as

$$\chi^2 = \sum_{i,j,k=0}^{k-1} \frac{(o_{ijk} - e)^2}{e}$$

where o_{ijk} = `dcube_count`(i, j, k).

The computed chi-squared has $m^3 - 1$ degrees of freedom, and the null hypothesis of pseudorandom uniform (0, 1) deviates is rejected if χ^2 is too large.

Example 1

The following example illustrates the use of the runs test on 10^4 pseudo-random uniform deviates. In the example, 2000 deviates are generated for each call to `IMSLS_RUNS`. Since the probability of a larger chi-squared statistic is 0.1872, there is no strong evidence to support rejection of this null hypothesis of randomness.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        int nran = 10000, n_run = 6;
        char *fmt = "%8.1f";
        float *x, pvalue, *runs_counts, *runs_expect, chisq, df;
        imsls_random_seed_set(123457);
        x = imsls_f_random_uniform(nran, 0);
        pvalue = imsls_f_randomness_test(nran, x, n_run,
                              IMSLS_CHI_SQUARED, &chisq,
                              IMSLS_DF, &df,
                              IMSLS_RUNS_EXPECT, &runs_expect,
                              IMSLS_RUNS, &runs_counts, &covariances,
                              0);
        imsls_f_write_matrix("runs_counts", 1, n_run, runs_counts, 0);
        imsls_f_write_matrix("runs_expect", 1, n_run, runs_expect,
                              IMSLS_WRITE_FORMAT, fmt,
                              0);
        imsls_f_write_matrix("covariances", n_run, n_run, covariances,
                              IMSLS_WRITE_FORMAT, fmt,
                              0);
        printf("chisq  =  %f\n", chisq);
        printf("df     =  %f\n", df);
        printf("pvalue =  %f\n", pvalue);

}
```

Output

```
                  runs_count
     1         2         3         4         5         6
1709.0    2046.0     953.0     260.0      55.0       4.0

                    runs_expect
     1         2         3         4         5         6
1667.3    2083.4     916.5     263.8      57.5      11.9

                    covariances
         1         2         3         4         5         6
1   1278.2    -194.6    -148.9     -71.6     -22.9      -6.7
2   -194.6    1410.1    -490.6    -197.2     -55.2     -14.4
3   -148.9    -490.6     601.4    -117.4     -31.2      -7.8
4    -71.6    -197.2    -117.4     222.1     -10.8      -2.6
5    -22.9     -55.2     -31.2     -10.8      54.8      -0.6
6     -6.7     -14.4      -7.8      -2.6      -0.6      11.7
chisq   =     8.76514
df      =     6.00000
pvalue  =    0.187225
```

Example 2

The following example illustrates the calculations of the `IMSLS_PAIRS` statistics when a random sample of size 10^4 is used and the `pairs_lag` is 1. The results are not significant. IMSL routine i`msls_f_random_uniform` (Chapter 12, "Random Number Generation) is used in obtaining the pseudorandom deviates.

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

```
void main()
{
       int nran = 10000, n_run = 10;
       float *x, pvalue, *pairs_counts, expect, chisq, df;
       imsls_random_seed_set(123467);
       x = imsls_f_random_uniform(nran, 0);
       pvalue = imsls_f_randomness_test(nran, x, n_run,
                                   IMSLS_CHI_SQUARED, &chisq,
                                   IMSLS_DF, &df,
                                   IMSLS_EXPECT, &expect,
                                   IMSLS_PAIRS, 5, &pairs_counts,
                                   0);
       imsls_f_write_matrix("pairs_counts", n_run, n_run, pairs_counts, 0);
       printf("expect =  %8.2f\n", expect);
       printf("chisq  =  %8.2f\n", chisq);
       printf("df     =  %8.2f\n", df);
       printf("pvalue =  %10.4f\n", pvalue);
}
```

Output

```
pairs_counts
      1     2     3     4     5     6     7     8     9     10
 1   112    82    95   118   103   103   113   84    90     74
 2   104   106   109   108   101    98   102   92    109    88
 3    88   111    86   106   112    79   103  105    106   101
 4    91   110   108   92     88   108   113   93    105   114
 5   104   105   103   104   101    94    96   87     93   104
 6    98   104   103   104    79    89    92   104    92   100
 7   103    91    97   101   116    83   118   118   106    99
 8   105   105   111    91    93    82   100   104   110    89
 9    92   102    82   101    94    128  102   110   125    98
10    79    99   103    98   104    101   93    93    98   105

expect =     99.95
chisq  =    104.86
df     =     99.00
pvalue =      0.3242
```

Example 3

In the following example, 2000 observations generated via IMSL routine `imsls_f_random_uniform` (Chapter 12, "Random Number Generation") are input to `IMSLS_DSQAR` in one call. In the example, the null hypothesis of a uniform distribution is not rejected.

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

```
void main()
{
 int nran = 2000, n_run = 6;
       float *x, pvalue, *dsquare_counts, *covariances, expect, chisq, df;
       imsls_random_seed_set(123457);
       x = imsls_f_random_uniform(nran, 0);
       pvalue = imsls_f_randomness_test(nran, x, n_run,
                                IMSLS_CHI_SQUARED, &chisq,
                                IMSLS_DF, &df,
                                IMSLS_EXPECT, &expect,
                                IMSLS_DSQUARE, &dsquare_counts,
                                0);
       imsls_f_write_matrix("dsquare_counts", 1, n_run, dsquare_counts, 0);
       printf("expect = %10.4f\n", expect);
       printf("chisq  = %10.4f\n", chisq);
       printf("df     = %8.2f\n", df);
       printf("pvalue = %10.4f\n", pvalue);
}
```

Output

```
             dsquare_counts
    1        2        3        4        5        6
   87       84       78       76       92       83
expect   =     83.3333
chisq    =      2.0560
df       =      5.00
pvalue   =      0.8413
```

Example 4

In the following example, 2001 deviates generated by IMSL routine `imsls_f_random_uniform` (Chapter 12, "Regression") are input to `IMSLS_DCUBE`, and tabulated in 27 equally sized cubes. In the example, the null hypothesis is not rejected.

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

void main()
{
       int nran = 2001, n_run = 3;
       float *x, pvalue, *dcube_counts, expect, chisq, df;
       imsls_random_seed_set(123457);
       x = imsls_f_random_uniform(nran, 0);
       pvalue = imsls_f_randomness_test(nran, x, n_run,
                              IMSLS_CHI_SQUARED, &chisq,
                              IMSLS_DF, &df,
                              IMSLS_EXPECT, &expect,
                              IMSLS_DCUBE, &dcube_counts,
                              0);
       imsls_f_write_matrix("dcube_counts", n_run, n_run, dcube_counts, 0);
       imsls_f_write_matrix("dcube_counts", n_run, n_run,
                              &dcube_counts[n_run*n_run], 0);
       imsls_f_write_matrix("dcube_counts", n_run, n_run,
                              &dcube_counts[2*n_run*n_run], 0);
       printf("expect = %10.4f\n", expect);
       printf("chisq  = %10.4f\n", chisq);
       printf("df     = %8.2f\n", df);
       printf("pvalue = %10.4f\n", pvalue);
}
```

Output

```
        dcube_counts
            1       2       3
1          26      27      24
2          20      17      32
3          30      18      21

           dcube_counts
            1       2       3
1          20      16      26
2          22      22      27
3          30      24      26

           dcube_counts
           1        2       3
1          28      30      22
2          23      24      22
3          33      30      27
expect =     24.7037
chisq  =     21.7631
df     =     26.0000
pvalue =    0.701586
```

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.

Afifi and Azen

Afifi, A.A. and S.P. Azen (1979), *Statistical Analysis: A Computer Oriented Approach*, 2d ed., Academic Press, New York.

Agresti, Wackerly, and Boyette

Agresti, Alan, Dennis Wackerly, and James M. Boyette (1979), Exact conditional tests for cross-classifications: Approximation of attained significance levels, *Psychometrika*, **44**, 75-83.

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.

Ahrens, J.H., and U. Dieter (1985), Sequential random sampling, *ACM Transactions on Mathematical Software*, **11**, 157−169.

Anderberg

Anderberg, Michael R. (1973), *Cluster Analysis for Applications*, Academic Press, New York.

Anderson

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, John Wiley & Sons, New York.

Anderson and Bancroft

Anderson, R.L. and T.A. Bancroft (1952), *Statistical Theory in Research*, McGraw-Hill Book Company, New York.

Atkinson

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.

Atkinson, A.C. (1985), *Plots, Transformations, and Regression*, Claredon Press, Oxford.

Barrodale and Roberts

Barrodale, I., and F.D.K. Roberts (1973), An improved algorithm for discrete L_1 approximation, *SIAM Journal on Numerical Analysis*, **10**, 839–848.

Barrodale, I., and F.D.K. Roberts (1974), Solution of an overdetermined system of equations in the l_1 norm, *Communications of the ACM*, **17**, 319–320.

Barrodale, I., and C. Phillips (1975), Algorithm 495. Solution of an overdetermined system of linear equations in the Chebyshev norm, *ACM Transactions on Mathematical Software*, **1**, 264–270.

Bartlett, M. S.

Bartlett, M.S. (1935), Contingency table interactions, *Journal of the Royal Statistics Society Supplement*, **2**, 248–252.

Bartlett, M. S. (1937) Some examples of statistical methods of research in agriculture and applied biology*, Supplement to the Journal of the Royal Statistical Society*, **4**, 137-183.

Bartlett, M. (1937), The statistical conception of mental factors, *British Journal of Psychology*, **28**, 97–104.

Bartlett, M.S. (1946), On the theoretical specification and sampling properties of autocorrelated time series, *Supplement to the Journal of the Royal Statistical Society*, **8**, 27–41.

Bartlett, M.S. (1978), *Stochastic Processes,* 3rd. ed., Cambridge University Press, Cambridge.

Bays and Durham

Bays, Carter and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59–64.

Bendel and Mickey

Bendel, Robert B., and M. Ray Mickey (1978), Population correlation matrices for sampling experiments, *Communications in Statistics*, **B7**, 163–182.

Best and Fisher

Best, D.J., and N.I. Fisher (1979), Efficient simulation of the von Mises distribution, *Applied Statistics*, **28**, 152−157.

Bishop et al

Bishop, Yvonne M.M., Stephen E. Feinberg, and Paul W. Holland (1975), *Discrete Multivariate Analysis: Theory and Practice*, MIT Press, Cambridge, Mass.

Bjorck and Golub

Bjorck, Ake, and Gene H. Golub (1973), Numerical Methods for Computing Angles Between Subspaces, *Mathematics of Computation,* **27**, 579−594.

Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

Bosten and Battiste

Bosten, Nancy E. and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156s−157.

Box and Jenkins

Box, George E.P. and Gwilym M. Jenkins (1976), *Time Series Analysis: Forecasting and Control*, revised ed., Holden-Day, Oakland.

Box and Pierce

Box, G.E.P., and David A. Pierce (1970), Distribution of residual autocorrelations in autoregressive-integrated moving average time series models, *Journal of the American Statistical Association*, **65**, 1509–1526.

Box and Tidwell

Box, G.E.P. and P.W. Tidwell (1962), Transformation of the independent variables, *Technometrics*, **4**, 531−550.

Boyette

Boyette, James M. (1979), Random RC tables with given row and column totals, *Applied Statistics*, **28**, 329−332.

Bradley

Bradley, J.V. (1968), *Distribution-Free Statistical Tests*, Prentice-Hall, New Jersey.

Breslow

Breslow, N.E. (1974), Covariance analysis of censored survival data, *Biometrics*, **30**, 89−99.

Brown

Brown, Morton E. (1983), MCDP4F, two-way and multiway frequency tables-measures of association and the log-linear model (complete and incomplete tables), in *BMDP Statistical Software, 1983 Printing with Additions*, (edited by W.J. Dixon), University of California Press, Berkeley.

Brown and Benedetti

Brown, Morton B. and Jacqualine K. Benedetti (1977), Sampling behavior and tests for correlation in two-way contingency tables, *Journal of the American Statistical Association*, **42**, 309−315.

Cheng

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317−322.

Chiang

Chiang, Chin Long (1968), *Introduction to Stochastic Processes in Statistics*, John Wiley & Sons, New York.

Conover

Conover, W.J. (1980), *Practical Nonparametric Statistics*, 2d ed., John Wiley & Sons, New York.

Conover and Iman

Conover, W.J. and Ronald L. Iman (1983), *Introduction to Modern Business Statistics*, John Wiley & Sons, New York.

Conover, W. J., Johnson, M. E., and Johnson, M. M

Conover, W. J., Johnson, M. E., and Johnson, M. M. (1981) A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data, *Technometrics*, **23**, 351-361.

Cook and Weisberg

Cook, R. Dennis and Sanford Weisberg (1982), *Residuals and Influence in Regression*, Chapman and Hall, New York.

Cooper

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190−192.

Cox

Cox, David R. (1970), *The Analysis of Binary Data*, Methuen, London.

Cox, D.R. (1972), Regression models and life tables (with discussion), *Journal of the Royal Statistical Society*, Series B, *Methodology*, **34**, 187–220.

Cox and Lewis

Cox, D.R., and P.A.W. Lewis (1966), *The Statistical Analysis of Series of Events*, Methuen, London.

Cox and Oakes

Cox, D.R., and D. Oakes (1984), *Analysis of Survival Data*, Chapman and Hall, London.

Cox and Stuart

Cox, D.R., and A. Stuart (1955), Some quick sign tests for trend in location and dispersion, *Biometrika*, **42**, 80–95.

D'Agostino and Stevens

D'Agostino, Ralph B. and Michael A. Stevens (1986), *Goodness-of-Fit Techniques*, Marcel Dekker, New York.

Dallal and Wilkinson

Dallal, Gerald E. and Leland Wilkinson (1986), An analytic approximation to the distribution of Lilliefor's test statistic for normality, *The American Statistician*, **40**, 294–296.

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.

Devore

Devore, Jay L (1982), *Probability and Statistics for Engineering and Sciences*, Brooks/Cole Publishing Company, Monterey, Calif.

Draper and Smith

Draper, N.R. and H. Smith (1981), *Applied Regression Analysis*, 2d ed., John Wiley & Sons, New York.

Durbin

Durbin, J. (1960), The fitting of time series models, *Revue Institute Internationale de Statistics*, **28**, 233–243.

Efroymson

Efroymson, M.A. (1960), Multiple regression analysis, *Mathematical Methods for Digital Computers*, Volume 1, (edited by A. Ralston and H. Wilf), John Wiley & Sons, New York, 191−203.

Ekblom

Ekblom, Hakan (1973), Calculation of linear best L_p-approximations, *BIT*, **13**, 292−300.

Ekblom, Hakan (1987), The L_1-estimate as limiting case of an L_p or Huber-estimate, in *Statistical Data Analysis Based on the L_1-Norm and Related Methods* (edited by Yadolah Dodge), North-Holland, Amsterdam, 109−116.

Elandt-Johnson and Johnson

Elandt-Johnson, Regina C., and Norman L. Johnson (1980), *Survival Models and Data Analysis*, John Wiley & Sons, New York, 172−173.

Emmett

Emmett, W.G. (1949), Factor analysis by Lawless method of maximum likelihood, *British Journal of Psychology, Statistical Section*, **2**, 90−97.

Engle

Engle, C. (1982), Autoregressive conditional heteroskedasticity with estimates of the variance of U.K. inflation, *Econometrica ,* **50**, 987−1008.

Fisher

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *The Annals of Eugenics*, **7**, 179−188.

Fishman

Fishman, George S. (1978), *Principles of Discrete Event Simulation*, John Wiley & Sons, New York.

Fishman and Moore

Fishman, George S. and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus , *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.

Fuller

Fuller, Wayne A. (1976), *Introduction to Statistical Time Series*, John Wiley & Sons, New York.

Furnival and Wilson

Furnival, G.M. and R.W. Wilson, Jr. (1974), Regressions by leaps and bounds, *Technometrics*, **16**, 499–511.

Fushimi

Fushimi, Masanori (1990), Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$, *Journal of Computational and Applied Mathematics*, **31**, 105–118.

Gentleman

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448–454.

Gibbons

Gibbons, J.D. (1971), *Nonparametric Statistical Inference*, McGraw-Hill, New York.

Girschick

Girschick, M.A. (1939), On the sampling theory of roots of determinantal equations, *Annals of Mathematical Statistics*, **10**, 203–224.

Golub and Van Loan

Golub, Gene H. and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Md.

Gonin and Money

Gonin, Rene, and Arthur H. Money (1989), *Nonlinear L_p-Norm Estimation*, Marcel Dekker, New York.

Goodnight

Goodnight, James H. (1979), A tutorial on the SWEEP operator, *The American Statistician*, **33**, 149–158.

Graybill

Graybill, Franklin A. (1976), *Theory and Application of the Linear Model*, Duxbury Press, North Scituate, Mass.

Griffin and Redish

Griffin, R. and K.A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

Gross and Clark

Gross, Alan J., and Virginia A. Clark (1975), *Survival Distributions: Reliability Applications in the Biomedical Sciences*, John Wiley & Sons, New York.

Gruenberger and Mark

Gruenberger, F., and A.M. Mark (1951), The d^2 test of random digits, *Mathematical Tables and Other Aids in Computation*, **5**, 109–110.

Guerra et al.

Guerra, Victor O., Richard A. Tapia, and James R. Thompson (1976), A random number generator for continuous random variables based on an interpolation procedure of Akima, in *Proceedings of the Ninth Interface Symposium on Computer Science and Statistics*, (edited by David C. Hoaglin and Roy E. Welsch), Prindle, Weber & Schmidt, Boston, 228–230.

Haldane

Haldane, J.B.S. (1939), The mean and variance of when used as a test of homogeneity, when expectations are small, *Biometrika*, **31**, 346.

Harman

Harman, Harry H. (1976), *Modern Factor Analysis*, 3d ed. revised, University of Chicago Press, Chicago.

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.

Hartigan

Hartigan, John A. (1975), *Clustering Algorithms*, John Wiley & Sons, New York.

Hartigan and Wong

Hartigan, J.A. and M.A. Wong (1979), Algorithm AS 136: A *K*-means clustering algorithm, *Applied Statistics*, **28**, 100−108.

Hayter

Hayter, Anthony J. (1984), A proof of the conjecture that the Tukey-Kramer multiple comparisons procedure is conservative, *Annals of Statistics*, **12**, 61−75.

Heiberger

Heiberger, Richard M. (1978), Generation of random orthogonal matrices, *Applied Statistics*, **27**, 199−206.

Hemmerle.

Hemmerle, William J. (1967), *Statistical Computations on a Digital Computer*, Blaisdell Publishing Company, Waltham, Mass.

Herraman

Herraman, C. (1968), Sums of squares and products matrix, *Applied Statistics*, **17**, 289−292.

Hill

Hill, G.W. (1970), Student's *t*-distribution, *Communications of the ACM*, **13**, 617−619.

Hill, G.W. (1970), Student's *t*-quantiles, *Communications of the ACM*, **13**, 619−620.

Hinkelmann, K and Kemthorne

Hinkelmann, K and Kemthorne, O (1994) *Design and Analysis of Experiments – Vol 1*, John Wiley.

Hinkley

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67−69.

Hoaglin and Welsch

Hoaglin, David C. and Roy E. Welsch (1978), The hat matrix in regression and ANOVA, *The American Statistician*, **32**, 17−22.

Hocking

Hocking, R.R. (1972), Criteria for selection of a subset regression: Which one should be used?, *Technometrics*, **14**, 967−970.

Hocking, R.R. (1973), A discussion of the two-way mixed model, *The American Statistician*, **27**, 148−152.

Hocking, R.R. (1985), *The Analysis of Linear Models*, Brooks/Cole Publishing Company, Monterey, California.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hughes and Saw

Hughes, David T., and John G. Saw (1972), Approximating the percentage points of Hotelling's generalized T_0^2 statistic, *Biometrika*, **59**, 224–226.

Iman and Davenport

Iman, R.L., and J.M. Davenport (1980), Approximations of the critical region of the Friedman statistic, *Communications in Statistics*, **A9(6)**,
571−595.

Jennrich and Robinson

Jennrich, R.I. and S.M. Robinson (1969), A Newton-Raphson algorithm for maximum likelihood factor analysis, *Psychometrika*, **34**, 111−123.

Jennrich and Sampson

Jennrich, R.I. and P.F. Sampson (1966), Rotation for simple loadings, *Psychometrika*, **31**, 313–323.

John

John, Peter W.M. (1971), *Statistical Design and Analysis of Experiments*, Macmillan Company, New York.

Jöhnk

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufallszahlen, *Metrika*, **8**, 5−15.

Johnson and Kotz

Johnson, Norman L., and Samuel Kotz (1969), *Discrete Distributions*, Houghton Mifflin Company, Boston.

Johnson, Norman L., and Samuel Kotz (1970a), *Continuous Univariate Distributions*-1, John Wiley & Sons, New York.

Johnson, Norman L., and Samuel Kotz (1970b), *Continuous Univariate Distributions*-2, John Wiley & Sons, New York.

Johnson and Welch

Johnson, D.G., and W.J. Welch (1980), The generation of pseudo-random correlation matrices, *Journal of Statistical Computation and Simulation*, **11**, 55−69.

Jonckheere

Jonckheere, A.R. (1954), A distribution-free *k*-sample test against ordered alternatives, *Biometrika*, **41**, 133−143.

Jöreskog

Jöreskog, K.G. (1977), Factor analysis by least squares and maximum-likelihood methods, *Statistical Methods for Digital Computers*, (edited by Kurt Enslein, Anthony Ralston, and Herbert S. Wilf), John Wiley & Sons, New York, 125−153.

Kachitvichyanukul

Kachitvichyanukul, Voratas (1982), *Computer generation of Poisson, binomial, and hypergeometric random variates*, Ph.D. dissertation, Purdue University, West Lafayette, Indiana.

Kaiser

Kaiser, H.F. (1963), Image analysis, *Problems in Measuring Change*, (edited by C. Harris), University of Wisconsin Press, Madison, Wis.

Kaiser and Caffrey

Kaiser, H.F. and J. Caffrey (1965), Alpha factor analysis, *Psychometrika*, **30**, 1−14.

Kalbfleisch and Prentice

Kalbfleisch, John D., and Ross L. Prentice (1980), *The Statistical Analysis of Failure Time Data*, John Wiley & Sons, New York.

Kemp

Kemp, A.W., (1981), Efficient generation of logarithmically distributed pseudo-random variables, *Applied Statistics,* **30**, 249−253.

Kendall and Stuart

Kendall, Maurice G. and Alan Stuart (1973), *The Advanced Theory of Statistics*, Volume 2: *Inference and Relationship*, 3d ed., Charles Griffin & Company, London.

Kendall, Maurice G. and Alan Stuart (1979), *The Advanced Theory of Statistics*, Volume 2: *Inference and Relationship*, 4th ed., Oxford University Press, New York.

Kendall et al.

Kendall, Maurice G., Alan Stuart, and J. Keith Ord (1983), *The Advanced Theory of Statistics*, Volume 3: *Design and Analysis, and Time Series*, 4th. ed., Oxford University Press, New York.

Kennedy and Gentle

Kennedy, William J., Jr. and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kuehl, R. O.

Kuehl, R. O. (2000) *Design of Experiments: Statistical Principles of Research Design and Analysis*, 2nd edition, Duxbury Press.

Kim and Jennrich

Kim, P.J., and R.I. Jennrich (1973), Tables of the exact sampling distribution of the two sample Kolmogorov-Smirnov criterion D_{mn} ($m < n$), in *Selected Tables in Mathematical Statistics*, Volume 1, (edited by H. L. Harter and D.B. Owen), American Mathematical Society, Providence, Rhode Island.

Kinderman and Ramage

Kinderman, A.J., and J.G. Ramage (1976), Computer generation of normal random variables, *Journal of the American Statistical Association*, **71**, 893−896.

Kinderman et al.

Kinderman, A.J., J.F. Monahan, and J.G. Ramage (1977), Computer methods for sampling from Student's *t* distribution, *Mathematics of Computation* **31**, 1009−1018.

Kinnucan and Kuki

Kinnucan, P. and H. Kuki (1968), *A Single Precision INVERSE Error Function Subroutine*, Computation Center, University of Chicago.

Kirk

Kirk, Roger E. (1982), *Experimental Design: Procedures for the Behavioral Sciences*, 2d ed., Brooks/Cole Publishing Company, Monterey, Calif.

Knuth

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, 2d ed., Addison-Wesley, Reading, Mass.

Kshirsagar

Kshirsagar, Anant M. (1972), *Multivariate Analysis*, Marcel Dekker, New York.

Lachenbruch

Lachenbruch, Peter A. (1975), *Discriminant Analysis*, Hafner Press, London.

Lai

Lai, D. (1998a), Local asymptotic normality for location-scale type processes. *Far East Journal of Theorectical Statistics*, (in press).

Lai, D. (1998b), Asymptotic distributions of the correlation integral based statistics. *Journal of Nonparametric Statistics*, (in press).

Lai, D. (1998c), Asymptotic distributions of the estimated BDS statistic and residual analysis of AR Models on the Canadian lynx data. *Journal of Biological Systems*, (in press).

Laird and Oliver

Laird, N.M., and D. Fisher (1981), Covariance analysis of censored survival data using log-linear analysis techniques, *JASA* **76**, 1231−1240.

Lawless

Lawless, J.F. (1982), *Statistical Models and Methods for Lifetime Data*, John Wiley & Sons, New York.

Lawley and Maxwell

Lawley, D.N. and A.E. Maxwell (1971), *Factor Analysis as a Statistical Method*, 2d ed., Butterworth, London.

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, Calif.

Lee

Lee, Elisa T. (1980), *Statistical Methods for Survival Data Analysis*, Lifetime Learning Publications, Belmont, Calif.

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.

Levene, H.

Levene, H. (1960) In *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, I. Olkin et al. editors, Stanford University Press, 278-292.

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.

Liffiefors

Lilliefors, H.W. (1967), On the Kolmogorov-Smirnov test for normality with mean and variance unknown, *Journal of the American Statistical Association*, **62**, 534−544.

Ljung and Box

Ljung, G.M., and G.E.P. Box (1978), On a measure of lack of fit in time series models, *Biometrika*, **65**, 297–303.

Longley

Longley, James W. (1967), An appraisal of least-squares programs for the electronic computer from the point of view of the user, *Journal of the American Statistical Association*, **62**, 819−841.

Marsaglia

Marsaglia, George (1964), Generating a variable from the tail of a normal distribution, *Technometrics*, **6**, 101−102.

Marsaglia, G. (1968), Random numbers fall mainly in the planes, *Proceedings of the National Academy of Sciences*, **61**, 25−28.

Marsaglia, G. (1972), The structure of linear congruential sequences, in *Applications of Number Theory to Numerical Analysis*, (edited by S. K. Zaremba), Academic Press, New York, 249−286.

Marsaglia, George (1972), Choosing a point from the surface of a sphere, *The Annals of Mathematical Statistics*, **43**, 645−646.

McKean and Schrader

McKean, Joseph W., and Ronald M. Schrader (1987), Least absolute errors analysis of variance, in *Statistical Data Analysis Based on the* L_1*-Norm and Related Methods* (edited by Yadolah Dodge), North-Holland, Amsterdam, 297−305.

McKeon

McKeon, James J. (1974), *F* approximations to the distribution of Hotelling's T_0^2, *Biometrika*, **61**, 381–383.

McCullagh and Nelder

McCullagh, P., and J.A. Nelder, (1983), *Generalized Linear Models*, Chapman and Hall, London.

Maindonald

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

Marazzi

Marazzi, Alfio (1985), Robust affine invariant covariances in ROBETH, ROBETH-85 document No. 6, Division de Statistique et Informatique, Institut Universitaire de Medecine Sociale et Preventive, Laussanne.

Mardia et al.

Mardia, K.V. (1970), Measures of multivariate skewness and kurtosis with applications, *Biometrics*, **57**, 519−530.

Mardia, K.V., J.T. Kent, J.M. Bibby (1979), *Multivariate Analysis*, Academic Press, New York.

Mardia and Foster

Mardia, K.V. and K. Foster (1983), Omnibus tests of multinormality based on skewness and kurtosis, *Communications in Statistics A, Theory and Methods*, **12**, 207−221.

Marquardt

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431−441.

Marsaglia

Marsaglia, George (1964), Generating a variable from the tail of a normal distribution, *Technometrics*, **6**, 101−102.

Marsaglia and Bray

Marsaglia, G. and T.A. Bray (1964), A convenient method for generating normal variables, *SIAM Review*, **6**, 260−264.

Marsaglia et al.

Marsaglia, G., M.D. MacLaren, and T.A. Bray (1964), A fast procedure for generating normal random variables, *Communications of the ACM*, **7**, 4−10.

Merle and Spath

Merle, G., and H. Spath (1974), Computational experiences with discrete L_p approximation, *Computing*, **12**, 315−321.

Miller

Miller, Rupert G., Jr. (1980), *Simultaneous Statistical Inference*, 2d ed., Springer-Verlag, New York.

Milliken and Johnson

Milliken, George A. and Dallas E. Johnson (1984), *Analysis of Messy Data, Volume 1: Designed Experiments*, Van Nostrand Reinhold, New York.

Moran

Moran, P.A.P. (1947), Some theorems on time series I, *Biometrika*, **34**, 281–291.

Moré et al.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for [4] MINPACK-1*, Argonne National Laboratory Report ANL-80_74, Argonne, Ill.

Morrison

Morrison, Donald F. (1976), *Multivariate Statistical Methods*, 2nd. ed. McGraw-Hill Book Company, New York.

Muller

Muller, M.E. (1959), A note on a method for generating points uniformly on N-dimensional spheres, *Communications of the ACM*, **2**, 19–20.

Nelson

Nelson, D. B. (1991), Conditional heteroskedasticity in asset returns: A new approach. *Econometrica*, , **59**, 347–370.

Nelson

Nelson, Peter (1989), Multiple Comparisons of Means Using Simultaneous Confidence Intervals, *Journal of Quality Technology*, **21**, 232–241.

Neter

Neter, John (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Ill.

Neter and Wasserman

Neter, John and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

Noether

Noether, G.E. (1956), Two sequential tests against trend, *Journal of the American Statistical Association*, **51**, 440–450.

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.

Palm

Palm, F. C. (1996), GARCH models of volatility. In *Handbook of Statistics*, Vol. 14, 209-240. Eds: Maddala and Rao. Elsevier,New York.

Patefield

Patefield, W.M. (1981), An efficient method of generating $R \times C$ tables with given row and column totals, *Applied Statistics*, **30**, 91–97.

Patefield and Tandy

Patefield, W.M. (1981), and Tandy D. (2000) Fast and Accurate Calculation of Owen’s T-Function, *J. Statistical Software*, **5**, Issue 5.

Peixoto

Peixoto, Julio L. (1986), Testable hypotheses in singular fixed linear models, Communications in Statistics: *Theory and Methods*, **15**,
1957−1973.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Pillai

Pillai, K.C.S. (1985), Pillai’s trace, in *Encyclopedia of Statistical Sciences, Volume 6*, (edited by Samuel Kotz and Norman L. Johnson), John Wiley & Sons, New York, 725−729.

Pregibon

Pregibon, Daryl (1981), Logistic regression diagnostics, *The Annals of Statistics*, **9**, 705−724.

Prentice

Prentice, Ross L. (1976), A generalization of the probit and logit methods for dose response curves, *Biometrics*, **32**, 761−768.

Priestley

Priestley, M.B. (1981), *Spectral Analysis and Time Series*, Volumes 1 and 2, Academic Press, New York.

Rao

Rao, C. Radhakrishna (1973), *Linear Statistical Inference and Its Applications*, 2d ed., John Wiley & Sons, New York.

Robinson

Robinson, Enders A. (1967), *Multichannel Time Series Analysis with Digital Computer Programs*, Holden-Day, San Francisco.

Royston

Royston, J.P. (1982a), An extension of Shapiro and Wilk's *W* test for normality to large samples, *Applied Statistics*, **31**, 115−124.

Royston, J.P. (1982b), The *W* test for normality, *Applied Statistics*, **31**, 176−180.

Royston, J.P. (1982c), Expected normal order statistics (exact and approximate), *Applied Statistics*, **31**, 161−165.

Sallas

Sallas, William M. (1990), An algorithm for an L_p norm fit of a multiple linear regression model, *American Statistical Association 1990 Proceedings of the Statistical Computing Section*, 131−136.

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.

Scheffe

Scheffe, Henry (1959), *The Analysis of Variance*, John Wiley & Sons, New York.

Schmeiser

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, *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, Ind.

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.

Searle

Searle, S.R. (1971), *Linear Models*, John Wiley & Sons, New York.

Seber

Seber, G.A.F. (1984), *Multivariate Observations*, John Wiley & Sons, New York.

Snedecor and Cochran

Snedecor and Cochran (1967) *Statistical Methods*, 6th edition, Iowa State University Press.

Snedecor, George W. & Cochran, William G.

Snedecor, George W. and Cochran, William G. (1967) *Statistical Methods*, 6th edition, Iowa State University Press, 296-298.

Shampine

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179−180.

Siegal

Siegal, Sidney (1956), *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, New York.

Singleton

Singleton, R.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185−187.

Smirnov

Smirnov, N.V. (1939), Estimate of deviation between empirical distribution functions in two independent samples (in Russian), *Bulletin of Moscow University*, **2**, 3−16.

Smith and Dubey

Smith, H., and S. D. Dubey (1964), "Some reliability problems in the chemical industry", Industrial Quality Control, 21 (2), 1964, 64-70.

Snedecor and Cochran

Snedecor, George W. and William G. Cochran (1967), *Statistical Methods*, 6th ed., Iowa State University Press, Ames, Iowa.

Sposito

Sposito, Vincent A. (1989), Some properties of L_p-estimators, in *Robust Regression*: *Analysis and Applications* (edited by Kenneth D. Lawrence and Jeffrey L. Arthur), Marcel Dekker, New York, 23−58.

Spurrier and Isham

Spurrier, John D. and Steven P. Isham (1985), Exact simultaneous confidence intervals for pairwise comparisons of three normal means, *Journal of the American Statistical Association*, **80**, 438−442.

Stablein, Carter, and Novak

Stablein, D.M, W.H. Carter, and J.W. Novak (1981), Analysis of survival data with nonproportional hazard functions, *Controlled Clinical Trials*, **2**, 149–159.

Stahel

Stahel, W. (1981), Robuste Schatzugen: Infinitesimale Opimalitat und Schatzugen von Kovarianzmatrizen, Dissertation no. 6881, ETH, Zurich.

Steel and Torrie

Steel and Torrie (1960) *Principles and Procedures of Statistics*, McGraw-Hill.

Stephens

Stephens, M.A. (1974), EDF statistics for goodness of fit and some comparisons, *Journal of the American Statistical Association*, **69**, 730−737.

Stirling

Stirling, W.D. (1981), Least squares subject to linear constraints, *Applied Statistics*, **30**, 204−212. (See correction, p. 357.)

Stoline

Stoline, Michael R. (1981), The status of multiple comparisons: simultaneous estimation of all pairwise comparisons in one-way ANOVA designs, *The American Statistician*, **35**, 134−141.

Strecok

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144−158.

Tanner and Wong

Tanner, Martin A., and Wing H. Wong (1983), The estimation of the hazard function from randomly censored data by the kernel method, *Annals of Statistics*, **11**, 989–993.

Tanner, Martin A., and Wing H. Wong (1984), Data-based nonparametric estimation of the hazard function with applications to model diagnostics and exploratory analysis, *Journal of the American Statistical Association*, **79**, 123–456.

Taylor and Thompson

Taylor, Malcolm S., and James R. Thompson (1986), Data based random number generation for a multivariate distribution via stochastic simulation, *Computational Statistics & Data Analysis*, **4**, 93−101.

Tezuka

Tezuka, S. (1995), *Uniform Random Numbers: Theory and Practice*. Academic Publishers, Boston.

Thompson

Thompson, James R, (1989), *Empirical Model Building*, John Wiley & Sons, New York.

Tucker and Lewis

Tucker, Ledyard and Charles Lewis (1973), A reliability coefficient for maximum likelihood factor analysis, *Psychometrika*, **38**, 1−10.

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.

Verdooren

Verdooren, L. R. (1963), Extended tables of critical values for Wilcoxon's test statistic, *Biometrika*, **50**, 177−186.

Wallace

Wallace, D.L. (1959), Simplified Beta-approximations to the Kruskal-Wallis H-test, *Journal of the American Statistical Association*, **54**, 225−230.

Weisberg

Weisberg, S. (1985), *Applied Linear Regression*, 2d ed., John Wiley & Sons, New York.

Woodfield

Woodfield, Terry J. (1990), Some notes on the Ljung-Box portmanteau statistic, *American Statistical Association 1990 Proceedings of the Statistical Computing Section*, 155–160.

Yates, F.

Yates, F. (1936) A new method of arranging variety trials involving a large number of varieties. *Journal of Agricultural Science*, **26**, 424-455.

Appendix B: Alphabetical Summary of Routines

Function	Purpose Statement	Page
`anova_balanced`	Analyzes a balanced complete experimental design for a fixed, random, or mixed model.	256
`anova_factorial`	Analyzes a balanced factorial design with fixed effects.	239
`anova_nested`	Analyzes a completely nested random model with possibly unequal numbers in the subgroups.	247
`anova_oneway`	Analyzes a one-way classification model.	230
`arma`	Computes least-square estimates of parameters for an ARMA model.	517
`arma_forecast`	Computes forecasts and their associated probability limits for an ARMA model.	527
`autocorrelation`	Computes the sample autocorrelation function of a stationary time series.	541
`beta`	Evaluates the complete beta function.	901
`beta_cdf`	Evaluates the beta probability distribution function.	730
`beta_incomplete`	Evaluates the real incomplete beta function.	903
`beta_inverse_cdf`	Evaluates the inverse of the beta distribution function.	731
`binomial_cdf`	Evaluates the binomial distribution function.	720
`binomial_coefficient`	Evaluates the binomial coefficient.	900
`binomial_pdf`	Evaluates the binomial probability function.	722
`bivariate_normal_cdf`	Evaluates the bivariate normal distribution function.	732
`box_cox_transform`	Performs a Box-Cox transformation.	537
`categorical_glm`	Analyzes categorical data using logistic, Probit, Poisson, and other generalized linear models.	425
`chi_squared_cdf`	Evaluates the chi-squared distribution function.	734

Function	Purpose Statement	Page
`chi_squared_inverse_cdf`	Evaluates the inverse of the chi-squared distribution function.	736
`chi_squared_test`	Performs a chi-squared goodness-of-fit test.	482
`cluster_hierarchical`	Performs a hierarchical cluster analysis given a distance matrix.	590
`cluster_k_means`	Performs a *K*-means (centroid) cluster analysis.	598
`cluster_number`	Computes cluster membership for a hierarchical cluster tree.	594
`cochran_q_test`	Performs a Cochran *Q* test for related observations.	472
`contingency_table`	Performs a chi-squared analysis of a two-way contingency table.	404
`continuous_table_setup`	Sets up table to generate pseudorandom numbers from a general continuous distribution.	812
`covariances`	Computes the sample variance-covariance or correlation matrix.	185
`cox_stuart_trends_test`	Performs the Cox and Stuart' sign test for trends in location and dispersion.	452
`crd_factorial`	Analyzes data from balanced and unbalanced completely randomized experiments.	267
`crosscorrelation`	Computes the sample cross-correlation function of two stationary time series	546
`ctime`	Returns the number of CPU seconds used.	911
`data_sets`	Retrieves a commonly analyzed data set.	890
`difference`	Differences a seasonal or nonseasonal time series.	532
`discrete_table_setup`	Sets up a table to generate pseudorandom numbers from a general discrete distribution.	781
`discriminant_analysis`	Performs discriminant function analysis.	628
`dissimilarities`	Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.	586
`error_code`	Returns the code corresponding to the error message from the last function called.	885
`error_options`	Sets various error handling options.	879
`exact_enumeration`	Computes exact probabilities in a two-way contingency table, using the total enumeration method.	417
`exact_network`	Computes exact probabilities in a two-way contingency table using the network algorithm.	419

Function	**Purpose Statement**	**Page**
`F_cdf`	Evaluates the *F* distribution function.	742
`F_inverse_cdf`	Evaluates the inverse of the *F* distribution function.	744
`factor_analysis`	Extracts initial factor-loading estimates in factor analysis.	609
`faure_next_point`	Computes a shuffled Faure sequence	856
`friedmans_test`	Performs Friedman's test for a randomized complete block design.	467
`gamma`	Evaluates the real gamma functions.	905
`gamma_cdf`	Evaluates the gamma distribution function.	745
`gamma_incomplete`	Evaluates the incomplete gamma function.	907
`gamma_inverse_cdf`	Evaluates the inverse of the gamma distribution function.	747
`garch`	Computes estimates of the parameters of a GARCH(p, q) model	566
`homogeneity`	Conducts Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.	378
`hypergeometric_cdf`	Evaluates the hypergeometric distribution function.	723
`hypergeometric_pdf`	Evaluates the hypergeometric probability function.	725
`hypothesis_partial`	Constructs a completely testable hypothesis.	96
`hypothesis_scph`	Sums of cross products for a multivariate hypothesis.	101
`hypothesis_test`	Tests for the multivariate linear hypothesis.	106
`k_trends_test`	Performs k-sample trends test against ordered alternatives.	475
`kalman`	Performs Kalman filtering and evaluates the likelihood function for the state-space model.	571
`kaplan_meier_estimates`	Computes Kaplan-Meier estimates of survival probabilities in stratified samples.	654
`kolmogorov_one`	Performs a Kolmogorov-Smirnov's one-sample test for continuos distributions.	494
`kolmogorov_two`	Performs a Kolmogorov-Smirnov's two-sample test	497
`kruskal_wallis_test`	Performs a Kruskal-Wallis's test for identical population medians.	465
`lack_of_fit`	Performs lack-of-fit test for an univariate time series or transfer function given the appropriate correlation function.	563

Function	Purpose Statement	Page
`latin_square`	Analyzes data from latin-square experiments.	288
`lattice`	Analyzes balanced and partially-balanced lattice experiments.	297
`life_tables`	Produces population and cohort life tables.	712
`Lnorm_regression`	Fits a multiple linear regression model using criteria other than least squares.	168
`log_beta`	Evaluates the log of the real beta function.	904
`log_gamma`	Evaluates the logarithm of the absolute value of the gamma function.	909
`machine (float)`	Returns information describing the computer's floating-point arithmetic.	888
`machine (integer)`	Returns integer information describing the computer's arithmetic.	886
`mat_mul_rect`	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any triple product.	893
`multi_crosscorrelation`	Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.	552
`multiple_comparisons`	Performs Student-Newman-Keuls multiple comparisons test.	385
`multivar_normality_test`	Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.	501
`noether_cyclical_trend`	Performs the Noether's test for cyclical trend.	449
`non_central_chi_sq`	Evaluates the noncentral chi-squared distribution function.	738
`non_central_chi_sq_inv`	Evaluates the inverse of the noncentral chi-squared function.	740
`non_central_t_cdf`	Evaluates the noncentral Student's *t* distribution function.	754
`non_central_t_inv_cdf`	Evaluates the inverse of the noncentral Student's *t* distribution function.	757
`nonlinear_optimization`	Fits a nonlinear regression model using Powell's algorithm.	159
`nonlinear_regression`	Fits a nonlinear regression model.	149
`nonparam_hazard_rate`	Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.	703

Function	Purpose Statement	Page
`normal_cdf`	Evaluates the standard normal (Gaussian) distribution function.	748
`normal_inverse_cdf`	Evaluates the inverse of the standard normal (Gaussian) distribution function.	750
`normal_one_sample`	Computes statistics for mean and variance inferences using a sample from a normal population.	7
`normal_two_sample`	Computes statistics for mean and variance inferences using samples from two normal population.	11
`normality_test`	Performs a test for normality.	490
`output_file`	Sets the output file or the error message output file.	874
`page`	Sets or retrieves the page width or length.	867
`partial_autocorrelation`	Computes the sample partial autocorrelation function of a stationary time series.	560
`partial_covariances`	Computes partial covariances or partial correlations from the covariance or correlation matrix.	193
`permute_matrix`	Permutes the rows or columns of a matrix.	898
`permute_vector`	Rearranges the elements of a vector as specified by a permutation.	897
`poisson_cdf`	Evaluates the Poisson distribution function.	726
`poisson_pdf`	Evaluates the Poisson probability function.	728
`poly_prediction`	Computes predicted values, confidence intervals, and diagnostics after fitting a polynomial regression model.	140
`poly_regression`	Performs a polynomial least-squares regression.	132
`pooled_covariances`	Computes a pooled variance-covariance from the observations.	198
`principal_components`	Computes principal components.	603
`prop_hazard_gen_lin`	Analyzes time event data via the proportional hazards model.	660
`random_arma`	Generates pseudorandom ARMA process numbers.	831
`random_beta`	Generates pseudorandom numbers from a beta distribution.	786
`random_binomial`	Generates pseudorandom binomial numbers.	765
`random_cauchy`	Generates pseudorandom numbers from a Cauchy distribution.	788
`random_chi_squared`	Generates pseudorandom numbers from a chi-squared distribution.	789

Function	Purpose Statement	Page
`random_exponential`	Generates pseudorandom numbers from a standard exponential distribution.	791
`random_exponential_mix`	Generates pseudorandom mixed numbers from a standard exponential distribution.	792
`random_gamma`	Generates pseudorandom numbers from a standard gamma distribution.	794
`random_general_continuous`	Generates pseudorandom numbers from a general continuous distribution.	810
`random_general_discrete`	Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.	777
`random_geometric`	Generates pseudorandom numbers from a geometric distribution.	766
`random_GFSR_table_get`	Retrieves the current table used in the GFSR generator.	853
`random_GFSR_table_set`	Sets the current table used in the GFSR generator.	853
`random_hypergeometric`	Generates pseudorandom numbers from a hypergeometric distribution.	768
`random_logarithmic`	Generates pseudorandom numbers from a logarithmic distribution.	770
`random_lognormal`	Generates pseudorandom numbers from a lognormal distribution.	796
`random_multinomial`	Generates pseudorandom numbers from a multinomial distribution.	821
`random_mvar_from_data`	Generates pseudorandom numbers from a multivariate distribution determined from a given sample.	819
`random_neg_binomial`	Generates pseudorandom numbers from a negative binomial distribution.	772
`random_normal`	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.	798
`random_normal_multivariate`	Generates pseudorandom numbers from a multivariate normal distribution.	815
`random_npp`	Generates pseudorandom numbers from a nonhomogeneous Poisson process.	835
`random_option`	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.	845
`random_option_get`	Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.	846

Function	Purpose Statement	Page
`random_order_normal`	Generates pseudorandom order statistics from a standard normal distribution.	827
`random_order_uniform`	Generates pseudorandom order statistics from a uniform (0, 1) distribution	829
`random_orthogonal_matrix`	Generates a pseudorandom orthogonal matrix or a correlation matrix.	816
`random_permutation`	Generates a pseudorandom permutation.	839
`random_poisson`	Generates pseudorandom numbers from a Poisson distribution.	774
`random_sample`	Generates a simple pseudorandom sample from a finite population.	842
`random_sample_indices`	Generates a simple pseudorandom sample of indices.	840
`random_seed_get`	Retrieves the current value of the seed used in the IMSL random number generators.	847
`random_seed_set`	Initializes a random seed for use in the IMSL random number generators.	850
`random_sphere`	Generates pseudorandom points on a unit circle or `K`-dimensional sphere.	823
`random_stable`	Sets up a table to generate pseudorandom numbers from a general discrete distribution.	800
`random_student_t`	Generates pseudorandom Student's *t*.	802
`random_substream_seed_get`	Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.	848
`random_table_get`	Retrieves the current table used in the shuffled generator.	852
`random_table_set`	Sets the current table used in the shuffled generator.	851
`random_table_twoway`	Generates a pseudorandom two-way table.	825
`random_triangular`	Generates pseudorandom numbers from a triangular distribution.	803
`random_uniform`	Generates pseudorandom numbers from a uniform (0, 1) distribution.	804
`random_uniform_discrete`	Generates pseudorandom numbers from a discrete uniform distribution.	775
`random_von_mises`	Generates pseudorandom numbers from a von Mises distribution.	806

Function	Purpose Statement	Page
`random_weibull`	Generates pseudorandom numbers from a Weibull distribution.	808
`randomness_test`	Performs a test for randomness.	505
`ranks`	Computes the ranks, normal scores, or exponential scores for a vector of observations.	36
`rcbd_factorial`	Analyzes data from balanced and unbalanced randomized complete-block experiments.	279
`regression`	Fits a multiple linear regression model using least squares.	64
`regression_prediction`	Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.	85
`regression_selection`	Selects the best multiple linear regression models.	112
`regression_stepwise`	Builds multiple linear regression models using forward selection, backward selection or stepwise selection.	123
`regression_summary`	Produces summary statistics for a regression model given the information from the fit.	77
`regressors_for_glm`	Generates regressors for a general linear model.	56
`robust_covariances`	Computes a robust estimate of a covariance matrix and mean vector.	204
`sign_test`	Performs a sign test.	442
`simple_statistics`	Computes basic univariate statistics.	2
`sort_data`	Sorts observations by specified keys, with option to tally cases into a multi-way frequency table.	27
`split_plot`	Analyzes a wide variety of split-plot experiments with fixed, mixed or random factors.	316
`split_split_plot`	Analyzes data from split-split-plot experiments.	329
`strip_plot`	Analyzes data from strip-plot experiments.	345
`strip_split_plot`	Analyzes data from strip-split-plot experiments.	355
`survival_estimates`	Estimates using various parametric models.	697
`survival_glm`	Analyzes survival data using a generalized linear model.	673
`t_cdf`	Evaluates the Student's *t* distribution function.	751
`t_inverse_cdf`	Evaluates the inverse of the Student's *t* distribution function.	753
`table_oneway`	Tallies observations into one-way frequency table.	18
`table_twoway`	Tallies observations into a two-way frequency table.	22

Function	Purpose Statement	Page
`tie_statistics`	Computes tie statistics for a sample of observations.	458
`version`	Returns integer information describing the version of the library, license number, operating system, and compiler.	878
`wilcoxon_rank_sum`	Performs a Wilcoxon rank sum test.	460
`wilcoxon_sign_rank`	Performs a Wilcoxon sign rank test.	445
`write_matrix`	Prints a rectangular matrix (or vector) stored in contiguous memory locations.	861
`write_options`	Sets or retrieves an option for printing a matrix.	868
`yates`	Estimates missing observations in designed experiments using Yate's method.	390

Index