Visual Numerics®

IMSL™

C Numerical Library

User's Guide

VOLUME 2 of 2: C Stat Library™

VERSION 6.0

IMSL™ C Numerical Library Version 6.0

Volume 2 of 2: C Stat Library User's Guide

Trusted for Over 30 Years

Visual Numerics, Inc. United States
Corporate Headquarters
12657 Alcosta Boulevard, Suite 450
San Ramon, CA 94583

PHONE: 925.415.8300
FAX: 925.415.9500
e-mail: info@vni.com

Westminster, Colorado
10955 Westmoor Drive, Suite 400
Westminster, CO, 80021

PHONE: 303.379.3040
FAX: 303.379.2140
e-mail: info@vni.com

Houston, Texas
2500 Wilcrest, Suite 200
Houston, TX 77042

PHONE: 713.784.3131
FAX: 713.781.9260
e-mail: info@vni.com

Visual Numerics S. A. de C. V.
Florencia 57 Piso 10-01
Col. Juarez
Mexico D. F. C. P. 06600
MEXICO

PHONE: +52-55-5514 9730 or 9628
FAX: +52-55-5514-5880
e-mail:avadillo@mail.internet.com.mx

Visual Numerics
International Ltd.
SoanePoint
6-8 Market Place
Reading, Berkshire RG1 2EG
UNITED KINGDOM

PHONE: +44 118.925.5910
FAX: +44 118.925.5912
e-mail: info@vniuk.co.uk

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Road
Taipei, Taiwan 110
ROC
PHONE: +88 622-727-2255
FAX: +88 622-727-6798
e-mail: info@vni.com.tw

Visual Numerics International GmbH
Zettachring 10
D-70567Stuttgart
GERMANY

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

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-1, MAPO-DONG, MAPO-GU
SEOUL, 121-050
KOREA SOUTH

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

Visual Numerics SARL
Immeuble le Wilson 1
70, avenue du General de Gaulle
92058 Paris La Defense, Cedex
FRANCE

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

Visual Numerics Japan, Inc.
GOBANCHO HIKARI BLDG. 4TH Floor
14 GOBAN-CHO CHIYODA-KU
TOKYO, JAPAN 102-0076

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

IMSL Fortran, C, and Java
Application Development Tools

Table of Contents

Introduction ix

IMSL C Stat Library ix
Getting Started ix
  ANSI C vs. Non-ANSI C ix
  The imsls.h File x
Thread Safe Usage x
  Signal Handling x
  Routines that Produce Output xi
  Input Arguments xi
Matrix Storage Modes xi
  General Mode xii
  Rectangular Mode xii
  Symmetric Mode xii
Memory Allocation for Output Arrays xii
Finding the Right Function xii
Organization of the Documentation xiii
Naming Conventions xiv
Error Handling, Underflow, and Overflow xiv
Printing Results xv
Missing Values xv
Passing Data to User-Supplied Functions xv

Chapter 1: Basic Statistics 1

Routines 1
Usage Notes 1
simple_statistics 1
normal_one_sample 7
normal_two_sample 11
table_oneway 17
table_twoway 22
sort_data 26
ranks 34

Chapter 2: Regression 43

Routines 43
Usage Notes 44
  Simple and Multiple Linear Regression 44

No Intercept Model 44
Variable Selection 45
Polynomial Model 45
Specification of X for the General Linear Model 45
Functions for Fitting the Model 48
Linear Dependence and the *R* Matrix 48
Nonlinear Regression Model 49
Weighted Least Squares 50
Summary Statistics 50
Tests for Lack-of-Fit 52
Transformations 54
Alternatives to Least Squares 54
Missing Values 55
regressors_for_glm 55
regression 64
regression_summary 76
regression_prediction 84
hypothesis_partial 95
hypothesis_scph 100
hypothesis_test 105
regression_selection 112
regression_stepwise 122
poly_regression 130
poly_prediction 137
nonlinear_regression 147
nonlinear_optimization 157
Lnorm_regression 166

Chapter 3: Correlation and Covariance 185

Routines 185
Usage Notes 185
covariances 185
partial_covariances 192
pooled_covariances 197
robust_covariances 203

Chapter 4: Analysis of Variance and Designed Experiments 215

Routines 215
Usage Notes 216
Completely Randomized Experiments 216
Factorial Experiments 217
Blocking 218
Multiple Locations 221
Split-Plot Designs – Nesting and Restricted Randomization 221
Strip-Plot Designs 223
Split-Split Plot and Strip-Split Plot Experiments 225
Validating Key Assumptions in Anova 226
Missing Observations 227

anova_oneway....228
anova_factorial....237
anova_nested....245
anova_balanced....254
crd_factorial....266
rcbd_factorial....277
latin_square....287
lattice....296
split_plot....314
split_split_plot....326
strip_plot....342
strip_split_plot....353
homogeneity....376
multiple_comparisons....383
yates....388

Chapter 5: Categorical and Discrete Data Analysis 401

Routines....401
Usage Notes....401
contingency_table....402
exact_enumeration....414
exact_network....416
categorical_glm....422

Chapter 6: Nonparametric Statistics 437

Routines....437
Usage Notes....437
sign_test....438
wilcoxon_sign_rank....441
noether_cyclical_trend....444
cox_stuart_trends_test....448
tie_statistics....453
wilcoxon_rank_sum....455
kruskal_wallis_test....459
friedmans_test....462
cochran_q_test....466
k_trends_test....469

Chapter 7: Tests of Goodness of Fit 475

Routines....475
Usage Notes....475
chi_squared_test....475
normality_test....483
kolmogorov_one....487
kolmogorov_two....490
multivar_normality_test....493
randomness_test....497

Chapter 8: Time Series and Forecasting 509

Routines 509
Usage Notes 510
arma....511
max_arma....521
arma_forecast....527
auto_uni_ar....532
ts_outlier_identification 537
ts_outlier_forecast....547
auto_arima....555
difference 572
seasonal_fit....576
box_cox_transform 584
autocorrelation 588
crosscorrelation 593
multi_crosscorrelation....599
partial_autocorrelation 608
lack_of_fit 611
estimate_missing....614
garch....621
kalman....626

Chapter 9: Multivariate Analysis 639

Routines 639
Usage Notes 639
    Cluster Analysis....639
    Principal Components....640
    Factor Analysis 640
dissimilarities 641
cluster_hierarchical 645
cluster_number....649
cluster_k_means....653
principal_components 657
factor_analysis 663
discriminant_analysis....682

Chapter 10: Survival and Reliability Analysis 707

Routines 707
Usage Notes 707
kaplan_meier_estimates 708
prop_hazards_gen_lin 713
survival_glm....727
survival_estimates....750
nonparam_hazard_rate 756
life_tables....764

Chapter 11: Probability Distribution Functions and Inverses 771

Routines 771
Usage Notes 772
  Continuous Distributions 773
binomial_cdf 774
binomial_pdf 775
hypergeometric_cdf 777
hypergeometric_pdf 778
poisson_cdf 779
poisson_pdf 781
beta_cdf 783
beta_inverse_cdf 785
bivariate_normal_cdf 786
chi_squared_cdf 788
chi_squared_inverse_cdf 789
non_central_chi_sq 791
non_central_chi_sq_inv 793
F_cdf 794
F_inverse_cdf 796
gamma_cdf 798
gamma_inverse_cdf 799
normal_cdf 801
normal_inverse_cdf 802
t_cdf 804
t_inverse_cdf 805
non_central_t_cdf 807
non_central_t_inv_cdf 809

Chapter 12: Random Number Generation 811

Routines 811
Usage Notes 813
  Overview of Random Number Generation 813
  Basic Uniform Generators 814
  The Multiplicative Congruential Generators 814
  Shuffled Generators 814
  The Generalized Feedback Shift Register Generator 815
  Setting the Seed 815
  Timing Considerations 815
  Distributions Other than the Uniform 815
  Tests 816
random_binomial 816
random_geometric 818
random_hypergeometric 819
random_logarithmic 822
random_neg_binomial 823
random_poisson 825
random_uniform_discrete 826
random_general_discrete 828

discrete_table_setup 832
random_beta 837
random_cauchy 838
random_chi_squared 840
random_exponential 841
random_exponential_mix 843
random_gamma 845
random_lognormal 846
random_normal 848
random_stable 850
random_student_t 852
random_triangular 853
random_uniform 854
random_von_mises 856
random_weibull 857
random_general_continuous 859
continuous_table_setup 862
random_normal_multivariate 864
random_orthogonal_matrix 866
random_mvar_from_data 868
random_multinomial 871
random_sphere 873
random_table_twoway 875
random_order_normal 876
random_order_uniform 878
random_arma 880
random_npp 884
random_permutation 887
random_sample_indices 889
random_sample 890
random_option 894
random_option_get 895
random_seed_get 896
random_substream_seed_get 897
random_seed_set 899
random_table_set 900
random_table_get 900
random_GFSR_table_set 901
random_GFSR_table_get 902
random_MT32_init 905
random_MT32_table_get 905
random_MT32_table_set 907
random_MT64_init 908
random_MT64_table_get 908
random_MT64_table_set 910
faure_next_point 911

Chapter 13: Neural Networks 915

Routines 915
Usage Notes 915
    Neural Networks – An Overview 915
    Neural Networks – History and Terminology....918
    Network Applications....922
Multilayer Feedforward Neural Networks 925
    Neural Network Error Calculations 927
mlff_network....934
mlff_network_trainer 944
mlff_network_forecast....954
scale_filter....960
time_series_filter....966
time_series_class_filter....969
unsupervised_nominal_filter....973
unsupervised_ordinal_filter....976

Chapter 14: Printing Functions 981

Routines 981
write_matrix....981
page....986
write_options....987

Chapter 15: Utilities 993

Routines 993
output_file....993
version....997
error_options....998
error_code 1004
machine (integer) 1005
machine (float)....1007
data_sets....1009
mat_mul_rect 1012
permute_vector 1015
permute_matrix....1017
binomial_coefficient 1018
beta....1020
beta_incomplete....1021
log_beta....1022
gamma....1023
gamma_incomplete....1025
log_gamma....1027
ctime....1029

Reference Material 1031

User Errors....1031

What Determines Error Severity 1031
Kinds of Errors and Default Actions 1031
Errors in Lower-level Functions 1032
Functions for Error Handling 1032
Threads and Error Handling 1033
Use of Informational Error to Determine Program Action 1033
Additional Examples 1033

Product Support **1035**

Contacting Visual Numerics Support 1035

Appendix A: References **1037**

Appendix B: Alphabetical Summary of Routines **1065**

Routines 1065

Index **1079**

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;
    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, C Stat Library can be safely called from a multithreaded application. When 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' 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' internal-signal handling , see Chapter 15, "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 15, "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 15, "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 15, "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

Simple Summary Statistics

Univariate summary statistics	simple_statistics	1
Mean and variance inference for a single normal population	normal_one_sample	7
Inferences for two normal populations	normal_two_sample	11

Tabulate, Sort, and Rank

Tally observations into a one-way frequency table	table_oneway	17
Tally observations into a two-way frequency table	table_twoway	22
Sort data with options to tally cases into a multi-way frequency table	sort_data	26
Ranks, normal scores, or exponential scores	ranks	34

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 15, "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)
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

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

Skewness

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

Excess or Kurtosis

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

Minimum

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

Maximum

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

Range

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

Coefficient of Variation

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

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 × (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 equal 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`.

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

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

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:

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

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 $\mu_1 \leq \mu_2$ 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, …, `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_datasorts 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

Multivariate Linear Regression—Model Fitting

Generate regressors for a general linear model	regressors_for_glm	55
Fit a multivariate linear regression model	regression	64

Multivariate Linear Regression—Statistical Inference and Diagnostics

Produce summary statistics for a regression model	regression_summary	76
Compute predicted values, confidence intervals, and diagnostics	regression_prediction	84
Construction of a completely testable hypothesis	hypothesis_partial	95
Sums of cross products for a multivariate hypothesis	hypothesis_scph	100
Tests for the multivariate linear hypothesis	hypothesis_test	105

Variable Selection

All best regressions	regression_selection	112
Stepwise regression	regression_stepwise	122

Polynomial and Nonlinear Regression

Fit a polynomial regression model	poly_regression	130
Compute predicted values, confidence intervals, and diagnostics	poly_prediction	137
Fit a nonlinear regression model.	nonlinear_regression	147
Fit a nonlinear regression model using Powell's algorithm	nonlinear_optimization	157

Alternatives to Least Squares Regression

LAV, Lpnorm, and LMV criteria regression	Lnorm_regression	166

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

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

where the observed values of the y_i's constitute the responses or values of the dependent variable, the x_i's are the 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` 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` and `imsls_f_regression_prediction` 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`, which computes all best-subset regressions, or by `imsls_f_regression_stepwise`, 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

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

where the observed values of the y_i's constitute the responses or values of the dependent variable; the x_i's are the 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` 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` 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` 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` 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 15, "Utilities.")

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

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

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` 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`. 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` 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:$\beta_1 = \beta_2 = ... = \beta_k = 0$, versus the alternative that at least one coefficient is nonzero is given by F = 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 $\overline{y}$ is output in `anova_table`[13].

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

`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`. 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

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

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:

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

The jackknife residual is defined as

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

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:

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

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.

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

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` 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 ($p \le 1$), is given by option IMSLS_METHOD_LLP. 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 $p \ge 1$ 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. 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 15, "Utilities" (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).
`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 $a_1, a_2, \ldots, a_n$. From this classification variable, `imsls_f_regressors_for_glm` creates *n* indicator variables. For $k = 1, 2, \ldots, n$, 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

$$\texttt{n\_continuous} = 1$$

$$\texttt{n\_effects} = 7$$

$$\texttt{n\_var\_effects} = (1, 1, 2, 1, 2, 2, 3)$$

$$\texttt{indices\_effects} = (0, 1, 0, 1, 2, 0, 2, 1, 2, 0, 1, 2)$$

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_SCPE, *float* **scpe[],
IMSLS_SCPE_USER, *float* scpe_user[],
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 15, Utilities.)

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
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_SCPE`, *float* `**scpe` (Output)
The address of a pointer to an internally allocated array of size `n_dependent` x `n_dependent` containing the error (residual) sums of squares and crossproducts. `scpe` [m][n] contains the sum of crossproducts for the *m*-th and *n*-th dependent variables.

`IMSLS_SCPE_USER`, *float* `scpe[]` (Output)
Storage for array `scpe` is provided by the user. See `IMSLS_SCPE`.

`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` × 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.,

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

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

$$\overline{y}$$

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

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

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[]`,
`IMSLS_SQSS`, *float* `**sqss`,
`IMSLS_SQSS_USER`, *float* `sqss[]`,
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, …, *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" 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.

IMSLS_SQSS, *float* **sqss (Output)

Address of a pointer to an internally allocated array of size *npar* by 4, where *npar* is equal to the numbers of parameters in the model, containing in columns 1 through 4 the sequential degrees of freedom, sum of squares, *F*-statistic, and *p*-value. Each row corresponds to an effect. 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, …, *npar* – 1.

IMSLS_SQSS_USER, *float* sqss[] (Output)

Storage for sqss is provided by the user. See IMSLS_SQSS.

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`.

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 = diag (w_1, w_2, …, w_n). The leverage is defined as

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

Put D = diag (d_1, d_2, …, d_n) with $d_j = 1$ if the j-th diagonal element of R is positive and 0 otherwise. The leverage is computed as $h_i = (a^T Da)\, w_i$ 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. For a detailed discussion, see case diagnostics.

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`.

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) 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). 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*

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

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

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

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. The partially testable hypothesis

$$H_0 : {}^{\alpha_1=5}_{\alpha_2=3}$$

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`.

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` 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 × `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

$$S_H = \left(H\hat{\beta}U - G\right)^T \left(C^T D C\right)^- \left(H\hat{\beta}U - G\right)$$

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*.

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.,

$$p = \begin{cases} \text{NU} & \text{if NU} > 0 \\ \text{NDEP} & \text{otherwise} \end{cases}$$

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

$$s = \begin{cases} 1 & \text{if } p = 1 \text{ or } q = 1 \\ \sqrt{\dfrac{p^2 q^2 - 4}{p^2 + q^2 - 5}} & \text{otherwise} \end{cases}$$

and

$$m = \upsilon - \frac{(p + q - 1)}{2}$$

The F test is exact if min $(p, q) \leq 2$ (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

$$U = \operatorname{tr}\left(HE^{-1}\right) = \sum_{i=1}^{p} \lambda_i$$

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^TA, 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 15, Utilities), some results can be incorrect. This limitation arises because the possible models indicated (the model numbers 1, 2, ..., $2^{\texttt{n\_candidate}}$) 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`) 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, $i = 0, ...,$ `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

Column	Description
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 `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
```

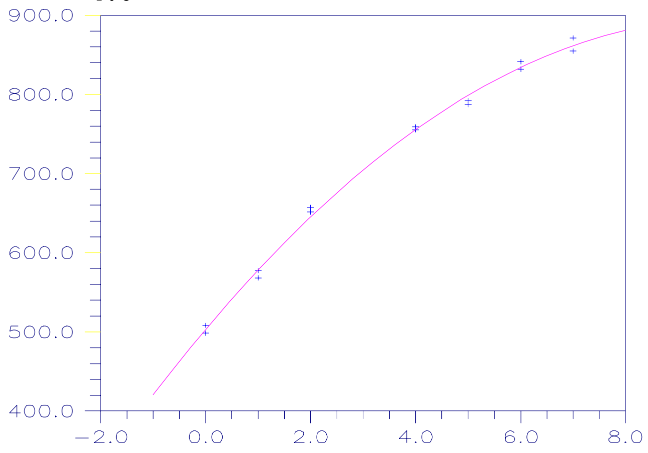


Figure 2- 1 A Polynomial Fit

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`.

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

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

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 1.

Results from function `imsls_f_poly_prediction`, 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

$$\hat{y}_i = \hat{\alpha}_0 p_0(z_i) + \hat{\alpha}_1 p_1(z_i) + \ldots + \hat{\alpha}_k p_k(z_i)$$

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^TX 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:

$$h_i = w_i \sum_{j=0}^{k} d_j^{-1} p_j^2(z_i)$$

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" 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

$$\frac{|g_j| * \max\left(|\theta_j|, 1/t_j\right)}{\frac{1}{2}\|F(\theta)\|_2^2}$$

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

$$\frac{|\theta_j - \theta'_j|}{\max\left(|\theta_j|, 1/t_j\right)}$$

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 = $\max(10^{-20},\varepsilon^2)$, $\max(10^{-40}, \varepsilon^2)$ 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

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

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:

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

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

   $$f\left(x;\theta\right) = \theta_1 e^{\theta_2 x}$$

   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_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.

Fatal Errors

`IMSLS_TOO_MANY_FCN_EVAL`	Maximum number of function evaluations exceeded.

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:
$a_{i1}\,\theta_1 + \ldots + a_{ij}\,\theta_j = b_i$ for $i = 1$, n_equality and $j = 1$, n_parameter, and
$a_{k1}\,\theta_1 + \ldots + a_{kj}\,\theta_j \le b_k$ 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.
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

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

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

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

$$(d^k)^T \nabla f(\theta^k + \alpha^k d^k) \geq 0.7\, (d^k)^T \nabla f(\theta^k)$$

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

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

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

$$y_i = \sin\left(\theta_0 x_i\right) + \varepsilon_i$$

```
#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

Function `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

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \ldots + \hat{\beta}_k x_k$$

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_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

$$\text{asy.}\operatorname{var}(\hat{\beta}) = \lambda^2 (R^T R)^{-1}$$

where R is from the QR decomposition of the matrix of regressors (output in `R-Matrix`) ere an estimate of $\lambda 2$ 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 function `imsls_f_regression`. 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

$$[V(^k+1)]1/2X$$

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:

$$\delta_i^{(k+1)} = \begin{cases} 0 & \text{if } e_i^{(k+1)} = e_i^{(k)} = 0 \\ \left|e_i^{(k+1)} - e_i^{(k)}\right| / \max(\left|e_i^{(k)}\right|, \left|e_i^{(k+1)}\right|, s) & \text{otherwise} \end{cases}$$

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

$$\mathit{Minimize} \sum_{i=0}^{n-1} (e_i^2 + c^2)^{p/2}$$

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 $[V(^{k}+1)]1/2X$. 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 $[R(^{k})]\mathrm{T} R(^{k}) d = X^{T} t$.

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

$z_i^{(k+1)}$ by $f_i\sqrt{w_i}z_i^{(k+1)}$, $v_i^{(k+1)}$ by $f_iw_iv_i^{(k+1)}$, and $t_i^{(k+1)}$ 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 λ2 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 λ2 depends on *p*.

For *p* = 1, the estimator for λ2 is given by (McKean and Schrader 1987)

$$\hat{\lambda}^2 = \left[\frac{\sqrt{\text{DFE}}\left(\tilde{e}_{(\text{DFE}-k+1)} - \tilde{e}_{(k)}\right)}{2z_{0.975}}\right]^2$$

with

$$k = \frac{\text{DFE}+k}{2} - z_{0.975}\sqrt{\frac{\text{DFE}}{4}}$$

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

$$DFE = \sum\nolimits_{i=0}^{n-1} f_i - \text{rank}$$

For *p* = 2, the estimator of λ2 is the customary least-squares estimator given by

$$s^2 = \frac{SSE}{DFE} = \frac{\sum_{i=0}^{n-1} f_iw_i(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1} f_i - \text{rank}}$$

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
```

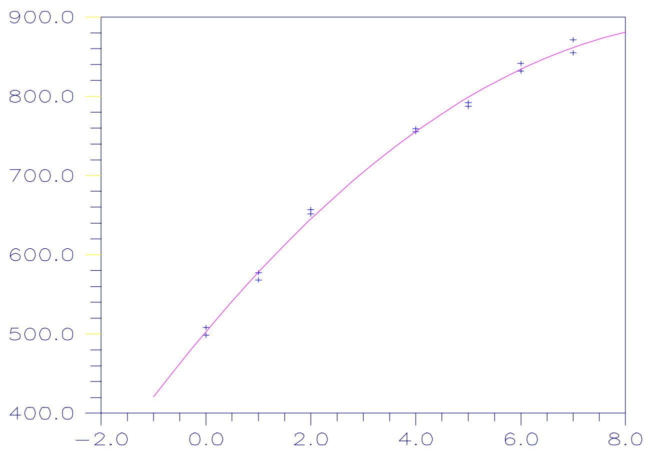


Figure 2- 2 Least Squares and Least Absolute Value Fitted Lines

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
```

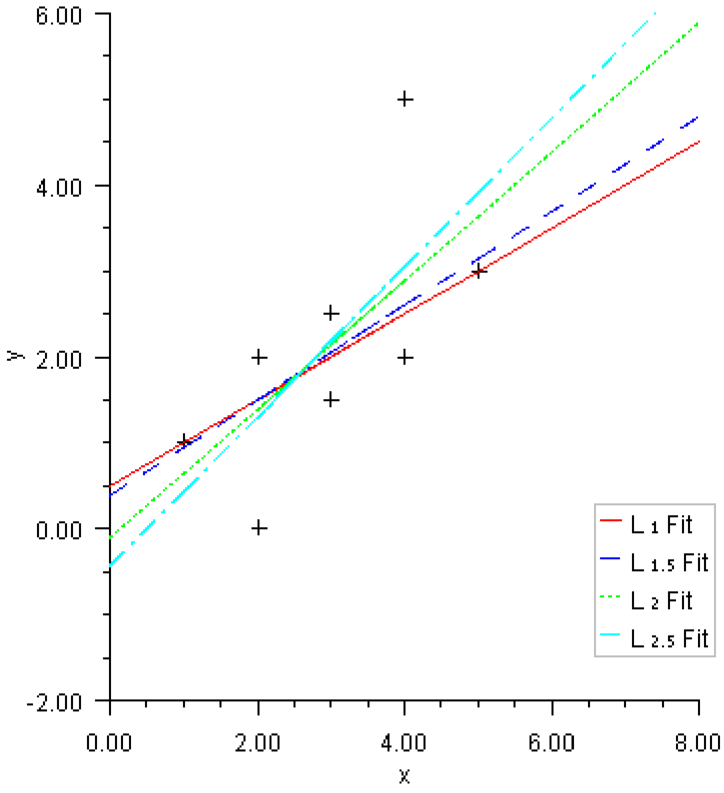


Figure 2- 3 Various L_p *Fitted Lines*

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
```

5.

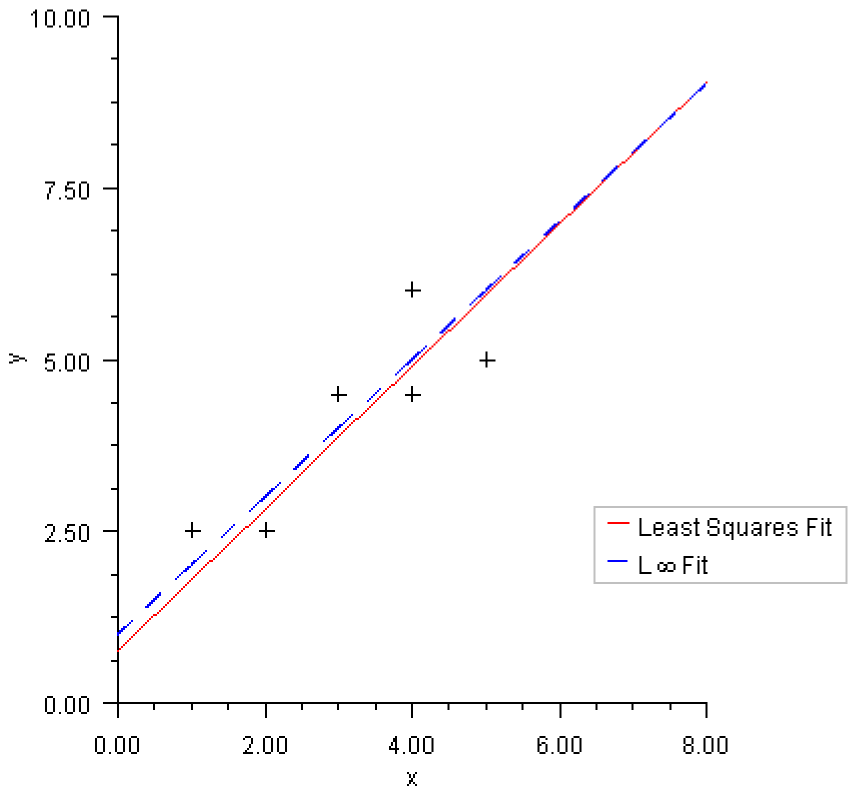


Figure 2- 4 Least Squares and Least Maximum Value Fitted Lines

Chapter 3: Correlation and Covariance

Routines

Variances, Covariances, and Correlations

Variance-covariance or correlation matrix	covariances	185
Partial correlations and covariances	partial_covariances	192
Pooled covariance matrix	pooled_covariances	197
Robust estimate of covariance matrix	robust_covariances	203

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 15, "Utilities"). 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

$$\overline{x_{i\bullet}} = \frac{T_i}{\sum_j w_i f_i}$$

$$S = \frac{1}{\sum_{ij} f_{ij} - g} \sum_{i,j} w_{ij} f_{ij} \left(x_{ij} - \overline{x}_{i\bullet} \right) \left(x_{ij} - \overline{x}_{ii\bullet} \right)^T$$

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:

$$u(r) = \begin{cases} \dfrac{a^2}{r^2} & r < a \\ 1 & a \le r \le b \\ \dfrac{b^2}{r^2} & r > b \end{cases}$$

$$w(r) = \min\left(1, \frac{c}{r}\right)$$

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

General Analysis of Variance

One-way analysis of variance	anova_oneway	228
Analysis of variance for fixed effects, balanced factorial designs	anova_factorial	237
Nested random effects analysis of variance	anova_nested	245
Analysis of variance for balanced fixed, random, or mixed models	anova_balanced	254

Designed Experiments

Analysis of balanced and unbalanced completely randomized factorial experiments	crd_factorial	266
Analysis of balanced and unbalanced randomized complete block factorial experiments	rcbd_factorial	277
Analysis of latin-square experiments	latin_square	287
Analysis of balanced and partially-balanced data from lattice experiments	lattice	296
Analysis of split-plot experiments	split_plot	314
Analysis of split-split-plot experiments	split_split_plot	326
Analysis of strip-plot experiments	strip_plot	342
Analysis of strip-split-plot experiments	strip_split_plot	353

Utilities

Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance	homogeneity	376
Multiple comparisons of means	multiple_comparisons	383
Yates' method for estimating missing observations in designed experiments	yates	388

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` and `imsls_f_anova_factorial` 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` 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 15, "Utilities " for a description. Functions `imsls_f_anova_factorial`, `imsls_f_anova_nested` and `imsls_f_anova_balanced` 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` and `imsls_f_crd_factorial`.

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` conducts multiple comparisons of treatment functions; whereas `imsls_f_crd_factorial` requires users to make a call to `imsls_f_multiple_comparisons` to compare treatment means.
2. `imsls_f_crd_factorial` can analyze treatments with a factorial treatment structure; whereas `imsls_f_anova_oneway` does not analyze factorial structures.
3. `imsls_f_crd_factorial` 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: 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 four different treatments. Three functions, `imsls_f_crd_factorial`, `imsls_f_rcbd_factorial` and `imsls_f_anova_factorial` 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` and `imsls_f_rcbd_factorial` support analysis of a factorial experiment with missing values and multiple locations. The function `imsls_f_anova_factorial` 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`, `imsls_f_lattice`, and `imsls_f_latin_square`. 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` 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

Source	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, imsls_f_rcbd_factorial, imsls_f_lattice, imsls_f_latin_square, imsls_f_split_plot, imsls_f_split_split_plot, imsls_f_strip_plot, imsls_f_strip_split_plot. 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

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.

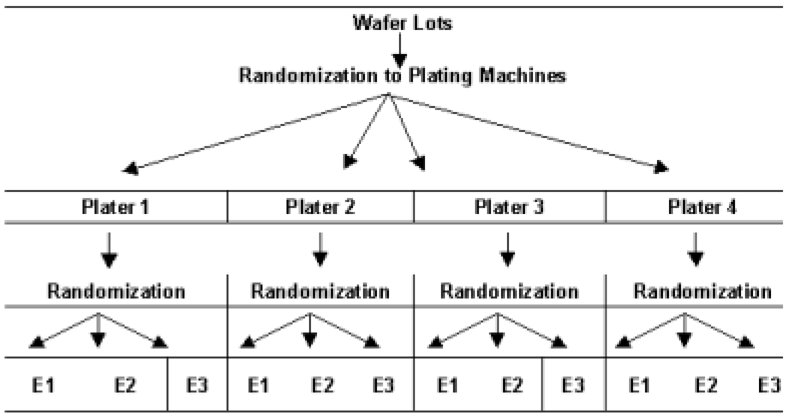


Figure 1 - Split-Plot Randomization

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.

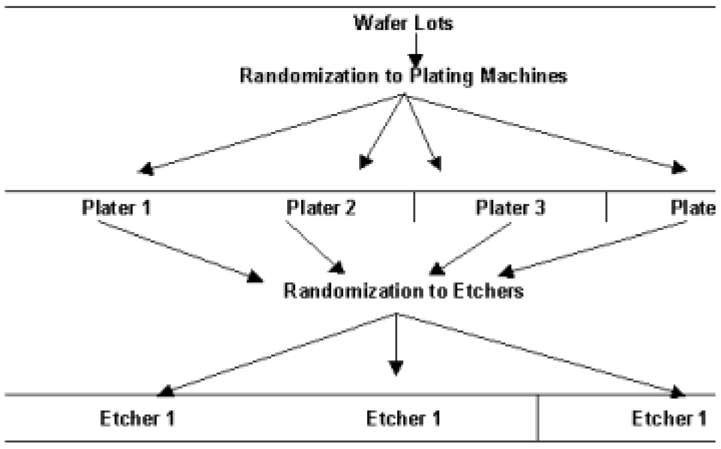


Figure 2 - Strip-Plot Semiconductor Experiment

Strip-plot experiments can be analyzed using `imsls_f_strip_plot`. 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` and `imsls_f_strip_split_plot` 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`. 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`, 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

$$\binom{\texttt{ngroups}}{2} \times 5$$

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

$$y_{ij} = \mu_i + \varepsilon_{ij} \quad i = 1, 2, \ldots, k; j = 1, 2, \ldots, n_i$$

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

$$\bar{y}_i - \bar{y}_j \pm q_{1-\alpha;k,v}\sqrt{\frac{s^2}{n}}$$

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 − α) 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

$$\bar{y}_i - \bar{y}_j \pm \sqrt{(k-1)F_{1-\alpha;k-1,v}\left(\frac{s^2}{n_i} + \frac{s^2}{n_j}\right)}$$

where $F_{1-\alpha;(k-1),v}$ 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:

$$\bar{y}_i - \bar{y}_j \pm t_{1-\frac{\alpha}{2};v\sqrt{\frac{s^2}{n_i}+\frac{s^2}{n_j}}}$$

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

Element	Analysis of Variance Statistics
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_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 Peter W.M. 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 Peter W.M. 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

$$y_{ijk} = \mu + \alpha_i + \beta_{ij} + e_{ijk} \qquad i = 1, 2, 3, 4; j = 1, 2, 3; k = 1, 2$$

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 $N(0, \sigma2)$. 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

Element	**Analysis of Variance Statistics**
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]	
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

$$\textstyle\sum_{i=1}^{a} \alpha_i = 0$$

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

$$\textstyle\sum_{i=1}^{a} c_{ij} = 0 \text{ for } j = 1, 2, \ldots, b$$

and the e_{ijk}'s are errors identically and independently distributed $N(0, \sigma2)$. 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

$$\mathrm{var}(y_{ijk}) = \sigma_B^2 + \frac{a-1}{a}\sigma_{AB}^2 + \sigma^2$$

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,

$$\mathrm{var}\left(y_{ijk}\right) = \tilde{\sigma}_B^2 + \tilde{\sigma}_{AB}^2 + \sigma^2$$

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

$$\sigma_B^2 = \tilde{\sigma}_B^2 + \frac{1}{a}\tilde{\sigma}_{AB}^2$$
$$\sigma_{AB}^2 = \tilde{\sigma}_{AB}^2$$

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

$$y_{ijk} = \mu + \alpha_i + b_j + c_{ij} + e_{ijk} \quad i = 1, 2, 3, 4; j = 1, 2, 3, 4; k = 1, 2$$

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

$$\textstyle\sum_{i=1}^{2} \alpha_i = 0$$

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 $\texttt{anova\_table}_{i,j}$ 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, 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	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:

$$\mathrm{CV} = \frac{100 \cdot \sqrt{\mathrm{MS}_{residual}}}{\texttt{grand\_mean}}$$

`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

$$F = \frac{\mathrm{MS}_{effect}}{\mathrm{MS}_{residual}}, \text{ when } \texttt{n\_locations} = 1.$$

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:

$$\mathrm{SS}_{error} = \mathrm{SS}_{A\times Locations} + \mathrm{SS}_{B\times Locations} + \mathrm{SS}_{A\times B\times Locations}$$

$$\mathrm{df}_{error} = \mathrm{df}_{A\times Locations} + \mathrm{df}_{B\times Locations} + \mathrm{df}_{A\times B\times Locations}$$

$$\mathrm{MS}_{error} = \frac{\mathrm{SS}_{error}}{\mathrm{df}_{error}}$$

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

j	**anova_table$_{i,j}$ = anova_table[i*6+j]**
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, –3 and so on.

Source Identifier	**ANOVA Source**
-1	Main Effects †
-2	Two-Way Interactions ‡
-3	Three-Way Interactions ‡
.	.
.	.
.	.
-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:

$$CV = \frac{100 \cdot \sqrt{MS_{residual}}}{\texttt{grand\_mean}} \cdot$$

`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

$$\text{n\_levels}[0] \times \text{n\_levels}[1] \times \cdots \times \text{n\_levels}[\text{n\_factors} - 1]$$

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

$$F_{effect} = \frac{MS_{effect}}{MS_{residual}}$$, when `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:

$$\text{Std Err(A)} = \sqrt{\frac{2\cdot MS_{residual}}{N_A}}$$
$$\text{Std Err(B)} = \sqrt{\frac{2\cdot MS_{residual}}{N_B}},$$
$$\text{Std Err}(\text{A}\times\text{B}) = \sqrt{\frac{2\cdot MS_{residual}}{N_{A\times B}}}$$

where

N_A, N_B 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
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	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[$(i$-$1)\times42+(j-1)\times6+(k$-$1)$] 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 *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_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 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=$t\sum_{i=1}^{t}(\overline{y}_{i.}-\overline{y}_{..})^2$	MSR
COLUMNS	$t-1$	SSC=$t\sum_{j=1}^{t}(\overline{y}_{.j}-\overline{y}_{..})^2$	MSC
TREATMENTS	$t-1$	SST=$t\sum_{k=1}^{t}(\overline{y}_{k}-\overline{y}_{..})^2$	MST
ERROR	$(t-1)(t-2)$	SSE=SSTot-SSR-SSC-SST	MSE
TOTAL	t^2-1	SSTot=$\sum_{i=1}^{t}\sum_{j=1}^{t}\left(y_{ij}-\overline{y}_{..}\right)^2$	

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	$r-1$	$\text{SSL}=t^2\sum_{l=1}^{r}(\bar{y}_{l..}-\bar{y}_{...})^2$	MSL
ROWS	$r(t-1)$	$\text{SSR}=t\sum_{l=1}^{r}\sum_{i=1}^{t}(\bar{y}_{li.}-\bar{y}_{l..})^2$	MSR
COLUMNS	$r(t-1)$	$\text{SSC}=t\sum_{l=1}^{r}\sum_{j=1}^{t}(\bar{y}_{l.j}-\bar{y}_{l..})^2$	MSC
TREATMENTS	$t-1$	$\text{SST}=r\cdot t\sum_{k=1}^{t}(\bar{y}_{k}-\bar{y}_{...})^2$	MST
LOCATIONS X TREATMENTS	$(r-1)(t-1)$	SSLT by difference	MSLT

SOURCE	DF	Sum of Squares	Mean Squares
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 four 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()`.

```
#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:

$$\texttt{n} = \texttt{n\_locations} \times \texttt{t} \times \texttt{r} \quad \textit{where}$$

$$t = \texttt{n\_treatments} \text{ and } r = \texttt{n\_reps}$$

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-1)×42+(j-1)×6 + (k-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	$r \cdot t - 1$	SSTot	

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	$p \cdot r \cdot t - 1$	SSTot	

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, `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 $j > 0$, 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

Split-Plot Experiments – Split-Plot B Nested within Whole-Plot A

CRD			
A3B2	A1B3	A4B1	A4B3
A2B3	A1B1	A3B2	A1B2
A2B2	A3B1	A2B1	A4B2

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:

$$n = \sum_{i=1}^{\texttt{n\_locations}} \left(\texttt{n\_whole} \times \texttt{n\_split} \times \texttt{n\_sub} \times \texttt{n\_blocks}_i\right)$$

where $\texttt{n\_block}_i$ 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, $\texttt{anova\_table}_{i,0}$ = `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	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†
-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 *k*th sub-plot within 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 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

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, `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	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 *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_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 Table 2 below. In strip-plot experiments, Factors A and B are completely crossed, see Table 1 below. 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

Table 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

Table 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, `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	BLOCKs WITHIN LOCATION
-3	STRIP-PLOT A
-4	LOCATION × STRIP-PLOT A †
-5	STRIP-PLOT A ERROR
-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 $j > 0$, $\text{treatment\_means}_{i,j}$ = treatment_means[(i-1)*n_split +(j-1)] contains the mean of the observations, averaged over all locations, blocks and replicates, for the ith level of the strip-plot and the jth 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

Table 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 Table 2 below. 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

Table 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 Table 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

Table 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 Table 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

Table 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 ith 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:

$$F_0 = \frac{\sum_{i=1}^{t} n_i \left(\overline{z}_{i.} - \overline{z}_{..}\right)^2 /(t-1)}{\sum_{i=1}^{t}\sum_{j=1}^{n_i}\left(z_{ij} - \overline{z}_{i.}\right)^2 /(N-t)},$$

where

$$z_{ij} = | x_{ij} - \overline{x}_{i.} | ,$$

x_{ij} is the jth observation from the ith treatment and $\overline{x}_{i.}$ is the mean for the ith 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        =      0.944
Bartlett's test statistic =      2.257



 *** Levene's Test ***

Levene's p-value        =      0.994
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`.

Design	Description
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:

$$\hat{x}_{ij} = \frac{t \cdot y_{.j} + r \cdot y_{i.} - y_{..}}{(r-1)(t-1)}$$, 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:

$$Y = \frac{r \cdot W + s \cdot x_{ij.} - x_{i..}}{(r-1)(s-1)}$$, 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

Statistics in the Two-Way Contingency Table

Two-way contingency table analysis	contingency_table	402
Exact probabilities in an r × c table; total enumeration	exact_enumeration	414
Exact probabilities in an r × c table	exact_network	416

Generalized Categorical Models

Generalized linear models	categorical_glm	422

Usage Notes

Routine `imsls_f_contingency_table` 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`, but this routine uses the total enumeration algorithm and, thus, often uses orders of magnitude more computer time than `imsls_f_exact_network`, which computes the same probabilities by use of the network algorithm (but can still be quite expensive).

The routine `imsls_f_categorical_glm` 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
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:

$$\text{phi, } \phi = \sqrt{\chi^2/n}$$

$$\text{contingency coefficient, } P = \sqrt{\chi^2/\left(n+\chi^2\right)}$$

$$\text{Cramer's } V,\ V = \sqrt{\chi^2/\left(n\min\left(r,c\right)\right)}$$

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

$$\lambda_{c|r} = \frac{\left(1 - p_{\bullet m}\right) - (1 - \sum_i p_{im})}{1 - p_{\bullet m}}$$

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 λ^* 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:

$$n/2 - (\sum_i x^2_{i\bullet})/(2n)$$

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:

$$q_j = x_{\bullet j}/2 - (\sum_i x^2_{ij})/(2x_{i\bullet})$$

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:

$$U_{r|c} = \frac{\sum_{i,j} x_{ij} \log\left(x_{i\bullet} x_{\bullet j} / n x_{ij}\right)}{\sum_{i} x_{i\bullet} \log\left(x_{i\bullet} / n\right)}$$

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 (6`factor1` + 2`factor2`). 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 $f_{i\bullet}$ 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.

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,
        IMSLS_EFFECTS, int n_effects, int n_var_effects[],
```

[*]Relationship between the parameter, θ or λ, and a linear model of the explanatory variables, $X\beta$.

```
        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

$$\sum_{i=0}^{\text{n\_class}-1} \texttt{n\_class\_values}[i]$$

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	$f(y) = \binom{S+y-1}{y-1} \theta^S (1-\theta)^y$	$\theta = \frac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$
2	$f(y) = (1-\theta)^y / (y \ln \theta)$	$\theta = \frac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$
3	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$	$\theta = \frac{\exp(\omega+\eta)}{1+\exp(\omega+\eta)}$
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

$$\Delta\beta = (\sum \left| \Psi''(\gamma_i) \right| x_i x_i^T)^{-1} \sum \Psi'(\gamma_i) x_i$$

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 $l_i(\gamma_i)$ denote the log-likelihood of the *i*-th observation evaluated at γ_i. Then, the standardized residual is computed as

$$r_i = \frac{l_i'(\hat{\gamma}_i)}{\sqrt{l_i''(\hat{\gamma}_i)}}$$

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        5.1876      -11.7118        0.0000
     34.2985        2.9164       11.7607        0.0000

Log likelihood    -18.778181

          Coefficient statistics for model 4
coefficients          s.e            z           p
    -34.9441        2.6412      -13.2305        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        3.2489      -12.1930        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.

Age		Heart Valve Type	
		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      0.3456     -15.6837      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

One sample tests - Nonparametric Statistics

Sign test	sign_test	438
Wilcoxon rank sum test	wilcoxon_sign_rank	441
Noehter’s test for cyclical trend	noether_cyclical_trend	444
Cox and Stuarts’ sign test for trends in location and dispersion	cox_stuart_trends_test	448
Tie statistics	tie_statistics	453

Two or more samples

Wilcoxon’s rank sum test	wilcoxon_rank_sum	455
Kruskal-Wallis test	kruskal_wallis_test	459
Friedman’s test	friedmans_test	462
Cochran's Q test	cochran_q_test	466
K-sample trends test	k_trends_test	469

Usage Notes

Much of what is considered nonparametrik_trends_testc 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`, 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 $j = 1, 2, \ldots,$ `n_observations`.

`IMSLS_N_ZERO_DEVIATIONS`, *int* `*n_zero_deviations` (Output)
Number of zero differences (ties) `x`[$j - 1$] – `percentile` for $j = 1, 2, \ldots,$ `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 $\mathrm{x}_i = \mathrm{y}_i - \mathrm{z}_i$ 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).

Function `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:

1. $H_0 : M \le 0 \quad H_1 : M > 0$
   Reject if `stat[0]` [or `stat[4]`] is too large.
2. $H_0 : M \ge 0 \quad H_1 : M < 0$
   Reject if `stat[1]` [or `stat[5]`] is too large.
3. $H_0 : M = 0 \quad H_1 : M \ne 0$
   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:

$H_0 : q = \Pr(X_i > X_{i-1} > X_{i-2}) + \Pr(X_i < X_{i-1} < X_{i-2}) \le 1/3 \qquad H_1 : q > 1/3$

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` $\neq 0$) 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	$H_0{:}Pr(\texttt{x1} < \texttt{x2}) = 0.5$	$H_1{:}Pr(\texttt{x1} < \texttt{x2}) \neq 0.5$	Reject if `stat` [9] is less than the significance level of the test. Alternatively,
	$H_0{:}E(\texttt{x1}) = E(\texttt{x2})$	$H_1{:}E(\texttt{x1}) \neq E(\texttt{x2})$	reject the null hypothesis if `stat` [6] is too large or too small.
2	$H_0{:}Pr(\texttt{x1} < \texttt{x2}) \leq 0.5$	$H_1{:}Pr(\texttt{x1} < \texttt{x2}) > 0.5$	Reject if `stat` [6] is too small
	$H_0{:}E(\texttt{x1}) \geq E(\texttt{x2})$	$H_1{:}E(\texttt{x1}) < E(\texttt{x2})$	
3	$H_0{:}Pr(\texttt{x1} < \texttt{x2}) \geq 0.5$	$H_1{:}Pr(\texttt{x1} < \texttt{x2}) < 0.5$	Reject if `stat` [6] is too large
	$H_o{:}E(\texttt{x1}) \leq E(\texttt{x2}))$	$H_1{:}E(\texttt{x1}) > E(\texttt{x2})$	

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).

The *p*-value is calculated using the large-sample normal approximation. This approximate calculation is only valid when the size of one or both samples is greater than 50. For smaller samples, see the exact tables for the Wilcoxon Rank Sum Test.

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 15, "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` 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` 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

$$A = \sum_{i=1}^{k} \sum_{j=1}^{b} \mathrm{Rank}\left(Y_{ij}\right)^2$$

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

$$R_i = \sum_{j=1}^{b} \mathrm{Rank}\left(Y_{ij}\right)$$

The Friedman test statistic (`stat[0]`) is given by:

$$T = \frac{(k-1)\left(bB - b^2 k (k+1)^2 / 4\right)}{A - bk(k+1)^2 / 4}$$

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

$$D = t_{1-\alpha/2}\sqrt{2b\left(A-B\right)/\left(\left(b-1\right)\left(k-1\right)\right)}$$

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 $\alpha = .05$ 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 $\alpha = .05$ 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 $F_1(X) < F_2(X) < \ldots F_k(X)$, 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

$H_0 : F_1(\mathrm{x}) \ge F_2(\mathrm{x}) \ge \ldots \ge F_k(\mathrm{x})$
$H_1 : F_1(\mathrm{x}) < F_2(\mathrm{x}) < \ldots < F_k(\mathrm{x})$
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

General Goodness-of-fit tests

Chi-squared goodness-of-fit test	chi_squared_test	475
Shapiro-Wilk *W* test for normality	normality_test	483
One-sample continuous data Kolmogorov-Smirnov	kolmogorov_one	487
Two-sample continuous data Kolmogorov-Smirnov	kolmogorov_two	490
Mardia's test for multivariate normality	multivar_normality_test	493

Tests for Randomness

Runs test, Paris-serial test, d^2 test or triplets tests	randomness_test	497

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:

1. 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.
2. 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` 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:

$$\bullet H_0 : F(x) = G(x) \quad H_1 : F(x) \neq G(x)$$
$$\bullet H_0 : F(x) \leq G(x) \quad H_1 : F(x) > G(x)$$
$$\bullet H_0 : F(x) \geq G(x) \quad H_1 : F(x) < G(x)$$

The test statistics are given as follows:

$$D_{mn} = \max\left(D_{mn}^{+}, D_{mn}^{-}\right) \qquad \text{(diffs[0])}$$
$$D_{mn}^{+} = \max_x (F_n(x) - G_m(x)) \quad \text{(diffs[1])}$$
$$D_{mn}^{-} = \max_x (\mathrm{G}_m(x) - F_n(x)) \quad \text{(diffs[2])}$$

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 104, according to an algorithm given by Kim and Jennrich (1973). When $n*m$ is greater than 104, 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

3.
```
        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 $R^T R$ 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 $np3$, 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 ($\Pr(D2 \le d^2)$) 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.

Examples

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 104 is used and the `pairs_lag` is 1. The results are not significant. IMSL routine `imsls_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, "Random Number Generation) 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
```

Chapter 8: Time Series and Forecasting

Routines

ARIMA Models

Computes least-squares or method of moments estimates of parameters	arma	511
Computes maximum likelihood estimates of parameters	max_arma	521
Computes forecasts and their associated probability limits	arma_forecast	527
Automatic selection and fitting of a univariate autoregressive time series model.	auto_uni_ar	532
Detects and determines outliers and simultaneously estimates the model parameters in a time series	ts_outlier_identification	537
Computes forecasts for an outlier contaminated time series	ts_outlier_forecast	547
Automatically identifies time series outliers, determines parameters of a multiplicative seasonal ARIMA $(p,0,q)\times(0,d,0)_s$ model and produces forecasts that incorporate the effects of outliers whose effects persist beyond the end of the series	auto_arima	555
Performs differencing on a time series	difference	563
Estimates the optimum seasonality parameters for a time series using an autoregressive model	seasonal_fit	576

Model Construction and Evaluation Utilities

Performs a Box-Cox transformation	box_cox_transform	584
Sample autocorrelation function	autocorrelation	588
Computes the sample cross correlation function	crosscorrelation	593
Computes the multichannel cross-correlation function	multi_crosscorrelation	599
Sample partial autocorrelation function	partial_autocorrelation	608
Lack-of-fit test based on the corrleation function	lack_of_fit	611
Estimates missing values in a time series	estimate_missing	614

GARCH Modeling

Computes estimates of the parameters of a GARCH(*p*,*q*) model	garch	618

Frequency Domain Modeling
Performs Kalman filtering and evaluates the likelihood function for the state-space model kalman 626

Usage Notes

The functions in this chapter assume the time series does not contain any missing observations. If missing values are present, they should be set to NaN (see Chapter 15, "Utilities") routine `imsls_f_machine`), and the routine will return an appropriate error message. To enable fitting of the model, the missing values must be replaced by appropriate estimates.

General Methodology

A major component of the model identification step concerns determining if a given time series is stationary. The sample correlation functions computed by routines `imsls_f_autocorrelation`, `imsls_f_crosscorrelation`, `imsls_f_multi_crosscorrelation`, and `imsls_f_partial_autocorrelation` may be used to diagnose the presence of nonstationarity in the data, as well as to indicate the type of transformation required to induce stationarity. The family of power transformations provided by routine `imsls_f_box_cox_transform` coupled with the ability to difference the transformed data using routine `imsls_f_difference` affords a convenient method of transforming a wide class of nonstationary time series to stationarity.

The "raw" data, transformed data, and sample correlation functions also provide insight into the nature of the underlying model. Typically, this information is displayed in graphical form via time series plots, plots of the lagged data, and various correlation function plots.

The observed time series may also be compared with time series generated from various theoretical models to help identify possible candidates for model fitting. The routine `imsls_f_random_uniform` (Chapter 12, "Random Number Generation) may be used to generate a time series according to a specified autoregressive moving average model.

Time Domain Methodology

Once the data are transformed to stationarity, a tentative model in the time domain is often proposed and parameter estimation, diagnostic checking and forecasting are performed.

ARIMA Model (Autoregressive Integrated Moving Average)

A small, yet comprehensive, class of stationary time-series models consists of the nonseasonal ARMA processes defined by

$$\phi(B)\,(W_t - \mu) = \theta(B)A_t, t \in Z$$

where $Z = \{..., -2, -1, 0, 1, 2, ...\}$ denotes the set of integers, B is the backward shift operator defined by $B^k W_t = W_{t-k}$, μ is the mean of W_t, and the following equations are true:

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - ... - \phi_p B^p, p \geq 0$$

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - ... - \theta_q B^q, q \geq 0$$

The model is of order (p, q) and is referred to as an ARMA (p, q) model.

An equivalent version of the ARMA (p, q) model is given by

$$\phi(B)\ W_t = \theta_0 + \theta(B) A_t,\ \ t \in Z$$

where θ_0 is an overall constant defined by the following:

$$\theta_0 = \mu\left(1 - \sum_{i=1}^{p} \phi_i\right)$$

See Box and Jenkins (1976, pp. 92–93) for a discussion of the meaning and usefulness of the overall constant.

If the "raw" data, $\{Z_t\}$, are homogeneous and nonstationary, then differencing using `imsls_f_difference` induces stationarity, and the model is called ARIMA (AutoRegressive Integrated Moving Average). Parameter estimation is performed on the stationary time series $W_t, = \nabla^d Z_t$, where $\nabla^d = (1 - B)^d$ is the backward difference operator with period 1 and order d, d > 0.

Typically, the method of moments includes argument `IMSLS_METHOD_OF_MOMENTS` in a call to function `imsls_f_arma` for preliminary parameter estimates. These estimates can be used as initial values into the least-squares procedure by including argument `IMSLS_LEAST_SQUARES` in a call to function `imsls_f_arma`. Other initial estimates provided by the user can be used. The least-squares procedure can be used to compute conditional or unconditional least-squares estimates of the parameters, depending on the choice of the backcasting length. The parameter estimates from either the method of moments or least-squares procedures can be input to function `imsls_f_arma_forecast` through the `arma_info` structure. The functions for preliminary parameter estimation, least-squares parameter estimation, and forecasting follow the approach of Box and Jenkins (1976, Programs 2–4, pp. 498–509).

arma

Computes least-square estimates of parameters for an ARMA model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_arma` (*int* `n_observations`, *float* `z[]`, *int* `p`, *int* `q`, ..., 0)

The type *double* function is `imsls_d_arma`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `z[]` (Input)
Array of length `n_observations` containing the observations.

int `p` (Input)
Number of autoregressive parameters.

int `q` (Input)
Number of moving average parameters.

Return Value

Pointer to an array of length $1 + p + q$ with the estimated constant, AR, and MA parameters. If `IMSLS_NO_CONSTANT` is specified, the 0-th element of this array is 0.0.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_arma` (*int* `n_observations`, *float* `z[]`, *int* `p`, *int* `q`,
`IMSLS_NO_CONSTANT`, *or*
`IMSLS_CONSTANT`,
`IMSLS_AR_LAGS`, *int* `ar_lags[]`,
`IMSLS_MA_LAGS`,v*int* `ma_lags[]`,
`IMSLS_METHOD_OF_MOMENTS`, *or*
`IMSLS_LEAST_SQUARES`,
`IMSLS_BACKCASTING`, *int* `length`, *float* `tolerance`,
`IMSLS_CONVERGENCE_TOLERANCE`, *float* `convergence_tolerance`,
`IMSLS_RELATIVE_ERROR`, *float*`vrelative_error`,
`IMSLS_MAX_ITERATIONS`,v*int*`vmax_iterations`,
`IMSLS_MEAN_ESTIMATE`, *float* `*z_mean`,
`IMSLS_INITIAL_ESTIMATES`, *float* `ar[]`, *float* `ma[]`,
`IMSLS_RESIDUAL`, *float* `**residual`,
`IMSLS_RESIDUAL_USER`, *float* `residual[]`,
`IMSLS_PARAM_EST_COV`, *float* `**param_est_cov`,
`IMSLS_PARAM_EST_COV_USER`, *float* `param_est_cov[]`,
`IMSLS_AUTOCOV`, *float* `**autocov`,
`IMSLS_AUTOCOV_USER`, *float* `autocov[]`,
`IMSLS_SS_RESIDUAL`, *float* `*ss_residual`,
`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]`, *float* `ma[]`,
`IMSLS_ARMA_INFO`, *Imsls_f_arma* `**arma_info`,
0)

Optional Arguments

IMSLS_NO_CONSTANT, *or*
IMSLS_CONSTANT

If IMSLS_NO_CONSTANT is specified, the time series is not centered about its mean, z_mean. If IMSLS_CONSTANT, the default, is specified, the time series is centered about its mean.

IMSLS_AR_LAGS, *int* ar_lags[] (Input)

Array of length p containing the order of the autoregressive parameters. The elements of ar_lags must be greater than or equal to 1.
Default: ar_lags = [1, 2, ..., p]

IMSLS_MA_LAGS, *int* ma_lags[] (Input)

Array of length q containing the order of the moving average parameters. The ma_lags elements must be greater than or equal to 1.
Default: ma_lags = [1, 2, ..., q]

IMSLS_METHOD_OF_MOMENTS, *or*
IMSLS_LEAST_SQUARES

If IMSLS_METHOD_OF_MOMENTS is specified, the autoregressive and moving average parameters are estimated by a method of moments procedure. If IMSLS_LEAST_SQUARES is specified, the autoregressive and moving average parameters are estimated by a least-squares procedure.

IMSLS_BACKCASTING, *int* length, *float* tolerance (Input)

If IMSLS_BACKCASTING is specified, length is the maximum length of backcasting and must be greater than or equal to 0. Argument tolerance is the tolerance level used to determine convergence of the backcast algorithm. Typically, tolerance is set to a fraction of an estimate of the standard deviation of the time series.
Default: length = 10; tolerance = 0.01 × standard deviation of z

IMSLS_CONVERGENCE_TOLERANCE, *float* convergence_tolerance (Input)

Tolerance level used to determine convergence of the nonlinear least-squares algorithm. Argument convergence_tolerance represents the minimum relative decrease in sum of squares between two iterations required to determine convergence. Hence, convergence_tolerance must be greater than or equal to 0. The default value is max $\{10^{-10}, \text{eps}^{2/3}\}$ for single precision and max $\{10^{-20}, \text{eps}^{2/3}\}$ for double precision, where
eps = imsls_f_machine(4) for single precision and
eps = imsls_d_machine(4) for double precision.

IMSLS_RELATIVE_ERROR, *float* relative_error (Input)

Stopping criterion for use in the nonlinear equation solver used in both the method of moments and least-squares algorithms.
Default: relative_error = 100 × imsls_f_machine(4)
See documentation for function imsls_f_machine (Chapter 15, "Utilities").

IMSLS_MAX_ITERATIONS, *int* max_iterations (Input)

Maximum number of iterations allowed in the nonlinear equation solver used

in both the method of moments and least-squares algorithms.
Default: `max_iterations` = 200

`IMSLS_MEAN_ESTIMATE`, *float* `*z_mean` (Input or Input/Output)
On input, `z_mean` is an initial estimate of the mean of the time series `z`. On return, `z_mean` contains an update of the mean.
If `IMSLS_NO_CONSTANT` and `IMSLS_LEAST_SQUARES` are specified, `z_mean` is not used in parameter estimation.

`IMSLS_INITIAL_ESTIMATES`, *float* `ar[]`, *float* `ma[]` (Input)
If specified, `ar` is an array of length `p` containing preliminary estimates of the autoregressive parameters, and `ma` is an array of length `q` containing preliminary estimates of the moving average parameters; otherwise, these are computed internally. `IMSLS_INITIAL_ESTIMATES` is only applicable if `IMSLS_LEAST_SQUARES` is also specified.

`IMSLS_RESIDUAL`, *float* `**residual` (Output)
Address of a pointer to an internally allocated array of length `n_observations` – max (`ar_lags` [*i*]) + `length` containing the residuals (including backcasts) at the final parameter estimate point in the first `n_observations` – max (`ar_lags` [*i*]) + *nb*, where *nb* is the number of values backcast.

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

`IMSLS_PARAM_EST_COV`, *float* `**param_est_cov` (Output)
Address of a pointer to an internally allocated array of size $np \times np$, where np = `p` + `q` + 1 if `z` is centered about `z_mean`, and np = `p` + `q` if `z` is not centered. The ordering of variables in `param_est_cov` is `z_mean`, `ar`, and `ma`. Argument *np* must be 1 or larger.

`IMSLS_PARAM_EST_COV_USER`, *float* `param_est_cov[]` (Output)
Storage for array `param_est_cov` is provided by the user. See `IMSLS_PARAM_EST_COV`.

`IMSLS_AUTOCOV`, *float* `**autocov` (Output)
Address of a pointer to an array of length `p` + `q` + 1 containing the variance and autocovariances of the time series `z`. Argument `autocov` [0] contains the variance of the series `z`. Argument `autocov` [*k*] contains the autocovariance of lag *k*, where *k* = 1, ..., `p` + `q` + 1.

`IMSLS_AUTOCOV_USER`, *float* `autocov[]` (Output)
Storage for array `autocov` is provided by the user. See `IMSLS_AUTOCOV`.

`IMSLS_SS_RESIDUAL`, *float* `*ss_residual` (Output)
If specified, `ss_residual` contains the sum of squares of the random shock, `ss_residual` = `residual` $[1]^2$ + ... + `residual` $[\mathrm{na}]^2$.

`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]`, *float* `ma[]` (Output)
If specified, `constant` is the constant parameter estimate, `ar` is an array of

length `p` containing the final autoregressive parameter estimates, and `ma` is an array of length `q` containing the final moving average parameter estimates.

`IMSLS_ARMA_INFO`, *Imsls_f_arma* `**arma_info` (Output)
Address of a pointer to an internally allocated structure of type *Imsls_f_arma* that contains information necessary in the call to `imsls_forecast`.

Description

Function `imsls_f_arma` computes estimates of parameters for a nonseasonal ARMA model given a sample of observations, $\{W_t\}$, for $t = 1, 2, ..., n$, where n = `n_observations`. There are two methods, method of moments and least squares, from which to choose. The default is method of moments.

Two methods of parameter estimation, method of moments and least squares, are provided. The user can choose the method of moments algorithm with the optional argument `IMSLS_METHOD_OF_MOMENTS`. The least-squares algorithm is used if the user specifies `IMSLS_LEAST_SQUARES`. If the user wishes to use the least-squares algorithm, the preliminary estimates are the method of moments estimates by default. Otherwise, the user can input initial estimates by specifying optional argument `IMSLS_INITIAL_ESTIMATES`. The following table lists the appropriate optional arguments for both the method of moments and least-squares algorithm:

Method of Moments only	Least Squares only	Both Method of Moments and Least Squares
`IMSLS_METHOD_OF_MOMENTS`	`IMSLS_LEAST_SQUARES`	`IMSLS_RELATIVE_ERROR`
	`IMSLS_CONSTANT`	`IMSLS_MAX_ITERATIONS`
	`(or IMSLS_NO_CONSTANT)`	
	`IMSLS_AR_LAGS`	`IMSLS_MEAN_ESTIMATE`
	`IMSLS_MA_LAGS`	`IMSLS_AUTOCOV(_USER)`
	`IMSLS_BACKCASTING`	`IMSLS_RETURN_USER`
	`IMSLS_CONVERGENCE_TOLERANCE`	`IMSLS_ARMA_INFO`
	`IMSLS_INITIAL_ESTIMATES`	
	`IMSLS_RESIDUAL (_USER)`	
	`IMSLS_PARAM_EST_COV (_USER)`	
	`IMSLS_SS_RESIDUAL`	

Method of Moments Estimation

Suppose the time series $\{Z_t\}$ is generated by an ARMA (p, q) model of the form

$$\phi(B)Z_t = \theta_0 + \theta(B)A_t$$

for $t \in \{0, \pm 1, \pm 2, ...\}$

Let $\hat{\mu}$ = `w_mean` be the estimate of the mean μ of the time series $\{Z_t\}$, where $\hat{\mu}$ equals the following:

$$\hat{\mu} = \begin{cases} \mu & \text{for } \mu \text{ known} \\ \dfrac{1}{n}\sum_{t=1}^{n} Z_t & \text{for } \mu \text{ unknown} \end{cases}$$

The autocovariance function is estimated by

$$\hat{\sigma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(Z_t - \hat{\mu})(Z_{t+k} - \hat{\mu})$$

for $k = 0, 1, ..., K$, where $K = p + q$. Note that $\hat{\sigma}(0)$ is an estimate of the sample variance.

Given the sample autocovariances, the function computes the method of moments estimates of the autoregressive parameters using the extended Yule-Walker equations as follows:

$$\hat{\Sigma}\hat{\phi} = \hat{\sigma}$$

where

$$\begin{aligned} \hat{\phi} &= \left(\hat{\phi}_1, \ldots, \hat{\phi}_p\right)^T \\ \hat{\Sigma}_{ij} &= \hat{\sigma}\left(|\, q + i - j \,|\right), && i, j = 1, \ldots, p \\ \hat{\sigma}_i &= \hat{\sigma}\left(q + i\right), && i = 1, \ldots, p \end{aligned}$$

The overall constant θ_0 is estimated by the following:

$$\hat{\theta}_0 = \begin{cases} \hat{\mu} & \text{for } p = 0 \\ \hat{\mu}\left(1 - \sum_{i=1}^{p}\hat{\phi}_i\right) & \text{for } p > 0 \end{cases}$$

The moving average parameters are estimated based on a system of nonlinear equations given $K = p + q + 1$ autocovariances, $\sigma(k)$ for $k = 1, ..., K$, and p autoregressive parameters ϕ_i for $i = 1, ..., p$.

Let $Z'_t = \phi(B)Z_t$. The autocovariances of the derived moving average process $Z'_t = \theta(B)A_t$ are estimated by the following relation:

$$\hat{\sigma}'(k) = \begin{cases} \hat{\sigma}(k) & \text{for } p = 0 \\ \sum_{i=0}^{p}\sum_{j=0}^{p}\hat{\phi}_i\hat{\phi}_j\left(\hat{\sigma}\left(|k + i - j|\right)\right) & \text{for } p \ge 1, \hat{\phi}_0 \equiv -1 \end{cases}$$

The iterative procedure for determining the moving average parameters is based on the relation

$$\sigma(k) = \begin{cases} \left(1+\theta_1^2+\ldots+\theta_q^2\right)\sigma_A^2 & \text{for } k=0 \\ \left(-\theta_k+\theta_1\theta_{k+1}+\ldots+\theta_{q-k}\theta_q\right)\sigma_A^2 & \text{for } k \ge 1 \end{cases}$$

where $\sigma(k)$ denotes the autocovariance function of the original Z_t process.

Let $\tau = (\tau_0, \tau_1, \ldots, \tau_q)^T$ and $f = (f_0, f_1, \ldots, f_q)^T$, where

$$\tau_j = \begin{cases} \sigma_A & \text{for } j=0 \\ -\theta_j / \tau_0 & \text{for } j = 1, \ldots, q \end{cases}$$

and

$$f_j = \sum_{i=0}^{q-j} \tau_i \tau_{i+j} - \hat{\sigma}'(j) \qquad \text{for } j = 0, 1, \ldots, q$$

Then, the value of τ at the $(i+1)$-th iteration is determined by the following:

$$\tau^{i+1} = \tau^i - \left(T^i\right)^{-1} f^i$$

The estimation procedure begins with the initial value

$$\tau^0 = (\sqrt{\hat{\sigma}'(0)}, \quad 0, \ldots, 0)^T$$

and terminates at iteration i when either $\|f^i\|$ is less than `relative_error` or i equals `max_iterations`. The moving average parameter estimates are obtained from the final estimate of τ by setting

$$\hat{\theta}_j = -\tau_j / \tau_0 \text{ for } j = 1, \ldots, q$$

The random shock variance is estimated by the following:

$$\hat{\sigma}_A^2 = \begin{cases} \hat{\sigma}(0) - \sum_{i=1}^{p} \hat{\phi}_i \hat{\sigma}(i) & \text{for } q = 0 \\ \tau_0^2 & \text{for } q \ge 0 \end{cases}$$

See Box and Jenkins (1976, pp. 498–500) for a description of a function that performs similar computations.

Least-squares Estimation

Suppose the time series $\{Z_t\}$ is generated by a nonseasonal ARMA model of the form,

$$\phi(B)(Z_t - \mu) = \theta(B)A_t \quad \text{for t} \in \{0, \pm 1, \pm 2, \ldots\}$$

where B is the backward shift operator, μ is the mean of Z_t, and

$$\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ldots - \phi_p B^{l_\phi(p)} \quad \text{for } p \geq 0$$
$$\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ldots - \theta_q B^{l_\theta(q)} \quad \text{for } q \geq 0$$

with p autoregressive and q moving average parameters. Without loss of generality, the following is assumed:

$$1 \leq l_\phi(1) \leq l_\phi(2) \leq \ldots \leq l_\phi(p)$$

$$1 \leq l_\theta(1) \leq l_\theta(2) \leq \ldots \leq l_\theta(q)$$

so that the nonseasonal ARMA model is of order (p', q'), where $p' = l_\theta(p)$ and $q' = l_\theta(q)$. Note that the usual hierarchical model assumes the following:

$$l_\phi(i) = i, 1 \leq i \leq p$$

$$l_\theta(j) = j, 1 \leq j \leq q$$

Consider the sum-of-squares function

$$S_T(\mu, \phi, \theta) = \sum_{-T+1}^{n} [A_t]^2$$

where

$$[A_t] = E\left[A_t \middle| (\mu, \phi, \theta, Z)\right]$$

and T is the backward origin. The random shocks $\{A_t\}$ are assumed to be independent and identically distributed

$$N(0, \sigma_A^2)$$

random variables. Hence, the log-likelihood function is given by

$$l(\mu, \phi, \theta, \sigma_A) = f(\mu, \phi, \theta) - n\ln(\sigma_A) - \frac{S_T(\mu, \phi, \theta)}{2\sigma_A^2}$$

where $f(\mu, \phi, \theta)$ is a function of μ, ϕ, and θ.

For $T = 0$, the log-likelihood function is conditional on the past values of both Z_t and A_t required to initialize the model. The method of selecting these initial values usually introduces transient bias into the model (Box and Jenkins 1976, pp. 210–211). For $T = \infty$, this dependency vanishes, and estimation problem concerns maximization of the unconditional log-likelihood function. Box and Jenkins (1976, p. 213) argue that

$$S_\infty(\mu,\phi,\theta)/\left(2\sigma_A^2\right)$$

dominates

$$l\left(\mu,\phi,\theta,\sigma_A^2\right)$$

The parameter estimates that minimize the sum-of-squares function are called least-squares estimates. For large n, the unconditional least-squares estimates are approximately equal to the maximum likelihood-estimates.

In practice, a finite value of T will enable sufficient approximation of the unconditional sum-of-squares function. The values of $[A_T]$ needed to compute the unconditional sum of squares are computed iteratively with initial values of Z_t obtained by back forecasting. The residuals (including backcasts), estimate of random shock variance, and covariance matrix of the final parameter estimates also are computed. ARIMA parameters can be computed by using imsls_f_difference with imsls_f_arma.

Examples

Example 1

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. The method of moments estimates

$$\hat{\theta}_0, \hat{\varphi}_1, \hat{\varphi}_2, \text{ and } \hat{\theta}_1$$

for the ARMA(2, 1) model

$$z_t = \theta_0 + \phi_0 z_{t-1} + \phi_2 z_{t-2} - \theta_1 A_{t-1} + A_t$$

where the errors A_t are independently normally distributed with mean zero and variance

$$\sigma_A^2$$

```
#include <imsls.h>

void main()
{
    int    p = 2;
    int    q = 1;
    int    i;
    int    n_observations = 100;
    int    max_iterations = 0;
    float  w[176][2];
    float  z[100];
    float  *parameters;
    float  relative_error = 0.0;

    imsls_f_data_sets(2, IMSLS_X_COL_DIM,
                      2, IMSLS_RETURN_USER, w,
                      0);
    for (i=0; i<n_observations; i++) z[i] = w[21+i][1];

    parameters = imsls_f_arma(n_observations, &z[0], p, q,
                              IMSLS_RELATIVE_ERROR, relative_error,
                              IMSLS_MAX_ITERATIONS, max_iterations,
                              0);
    printf("AR estimates are %11.4f and %11.4f.\n",
           parameters[1], parameters[2]);
    printf("MA estimate is %11.4f.\n", parameters[3]);
}
```

Output

```
AR estimates are      1.2443 and     -0.5751.
MA estimate is     -0.1241.
```

Example 2

The data for this example are the same as that for the initial example. Preliminary method of moments estimates are computed by default, and the method of least squares is used to find the final estimates. Note that at the end of the output, a warning error appears. In most cases, this error message can be ignored. There are three general reasons this error can occur:

1. Convergence is declared using the criterion based on tolerance, but the gradient of the residual sum-of-squares function is nonzero. This occurs in this example. Either the message can be ignored or `tolerance` can be reduced to allow more iterations and a slightly more accurate solution.
2. Convergence is declared based on the fact that a very small step was taken, but the gradient of the residual sum-of-squares function was nonzero. This message can usually be ignored. Sometimes, however, the algorithm is making very slow progress and is not near a minimum.
3. Convergence is not declared after 100 iterations.

Trying a smaller value for `tolerance` can help determine what caused the error message.

```
#include <imsls.h>

void main()
{
    int    p = 2;
    int    q = 1;
    int    i;
    int    n_observations = 100;
    float  w[176][2];
    float  z[100];
    float  *parameters;
    float  tolerance = 0.125;

    imsls_f_data_sets(2, IMSLS_X_COL_DIM,
                      2, IMSLS_RETURN_USER, w,
                      0);
    for (i=0; i<n_observations; i++) z[i] = w[21+i][1];

    parameters = imsls_f_arma(n_observations, &z[0], p, q,
                              IMSLS_LEAST_SQUARES,
                              IMSLS_CONVERGENCE_TOLERANCE,
                                 tolerance,
                              0);
    printf("AR estimates are %11.4f and %11.4f.\n",
           parameters[1], parameters[2]);
    printf("MA estimate is %11.4f.\n", parameters[3]);

}
```

Output

```
*** WARNING  Error IMSLS_LEAST_SQUARES_FAILED from imsls_f_arma.  Least
***          squares estimation of the parameters has failed to converge.
***          Increase "length" and/or "tolerance" and/or
***          "convergence_tolerance". The estimates of the parameters at
              the
***          last iteration may be used as new starting values.

AR estimates are      1.3926 and     -0.7329.
MA estimate is     -0.1375.
```

Warning Errors

`IMSLS_LEAST_SQUARES_FAILED`	Least-squares estimation of the parameters has failed to converge. Increase “length” and/or “tolerance” and/or “convergence_tolerance.” The estimates of the parameters at the last iteration may be used as new starting values.

max_arma

Exact maximum likelihood estimation of the parameters in a univariate ARMA (autoregressive, moving average) time series model.

Synopsis

#include <`imsls.h`>

float *`imsls_f_max_arma` (*int* `n_obs`, *float* `w[]`, *int* `p`, *int* `q`,…,0)

The type *double* function is `imsls_d_max_arma`.

Required Arguments

int `n_obs` (Input)
Number of observations in the time series.

float `w[]` (Input)
Array of length `n_obs` containing the time series.

int `p` (Input)
Non-negative number of autoregressive parameters.

int `q` (Input)
Non-negative number of moving average parameters.

Return Value

Pointer to an array of length 1+`p`+`q` with the estimated constant, AR and MA parameters. If no value can be computed, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_max_arma` (*int* `n_obs`, *float* `w[]`, *int* `p`, *int* `q`,
`IMSLS_INITIAL_ESTIMATES`, *float* `init_ar[]` *float* `init_ma[]`,
`IMSLS_PRINT_LEVEL`, *int* `iprint`,
`IMSLS_MAX_ITERATIONS`, *int* `maxit`,
`IMSLS_LOG_LIKELIHOOD`, *float* `*log_likeli`,
`IMSLS_VAR_NOISE`, *float* `*avar`,
`IMSLS_ARMA_INFO`, *Imsls_f_arma* `**arma_info`,
`IMSLS_MEAN_ESTIMATE`, *float* `*w_mean`,
`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]`, *float* `ma[]`,
0)

Optional Arguments

`IMSLS_INITIAL_ESTIMATES`, *float* `init_ar[]`, *float* `init_ma[]` (Input)
If specified, `init_ar` is an array of length `p` containing preliminary estimates of the autoregressive parameters, and `init_ma` is an array of length `q` containing preliminary estimates of the moving average parameters; otherwise, they are computed internally. If `p`=0 or `q`=0, then the corresponding arguments are ignored.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option:
0 — No printing.
1 — Prints final results only.
2 — Prints intermediate and final results.
Default: `iprint` = 0

`IMSLS_MAX_ITERATIONS`, *int* `maxit` (Input)
Maximum number of estimation iterations.
Default: `maxit` = 300

`IMSLS_VAR_NOISE`, *float* `*avar` (Output)
Estimate of innovation variance.

`IMSLS_LOG_LIKELIHOOD`, *float* `*log_likeli` (Output)
Value of -2*(ln(likelihood)) for the fitted model.

`IMSLS_ARMA_INFO`, *Imsls_f_arma* `**arma_info` (Output)
Address of a pointer to an internally allocated structure of type *Imsls_f_arma* that contains information necessary in the call to `imsls_f_arma_forecast`.

`IMSLS_MEAN_ESTIMATE`, *float* `*w_mean` (Input/Output)
Estimate of the mean of the time series `w`. On return, `w_mean` contains an update of the mean.
Default: Time series `w` is centered about its sample mean.

`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]`, *float* `ma[]` (Output)
If specified, `constant` is the constant parameter estimate, `ar` is an array of length `p` containing the final autoregressive parameter estimates, and `ma` is an array of length `q` containing the final moving average parameter estimates.

Description

The function `imsls_f_max_arma` is derived from the maximum likelihood estimation algorithm described by Akaike, Kitagawa, Arahata and Tada (1979), and the XSARMA routine published in the TIMSAC-78 Library.

Using the notation developed in the Time Domain Methodology at the beginning of this chapter, the stationary time series W_t with mean μ can be represented by the nonseasonal autoregressive moving average (ARMA) model by the following relationship:

$$\phi(B)(W_t - \mu) = \theta(B)a_t$$

where

$$t \in ZZ = \{\cdots, -2, -1, 0, 1, 2, \cdots\},$$

B is the backward shift operator defined by $B^k W_t = W_{t-k}$,

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p, \quad p \ge 0,$$

and

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \cdots - \theta_q B^q, \quad q \ge 0.$$

Function `imsls_f_max_arma` estimates the AR coefficients $\phi_1, \phi_2, \cdots, \phi_p$ and the MA coefficients $\theta_1, \theta_2, \cdots, \theta_q$ using maximum likelihood estimation.

Function `imsls_f_max_arma` checks the initial estimates for both the autoregressive and moving average coefficients to ensure that they represent a stationary and invertible series respectively.

If

$$\phi_1, \phi_2, \cdots, \phi_p$$

are the initial estimates for a stationary series then all (complex) roots of the following polynomial will fall outside the unit circle:

$$1 - \phi_1 z - \phi_2 z^2 - \cdots - \phi_p z^p .$$

If

$$\theta_1, \theta_2, \cdots, \theta_q$$

are initial estimates for an invertible series then all (complex) roots of the polynomial

$$1 - \theta_1 z - \theta_2 z^2 - \cdots - \theta_q z^q$$

will fall outside the unit circle.

Initial values for the AR and MA coefficients can be supplied by vectors `init_ar` and `init_ma`. Otherwise, estimates are computed internally by the method of moments. `imsls_f_max_arma` computes the roots of the associated polynomials. If the AR estimates represent a non-stationary series, `imsls_f_max_arma` issues a warning message and replaces `init_ar` with initial values that are stationary. If the MA estimates represent a non-invertible series, `imsls_f_max_arma` issues a terminal error, and new initial values have to be sought.

`imsls_f_max_arma` also validates the final estimates of the AR coefficients to ensure that they too represent a stationary series. This is done to guard against the possibility that the internal log-likelihood optimizer converged to a non-stationary solution. If non-stationary estimates are encountered, `imsls_f_max_arma` issues a fatal error message. Routines `imsls_error_options` and `imsls_error_code` (see Chapter 15, Utilities) can be used to verify that the stationarity condition was met.

For model selection, the ARMA model with the minimum value for *AIC* might be preferred,

$$AIC = \texttt{log\_likeli} + 2(\mathrm{p+q})$$

Function `imsls_f_max_arma` can also handle white noise processes, i.e. ARMA(0,0) Processes.

Examples

Example 1

Consider the Wolfer Sunspot data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1770 through 1869. In this example, `imsls_f_max_arma` is used to fit the following `ARMA(2,1)` model:

$$\tilde{w}_t = \phi_1 \tilde{w}_{t-1} + \phi_2 \tilde{w}_{t-2} + a_t - \theta_1 a_{t-1},$$

with $\tilde{w}_t := w_t - \mu$, μ the sample mean of the time series $\{w_t\}$.

For these data, imsls_f_max_arma calculated the following model:

$$\tilde{w}_t = 1.23\tilde{w}_{t-1} - 0.56\tilde{w}_{t-2} + a_t + 0.38a_{t-1}.$$

Defining the overall constant ϕ_0 by $\phi_0 := \mu(1 - \sum_{i=1}^{p} \phi_i)$, we obtain the following equivalent representations:

$$w_t = \phi_0 + \phi_1 w_{t-1} + \phi_2 w_{t-2} + a_t - \theta_1 a_{t-1},$$

and

$$w_t = 15.73 + 1.23w_{t-1} - 0.56w_{t-2} + a_t + 0.38a_{t-1}.$$

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

void main()
{
  int i;
  int n_obs = 100;
  int p = 2, q = 1;
  float z[176][2];
  float w[100];
  float *parameters = NULL;
  float avar, log_likeli;

  /* get wolfer sunspot data */
  imsls_f_data_sets (2, IMSLS_X_COL_DIM, 2,
                     IMSLS_RETURN_USER, w,
                     0);

  for (i=0; i<n_obs; i++)
      w[i] = z[21+i][1];

  parameters = imsls_f_max_arma (n_obs, w, p, q,
                       IMSLS_MAX_ITERATIONS, 12000,
                       IMSLS_VAR_NOISE, &avar,
                       IMSLS_LOG_LIKELIHOOD, &log_likeli,
                       0);

  printf("AR estimates are %11.4f and %11.4f.\n",
          parameters[1], parameters[2]);
  printf("MA estimate is %11.4f.\n", parameters[3]);
  printf("Constant estimate is %11.4f.\n", parameters[0]);
  printf("-2*ln(Maximum Log Likelihood) = %11.4f.\n", log_likeli);
  printf("White noise variance = %11.4f.\n", avar);


  if (parameters)
  {
     free(parameters);
     parameters = NULL;
  }
  return;
}
```

Output

```
AR estimates are      1.2273 and     -0.5626.
MA estimate is     -0.3808.
Constant estimate is     15.7508.
-2*ln(Maximum Log Likelihood) =    539.5843.
White noise variance =    214.5020.
```

Example 2

This is the same as

Example 1, but now initial values for the AR and MA parameters are explicitly given.

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

void main()
{
  int i;
  int n_obs = 100;
  int p = 2, q = 1;
  float z[176][2];
  float w[100];
  float parameters[4];
  float avar, log_likeli;
  float init_ar[2] = {1.244e0, -0.575e0};
  float init_ma[1] = {-0.1241e0};


  /* get wolfer sunspot data */
  imsls_f_data_sets (2, IMSLS_X_COL_DIM, 2,
                     IMSLS_RETURN_USER, w,
                     0);

  for (i=0; i<n_obs; i++)
      z[i] = w[21+i][1];

  imsls_f_max_arma (n_obs, w, p, q,
                       IMSLS_MAX_ITERATIONS, 12000,
                       IMSLS_VAR_NOISE, &avar,
                       IMSLS_LOG_LIKELIHOOD, &log_likeli,
                       IMSLS_INITIAL_ESTIMATES, init_ar, init_ma,
                       IMSLS_RETURN_USER, &parameters[0], &parameters[1],
                                          &parameters[3],
                       0);

  printf("AR estimates are %11.4f and %11.4f.\n",
          parameters[1], parameters[2]);
  printf("MA estimate is %11.4f.\n", parameters[3]);
  printf("Constant estimate is %11.4f.\n", parameters[0]);
  printf("-2*ln(Maximum Log Likelihood) = %11.4f.\n", log_likeli);
  printf("White noise variance = %11.4f.\n", avar);


  return;
}
```

Output

```
AR estimates are      1.2273 and     -0.5623.
MA estimate is     -0.3804.
Constant estimate is     15.7373.
-2*ln(Maximum Log Likelihood) =    539.5843.
White noise variance =    214.5052.
```

arma_forecast

Computes forecasts and their associated probability limits for an ARMA model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_arma_forecast` (*Imsls_f_arma* `*arma_info`, *int* `n_predict`, ..., 0)

The type *double* function is `imsls_d_arma_forecast`.

Required Arguments

Imsls_f_arma `*arma_info` (Input)
: Pointer to a structure of type *Imsls_f_arma* that is passed from the `imsls_f_arma` function.

int `n_predict` (Input)
: Maximum lead time for forecasts. Argument `n_predict` must be greater than 0.

Return Value

Pointer to an array of length `n_predict` × (`backward_origin` + 3) containing the forecasts up to `n_predict` steps ahead and the information necessary to obtain pairwise confidence intervals. More information is given in the description of argument `IMSLS_RETURN_USER`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_arma_forecast` (*Imsls_f_arma* `*arma_info`, *int* `n_predict`,
`IMSLS_CONFIDENCE`, *float* `confidence`,

```
IMSLS_BACKWARD_ORIGIN, int backward_origin,
IMSLS_RETURN_USER, float forecasts[],
0)
```

Optional Arguments

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Value in the exclusive interval (0, 100) used to specify the `confidence` percent probability limits of the forecasts. Typical choices for `confidence` are 90.0, 95.0, and 99.0.
Default: `confidence` = 95.0

`IMSLS_BACKWARD_ORIGIN`, *int* `backward_origin` (Input)
If specified, the maximum backward origin. Argument `backward_origin` must be greater than or equal to 0 and less than or equal to `n_observations` – max (*maxar*, *maxma*), where *maxar* = max (`ar_lags` [*i*]), *maxma* = max (`ma_lags` [*j*]), and `n_observations` = the number of observations in the series, as input in function `imsls_f_arma`. Forecasts at origins `n_observations` – `backward_origin` through `n_observations` are generated.
Default: `backward_origin` = 0

`IMSLS_RETURN_USER`, *float* `forecasts[]` (Output)
If specified, a user-specified array of length `n_predict` × (`backward_origin` + 3) as defined below.

Column	Content
J	forecasts for lead times *l* = 1, ..., `n_predict` at origins `n_observations` – `backward_origin` – 1 + *j*, where *j* = 0, ..., `backward_origin`
backward_origin + 2	deviations from each forecast that give the confidence percent probability limits
backward_origin + 3	psi weights of the infinite order moving average form of the model

If specified, the forecasts for lead times *l* = 1, ..., `n_predict` at origins `n_observations` – `backward_origin` – 1 + *j*, where *j* = 1, ..., `backward_origin` + 1.

Description

The Box-Jenkins forecasts and their associated probability limits for a nonseasonal ARMA model are computed given a sample of n = `n_observations` $\{Z_t\}$ for $t = 1, 2, ..., n,$ where `n_observations` = the number of observations in the series, as input in function `imsls_f_arma`.

Suppose the time series $\{Z_t\}$ is generated by a nonseasonal ARMA model of the form

$$\phi(B)Z_t = \theta_0 + \theta(B)A_t$$

for $t \in \{0, \pm1, \pm2, ...\}$, where B is the backward shift operator, θ_0 is the constant, and

$$\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ldots - \phi_p B^{l_\phi(p)}$$
$$\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ldots - \theta_q B^{l_\theta(q)}$$

with p autoregressive and q moving average parameters. Without loss of generality, the following is assumed:

$$1 \le l_\phi(1) \le l_\phi(2) \le \ldots \le l_\phi(p)$$

$$1 \le l_\theta(1) \le l_\theta(2) \le \ldots \le l_\theta(q)$$

so that the nonseasonal ARMA model is of order (p', q'), where $p' = l_\theta(p)$ and $q' = l_\theta(q)$. Note that the usual hierarchical model assumes the following:

$$l_\phi(i) = i,\ 1 \le i \le p$$

$$l_\theta(j) = j,\ 1 \le j \le q$$

The Box-Jenkins forecast at origin t for lead time l of Z_{t+1} is defined in terms of the difference equation

$$\hat{Z}_t(l) = \theta_0 + \phi_1\left[Z_{t+l-l_\phi(1)}\right] + \ldots + \phi_p\left[Z_{t+l-l_\phi(p)}\right]$$
$$+\left[A_{t+l}\right] - \theta_1\left[A_{t+l-l_\theta(1)}\right] - \ldots - \left[A_{t+l}\right] - \theta_1\left[A_{t+l-l\theta(1)}\right] - \ldots - \theta_q\left[A_{t+l-l_\theta(q)}\right]$$

where the following is true:

$$[Z_{t+k}] = \begin{cases} Z_{t+k} & \text{for } k = 0, -1, -2, \ldots \\ \hat{Z}_t(k) & \text{for } k = 1, 2, \ldots \end{cases}$$

$$[A_{t+k}] = \begin{cases} Z_{t+k} - \hat{Z}_{t+k-1}(1) & \text{for } k = 0, -1, -2, \ldots \\ 0 & \text{for } k = 1, 2, \ldots \end{cases}$$

The $100(1 - \alpha)$ percent probability limits for Z_{t+l} are given by

$$\hat{Z}_t(l) \pm z_{1/2}\left\{1 + \sum_{j=1}^{l-1} \psi_j^2\right\}^{1/2} \sigma_A$$

where $z_{(1-\alpha/2)}$ is the $100(1 - \alpha/2)$ percentile of the standard normal distribution

$$\sigma_A^2$$

(returned from imsls_f_arma) and

$$\left\{\psi_j^2\right\}$$

are the parameters of the random shock form of the difference equation. Note that the forecasts are computed for lead times $l = 1, 2, ..., L$ at origins
$t = (n - b), (n - b + 1), ..., n$, where L = `n_predict` and b = `backward_origin`.

The Box-Jenkins forecasts minimize the mean-square error

$$E\left[Z_{t+l} - \hat{Z}_t(l)\right]^2$$

Also, the forecasts can be easily updated according to the following equation:

$$\hat{Z}_{t+1}(l) = \hat{Z}_t(l+1) + \psi_l A_{t+1}$$

This approach and others are discussed in Chapter 5: "*Forecasting*" of Box and Jenkins (1976).

Example

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Function imsls_f_arma_forecast computes forecasts and 95-percent probability limits for the forecasts for an ARMA(2, 1) model fit using function imsls_f_max_arma with the method of moments option. With `backward_origin` = 3, columns zero through three of `forecasts` provide forecasts given the data through 1866, 1867, 1868, and 1869, respectively. Column four gives the deviations from the forecast for computing probability limits, and column six gives the psi weights, which can be used to update forecasts when more data is available. For example, the forecast for the 102nd observation (year 1871) given the data through the 100th observation (year 1869) is 77.21; and 95-percent probability limits are given by 77.21 ∓ 56.30. After observation 101 (Z_{101} for year 1870) is available, the forecast can be updated by using

$$\hat{Z}_t(l) \pm z_{\alpha/2}\left\{1 + \sum_{j=1}^{l-1} \psi_j^2\right\}^{1/2} \sigma_A$$

with the psi weight ($\psi_1 = 1.37$) and the one-step-ahead forecast error for observation 101 ($Z_{101} - 83.72$) to give the following:

$$77.21 + 1.37 \times (Z_{101} - 83.72)$$

Since this updated forecast is one step ahead, the 95-percent probability limits are now given by the forecast $\mp$ 33.22.

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

```
void main()
{
    int    p = 2;
    int    q = 1;
    int    i;
    int    n_observations = 100;
    int    max_iterations = 0;
    int    n_predict = 12;
    int    backward_origin = 3;
    float  w[176][2];
    float  z[100];
    float  *parameters;
    float  rel_error = 0.0;
    float  *forecasts;
    Imsls_f_arma *arma_info;

    char   *col_labels[] = {
           "Lead Time",
           "Forecast From 1866",
           "Forecast From 1867",
           "Forecast From 1868",
           "Forecast From 1869",
           "Dev. for Prob. Limits",
           "Psi"};

    imsls_f_data_sets(2, IMSLS_X_COL_DIM,
                      2, IMSLS_RETURN_USER, w,
                      0);
    for (i=0; i<n_observations; i++) z[i] = w[21+i][1];

    parameters = imsls_f_arma(n_observations, &z[0], p, q,
                              IMSLS_RELATIVE_ERROR,
                                 rel_error,
                              IMSLS_MAX_ITERATIONS,
                                 max_iterations,
                              IMSLS_ARMA_INFO,
                                 &arma_info,
                              0);
    printf("Method of Moments initial estimates:\n");
    printf("AR estimates are %11.4f and %11.4f.\n",
           parameters[1], parameters[2]);
    printf("MA estimate is %11.4f.\n", parameters[3]);

    forecasts = imsls_f_arma_forecast(arma_info, n_predict,
                              IMSLS_BACKWARD_ORIGIN,
                                 backward_origin,
                              0);

    imsls_f_write_matrix("* * * Forecast Table * * *\n",
                         n_predict, backward_origin+3,
                         forecasts,
                         IMSLS_COL_LABELS, col_labels,
                         IMSLS_WRITE_FORMAT, "%11.4f",
                         0);
}
```

Output

```
Method of Moments initial estimates:
AR estimates are      1.2443 and     -0.5751.
MA estimate is     -0.1241.

                     * * * Forecast Table * * *

Lead Time  Forecast From  Forecast From  Forecast From  Forecast From
                    1866           1867           1868           1869
        1        18.2833        16.6151        55.1893        83.7196
        2        28.9182        32.0189        62.7606        77.2092
        3        41.0101        45.8275        61.8922        63.4608
        4        49.9387        54.1496        56.4571        50.0987
        5        54.0937        56.5623        50.1939        41.3803
        6        54.1282        54.7780        45.5268        38.2174
        7        51.7815        51.1701        43.3221        39.2965
        8        48.8417        47.7072        43.2631        42.4582
        9        46.5335        45.4736        44.4577        45.7715
       10        45.3524        44.6861        45.9781        48.0758
       11        45.2103        44.9909        47.1827        49.0371
       12        45.7128        45.8230        47.8072        48.9080

Lead Time  Dev. for Prob.          Psi
                   Limits
        1         33.2179       1.3684
        2         56.2980       1.1274
        3         67.6168       0.6158
        4         70.6432       0.1178
        5         70.7515      -0.2076
        6         71.0869      -0.3261
        7         71.9074      -0.2863
        8         72.5337      -0.1687
        9         72.7498      -0.0452
       10         72.7653       0.0407
       11         72.7779       0.0767
       12         72.8225       0.0720
```

auto_uni_ar

Automatic selection and fitting of a univariate autoregressive time series model. The lag for the model is automatically selected using Akaike's information criterion (AIC). Estimates of the autoregressive parameters for the model with minimum AIC are calculated using method of moments, method of least squares, or maximum likelihood.

Synopsis

#include `<imsls.h>`

float `*imsls_f_auto_uni_ar(`*int* `n_obs`, *float* `z[]`, *int* `maxlag`,
int `*p,…,0)`

The type *double* function is `imsls_d_auto_uni_ar`.

Required Arguments

int `n_obs` (Input)
Number of observations in the time series.

float `z[]` (Input)
Array of length `n_obs` containing the stationary time series.

int `maxlag` (Input)
Maximum number of autoregressive parameters requested. It is required that $1 \le$ `maxlag` $\le$ `n_obs/2`.

int `*p` (Output)
Number of autoregressive parameters in the model with minimum AIC.

Return Value

Vector of length 1+ `maxlag` containing the estimates for the constant and the autoregressive parameters in the model with minimum AIC. The estimates are located in the first 1+ `p` locations of this array.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_auto_uni_ar` (*int* `n_obs`, *float* `z[]`, *int* `maxlag`,
int `*p`,
`IMSLS_PRINT_LEVEL`, *int* `iprint`,
`IMSLS_MAX_ITERATIONS`, *int* `maxit`,
`IMSLS_METHOD`, *int* `method`,
`IMSLS_VAR_NOISE`, *float* `*avar`,
`IMSLS_AIC`, *float* `*aic`,
`IMSLS_MEAN_ESTIMATE`, *float* `*z_mean`,
`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]`,
0)

Optional Arguments

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option:
0 — No printing.
1 — Prints final results only.
2 — Prints intermediate and final results.
Default: `iprint` = 0

`IMSLS_MAX_ITERATIONS`, *int* `maxit` (Input)
Maximum number of estimation iterations.
Default: `maxit` = 300

`IMSLS_METHOD`, *int* `method` (Input)
Estimation method option:
0 — Method of moments
1 — Method of least squares realized through Householder transformations

2 — Maximum likelihood
Default: `method` = 1

`IMSLS_VAR_NOISE`, *float* `*avar` (Output)
Estimate of innovation variance.

`IMSLS_AIC`, *float* `*aic` (Output)
Minimum AIC.

`IMSLS_MEAN_ESTIMATE`, *float* `*z_mean` (Input/Output)
Estimate of the mean of the time series `z`. On return, `z_mean` contains an update of the mean.
Default: Time series `z` is centered about its sample mean.

`IMSLS_RETURN_USER`, *float* `*constant`, *float* `ar[]` (Output)
If specified, `constant` is the constant parameter estimate, `ar` is an array of length `maxlag` containing the final autoregressive parameter estimates in its first `p` locations.

Description

Function `auto_uni_ar` automatically selects the order of the AR model that best fits the data and then computes the AR coefficients. The algorithm used in `auto_uni_ar` is derived from the work of Akaike, H., et. al (1979) and Kitagawa and Akaike (1978). This code was adapted from the UNIMAR procedure published as part of the TIMSAC-78 Library.

The best fit AR model is determined by successively fitting AR models with 0, 1, 2, ..., `maxlag` autoregressive coefficients. For each model, Akaike's Information Criterion (AIC) is calculated based on the formula

$$AIC = -2\ln(likelihood) + 2\,\mathrm{p}$$

Function `auto_uni_ar` uses the approximation to this formula developed by Ozaki and Oda (1979),

$$AIC = (\mathtt{n\_obs} - \mathtt{maxlag})\ln\left(\hat{\sigma}^2\right) + 2\,\mathrm{p} + (\mathtt{n\_obs} - \mathtt{maxlag})\left(\ln(2\pi) + 1\right),$$

where $\hat{\sigma}^2$ is an estimate of the residual variance of the series, commonly known in time series analysis as the innovation variance.

The best fit model is the model with minimum AIC. If the number of parameters in this model is equal to the highest order autoregressive model fitted, i.e., p=`maxlag`, then a model with smaller AIC might exist for larger values of `maxlag`. In this case, increasing `maxlag` to explore AR models with additional autoregressive parameters might be warranted.

If `method` = 0, estimates of the autoregressive coefficients for the model with minimum AIC are calculated using method of moments. If `method` =1, the coefficients are determined by the method of least squares applied in the form described by Kitagawa and Akaike (1978). Otherwise, if `method` =2, the coefficients are estimated using maximum likelihood.

Example

Consider the Wolfer Sunspot data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1770 through 1869. In this example, imsls_f_auto_uni_ar found the minimum AIC fit is an autoregressive model with 3 lags:

$$\tilde{w}_t = \phi_1 \tilde{w}_{t-1} + \phi_2 \tilde{w}_{t-2} + \phi_3 \tilde{w}_{t-3} + a_t ,$$

where

$$\tilde{w}_t := w_t - \mu ,$$

μ the sample mean of the time series $\{w_t\}$. Defining the overall constant ϕ_0 by $\phi_0 := \mu(1 - \sum_{i=1}^{3} \phi_i)$, we obtain the following equivalent representation:

$$w_t = \phi_0 + \phi_1 w_{t-1} + \phi_2 w_{t-2} + \phi_3 w_{t-3} + a_t .$$

The example computes estimates for ϕ_0, ϕ_1, ϕ_2, ϕ_3 for every of the three parameter estimation methods available.

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

void main()
{
  int i;
  int maxlag = 20;
  int n_obs = 100;
  int p;
  float w[176][2];
  float z[100];
  float *parameters = NULL;
  float avar, aic, constant;
  float ar[20];

  /* get wolfer sunspot data */
  imsls_f_data_sets (2, IMSLS_X_COL_DIM, 2,
                      IMSLS_RETURN_USER, w,
                      0);

  for (i=0; i<n_obs; i++)
      z[i] = w[21+i][1];

  /* Compute AR parameters for minimum AIC by method of moments */

  printf("\n\nAIC Automatic Order selection\n");
  printf("AR coefficients estimated using method of moments\n");

  parameters = imsls_f_auto_uni_ar(n_obs, z, maxlag, &p,
                                  IMSLS_VAR_NOISE, &avar,
                                  IMSLS_METHOD, 0,
                                  IMSLS_AIC, &aic,
                                  0);

  printf("Order selected: %d\n", p);
  printf("AIC =  %11.4f,  Variance = %11.4f\n", aic, avar);
  printf("Constant estimate is %11.4f.\n", parameters[0]);
  imsls_f_write_matrix("Final AR coefficients estimated by method of
moments",
                 p, 1, &parameters[1], 0);

  if (parameters)
  {
     free(parameters);
     parameters = NULL;
  }

  /* Compute AR parameters for minimum AIC by method of least squares */

  printf("\n\nAIC Automatic Order selection\n");
  printf("AR coefficients estimated using method of least squares\n");

  imsls_f_auto_uni_ar(n_obs, z, maxlag, &p,
                      IMSLS_VAR_NOISE, &avar,
                      IMSLS_METHOD, 1,
                      IMSLS_AIC, &aic,
                      IMSLS_RETURN_USER, &constant, ar,
                      0);

  printf("Order selected: %d\n", p);
  printf("AIC =  %11.4f,  Variance = %11.4f\n", aic, avar);
  printf("Constant estimate is %11.4f.\n", constant);
  imsls_f_write_matrix("Final AR coefficients estimated by method of least
squares", \
                           p, 1, ar, 0);

  /* Compute AR parameters for minimum AIC by maximum likelihood estimation
*/

  printf("\n\nAIC Automatic Order selection\n");
  printf("AR coefficients estimated using maximum likelihood\n");

  imsls_f_auto_uni_ar(n_obs, z, maxlag, &p,
                      IMSLS_VAR_NOISE, &avar,
                      IMSLS_METHOD, 2,
                      IMSLS_AIC, &aic,
                      IMSLS_RETURN_USER, &constant, ar,
                      0);

  printf("Order selected: %d\n", p);
  printf("AIC =  %11.4f,  Variance = %11.4f\n", aic, avar);
  printf("Constant estimate is %11.4f.\n", constant);
  imsls_f_write_matrix("Final AR coefficients estimated by maximum
likelihood", \
                        p, 1, ar, 0);


  return;
}
```

Output

```
AIC Automatic Order selection
AR coefficients estimated using method of moments
Order selected: 3
AIC =     554.0114,  Variance =    287.2694
Constant estimate is     13.7098.

Final AR coefficients estimated by method of moments
                    1       1.368
                    2      -0.738
                    3       0.078


              AIC Automatic Order selection
AR coefficients estimated using method of least squares
Order selected: 3
AIC =     554.0114,  Variance =    144.7149
Constant estimate is      9.8934.

Final AR coefficients estimated by method of least squares
                    1       1.604
                    2      -1.024
                    3       0.209


AIC Automatic Order selection
AR coefficients estimated using maximum likelihood
Order selected: 3
AIC =     554.0114,  Variance =    218.8337
Constant estimate is     11.3902.

Final AR coefficients estimated by maximum likelihood
                    1       1.553
                     2      -1.001
                     3       0.205
```

ts_outlier_identification

Detects and determines outliers and simultaneously estimates the model parameters in a time series whose underlying outlier free series follows a general seasonal or nonseasonal ARMA model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_ts_outlier_identification` (*int* `n_obs`, *int* `model[]`,
float `w[]`,…,0)

The type *double* function is `imsls_d_ts_outlier_identification`.

Required Arguments

int `n_obs` (Input)
Number of observations in the time series.

int `model[]` (Input)
Vector of length 4 containing the numbers p, q, s, d of the ARIMA $(p,0,q)\times(0,d,0)_s$ model the outlier free series is following.

float `w[]` (Input)
An array of length `n_obs` containing the time series.

Return Value

Pointer to an array of length `n_obs` containing the outlier free time series.
If an error occurred, `NULL` is returned.

Synopsis with Optional Arguments

#include <`imsls.h`>

```
float *imsls_f_ts_outlier_identification (int n_obs,
          int model[],float w[],
          IMSLS_RETURN_USER, float x[],
          IMSLS_DELTA, float delta,
          IMSLS_CRITICAL, float critical,
          IMSLS_EPSILON, float epsilon,
          IMSLS_RELATIVE_ERROR,float relative_error,
          IMSLS_RESIDUAL, float **residual,
          IMSLS_RESIDUAL_USER, float residual[],
          IMSLS_RESIDUAL_SIGMA, float *res_sigma,
          IMSLS_NUM_OUTLIERS, int *num_outliers,
          IMSLS_OUTLIER_STATISTICS, int **outlier_stat,
          IMSLS_OUTLIER_STATISTICS_USER, int outlier_stat[],
          IMSLS_TAU_STATISTICS, float **tau_stat,
          IMSLS_TAU_STATISTICS_USER, float tau_stat[],
          IMSLS_OMEGA_WEIGHTS, float **omega,
          IMSLS_OMEGA_WEIGHTS_USER, float omega[],
          IMSLS_ARMA_PARAM, float **parameters,
          IMSLS_ARMA_PARAM_USER, float parameters[],
          IMSLS_AIC, float *aic,
          0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `x[]` (Output)
A user supplied array of length `n_obs` containing the outlier free series.

`IMSLS_DELTA`, *float* `delta` (Input)
The dampening effect parameter used in the detection of a Temporary

Change Outlier (TC), 0<delta < 1.
Default: delta = 0.7

IMSLS_CRITICAL, *float* critical (Input)
Critical value used as a threshold for outlier detection, critical > 0.
Default: critical = 3.0

IMSLS_EPSILON, *float* epsilon (Input)
Positive tolerance value controlling the accuracy of parameter estimates during outlier detection.
Default: epsilon = 0.001

IMSLS_RELATIVE_ERROR, *float* relative_error (Input)
Stopping criterion for the nonlinear equation solver used in function imsls_f_arma.
Default: relative_error = 10^{-10}.

IMSLS_RESIDUAL, *float* **residual (Output)
Address of a pointer to an internally allocated array of length n_obs containing the residuals for the outlier free series.

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

IMSLS_RESIDUAL_SIGMA, *float* *res_sigma (Output)
Residual standard error of the outlier free series.

IMSLS_NUM_OUTLIERS, *int* *num_outliers (Output)
The number of outliers detected.

IMSLS_OUTLIER_STATISTICS, *int* **outlier_stat (Output)
Address of a pointer to an internally allocated array of length num_outliers × 2 containing outlier statistics. The first column contains the time at which the outlier was observed (t=1,2,...,n_obs) and the second column contains an identifier indicating the type of outlier observed. Outlier types fall into one of five categories:

0	Innovational Outliers (IO)
1	Additive outliers (AO)
2	Level Shift Outliers (LS)
3	Temporary Change Outliers (TC)
4	Unable to Identify (UI).

Use IMSLS_NUM_OUTLIERS to obtain num_outliers, the number of detected outliers.
If num_outliers = 0, NULL is returned.

IMSLS_OUTLIER_STATISTICS_USER, *int* outlier_stat[] (Output)
A user allocated array of length n_obs × 2 containing outlier statistics in the first num_outliers locations. Use IMSLS_NUM_OUTLIERS to obtain the number of outliers, num_outliers, detected by ts_outlier_identification. See IMSLS_OUTLIER_STATISTICS.
If num_outliers = 0, outlier_stat stays unchanged.

`IMSLS_TAU_STATISTICS`, *float* `**tau_stat` (Output)
Address of a pointer to an internally allocated array of length `num_outliers` containing the *t* value for each detected outlier.
If `num_outliers = 0`, `NULL` is returned.

`IMSLS_TAU_STATISTICS_USER`, *float* `tau_stat[]` (Output)
A user allocated array of length `n_obs` containing the *t* value for each detected outlier in its first `num_outliers` locations.
If `num_outliers = 0`, `tau_stat` stays unchanged.

`IMSLS_OMEGA_WEIGHTS`, *float* `**omega` (Output)
Address of a pointer to an internally allocated array of length `num_outliers` containing the computed ω weights for the detected outliers.
If `num_outliers = 0`, `NULL` is returned.

`IMSLS_OMEGA_WEIGHTS_USER` *float* `omega[]` (Output)
A user allocated array of length `n_obs` containing the computed ω weights for the detected outliers in its first `num_outliers` locations.
If `num_outliers = 0`, `omega` stays unchanged.

`IMSLS_ARMA_PARAM`, *float* `**parameters` (Output)
Address of a pointer to an internally allocated array of length 1+*p*+*q* containing the estimated constant, AR and MA parameters.

`IMSLS_ARMA_PARAM_USER` *float* `parameters[]` (Output)
A user allocated array of length 1+*p*+*q* containing the estimated constant, AR and MA parameters.

`IMSLS_AIC`, *float* `*aic` (Output)
Akaike's information criterion (AIC).

Description

Consider a univariate time series $\{Y_t\}$ that can be described by the following multiplicative seasonal ARIMA model of order $(p,0,q)\times(0,d,0)_s$:

$$Y_t - \mu = \frac{\theta(B)}{\Delta_s^d \phi(B)} a_t,\ t = 1,\ldots,n.$$

Here, $\Delta_s^d = (1-B^s)^d$, $\theta(B) = 1-\theta_1 B - \ldots - \theta_q B^q$, $\phi(B) = 1-\phi_1 B - \ldots - \phi_p B^p$. B is the lag operator, $B^k Y_t = Y_{t-k}$, $\{a_t\}$ is a white noise process, and μ denotes the mean of the series $\{Y_t\}$.

In general, $\{Y_t\}$ is not directly observable due to the influence of outliers. Chen and Liu (1993) distinguish between four types of outliers: innovational outliers (IO), additive outliers (AO), temporary changes (TC) and level shifts (LS). If an outlier occurs as the last observation of the series, then Chen and Liu's algorithm is unable to determine the outlier's classification. In `imsls_f_ts_outlier_identification`, such an outlier is called a UI (unable to identify) and is treated as an innovational outlier.

In order to take the effects of multiple outliers occurring at time points $t_1, t_2, \ldots, t_m$ into account, Chen and Liu consider the following model:

$$Y_t^* - \mu = \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j) + \frac{\theta(B)}{\Delta_s^d \phi(B)} a_t \,.$$

Here, $\{Y_t^*\}$ is the observed outlier contaminated series, and ω_j and $L_j(B)$ denote the magnitude and dynamic pattern of outlier j, respectively. $I_t(t_j)$ is an indicator function that determines the temporal course of the outlier effect, $I_{t_j}(t_j) = 1$, $I_t(t_j) = 0$ otherwise. **Note** that $L_j(B)$ operates on I_t via $B^k I_t = I_{t-k}$, $k = 0, 1, \ldots$.

The last formula shows that the outlier free series $\{Y_t\}$ can be obtained from the original series $\{Y_t^*\}$ by removing all occurring outlier effects:

$$Y_t = Y_t^* - \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j) \,.$$

The different types of outliers are charaterized by different values for $L_j(B)$:

1. $L_j(B) = \dfrac{\theta(B)}{\Delta_s^d \phi(B)}$ for an innovational outlier,

2. $L_j(B) = 1$ for an additive outlier,

3. $L_j(B) = (1-B)^{-1}$ for a level shift outlier *and*

4. $L_j(B) = (1-\delta B)^{-1}$, $0 < \delta < 1$, for a temporary change outlier.

Function imsls_f_ts_outlier_identification is an implementation of Chen and Liu's algorithm. It determines the coefficients in $\phi(B), \theta(B)$ and the outlier effects in the model for the observed series jointly in three stages. The magnitude of the outlier effects is determined by least squares estimates. Outlier detection itself is realized by examination of the maximum value of the standardized statistics of the outlier effects. For a detailed description, see Chen and Liu's original paper (1993).

Intermediate and final estimates for the coefficients in $\phi(B)$ and $\theta(B)$ are computed by functions imsls_f_arma and imsls_f_max_arma. If the roots of $\phi(B)$ or $\theta(B)$ lie on or within the unit circle, then the algorithm stops with an appropriate error message. In this case, different values for *p* and *q* should be tried.

Examples

Example 1

This example is based on estimates of the Canadian lynx population. Function `imsls_f_ts_outlier_identification` is used to fit an ARIMA(2,2,0) model of the form $(1-B)^2(1-\phi_1 B - \phi_2 B^2) Y_t = a_t$, $t = 1, 2, \ldots, 144$, $\{a_t\}$ Gaussian White noise, to the given series. Function `ts_outlier_identification` computes parameters $\phi_1 = 0.123609$ and $\phi_2 = -0.178963$ and identifies a LS outlier at time point $t = 16$.

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

void main()
{
  float series[114]={
   0.24300E01,0.25060E01,0.27670E01,0.29400E01,0.31690E01,0.34500E01,
   0.35940E01,0.37740E01,0.36950E01,0.34110E01,0.27180E01,0.19910E01,
   0.22650E01,0.24460E01,0.26120E01,0.33590E01,0.34290E01,0.35330E01,
   0.32610E01,0.26120E01,0.21790E01,0.16530E01,0.18320E01,0.23280E01,
   0.27370E01,0.30140E01,0.33280E01,0.34040E01,0.29810E01,0.25570E01,
   0.25760E01,0.23520E01,0.25560E01,0.28640E01,0.32140E01,0.34350E01,
   0.34580E01,0.33260E01,0.28350E01,0.24760E01,0.23730E01,0.23890E01,
   0.27420E01,0.32100E01,0.35200E01,0.38280E01,0.36280E01,0.28370E01,
   0.24060E01,0.26750E01,0.25540E01,0.28940E01,0.32020E01,0.32240E01,
   0.33520E01,0.31540E01,0.28780E01,0.24760E01,0.23030E01,0.23600E01,
   0.26710E01,0.28670E01,0.33100E01,0.34490E01,0.36460E01,0.34000E01,
   0.25900E01,0.18630E01,0.15810E01,0.16900E01,0.17710E01,0.22740E01,
   0.25760E01,0.31110E01,0.36050E01,0.35430E01,0.27690E01,0.20210E01,
   0.21850E01,0.25880E01,0.28800E01,0.31150E01,0.35400E01,0.38450E01,
   0.38000E01,0.35790E01,0.32640E01,0.25380E01,0.25820E01,0.29070E01,
   0.31420E01,0.34330E01,0.35800E01,0.34900E01,0.34750E01,0.35790E01,
   0.28290E01,0.19090E01,0.19030E01,0.20330E01,0.23600E01,0.26010E01,
   0.30540E01,0.33860E01,0.35530E01,0.34680E01,0.31870E01,0.27230E01,
   0.26860E01,0.28210E01,0.30000E01,0.32010E01,0.34240E01,0.35310E01};

  int n_obs = 114;
  float *parameters = NULL, *result = NULL;
  float res_sigma, aic;
  int *outlier_stat = NULL;
  int num_outliers;

  model[0] = 2;
  model[1] = 0;
  model[2] = 1;
  model[3] = 2;

  result = imsls_f_ts_outlier_identification(n_obs, model, series,
                                 IMSLS_CRITICAL, 3.5,
                                 IMSLS_NUM_OUTLIERS, &num_outliers,
                                 IMSLS_OUTLIER_STATISTICS, &outlier_stat,
                                 IMSLS_ARMA_PARAM, &parameters,
                                 IMSLS_RESIDUAL_SIGMA, &res_sigma,
                                 IMSLS_AIC, &aic,
                                 0);

  printf("Number of outliers: %d\n\n", num_outliers);
  printf("Outlier statistics:\n");
  printf("Time point\t\tOutlier type\n");
  for (i=0; i<num_outliers; i++)
     printf("%d\t\t%d\n", outlier_stat[2*i], outlier_stat[2*i+1]);
  printf("\n\n");
  printf("ARMA parameters:\n");
  for (i=0; i<=model[0]+model[1]; i++)
      printf("%d\t\t%lf\n", i, parameters[i]);

  printf("\n\n");
  printf("RSE:%lf\n", res_sigma);
  printf("\n\n");
  printf("AIC:%lf\n", aic);

  if (parameters)
  {
    free(parameters);
    parameters = NULL;
  }

  if (outlier_stat)
  {
    free(outlier_stat);
    outlier_stat = NULL;
  }

  if (result)
  {
    free(result);
    result = NULL;
  }

  return;
}
```

Output

```
ARMA parameters:
0               0.000000
1               0.123609
2               -0.178963

Number of outliers: 1

Outlier statistics:
Time point      Outlier type
16              2

RSE:0.319653
AIC:282.997314

Extract from the series:

time point      original series         outlier free series

1                 2.430000                2.430000
2                 2.506000                2.506000
3                 2.767000                2.767000
```

```
4                 2.940000                2.940000
5                 3.169000                3.169000
6                 3.450000                3.450000
7                 3.594000                3.594000
8                 3.774000                3.774000
9                 3.695000                3.695000
10                3.411000                3.411000
11                2.718000                2.718000
12                1.991000                1.991000
13                2.265000                2.265000
14                2.446000                2.446000
15                2.612000                2.612000
16                3.359000                2.702106
17                3.429000                2.772106
18                3.533000                2.876106
19                3.261000                2.604106
20                2.612000                1.955106
21                2.179000                1.522106
22                1.653000                0.996106
23                1.832000                1.175106
24                2.328000                1.671106
25                2.737000                2.080106
26                3.014000                2.357106
27                3.328000                2.671106
28                3.404000                2.747107
29                2.981000                2.324106
30                2.557000                1.900106
31                2.576000                1.919106
32                2.352000                1.695106
33                2.556000                1.899106
34                2.864000                2.207107
35                3.214000                2.557106
36                3.435000                2.778106
```

Example 2

This example is an artificial realization of an ARMA(1,1) process via formula $Y_t - 0.8Y_{t-1} = 10.0 + a_t + 0.5a_{t-1},\ t = 1,\ldots,300,\ \{a_t\}$ Gaussian white noise, $E[Y_t] = 50.0$.

An additive outlier with $\omega_1 = 4.5$ was added at time point $t = 150$, a temporary change outlier with $\omega_2 = 3.0$ was added at time point $t = 200$.

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

void main()
{

  int i, n_obs = 300;
  float parameters_user[300], result_user[300];
  float res_sigma, aic;
  int  outlier_stat[600];
  int num_outliers;
  int outlier_stat_user[300];
```

```
float omega_user[300];
int model[4];

float series[300]={
  50.0000000,50.2728081,50.6242599,51.0373917,51.9317627,50.3494759,
  51.6597252,52.7004929,53.5499802,53.1673279,50.2373505,49.3373871,
  49.5516472,48.6692696,47.6606636,46.8774185,45.7315445,45.6469727,
  45.9882355,45.5216560,46.0479660,48.1958656,48.6387749,49.9055367,
  49.8077278,47.7858467,47.9386749,49.7691956,48.5425873,49.1239853,
  49.8518791,50.3320694,50.9146347,51.8772049,51.8745689,52.3394470,
  52.7273712,51.4310036,50.6727448,50.8370399,51.2843437,51.8162918,
  51.6933670,49.7038231,49.0189247,49.455703,50.2718010,49.9605980,
  51.3775749,50.2285385,48.2692299,47.6495590,49.2938499,49.1924858,
  49.6449242,50.0446815,51.9972496,54.2576981,52.9835434,50.4193535,
  50.3617897,51.8276901,53.1239929,54.0682144,54.9238319,55.6877632,
  54.8896332,54.0701065,52.2754097,52.2522354,53.1248703,51.1287193,
  50.5003815,49.6504173,47.2453079,45.4555626,45.8449707,45.9765129,
  45.7682228,45.2343674,46.6496811,47.0894432,49.3368340,50.8058052,
  49.9132500,49.5893288,48.2470627,46.9779968,45.6760864,45.7070389,
  46.6158409,47.5303612,47.5630417,47.0389214,46.0352287,45.8161545,
  45.7974396,46.0015373,45.3796463,45.3461685,47.6444016,49.3327446,
  49.3810692,50.2027817,51.4567032,52.3986320,52.5819206,52.7721825,
  52.6919098,53.3274345,55.1345940,56.8962631,55.7791634,55.0616989,
  52.3551178,51.3264084,51.0968323,51.1980476,52.8001442,52.0545082,
  50.8742943,51.5150337,51.2242050,50.5033989,48.7760124,47.4179192,
  49.7319527,51.3320541,52.3918304,52.4140434,51.0845947,49.6485748,
  50.6893463,52.9840813,53.3246994,52.4568024,51.9196091,53.6683121,
  53.4555359,51.7755814,49.2915611,49.8755112,49.4546776,48.6171913,
  49.9643021,49.3766441,49.2551308,50.1021881,51.0769119,55.8328133,
  52.0212708,53.4930801,53.2147255,52.2356453,51.9648819,52.1816330,
  51.9898071,52.5623627,51.0717278,52.2431946,53.6943054,54.3752098,
  54.1492615,53.8523254,52.1093712,52.3982697,51.2405128,50.3018112,
  51.3819618,49.5479546,47.5024452,47.4447708,47.8939056,48.4070015,
  48.2440681,48.7389755,49.7309227,49.1998024,49.5798340,51.1196213,
  50.6288414,50.3971405,51.6084099,52.4564743,51.6443901,52.4080658,
  52.4643364,52.6257210,53.1604691,51.9309731,51.4137230,52.1233368,
  52.9867249,53.3180733,51.9647636,50.7947655,52.3815842,50.8353729,
  49.4136009,52.8355217,52.2234840,51.1392517,48.5245132,46.8700218,
  46.1607285,45.2324257,47.4157829,48.9989090,49.6230736,50.4352913,
  51.1652985,50.2588654,50.7820129,51.0448799,51.2880516,49.6898804,
  49.0288200,49.9338837,48.2214432,46.2103348,46.9550171,47.5595894,
  47.7176018,48.4502945,50.9816895,51.6950073,51.6973495,52.1941261,
  51.8988075,52.5617599,52.0218391,49.5236053,47.9684906,48.2445183,
  48.8275146,49.7176971,51.5649338,52.5627213,52.0182419,50.9688835,
  51.5846901,50.9486771,48.8685837,48.5600624,48.4760094,48.5348396,
  50.4187813,51.2542381,50.1872864,50.4407692,50.6222687,50.4972000,
  51.0036087,51.3367500,51.7368202,53.0463791,53.6261253,52.0728683,
  48.9740753,49.3280830,49.2733917,49.8519020,50.8562126,49.5594254,
  49.6109200,48.3785629,48.0026474,49.4874268,50.1596375,51.8059540,
  53.0288620,51.3321075,49.3114815,48.7999306,47.7201881,46.3433914,
  46.5303612,47.6294632,48.6012459,47.8567657,48.0604057,47.1352806,
  49.5724792,50.5566483,49.4182968,50.5578079,50.6883736,50.6333389,
  51.9766159,51.0595245,49.3751640,46.9667702,47.1658173,47.4411278,
  47.5360374,48.9914742,50.4747620,50.2728043,51.9117165,53.7627792};
  model[0] = 1;
  model[1] = 1;
  model[2] = 1;
  model[3] = 0;

  imsls_f_ts_outlier_identification(n_obs, model, series,
                            IMSLS_NUM_OUTLIERS, &num_outliers,
                            IMSLS_OUTLIER_STATISTICS_USER,
outlier_stat_user,
                            IMSLS_OMEGA_WEIGHTS_USER, omega_user,
                            IMSLS_ARMA_PARAM_USER, parameters_user,
                            IMSLS_RETURN_USER, result_user,
                            IMSLS_RESIDUAL_SIGMA, &res_sigma,
                            IMSLS_AIC, &aic,
                            IMSLS_RELATIVE_ERROR, 1.0e-05,
                            0);

   printf("\n");
   printf("ARMA parameters:\n");
   for (i=0; i<=model[0]+model[1]; i++)
      printf("%d\t\t%lf\n", i, parameters_user[i]);

   printf("\nNumber of outliers: %d\n\n", num_outliers);
   printf("Outlier statistics:\n");
   printf("Time point\tOutlier type\n");
   for (i=0; i<num_outliers; i++)
     printf("%d\t\t%d\n", outlier_stat_user[2*i], outlier_stat_user[2*i+1]);

  printf("\nOmega statistics:\n");
  printf("Time point\tomega\n");
  for (i=0; i<num_outliers; i++)
     printf("%d\t%18.6f\n", outlier_stat_user[2*i], omega_user[i]);

  printf("\n");
  printf("RSE:%lf\n", res_sigma);
  printf("AIC:%lf\n\n", aic);

  return;
}
```

Output

```
ARMA parameters:
0               10.808282
1               0.785631
2               -0.496392

Number of outliers: 2

Outlier statistics:
Time point      Outlier type
150             1
200             3

Omega statistics:
```

```
Time point      omega
150               4.477811
200               3.382051

RSE:1.007220
AIC:1417.042480
```

ts_outlier_forecast

Computes forecasts, their associated probability limits and ψ weights for an outlier contaminated time series whose underlying outlier free series follows a general seasonal or nonseasonal ARMA model.

Synopsis

#include <imsls.h>

float *imsls_f_ts_outlier_forecast (*int* n_obs, *float* series[],
int num_outliers, *int* outlier_statistics[], *float* omega[],
float delta, *int* model[], *float* parameters[], *int* n_predict,…,0)

The type *double* function is imsls_d_ts_outlier_forecast.

Required Arguments

int n_obs (Input)
Number of observations in the time series.

float series[] (Input)
An array of length n_obs by 2 containing the outlier free time series in its first column and the residuals of the series in the second column.

int num_outliers (Input)
Number of detected outliers in the original outlier contaminated series as computed in imsls_f_ts_outlier_identification.

int outlier_statistics[] (Input)
An array of length num_outliers by 2 containing the outlier statistics from imsls_f_ts_outlier_identification. If num_outliers=0, this array is ignored.

float omega[] (Input)
Array of length num_outliers containing the ψ weights for the outliers determined in imsls_f_ts_outlier_identification. Ignored, if num_outliers=0.

float delta (Input)
The dynamic dampening effect parameter used in the outlier detection.

int model[] (Input)
Vector of length 4 containing the numbers p, q, s, d of the ARIMA $(p,0,q)\times(0,d,0)_s$ model the outlier free series is following.

float `parameters[]` (Input)

Vector of length 1+p+q containing the estimated constant, AR and MA parameters as output from `imsls_f_ts_outlier_identification`.

int `n_predict` (Input)

Maximum lead time for forecasts. The forecasts are taken at origin t=`n_obs`, the time point of the last observed value, for lead times 1,2,...,`n_predict`.

Return Value

Pointer to an array of length `n_predict` by 3. The first column contains the forecasted values for the original outlier contaminated series. The second column contains the deviations from each forecast for computing confidence probability limits, and the third column contains the ψ weights of the infinite moving average form of the model.

If an error occurred, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_ts_outlier_forecast`(*int* `n_obs`, *float* `series[]`,
int `num_outliers`, *int* `outlier_statistics[]`,
float `omega[]`, *float* `delta`, *int* `model[]`,
float `parameters[]`, *int* `n_predict`,
`IMSLS_RETURN_USER`, *float* `forecast[]`,
`IMSLS_CONFIDENCE`, *float* `confidence`,
`IMSLS_OUT_FREE_FORECAST`, *float* `**outfree_forecast`,
`IMSLS_OUT_FREE_FORECAST_USER`, *float* `outfree_forecast[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `forecast[]` (Output)

An array of length `n_predict` by 3 supplied by the user containing the forecasts for the original outlier contaminated series in column 1, deviations from each forecast in column 2 and the ψ weights of the infinite moving average form of the model in column 3.

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)

Value in the exclusive interval (0,100) used to specify the `confidence` percent probability limits of the forecast.Typical choices for `confidence` are 90.0, 95.0 and 99.0.
Default: `confidence` = 95.0

`IMSLS_OUT_FREE_FORECAST`, *float* `**outfree_forecast` (Output)

Address of a pointer to an array of length `n_predict` by 3 containing the forecasts for the original outlier free series in column 1, deviations from each forecast in column 2 and the ψ weights of the infinite moving average form of the model in column 3.

`IMSLS_OUT_FREE_FORECAST_USER`, *float* `outfree_forecast[]` Output)

Storage for array `outfree_forecast` is provided by the user. For a description, see `IMSLS_OUT_FREE_FORECAST`.

Description

Consider the following model for a given outlier contaminated univariate time series $\{Y_t^*\}_{t=1,\ldots,n}$:

$$Y_t^* = Y_t + \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j).$$

For an explanation of the notation, see the "Description" section for `imsls_f_ts_outlier_identification`. It follows from the formula above that the Box-Jenkins forecast at origin t for lead time l, $\hat{Y}_t^*(l)$, can be computed as:

$$\hat{Y}_t^*(l) = \hat{Y}_t(l) + \sum_{j=1}^{m} \omega_j L_j(B) I_{t+l}(t_j), \quad l = 1,\ldots,\texttt{n\_predict}.$$

Therefore, computation of the forecasts for $\{Y_t^*\}$ is done in two steps:

1. Computation of the forecasts for the outlier free series $\{Y_t\}$.
2. Computation of the forecasts for the original series $\{Y_t^*\}$ by adding the multiple outlier effects to the forecasts for $\{Y_t\}$.

Step 1 above:

Since

$$\varphi(B)(Y_t - \mu) = \theta(B) a_t,$$

where

$$\varphi(B) := \Delta_s^d \phi(B) = 1 - \varphi_1 B - \ldots - \varphi_{p+sd} B^{p+sd},$$

the Box-Jenkins forecast at origin t for lead time l, $\hat{Y}_t(l)$, can be computed recursively as:

$$\hat{Y}_t(l) = (1 - \sum_{j=1}^{p+sd} \varphi_j)\mu + \sum_{j=1}^{p+sd} \varphi_j \hat{Y}_t(l-j) - \sum_{j=l}^{q} \theta_j a_{t+l-j}.$$

Here,

$$\hat{Y}_t(l-j) = \begin{cases} Y_{t+l-j} & \text{for} \quad l-j \le 0 \\ \hat{Y}_t(l-j) & \text{for} \quad l-j > 0 \end{cases},$$

and

$$a_k = \begin{cases} 0 & \text{for} \quad k \le \max\{1, p+sd\} \\ Y_k - \hat{Y}_{k-1}(1) & \text{for} \quad k = \max\{1, p+sd\}+1,\ldots,n \end{cases}.$$

Step 2 above:

The formulas for $L_j(B)$ for the different types of outliers are as follows:

Innovational outliers (IO)
$$L_j(B) = \frac{\theta(B)}{\Delta_s^d \phi(B)} := \psi(B) = \sum_{k=0}^{\infty} \psi_k B^k,\ \psi_0 = 1$$

Additive outliers (AO)
$$L_j(B) = 1$$

Level shifts (LS)
$$L_j(B) = \frac{1}{1-B} = \sum_{k=0}^{\infty} B^k$$

Temporary changes (TC)
$$L_j(B) = \frac{1}{1-\delta B} = \sum_{k=0}^{\infty} \delta^k B^k$$

Assuming the outlier occurs at time point t_j, the outlier impact is therefore:

Innovational outliers (IO)
$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j, \\ \omega_j \psi_k & \text{for} \quad t = t_j + k, k \ge 0, \end{cases}$$

Additive outliers (AO)
$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t \ne t_j, \\ \omega_j & \text{for} \quad t = t_j, \end{cases}$$

Level shifts (LS)
$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j, \\ \omega_j & \text{for} \quad t = t_j + k, k \ge 0, \end{cases}$$

Temporary changes (TC)
$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j, \\ \omega_j \delta^k & \text{for} \quad t = t_j + k, k \ge 0. \end{cases}$$

From these formulas, the forecasts $\hat{Y}_t^*(l)$ can be computed easily.

The $100(1-\alpha)$ percent probability limits for Y_{t+l}^* and Y_{t+l} are given by

$$\hat{Y}_t^*(l) \text{ (or } \hat{Y}_t(l), \text{ resp.)} \pm u_{\alpha/2}(1+\sum_{j=1}^{l-1}\psi_j^2)^{1/2} s_a,$$

where $u_{\alpha/2}$ is the $100(1-\alpha/2)$ percentile of the standard normal distribution, s_a^2 is an estimate of the variance σ_a^2 of the random shocks (returned from imsls_f_ts_outlier_identification), and the ψ weights $\{\psi_j\}$ are the coefficients in

$$\psi(B) := \sum\nolimits_{k=0}^{\infty} \psi_k B^k := \frac{\theta(B)}{\Delta_s^d \phi(B)},\ \psi_0 = 1.$$

For a detailed explanation of these concepts, see Chapter 5: “Forecasting,” Box, Jenkins and Reinsel (1994).

Example

This example is a realization of an ARMA(2,1) process described by the model $Y_t - Y_{t-1} + 0.24Y_{t-2} = 10.0 + a_t + 0.5\,a_{t-1}$, $\{a_t\}$ a Gaussian white noise process.

Outliers were artificially added to the outlier free series $\{Y_t\}_{t=1,\ldots,280}$ at time points $t = 150$ (level shift, $\omega_1 = +2.5$) and $t = 200$ (additive outlier, $\omega_2 = +3.2$), resulting in the outlier contaminated series $\{Z_t\}_{t=1,\ldots,280}$. For both series, forecasts were determined for time points $t = 281,\ldots,290$ and compared with the actual values of the series.

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

void main()
{
  float time_series[290] ={
    41.6699982,41.6699982,42.0752144,42.6123962,43.6161919,42.1932831,
    43.1055450,44.3518715,45.3961258,45.0790215,41.8874397,40.2159805,
    40.2447319,39.6208458,38.6873589,37.9272423,36.8718872,36.8310852,
    37.4524879,37.3440933,37.9861374,40.3810501,41.3464622,42.6495285,
    42.6096764,40.3134537,39.7971268,41.5401535,40.7160759,41.0363541,
    41.8171883,42.4190292,43.0318832,43.9968109,44.0419617,44.3225212,
    44.6082611,43.2199631,42.0419197,41.9679718,42.4926224,43.2091255,
    43.2512283,41.2301674,40.1057358,40.4510574,41.5329170,41.5678177,
    43.0090141,42.1592140,39.9234505,38.8394127,40.4319878,40.8679352,
    41.4551926,41.9756317,43.9878922,46.5736389,45.5939293,42.4487762,
    41.5325394,42.8830910,44.5771217,45.8541985,46.8249474,47.5686378,
    46.6700745,45.4120026,43.2305107,42.7635345,43.7112923,42.0768661,
    41.1835632,40.3352280,37.9761467,35.9550056,36.3212509,36.9925880,
    37.2625008,37.0040665,38.5232544,39.4119797,41.8316803,43.7091446,
    42.9381447,42.1066780,40.3771248,38.6518707,37.0550499,36.9447708,
    38.1017685,39.4727097,39.8670387,39.3820763,38.2180786,37.7543488,
    37.7265244,38.0290642,37.5531158,37.4685936,39.8233147,42.0480766,
    42.4053535,43.0117416,44.1289330,45.0393829,45.1114540,45.0086479,
    44.6560631,45.0278931,46.7830849,48.7649765,47.7991905,46.5339661,
    43.3679199,41.6420822,41.2694893,41.5959740,43.5330009,43.3643608,
    42.147129  1,42.5552788,42.4521446,41.7629128,39.9476891,38.3217010,
    40.5318718,42.8811569,44.4796944,44.6887932,43.1670265,41.2226143,
    41.8330154,44.3721924,45.2697029,44.4174194,43.5068550,44.9793015,
    45.0585403,43.2746620,40.3317070,40.3880501,40.2627106,39.6230278,
    41.0305252,40.9262009,40.8326912,41.7084885,42.9038048,45.8650513,
    46.5231590,47.9916115,47.8463135,46.5921936,45.8854408,45.9130440,
    45.7450371,46.2964249,44.9394569,45.8141251,47.5284042,48.5527802,
    48.3950577,47.8753052,45.8880005,45.7086983,44.6174774,43.5567932,
    44.5891113,43.1778679,40.9405632,40.6206894,41.3330421,42.2759552,
    42.4744949,43.0719833,44.2178459,43.8956337,44.1033440,45.6241455,
    45.3724861,44.9167595,45.9180603,46.9077835,46.1666603,46.6013489,
    46.6592331,46.7291603,47.1908340,45.9784355,45.1215782,45.6791115,
    46.7379875,47.3036957,45.9968834,44.4669495,45.7734680,44.6315041,
    42.9911766,46.3842583,43.7214432,43.5276833,41.3946495,39.7013168,
    39.1033401,38.5292892,41.0096245,43.4535828,44.6525154,45.5725899,
    46.2815285,45.2766647,45.3481712,45.5039482,45.6745682,44.0144806,
    42.9305000,43.6785469,42.2500534,40.0007210,40.4477005,41.4432716,
    42.0058670,42.9357758,45.6758842,46.8809929,46.8601494,47.0449791,
    46.5420647,46.8939934,46.2963371,43.5479164,41.3864059,41.4046364,
    42.3037987,43.6223717,45.8602371,47.3016396,46.8632469,45.4651413,
    45.6275482,44.9968376,42.7558670,42.0218239,41.9883728,42.2571678,
    44.3708687,45.7483635,44.8832512,44.7945862,44.8922577,44.7409401,
    45.1726494,45.5686874,45.9946709,47.3151054,48.0654068,46.4817467,
    42.8618279,42.4550323,42.5791168,43.4230957,44.7787971,43.8317108,
    43.6481781,42.4183960,41.8426285,43.3475227,44.4749908,46.3498306,
    47.8599319,46.2449913,43.6044006,42.4563484,41.2715340,39.8492508,
    39.9997292,41.4410820,42.9388237,42.5687332,42.6384087,41.7088661,
    43.9399033,45.4284401,44.4558411,45.1761856,45.3489113,45.1892662,
    46.3754730,45.6082802 };

  int n_obs = 280, i;
  float *parameters = NULL, *result = NULL, *forecast = NULL;
  float *outfree_forecast = NULL, *omega = NULL, *residual = NULL;
  float res_sigma, aic;
  float delta = 0.7;
  float series[560];
  int *outlier_stat = NULL;
  int num_outliers;
  int n_predict = 10;
  int model[4];
  float forecast_table[40];

  model[0] = 2;
  model[1] = 1;
  model[2] = 1;
  model[3] = 0;

  result = imsls_f_ts_outlier_identification(n_obs, model,
                            time_series,
                            IMSLS_RELATIVE_ERROR, 1.0e-5,
                            IMSLS_NUM_OUTLIERS, &num_outliers,
                            IMSLS_RESIDUAL, &residual,
                            IMSLS_OUTLIER_STATISTICS, &outlier_stat,
                            IMSLS_OMEGA_WEIGHTS, &omega,
                            IMSLS_ARMA_PARAM, &parameters,
                            IMSLS_RESIDUAL_SIGMA, &res_sigma,
                            IMSLS_AIC, &aic,
                            0);

  printf("\nARMA parameters:\n");
  for (i=0; i<=model[0]+model[1]; i++)
      printf("%d\t\t%lf\n", i, parameters[i]);

  printf("\nNumber of outliers: %d\n\n", num_outliers);
  printf("Outlier statistics:\n");
```

```
printf("Time point\t\tOutlier type\n");
for (i=0; i<num_outliers; i++)
   printf("%d\t\t%d\n", outlier_stat[2*i], outlier_stat[2*i+1]);

printf("\n");
printf("RSE:%lf\n", res_sigma);
printf("AIC:%lf\n", aic);

for (i=0; i<n_obs; i++)
 {
    series[2*i] = time_series[i];
    series[2*i+1] = residual[i];
 }

forecast = imsls_f_ts_outlier_forecast(n_obs, series,
                num_outliers, outlier_stat, omega, delta,
                model, parameters, n_predict,
                IMSLS_OUT_FREE_FORECAST,&outfree_forecast, 0);

for (i=0; i<n_predict; i++)
{
   forecast_table[4*i] = time_series[n_obs+i];
   forecast_table[4*i+1] = forecast[3*i];
   forecast_table[4*i+2] = forecast[3*i+1];
   forecast_table[4*i+3] = forecast[3*i+2];
}

imsls_f_write_matrix("\t* * * Forecast Table for outlier"
                     "contaminated series * * *\nOrig. Series"
                     "\tforecast\tprob. limits\tpsi weights\n",
                     n_predict, 4, forecast_table,
                     IMSLS_WRITE_FORMAT, "%11.4f", 0);

for (i=0; i<n_predict; i++)
{
   forecast_table[4*i] = time_series[n_obs+i] - 2.5;
   forecast_table[4*i+1] = outfree_forecast[3*i];
   forecast_table[4*i+2] = outfree_forecast[3*i+1];
   forecast_table[4*i+3] = outfree_forecast[3*i+2];
}

printf("\n");
imsls_f_write_matrix("\t* * * Forecast Table for outlier free"
                     "series * * *\n\nOutlier free series\tforecast"
                     "\tprob. limits\tpsi weights\n",
                     n_predict, 4, forecast_table,
                     IMSLS_WRITE_FORMAT, "%11.4f", 0);

if (parameters)
{
  free(parameters);
  parameters = NULL;
}

if (outlier_stat)
```

```
  {
    free(outlier_stat);
    outlier_stat = NULL;
  }

  if (result)
  {
    free(result);
    result = NULL;
  }

  if (forecast)
  {
    free(forecast);
    forecast = NULL;
  }

  if (outfree_forecast)
  {
    free(outfree_forecast);
    outfree_forecast = NULL;
  }

  if (omega)
  {
    free(omega);
    omega = NULL;
  }

  if (residual)
  {
    free(residual);
    residual = NULL;
  }

  return;
}
```

Output

```
ARMA parameters:
0              8.839014
1              0.948735
2              -0.153870
3              -0.553387

Number of outliers: 2

Outlier statistics:
Time point              Outlier type
150            2
200            1

RSE:1.004321
AIC:1323.625977
```

```
 * * * Forecast Table for outlier contaminated series * * *

    Orig. series    forecast      prob. limits    psi weights

              1           2             3           4
 1      42.6384     43.6883       1.9684      1.5021
 2      41.7089     43.8260       3.5521      1.2712
 3      43.9399     44.0496       4.3450      0.9749
 4      45.4284     44.2406       4.7500      0.7294
 5      44.4558     44.3874       4.9622      0.5420
 6      45.1762     44.4973       5.0756      0.4019
 7      45.3489     44.5790       5.1369      0.2979
 8      45.1893     44.6395       5.1703      0.2208
 9      46.3755     44.6844       5.1885      0.1637
10      45.6083     44.7177       5.1985      0.1213


 * * * Forecast Table for outlier free series * * *

  Outlier free series    forecast   prob. limits    psi weights

                1           2             3           4
   1      40.1384     41.9641       1.9684      1.5021
   2      39.2089     42.1018       3.5521      1.2712
   3      41.4399     42.3254       4.3450      0.9749
   4      42.9284     42.5164       4.7500      0.7294
   5      41.9558     42.6632       4.9622      0.5420
   6      42.6762     42.7731       5.0756      0.4019
   7      42.8489     42.8548       5.1369      0.2979
   8      42.6893     42.9153       5.1703      0.2208
   9      43.8755     42.9602       5.1885      0.1637
  10      43.1083     42.9935       5.1985      0.1213
```

auto_arima

Automatically identifies time series outliers, determines parameters of a multiplicative seasonal ARIMA $(p,0,q)\times(0,d,0)_s$ model and produces forecasts that incorporate the effects of outliers whose effects persist beyond the end of the series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_auto_arima` (*int* `n_obs`, *int* `tpoints[]`, *float* `x[]`,...,0)

The type *double* function is `imsls_d_auto_arima`.

Required Arguments

int `n_obs` (Input)

Number of observations in the original time series. Assuming that the series is defined at time points $t_1,\ldots,t_{\text{n\_obs}}$, the actual length of the series, including missing observations is $n = t_{\text{n\_obs}} - t_1 + 1$.

int `tpoints[]` (Input)

A vector of length `n_obs` containing the time points $t_1, t_2, \ldots t_{n\_obs}$ the time series was observed. It is required that $t_1, t_2, \ldots t_{n\_obs}$ are in strictly ascending order.

float `x[]` (Input)

A vector of length `n_obs` containing the observed time series values $Y_1^*, Y_2^*, \cdots, Y_{n\_obs}^*$. This series can contain outliers and missing observations. Outliers are identified by this routine and missing values are identified by the time values in vector `tpoints`. If the time interval between two consecutive time points is greater than one, i.e. $t_{i+1} - t_i = m > 1$, then $m-1$ missing values are assumed to exist between t_i and t_{i+1} at times $t_i + 1, t_i + 2, \ldots, t_{i+1} - 1$. Therefore, the gap free series is assumed to be defined for equidistant time points $t_1, t_1 + 1, \ldots, t_{n\_obs}$. Missing values are automatically estimated prior to identifying outliers and producing forecasts. Forecasts are generated for both missing and observed values.

Return Value

Pointer to an array of length 1 + *p* + *q* with the estimated constant, AR and MA parameters used to fit the outlier-free series using an ARIMA $(p,0,q) \times (0,d,0)_s$ model. Upon completion, if *d*=`model[3]`=0, then an ARMA(*p, q*) model or AR(*p*) model is fitted to the outlier-free version of the observed series Y_t^*. If *d*=`model[3]`>0, these parameters are computed for an ARMA(*p*,*q*) representation of the seasonally adjusted series $Z_t^* = \Delta_s^d \cdot Y_t^* = (1 - B_s)^d \cdot Y_t^*$, where $B_s Y_t^* = Y_{t-s}^*$ and *s*=`model[2]`≥1.
If an error occurred, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_auto_arima(`*int* `n_obs`, *int* `tpoints[]`, *float* `x[]`,
    `IMSLS_METHOD`, *int* `method`,
    `IMSLS_MAX_LAG`, *int* `maxlag`,
    `IMSLS_MODEL`, *int* `model[]`,
    `IMSLS_DELTA`, *float* `delta`,
    `IMSLS_CRITICAL`, *float* `critical`,
    `IMSLS_EPSILON`, *float* `epsilon`,
    `IMSLS_RESIDUAL`, *float* `**residual`,
    `IMSLS_RESIDUAL_USER`, *float* `residual[]`,
    `IMSLS_RESIDUAL_SIGMA`, *float* `*res_sigma`,
    `IMSLS_NUM_OUTLIERS`, *int* `*num_outliers`,
    `IMSLS_P_INITIAL`, *int* `n_p_initial`, *int* `p_initial[]`,
    `IMSLS_Q_INITIAL`, *int* `n_q_initial`, *int* `q_initial[]`,
    `IMSLS_S_INITIAL`, *int* `n_s_initial`, *int* `s_initial[]`,
    `IMSLS_D_INITIAL`, *int* `n_d_initial`, *int* `d_initial[]`,
    `IMSLS_OUTLIER_STATISTICS`, *int* `**outlier_stat`,
    `IMSLS_OUTLIER_STATISTICS_USER`, *int* `outlier_stat[]`,
    `IMSLS_AIC`, *float* `*aic`,

IMSLS_OUT_FREE_SERIES, *float* **outfree_series,
IMSLS_OUT_FREE_SERIES_USER, *float* outfree_series[],
IMSLS_CONFIDENCE, *float* confidence,
IMSLS_NUM_PREDICT, *int* n_predict,
IMSLS_OUT_FREE_FORECAST, *float* **outfree_forecast,
IMSLS_OUT_FREE_FORECAST_USER, *float* outfree_forecast[],
IMSLS_OUTLIER_FORECAST, *float* **outlier_forecast,
IMSLS_OUTLIER_FORECAST_USER, *float* outlier_forecast[],
IMSLS_RETURN_USER, *float* parameters[],
0)

Optional Arguments

IMSLS_METHOD, *int* method (Input)
The method used in model selection:
1 — Automatic ARIMA $(p,0,0)\times(0,d,0)_s$ selection
2 — Grid search (Requires arguments IMSLS_P_INITIAL and IMSLS_Q_INITIAL.)
3 — Specified ARIMA $(p,0,q)\times(0,d,0)_s$ model (Requires argument IMSLS_MODEL.)
Default: method = 1
For more information, see the "Description" section.

IMSLS_MAX_LAG, *int* maxlag (Input)
The maximum lag allowed when fitting an AR(*p*) model.
Default: maxlag = 10

IMSLS_MODEL, *int* model[] (Input/Output)
Array of length 4 containing the values for *p, q, s, d*. If method=3 is chosen, then the values for *p* and *q* must be defined. If IMSLS_S_INITIAL and IMSLS_D_INITIAL are not defined, then also *s* and *d* must be given. If method=1 or method=2, then model is ignored as an input array. On output, model contains the optimum values for *p, q, s, d* in model[0], model[1], model[2] and model[3], respectively.

IMSLS_DELTA, *float* delta (Input)
The dampening effect parameter used in the detection of a Temporary Change Outlier (TC), 0<delta<1.
Default: delta = 0.7

IMSLS_CRITICAL, *float* critical (Input)
Critical value used as a threshold for outlier detection, critical > 0.
Default: critical = 3.0

IMSLS_EPSILON, *float* epsilon (Input)
Positive tolerance value controlling the accuracy of parameter estimates during outlier detection.
Default: epsilon = 0.001

IMSLS_RESIDUAL, *float* **residual (Output)
Address of a pointer to an internally allocated array of length

$n = t_{n\_obs} - t_1 + 1 \geq$ n_obs , containing $\hat{e}_t$, the estimates of the white noise in the outlier free original series.

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

IMSLS_RESIDUAL_SIGMA, *float* *res_sigma (Output)
Residual standard error (RSE) of the outlier free original series.

IMSLS_NUM_OUTLIERS, *int* *num_outliers (Output)
The number of outliers detected.

IMSLS_P_INITIAL, *int* n_p_initial, *int* p_initial[] (Input)
An array with n_p_initial elements containing the candidate values for *p*, from which the optimum is being selected. All candidate values in p_initial[] must be non-negative and n_p_initial ≥ 1. If method=2, then IMSLS_P_INITIAL must be defined. Otherwise, n_p_initial and p_initial are ignored.

IMSLS_Q_INITIAL, *int* n_q_initial, *int* q_initial[] (Input)
An array with n_q_initial elements containing the candidate values for *q*, from which the optimum is being selected. All candidate values in q_initial[] must be non-negative and n_q_initial ≥ 1. If method=2, then IMSLS_Q_INITIAL must be defined. Otherwise, n_q_initial and q_initial are ignored.

IMSLS_S_INITIAL, *int* n_s_initial, *int* s_initial[] (Input)
A vector of length n_s_initial containing the candidate values for s, from which the optimum is being selected. All candidate values in s_initial[] must be positive and n_s_initial ≥ 1.
Default: n_s_initial=1, s_initial={1}

IMSLS_D_INITIAL, *int* n_d_initial, *int* d_initial[] (Input)
A vector of length n_d_initial containing the candidate values for *d*, from which the optimum is being selected. All candidate values in d_initial[] must be non-negative and n_d_initial ≥ 1.
Default: n_d_initial=1, d_initial={0}

IMSLS_OUTLIER_STATISTICS, *int* **outlier_stat (Output)
Address of a pointer to an internally allocated array of length num_outliers by 2 containing outlier statistics. The first column contains the time at which the outlier was observed ($t = t_1, t_1 + 1, t_1 + 2, \ldots, t_{\text{n\_obs}}$) and the second column contains an identifier indicating the type of outlier observed. Outlier types fall into one of five categories:

0	Innovational Outliers (IO)
1	Additive Outliers (AO)
2	Level Shift Outliers (LS)
3	Temporary Change Outliers (TC)
4	Unable to Identify (UI).

If `num_outliers=0`, `NULL` is returned.

`IMSLS_OUTLIER_STATISTICS_USER`, *int* `outlier_stat[]` (Output)
A user allocated array of length $n \times 2$ containing outlier statistics in its first `num_outliers` rows. Here, $n = t_{\text{n\_obs}} - t_1 + 1 \geq \text{n\_obs}$.
See `IMSLS_OUTLIER_STATISTICS`.
If `num_outliers` = 0, `outlier_stat` stays unchanged.

`IMSLS_AIC`, *float* `*aic` (Output)
Akaike's information criterion (AIC) for the optimum model.

`IMSLS_OUT_FREE_SERIES`, *float* `**outfree_series` (Output)
Address of a pointer to an internally allocated array of length *n* by 2, where $n = t_{\text{n\_obs}} - t_1 + 1$. The first column of `outfree_series` contains the `n_obs` observations from the original series, Y_t^*, plus estimated values for any time gaps. The second column contains the same values as the first column adjusted by removing any outlier effects. In effect, the second column contains estimates of the underlying outlier-free series, Y_t. If no outliers are detected then both columns will contain identical values.

`IMSLS_OUT_FREE_SERIES_USER`, *float* `outfree_series[]` (Output)
A user allocated array of length *n* by 2, where $n = t_{\text{n\_obs}} - t_1 + 1$. For further details, see `IMSLS_OUT_FREE_SERIES`.

`IMSLS_CONFIDENCE`, *float* `confidence` (Input)
Confidence level for computing forecast confidence limits, taken from the exclusive interval (0, 100). Typical choices for `confidence` are 90.0, 95.0 and 99.0.
Default: `confidence` = 95.0

`IMSLS_NUM_PREDICT`, *int* `n_predict` (Input)
The number of forecasts requested. Forecasts are made at origin $t_{\text{n\_obs}}$, i.e. from the last observed value of the series.
Default: `n_predict` = 0

`IMSLS_OUT_FREE_FORECAST`, *float* `**outfree_forecast` (Output)
Address of a pointer to an internally allocated array of length `n_predict` by 3. The first column contains the forecasted values for the original outlier free series for $t = t_{\text{n\_obs}}+1$, $t_{n\_obs} + 2, \ldots, t_{\text{n\_obs}} +$ `n_predict`. The second column contains standard errors for these forecasts, and the third column contains the psi weights of the infinite order moving average form of the model.

`IMSLS_OUT_FREE_FORECAST_USER`, *float* `outfree_forecast[]` (Output)
A user allocated array of length `n_predict` by 3. For more information, see `IMSLS_OUT_FREE_FORECAST`.

`IMSLS_OUTLIER_FORECAST`, *float* `**outlier_forecast` (Output)
Address of a pointer to an internally allocated array of length `n_predict` by 3. The first column contains the forecasted values for the original series for $t = t_{\text{n\_obs}}+1,\ t_{\text{n\_obs}}+2, \ldots,\ t_{\text{n\_obs}}+\text{n\_predict}$. The second column contains standard errors for these forecasts, and the third column contains the ψ weights of the infinite order moving average form of the model.

`IMSLS_OUTLIER_FORECAST_USER`, *float* `outlier_forecast[]` (Output)
A user allocated array of length `n_predict` by `3`. For more information, see `IMSLS_OUTLIER_FORECAST`.

`IMSLS_RETURN_USER`, *float* `parameters[]` (Output)
A user allocated array containing the estimated constant, AR and MA parameters in its first $1+p+q$ locations. The values p and q can be estimated by upper bounds: If `method=1`, an upper bound for p would be `maxlag`, and $q = 0$. If `method=2`, upper bounds for p and q would be the maximum values in arrays `p_initial` and `q_initial`, respectively. If `method=3`, $p=$ `model[0]` and $q=$ `model[1]`.

Description

Overview

Function `imsls_f_auto_arima` determines the parameters of a multiplicative seasonal ARIMA $(p,0,q)\times(0,d,0)_s$ model, and then uses the fitted model to identify outliers and prepare forecasts. The order of this model can be specified or automatically determined.
The ARIMA $(p,0,q)\times(0,d,0)_s$ model handled by `imsls_f_auto_arima` has the following form:

$$\phi(B)\Delta_s^d(Y_t-\mu)=\theta(B)a_t, \quad t=1,2,\ldots,n,$$

where

$$\phi(B)=1-\phi_1B-\phi_2B^2-\cdots-\phi_pB^p\,,\quad \theta(B)=1-\theta_1B-\theta_2B^2-\cdots-\theta_qB^q\,,\quad \Delta_s^d=(1-B^s)^d$$

and

$$B^kY_t=Y_{t-k}\,.$$

It is assumed that all roots of $\phi(B)$ and $\theta(B)$ lie outside the unit circle. Clearly, if $s=1$ this reduces to the traditional ARIMA(p, d, q) model.

Y_t is the unobserved, outlier-free time series with mean μ, and white noise a_t. This model is referred to as the underlying, outlier-free model. Function `imsls_f_auto_arima` does not assume that this series is observable. It assumes that the observed values might be contaminated by one or more outliers, whose effects are added to the underlying outlier-free series:

$$Y_t^* = Y_t + outlier\_effect_t .$$

Outlier identification uses the algorithm developed by Chen and Liu (1993). Outliers are classified into 1 of 5 types:

0. innovational
1. additive
2. level shift
3. temporary change *and*
4. unable to identify

Once outliers are identified, imsls_f_auto_arima estimates Y_t, the outlier-free series representation of the data, by removing the estimated outlier effects.

Using the information about the adjusted ARIMA $(p,0,q)\times(0,d,0)_s$ model and the removed outliers, forecasts are then prepared for the outlier-free series. Outlier effects are added to these forecasts to produce a forecast for the observed series, Y_t^*. If there are no outliers, then the forecasts for the outlier-free series and the observed series will be identical.

Model Selection

Users have an option of either specifying specific values for *p, q , s* and *d* or have imsls_f_auto_arima automatically select best fit values. Model selection can be conducted in one of three methods listed below depending upon the value of variable `method`.

Method 1: Automatic ARIMA $(p,0,0)\times(0,d,0)_s$ Selection

This method initially searches for the AR(p) representation with minimum AIC for the noisy data, where *p* =0,...,`maxlag`.

If `IMSLS_D_INITIAL` is defined then the values in `s_initial` and `d_initial` are included in the search to find an optimum ARIMA $(p,0,0)\times(0,d,0)_s$ representation of the series. Here, every possible combination of values for *p, s* in `s_initial` and *d* in `d_initial` is examined. The best found ARIMA $(p,0,0)\times(0,d,0)_s$ representation is then used as input for the outlier detection routine.

The optimum values for *p, q, s* and *d* are returned in `model[0]`, `model[1]`, `model[2]` and `model[3]`, respectively.

Method 2: Grid Search

The second automatic method conducts a grid search for *p* and *q* using all possible combinations of candidate values in `p_initial` and `q_initial`. Therefore, for this method the definition of `IMSLS_P_INITIAL` and `IMSLS_Q_INITIAL` is required.

If `IMSLS_D_INITIAL` is defined, the grid search is extended to include the candidate values for `s` and `d` given in `s_initial` and `d_initial`, respectively.

If `IMSLS_D_INITIAL` is not defined, no seasonal adjustment is attempted, and the grid search is restricted to searching for optimum values of *p* and *q* only.

The optimum values of *p, q, s* and *d* are returned in `model[0]`, `model[1]`, `model[2]` and `model[3]`, respectively.

Method 3: Specified ARIMA $(p,0,q)\times(0,d,0)_s$ Model

In the third method, specific values for *p*, *q*, *s* and *d* are given. The values for *p* and *q* must be defined in `model[0]` and `model[1]`, respectively. If `IMSLS_S_INITIAL` and `IMSLS_D_INITIAL` are not defined, then values $s > 0$ and $d \geq 0$ must be specified in `model[2]` and `model[3]`. If `IMSLS_S_INITIAL` and `IMSLS_D_INITIAL` are defined, then a grid search for the optimum values of *s* and *d* is conducted using all possible combinations of input values in `s_initial` and `d_initial`. The optimum values of *s* and *d* can be found in `model[2]` and `model[3]`, respectively.

Outliers

The algorithm of Chen and Liu (1993) is used to identify outliers. The number of outliers identified is returned in `num_outliers`. Both the time and classification for these outliers are returned in `outlier_stat[]`. Outliers are classified into one of five categories based upon the standardized statistic for each outlier type. The time at which the outlier occurred is given in the first column of `outlier_stat`. The outlier identifier returned in the second column is according to the descriptions in the following table:

Outlier Identifier	**Name**	**General Description**
0	(IO) Innovational Outlier	Innovational outliers persist. That is, there is an initial impact at the time the outlier occurs. This effect continues in a lagged fashion with all future observations. The lag coefficients are determined by the coefficient of the underlying ARIMA $(p,0,q)\times(0,d,0)_s$ model.
1	(AO) Additive Outlier	Additive outliers do not persist. As the name implies, an additive outlier effects only the observation at the time the outlier occurs. Hence additive outliers have no effect on future forecasts.
2	(LS) Level Shift	Level shift outliers persist. They have the effect of either raising or lowering the mean of the series starting at the time the outlier occurs. This shift in the mean is abrupt and permanent.
3	(TC) Temporary Change	Temporary change outliers persist and are similar to level shift outliers with one major exception. Like level shift outliers, there is an abrupt change in the mean of the series at the time this outlier occurs. However, unlike level shift outliers, this shift is not permanent. The TC outlier gradually decays, eventually bringing the mean of the series back to its original value. The rate of this decay is modeled using the parameter `delta`. The default of `delta=` 0.7 is the value recommended for general use by Chen and Liu (1993).
4	(UI) Unable to Identify	If an outlier is identified as the last observation, then the algorithm is unable to determine the outlier's classification. For forecasting, a UI outlier is treated as an IO outlier. That is, its effect is lagged into the forecasts.

Except for additive outliers (AO), the effect of an outlier persists to observations following that outlier. Forecasts produced by `imsls_f_auto_arima` take this into account.

Examples

Example 1

This example uses time series LNU03327709 from the US Department of Labor, Bureau of Labor Statistics. It contains the unadjusted special unemployment rate, taken monthly from Janurary 1994 through September 2005. The values 01/2004 – 03/2005 are used by `imsls_f_auto_arima` for outlier detection and parameter estimation. In this example, Method 1 without seasonal adjustment is chosen to find an appropriate AR(p) model. A forecast is done for the following six months and compared with the actual values 04/2005 – 09/2005.

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

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

void main(void)
{
  float *parameters = NULL, *outlier_forecast = NULL;
  int *outlier_stat = NULL;
  int n_obs, n_predict, i, num_outliers;
  float aic, res_sigma;
  int model[4];
  float forecast_table[24];

  float x[141] = {
    12.8,12.2,11.9,10.9,10.6,11.3,11.1,10.4,10.0,9.7,9.7,9.7,
    11.1,10.5,10.3,9.8,9.8,10.4,10.4,10.0,9.7,9.3,9.6,9.7,
    10.8,10.7,10.3,9.7,9.5,10.0,10.0,9.3,9.0,8.8,8.9,9.2,
    10.4,10.0,9.6,9.0,8.5,9.2,9.0,8.6,8.3,7.9,8.0,8.2,
    9.3,8.9,8.9,7.7,7.6,8.4,8.5,7.8,7.6,7.3,7.2,7.3,
    8.5,8.2,7.9,7.4,7.1,7.9,7.7,7.2,7.0,6.7,6.8,6.9,
    7.8,7.6,7.4,6.6,6.8,7.2,7.2,7.0,6.6,6.3,6.8,6.7,
    8.1,7.9,7.6,7.1,7.2,8.2,8.1,8.1,8.2,8.7,9.0,9.3,
    10.5,10.1,9.9,9.4,9.2,9.8,9.9,9.5,9.0,9.0,9.4,9.6,
    11.0,10.8,10.4,9.8,9.7,10.6,10.5,10.0,9.8,9.5,9.7,9.6,
    10.9,10.3,10.4,9.3,9.3,9.8,9.8,9.3,8.9,9.1,9.1,9.1,
    10.2,9.9,9.4,8.7,8.6,9.3,9.1,8.8,8.5};

  int times[141] = {
       1,2,3,4,5,6,7,8,9,10,11,12,
      13,14,15,16,17,18,19,20,21,22,23,24,
      25,26,27,28,29,30,31,32,33,34,35,36,
      37,38,39,40,41,42,43,44,45,46,47,48,
      49,50,51,52,53,54,55,56,57,58,59,60,
      61,62,63,64,65,66,67,68,69,70,71,72,
      73,74,75,76,77,78,79,80,81,82,83,84,
      85,86,87,88,89,90,91,92,93,94,95,96,
      97,98,99,100,101,102,103,104,105,106,107,108,
     109,110,111,112,113,114,115,116,117,118,119,120,
     121,122,123,124,125,126,127,128,129,130,131,132,
     133,134,135,136,137,138,139,140,141};

   n_predict = 6;
   n_obs = 135;

   parameters = imsls_f_auto_arima(n_obs, times, x, IMSLS_MODEL, model,
                      IMSLS_AIC, &aic,
                      IMSLS_MAX_LAG, 5,
                      IMSLS_CRITICAL, 4.0,
                      IMSLS_NUM_OUTLIERS, &num_outliers,
                      IMSLS_OUTLIER_STATISTICS, &outlier_stat,
                      IMSLS_RESIDUAL_SIGMA, &res_sigma,
                      IMSLS_NUM_PREDICT, n_predict,
                      IMSLS_OUTLIER_FORECAST, &outlier_forecast,
                      0);

   printf("\nMethod 1: Automatic ARIMA model selection,"
                       " no differencing\n");
   printf("\nModel chosen: p=%d, q=%d, s=%d, d=%d\n", model[0],
                    model[1], model[2], model[3]);
   printf("\nNumber of outliers: %d\n\n", num_outliers);

   printf("Outlier statistics:\n\n");
   printf("Time point\t\tOutlier type\n");
   for (i=0; i<num_outliers; i++)
     printf("%d\t\t%d\n", outlier_stat[2*i], outlier_stat[2*i+1]);

   printf("\nAIC = %lf\n", aic);
   printf("RSE = %lf\n\n", res_sigma);

   printf("Parameters:\n");
   for (i=0; i<=model[0]+model[1]; i++)
     printf("parameters[%d]=%lf\n", i,  parameters[i]);

   for (i=0; i<n_predict; i++)
   {
      forecast_table[4*i] = x[n_obs+i];
      forecast_table[4*i+1] = outlier_forecast[3*i];
      forecast_table[4*i+2] = outlier_forecast[3*i+1];
      forecast_table[4*i+3] = outlier_forecast[3*i+2];
   }

   imsls_f_write_matrix("\t* * * Forecast Table * * *"
      "\nOrig. series\t  forecast\tprob. limits\tpsi weights\n",
      n_predict, 4, forecast_table, IMSLS_WRITE_FORMAT, "%11.4f", 0);

   if (parameters)
   {
      free(parameters);
      parameters = NULL;
   }

   if (outlier_forecast)
   {
      free(outlier_forecast);
      outlier_forecast = NULL;
   }

   if (outlier_stat)
   {
      free(outlier_stat);
      outlier_stat = NULL;
   }

   return;
}
```

Output

```
Method 1: Automatic ARIMA model selection, no differencing

Model chosen: p=5, q=0, s=1, d=0
```

```
Number of outliers: 6

Outlier statistics:

Time point      Outlier type
13              0
37              3
85              0
97              0
109             0
121             0

AIC = 380.951660
RSE = 0.372990

Parameters:
parameters[0]=0.078454
parameters[1]=0.905531
parameters[2]=-0.101995
parameters[3]=-0.184992
parameters[4]=0.218070
parameters[5]=0.154951

                * * * Forecast Table * * *
  Orig. series    forecast      prob. limits    psi weights

             1            2             3             4
1       8.7000       9.0883        0.7310        0.9055
2       8.6000       9.1523        0.9862        0.7180
3       9.3000       9.4397        1.1172        0.3728
4       9.1000       9.5955        1.1500        0.3149
5       8.8000       9.5500        1.1728        0.4667
6       8.5000       9.4054        1.2214        0.6184
```

Example 2

This is the same as Example 1, except now `imsls_f_auto_arima` uses Method 2 with a possible seasonal adjustment. As a result, the unadjusted model with $p = 3, q = 2, s = 1, d = 0$ is chosen as optimum.

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

void main(void)
{
  int n_obs, n_predict, i, num_outliers;
  float aic, res_sigma;
  int model[4];
  int n_s_initial = 2;
  int n_d_initial = 3;
  int s_initial[2] = {1,2};
  int d_initial[3] = {0,1,2};
  int n_p_initial = 4, n_q_initial = 4;
  int p_initial[4] = {0,1,2,3};
  int q_initial[4] = {0,1,2,3};
  float parameters_user[141];
  float outfree_series_user[282];
  int outlier_stat_user[282];
  float outlier_forecast_user[24];
  float forecast_table[24];

  float x[141] = {
    12.8,12.2,11.9,10.9,10.6,11.3,11.1,10.4,10.0,9.7,9.7,9.7,
    11.1,10.5,10.3,9.8,9.8,10.4,10.4,10.0,9.7,9.3,9.6,9.7,
    10.8,10.7,10.3,9.7,9.5,10.0,10.0,9.3,9.0,8.8,8.9,9.2,
    10.4,10.0,9.6,9.0,8.5,9.2,9.0,8.6,8.3,7.9,8.0,8.2,
    9.3,8.9,8.9,7.7,7.6,8.4,8.5,7.8,7.6,7.3,7.2,7.3,
    8.5,8.2,7.9,7.4,7.1,7.9,7.7,7.2,7.0,6.7,6.8,6.9,
    7.8,7.6,7.4,6.6,6.8,7.2,7.2,7.0,6.6,6.3,6.8,6.7,
    8.1,7.9,7.6,7.1,7.2,8.2,8.1,8.1,8.2,8.7,9.0,9.3,
    10.5,10.1,9.9,9.4,9.2,9.8,9.9,9.5,9.0,9.0,9.4,9.6,
    11.0,10.8,10.4,9.8,9.7,10.6,10.5,10.0,9.8,9.5,9.7,9.6,
    10.9,10.3,10.4,9.3,9.3,9.8,9.8,9.3,8.9,9.1,9.1,9.1,
    10.2,9.9,9.4,8.7,8.6,9.3,9.1,8.8,8.5};

  int times[141] = {
       1,2,3,4,5,6,7,8,9,10,11,12,
      13,14,15,16,17,18,19,20,21,22,23,24,
      25,26,27,28,29,30,31,32,33,34,35,36,
      37,38,39,40,41,42,43,44,45,46,47,48,
      49,50,51,52,53,54,55,56,57,58,59,60,
      61,62,63,64,65,66,67,68,69,70,71,72,
      73,74,75,76,77,78,79,80,81,82,83,84,
      85,86,87,88,89,90,91,92,93,94,95,96,
      97,98,99,100,101,102,103,104,105,106,107,108,
     109,110,111,112,113,114,115,116,117,118,119,120,
     121,122,123,124,125,126,127,128,129,130,131,132,
     133,134,135,136,137,138,139,140,141};

   n_predict = 6;
   n_obs = 135;

   imsls_f_auto_arima(n_obs, times, x, IMSLS_MODEL, model,
                  IMSLS_AIC, &aic,
                  IMSLS_CRITICAL, 4.0,
                  IMSLS_MAX_LAG, 5,
                  IMSLS_METHOD, 2,
                  IMSLS_P_INITIAL, n_p_initial, p_initial,
                  IMSLS_Q_INITIAL, n_q_initial, q_initial,
                  IMSLS_S_INITIAL, n_s_initial, s_initial,
                  IMSLS_D_INITIAL, n_d_initial, d_initial,
                  IMSLS_NUM_OUTLIERS, &num_outliers,
                  IMSLS_OUTLIER_STATISTICS_USER, outlier_stat_user,
                  IMSLS_RESIDUAL_SIGMA, &res_sigma,
                  IMSLS_NUM_PREDICT, 6,
                  IMSLS_OUTLIER_FORECAST_USER, outlier_forecast_user,
                  IMSLS_RETURN_USER, parameters_user,
                  0);
   for (i=0; i<n_predict; i++)
   {
      forecast_table[4*i] = x[n_obs+i];
      forecast_table[4*i+1] = outlier_forecast_user[3*i];
      forecast_table[4*i+2] = outlier_forecast_user[3*i+1];
      forecast_table[4*i+3] = outlier_forecast_user[3*i+2];
   }

   printf("\nMethod 2: Grid search, differencing allowed\n");

   printf("\nModel chosen: p=%d, q=%d, s=%d, d=%d\n", model[0],
                    model[1], model[2], model[3]);
   printf("\nNumber of outliers: %d\n\n", num_outliers);

   printf("Outlier statistics:\n\n");
   printf("Time point\t\tOutlier type\n");
   for (i=0; i<num_outliers; i++)
     printf("%d\t\t%d\n", outlier_stat_user[2*i],
                               outlier_stat_user[2*i+1]);

   printf("\nAIC = %lf\n", aic);
   printf("RSE = %lf\n\n", res_sigma);

   printf("Parameters:\n");
   for (i=0; i<=model[0]+model[1]; i++)
     printf("parameters[%d]=%lf\n", i,  parameters_user[i]);

   imsls_f_write_matrix("\n\t* * * Forecast Table * * *"
      "\nOrig. series\t  forecast\tprob. limits\tpsi weights\n",
      n_predict, 4, forecast_table, IMSLS_WRITE_FORMAT, "%11.4f", 0);

   return;
}
```

Output

```
Method 2: Grid search, differencing allowed

Model chosen: p=3, q=2, s=1, d=0

Number of outliers: 1

Outlier statistics:

Time point              Outlier type
109             0

AIC = 408.076813
RSE = 0.412409

Parameters:
parameters[0]=0.509478
parameters[1]=1.944665
parameters[2]=-1.901104
parameters[3]=0.901657
```

```
parameters[4]=1.113017
parameters[5]=-0.914998


                * * * Forecast Table * * *
  Orig. series    forecast      prob. limits    psi weights

             1           2             3             4
1       8.7000       9.1109       0.8083       0.8316
2       8.6000       9.1811       1.0513       0.6312
3       9.3000       9.5185       1.1686       0.5480
4       9.1000       9.7804       1.2497       0.6157
5       8.8000       9.7117       1.3451       0.7245
6       8.5000       9.3842       1.4671       0.7326
```

Example 3

This example is the same as Example 2 but now Method 3 with the optimum model parameters $p = 3, q = 2, s = 1, d = 0$ from Example 2 are chosen for outlier detection and forecasting.

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

void main(void)
{
  float *parameters = NULL, *outlier_forecast = NULL;
  int *outlier_stat = NULL;
  int n_obs, n_predict, i, num_outliers;
  float aic, res_sigma;
  int model[4];
  float forecast_table[24];

  float x[141] = {
    12.8,12.2,11.9,10.9,10.6,11.3,11.1,10.4,10.0,9.7,9.7,9.7,
    11.1,10.5,10.3,9.8,9.8,10.4,10.4,10.0,9.7,9.3,9.6,9.7,
    10.8,10.7,10.3,9.7,9.5,10.0,10.0,9.3,9.0,8.8,8.9,9.2,
    10.4,10.0,9.6,9.0,8.5,9.2,9.0,8.6,8.3,7.9,8.0,8.2,
    9.3,8.9,8.9,7.7,7.6,8.4,8.5,7.8,7.6,7.3,7.2,7.3,
    8.5,8.2,7.9,7.4,7.1,7.9,7.7,7.2,7.0,6.7,6.8,6.9,
    7.8,7.6,7.4,6.6,6.8,7.2,7.2,7.0,6.6,6.3,6.8,6.7,
    8.1,7.9,7.6,7.1,7.2,8.2,8.1,8.1,8.2,8.7,9.0,9.3,
    10.5,10.1,9.9,9.4,9.2,9.8,9.9,9.5,9.0,9.0,9.4,9.6,
    11.0,10.8,10.4,9.8,9.7,10.6,10.5,10.0,9.8,9.5,9.7,9.6,
    10.9,10.3,10.4,9.3,9.3,9.8,9.8,9.3,8.9,9.1,9.1,9.1,
    10.2,9.9,9.4,8.7,8.6,9.3,9.1,8.8,8.5};

  int times[141] = {
       1,2,3,4,5,6,7,8,9,10,11,12,
      13,14,15,16,17,18,19,20,21,22,23,24,
      25,26,27,28,29,30,31,32,33,34,35,36,
      37,38,39,40,41,42,43,44,45,46,47,48,
      49,50,51,52,53,54,55,56,57,58,59,60,
      61,62,63,64,65,66,67,68,69,70,71,72,
      73,74,75,76,77,78,79,80,81,82,83,84,
      85,86,87,88,89,90,91,92,93,94,95,96,
      97,98,99,100,101,102,103,104,105,106,107,108,
    109,110,111,112,113,114,115,116,117,118,119,120,
    121,122,123,124,125,126,127,128,129,130,131,132,
    133,134,135,136,137,138,139,140,141};

  n_predict = 6;
  n_obs = 135;

  model[0] = 3;
  model[1] = 2;
  model[2] = 1;
  model[3] = 0;

  parameters = imsls_f_auto_arima(n_obs, times, x, IMSLS_MODEL, model,
                      IMSLS_AIC, &aic,
                      IMSLS_CRITICAL, 4.0,
                      IMSLS_METHOD, 3,
                      IMSLS_NUM_OUTLIERS, &num_outliers,
                      IMSLS_OUTLIER_STATISTICS, &outlier_stat,
                      IMSLS_RESIDUAL_SIGMA, &res_sigma,
                      IMSLS_NUM_PREDICT, 6,
                      IMSLS_OUTLIER_FORECAST, &outlier_forecast,
                      0);

  printf("\nMethod 3: Specified ARIMA model\n");
  printf("\nModel: p=%d, q=%d, s=%d, d=%d\n", model[0], model[1],
                        model[2], model[3]);
  printf("\nNumber of outliers: %d\n\n", num_outliers);

  printf("Outlier statistics:\n\n");
  printf("Time point\t\tOutlier type\n");
  for (i=0; i<num_outliers; i++)
    printf("%d\t\t%d\n", outlier_stat[2*i], outlier_stat[2*i+1]);

  printf("\nAIC = %lf\n", aic);
  printf("RSE = %lf\n", res_sigma);

  printf("\nParameters:\n");
  for (i=0; i<=model[0]+model[1]; i++)
    printf("parameters[%d]=%lf\n", i,  parameters[i]);

  for (i=0; i<n_predict; i++)
  {
     forecast_table[4*i] = x[n_obs+i];
     forecast_table[4*i+1] = outlier_forecast[3*i];
     forecast_table[4*i+2] = outlier_forecast[3*i+1];
     forecast_table[4*i+3] = outlier_forecast[3*i+2];
  }

  imsls_f_write_matrix("\t* * * Forecast Table * * *"
      "\nOrig. series\t  forecast\tprob. limits\tpsi weights\n",
      n_predict, 4, forecast_table, IMSLS_WRITE_FORMAT, "%11.4f", 0);
```

```
   if (parameters)
   {
      free(parameters);
      parameters = NULL;
   }

   if (outlier_forecast)
   {
      free(outlier_forecast);
      outlier_forecast = NULL;
   }

   if (outlier_stat)
   {
      free(outlier_stat);
      outlier_stat = NULL;
   }

   return;

}
```

Output

```
Method 3: Specified ARIMA model

Model: p=3, q=2, s=1, d=0

Number of outliers: 1

Outlier statistics:

Time point              Outlier type
109            0

AIC = 408.076813
RSE = 0.412409

Parameters:
parameters[0]=0.509478
parameters[1]=1.944665
parameters[2]=-1.901104
parameters[3]=0.901657
parameters[4]=1.113017
parameters[5]=-0.914998

                * * * Forecast Table * * *
  Orig. series    forecast      prob. limits    psi weights

            1            2            3            4
1       8.7000       9.1109       0.8083       0.8316
2       8.6000       9.1811       1.0513       0.6312
3       9.3000       9.5185       1.1686       0.5480
4       9.1000       9.7804       1.2497       0.6157
5       8.8000       9.7117       1.3451       0.7245
6       8.5000       9.3842       1.4671       0.7326
```

difference

Differences a seasonal or nonseasonal time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_difference` (*int* `n_observations`, *float* `z[]`,
int `n_differences`, *int* `periods[]`, ..., 0)

The type *double* function is `imsls_d_difference`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `z[]` (Input)
Array of length `n_observations` containing the time series.

int `n_differences` (Input)
Number of differences to perform. Argument `n_differences` must be greater than or equal to 1.

int `periods[]` (Input)
Array of length `n_differences` containing the periods at which `z` is to be differenced.

Return Value

Pointer to an array of length `n_observations` containing the differenced series.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_difference` (*int* `n_observations`, *float* `z[]`,
int `n_differences`, *int* `periods[]`,
`IMSLS_ORDERS`, *int* `orders[]`,
`IMSLS_LOST`, *int*`v*n_lost`,
`IMSLS_EXCLUDE_FIRST`, *or*
`IMSLS_SET_FIRST_TO_NAN`,
`IMSLS_RETURN_USER`, *float* `w[]`,
0)

Optional Arguments

`IMSLS_ORDERS`, *int* `orders[]` (Input)
Array of length `n_differences` containing the order of each difference given in periods. The elements of orders must be greater than or equal to 0.

`IMSLS_LOST`, *int* `*n_lost` (Output)
Number of observations lost because of differencing the time series `z`.

IMSLS_EXCLUDE_FIRST, *or*
IMSLS_SET_FIRST_TO_NAN

If IMSLS_EXCLUDE_FIRST is specified, the first n_lost are excluded from w due to differencing. The differenced series w is of length n_observations – n_lost. If IMSLS_SET_FIRST_TO_NAN is specified, the first n_lost observations are set to NaN (Not a Number). This is the default if neither IMSLS_EXCLUDE_FIRST nor IMSLS_SET_FIRST_TO_NAN is specified.

IMSLS_RETURN_USER, *float* w[] (Output)

If specified, w contains the differenced series. If IMSLS_EXCLUDE_FIRST also is specified, w is of length n_observations. If IMSLS_SET_FIRST_TO_NAN is specified or neither IMSLS_EXCLUDE_FIRST nor IMSLS_SET_FIRST_TO_NAN is specified, w is of length n_observations – n_lost.

Description

Function imsls_f_difference performs m = n_differences successive backward differences of period s_i = periods $[i-1]$ and order
d_i = orders $[i-1]$ for $i = 1, ..., m$ on the n = n_observations observations $\{Z_t\}$ for $t = 1, 2, ..., n$.

Consider the backward shift operator B given by

$$B^k Z_t = Z_{t-k}$$

for all k. Then, the *backward difference operator* with period s is defined by the following:

$$\Delta_s Z_t = (1 - B^s) Z_t = Z_t - Z_{t-s} \quad \text{for} \quad s > 0.$$

Note that $B^s Z_t$ and $\Delta^s Z_t$ are defined only for $t = (s + 1), ..., n$. Repeated differencing with period s is simply

$$\Delta_s^d Z_t = \left(1 - B^s\right)^d Z_t = \sum_{j=0}^{d} \frac{d!}{j!(d-j)!} (-1)^j B^{sj} Z_t$$

where $d \geq 0$ is the order of differencing. Note that

$$\Delta_s^d Z_t$$

is defined only for $t = (sd + 1), ..., n$.

The general difference formula used in the function imsls_f_difference is given by

$$W_t = \begin{cases} \text{NaN} & \text{for } t = 1, \ldots, n_L \\ \Delta_{s_1}^{d_1} \Delta_{s_2}^{d_2} \ldots \Delta_{s_m}^{d_m} Z_t & \text{for } t = n_L + 1, \ldots, n \end{cases}$$

where n_L represents the number of observations "lost" because of differencing and NaN represents the missing value code. See the functions `imsls_f_machine` and `imsls_d_machine` (Chapter 15, "Utilities") to retrieve missing values. Note that

$$n_L = \sum_j s_j d_j$$

A homogeneous, stationary time series can be arrived at by appropriately differencing a homogeneous, nonstationary time series (Box and Jenkins 1976, p. 85). Preliminary application of an appropriate transformation followed by differencing of a series can enable model identification and parameter estimation in the class of homogeneous stationary autoregressive moving average models.

Examples

Example 1

Consider the Airline Data (Box and Jenkins 1976, p. 531) consisting of the monthly total number of international airline passengers from January 1949 through December 1960. Function `imsls_f_difference` is used to compute

$$W_t = \Delta_1 \Delta_{12} Z_t = (Z_t - Z_{t-12}) - (Z_{t-1} - Z_{t-13})$$

for $t = 14, 15, \ldots, 24$.

```
#include <imsls.h>

void main()

{
    int     i;
    int     n_observations = 24;
    int     n_differences = 2;
    int     periods[2] = {1, 12};
    float   *z;
    float   *difference;

    z = imsls_f_data_sets (4, 0);
    difference = imsls_f_difference (n_observations, z,
                                     n_differences, periods,
                                     0);
    printf ("i\tz[i]\tdifference[i]\n");
    for (i = 0; i < n_observations; i++)
        printf ("%d\t%f\t%f\n", i, z[i], difference[i]);

}
```

Output

```
 i      z[i]         difference[i]
 0      112.000000   NaN
 1      118.000000   NaN
 2      132.000000   NaN
 3      129.000000   NaN
 4      121.000000   NaN
 5      135.000000   NaN
 6      148.000000   NaN
 7      148.000000   NaN
 8      136.000000   NaN
 9      119.000000   NaN
10      104.000000   NaN
11      118.000000   NaN
12      115.000000   NaN
13      126.000000    5.000000
14      141.000000    1.000000
15      135.000000   -3.000000
16      125.000000   -2.000000
17      149.000000   10.000000
18      170.000000    8.000000
19      170.000000    0.000000
20      158.000000    0.000000
21      133.000000   -8.000000
22      114.000000   -4.000000
23      140.000000   12.000000
```

Example 2

The data for this example is the same as that for the initial example. The first `n_lost` observations are excluded from *W* due to differencing, and `n_lost` is also output.

```
#include <imsls.h>

void main()
{

    int    i;
    int    n_observations = 24;
    int    n_differences = 2;
    int    periods[2] = {1, 12};
    int    n_lost;
    float  *z;
    float  *difference;
                  /* Get airline data */
    z = imsls_f_data_sets (4, 0);
                  /* Compute differenced time series when observations
                     lost are excluded from the differencing */
    difference = imsls_f_difference (n_observations, z,
                                     n_differences, periods,
                                     IMSLS_EXCLUDE_FIRST,
                                     IMSLS_LOST, &n_lost,
                                     0);
                  /* Print the number of lost observations */
    printf ("n_lost equals %d\n", n_lost);
    printf ("\n\ni\tz[i]\t        difference[i]\n");
```

```
                  /* Print the original time series and the differenced
                     time series */
    for (i = 0; i < n_observations - n_lost; i++)
        printf ("%d\t%f\t%f\n", i, z[i], difference[i]);
}
```

Output

```
n_lost equals 13


 i      z[i]          difference[i]
 0      112.000000     5.000000
 1      118.000000     1.000000
 2      132.000000    -3.000000
 3      129.000000    -2.000000
 4      121.000000    10.000000
 5      135.000000     8.000000
 6      148.000000     0.000000
 7      148.000000     0.000000
 8      136.000000    -8.000000
 9      119.000000    -4.000000
10      104.000000    12.000000
```

Fatal Errors

`IMSLS_PERIODS_LT_ZERO`	"period[#]" = #. All elements of "period" must be greater than 0.
`IMSLS_ORDER_NEGATIVE`	"order[#]" = #. All elements of "order" must be nonnegative.
`IMSLS_Z_CONTAINS_NAN`	"z[#]" = NaN; "z" can not contain missing values. There may be other elements of "z" that are equal to NaN.

seasonal_fit

Estimates the optimum seasonality parameters for a time series using an autoregressive model, AR(p), to represent the time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_seasonal_fit`(*int* `n_obs`, *float* `z[]`, *int* `maxlag`,
int `n_differences`, *int* `n_s_initial`, *int* `s_initial[]`,…,0)

The type *double* function is `imsls_d_seasonal_fit`.

Required Arguments

int `n_obs` (Input)
Number of observations in the time series.

float `z[]` (Input)
: An array of length `n_obs` containing the time series. No missing values in the series are allowed.

int `maxlag` (Input)
: The maximum lag allowed when fitting an AR(p) model.

int `n_differences` (Input)
: The number of differences to perform. Argument `n_differences` must be greater than or equal to one.

int `n_s_initial` (Input)
: The number of rows of the array containing the seasonal differences.

int `s_initial[]` (Input)
: Array of dimension `n_s_initial` by `n_differences` containing the seasonal differences to test. All values of `s_initial` must be greater than or equal to one.

Return Value

Pointer to an array of length `n_obs` or `n_obs-n_lost` containing the optimum seasonally adjusted, autoregressive series. The first `n_lost` observations in this series are set to NaN, missing values. The seasonal adjustment is done by selecting optimum values for $d_1, \ldots, d_m, s_1, \ldots, s_m$ (m=`n_differences`) and p in the AR model:

$$\phi_p(B)(\Delta_{s_1}^{d_1}\Delta_{s_2}^{d_2}\cdots\Delta_{s_m}^{d_m}Z_t-\mu)=a_t\ ,$$

where $\{Z_t\}$ is the original time series, *B* is the backward shift operator defined by $B^kZ_t = Z_{t-k}$, $k \ge 0$, a_t is Gaussian white noise with $E[a_t]=0$ and $VAR[a_t]=\sigma^2$,
$\phi_p(B) = 1-\phi_1B-\phi_2B^2-\cdots-\phi_pB^p$, $0 \le p \le \texttt{maxlag}$,
$\Delta_s^d = (1-B^s)^d$, with $s > 0,\ d \ge 0$, and μ is a centering parameter for the differenced series.

NOTE that $\Delta_s^0 = 1$, the identity operator, i.e. $\Delta_s^0\,Y_t = Y_t$.

If an error occurred, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float * `imsls_f_seasonal_fit` (*int* `n_obs`, *float* `z[]`, *int* `maxlag`,
int `n_differences`, *int* `n_s_initial`, *int* `s_initial[]`,
`IMSLS_RETURN_USER`, *float* `w[]`,
`IMSLS_D_INITIAL`, *int* `n_d_initial`, *int* `d_initial[]`,
`IMSLS_SET_FIRST_TO_NAN`, *or* `IMSLS_EXCLUDE_FIRST`,
`IMSLS_CENTER`, *int* `n_center`,
`IMSLS_LOST`, *int* `*n_lost`,
`IMSLS_BEST_PERIODS`, *int* `**s`,
`IMSLS_BEST_PERIODS_USER`, *int* `s[]`,

IMSLS_BEST_ORDERS, *int* **d,
IMSLS_BEST_ORDERS_USER, *int* d[],
IMSLS_AR_ORDER, *int* *p,
IMSLS_AIC, *float* *aic,
0*)*

Optional Arguments

IMSLS_RETURN_USER, *float* w[] (Output)
An array of length n_obs supplied by the user to hold the seasonally adjusted series returned by imsls_f_seasonal_fit.

IMSLS_D_INITIAL, *int* n_d_initial, *int* d_initial[] (Input)
An array of dimension n_d_initial by n_differences containing the candidate values for d[], from which the optimum is being selected. All candidate values in d_initial[] must be non-negative and n_d_initial ≥ 1.
Default: n_d_initial=1, d_initial an array of length n_differences filled with ones.

IMSLS_SET_FIRST_TO_NAN, *or* IMSLS_EXCLUDE_FIRST (Input)
If IMSLS_EXCLUDE_FIRST is specified, the first n_lost values are excluded from w due to differencing. The differenced series w is of length n_obs-n_lost. If IMSLS_SET_FIRST_TO_NAN is specified, the first n_lost observations are set to NaN (Not a Number).
Default: IMSLS_SET_FIRST_TO_NAN.

IMSLS_CENTER, *int* n_center (Input)
If supplied, IMSLS_CENTER controls the method used to center the differenced series. If n_center=0 then the series is not centered. If n_center=1, the mean of the series is used to center the data, and if n_center=2, the median is used.
Default: n_center=1.

IMSLS_LOST, *int* *n_lost (Output)
The number of observations lost due to differencing the time series. This is also equal to the number of NaN values that appear in the first n_lost locations of the returned seasonally adjusted series.

IMSLS_BEST_PERIODS, *int* **s (Output)
Address of a pointer to an internally allocated array of length *m*=n_differences containing the optimum values for the seasonal adjustment parameters $s_1, s_2, \ldots, s_m$ selected from the list of candidates contained in s_initial[].

IMSLS_BEST_PERIODS_USER, *int* s[] (Output)
A user supplied array of length n_differences for storage of the array s.

IMSLS_BEST_ORDERS, *int* **d (Output)
Address of a pointer to an internally allocated array of length *m*=n_differences containing the optimum values for the seasonal adjustment parameters $d_1, d_2, \ldots, d_m$ selected from the list of candidates contained in d_initial[].

IMSLS_BEST_ORDERS_USER, *int* d[] (Output)
A user supplied array of length n_differences for storage of the array d.

IMSLS_AR_ORDER, *int* *p (Output)

The optimum value for the autoregressive lag.

IMSLS_AIC, *float* *aic (Output)

Akaike's Information Criterion (AIC) for the optimum seasonally adjusted model.

Description

Many time series contain seasonal trends and cycles that can be modeled by first differencing the series. For example, if the correlation is strong from one period to the next, the series might be differenced by a lag of 1. Instead of fitting a model to the series Z_t, the model is fitted to the transformed series: $W_t = Z_t - Z_{t-1}$. Higher order lags or differences are warranted if the series has a cycle every 4 or 13 weeks.

Function imsls_f_seasonal_fit does not center the original series. If IMSLS_CENTER is specified with either n_center =1 or n_center =2, then the differenced series, W_t, is centered before determination of minimum AIC and optimum lag. For every combination of rows in s_initial and d_initial, the series Z_t is converted to the seasonally adjusted series using the following computation

$$W_t(s,d) = \Delta_{s_1}^{d_1} \Delta_{s_2}^{d_2} \cdots \Delta_{s_m}^{d_m} Z_t = \prod_{i=1}^{m} (1-B^{s_i})^{d_i} Z_t = \prod_{i=1}^{m} \sum_{j=0}^{d_i} \binom{d_i}{j} (-1)^j B^{j \cdot s_i} Z_t \, .$$

where $s := (s_1, \ldots, s_m)$, $d := (d_1, \ldots, d_m)$ represent specific rows of arrays s_initial and d_initial respectively, and m =n_differences.

This transformation of the series Z_t to $W_t(s,d)$ is accomplished using function imsls_f_difference(). After this transformation,

$$W_t(s,d)$$

is (optionally) centered and a call is made to imsls_f_auto_uni_ar to automatically determine the optimum lag for an AR(*p*) representation for $W_t(s,d)$. This procedure is repeated for every possible combination of rows of s_initial and d_initial. The series with the minimum AIC is identified as the optimum representation and returned.

Example

Consider the Airline Data (Box, Jenkins and Reinsel 1994, p. 547) consisting of the monthly total number of international airline passengers from January 1949 through December 1960. Function imsls_f_seasonal_fit is used to compute the optimum seasonality representation of the adjusted series

$$W_t(s,d) = \Delta_{s_1}^{d_1} \Delta_{s_2}^{d_2} Z_t = (1-B^{s_1})^{d_1} (1-B^{s_2})^{d_2} Z_t \, ,$$

where

$$s = (1,1)$$

or

$$s = (1,12)$$

and

$$d = (1,1).$$

As differenced series with minimum AIC,

$$W_t = \Delta_1^1 \Delta_{12}^2 Z_t = \left(Z_t - Z_{t-12}\right) - \left(Z_{t-1} - Z_{t-13}\right),$$

is identified.

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

void main()
{
  int i;
  int maxlag = 10;
  int nobs = 144;
  int n_differences = 2;
  int n_s_initial = 2;
  int nlost;
  int npar;
  float aic;
  int s_init[] = { 1, 1,
                   1, 12};
  int *s = NULL;
  int *d = NULL;
  float *z = NULL;
  float *difference = NULL;

  z = imsls_f_data_sets(4, 0);

  difference = imsls_f_seasonal_fit(nobs, z, maxlag, n_differences,
                                  n_s_initial, s_init,
                                  IMSLS_LOST, &nlost,
                                  IMSLS_BEST_PERIODS, &s,
                                  IMSLS_BEST_ORDERS, &d,
                                  IMSLS_AIC, &aic,
                                  IMSLS_AR_ORDER, &npar,
                                  0);

  printf("\nnlost = %d\n", nlost);
  printf("s = (%d, %d)\n", s[0], s[1]);
  printf("d = (%d, %d)\n", d[0], d[1]);
  printf("Order of optimum AR process: %d\n", npar);
  printf("aic = %lf\n", aic);
  printf("\ni\tz[i]\tdifference[i]\n");
  for (i=0; i<nobs; i++)
    printf("%d\t%f\t%f\n", i, z[i], difference[i]);

  if (s)
  {
     free(s);
     s = NULL;
  }

  if (d)
  {
     free(d);
     d = NULL;
  }

  if (z)
  {
     free(z);
     z = NULL;
  }

  if (difference)
  {
     free(difference);
     difference = NULL;
  }

  return;
}
```

Output

```
nlost = 13
s = (1, 12)
d = (1, 1)
Order of optimum AR process: 1
aic = 829.780334

i       z[i]    difference[i]
0       112.000000      NaN
1       118.000000      NaN
2       132.000000      NaN
3       129.000000      NaN
4       121.000000      NaN
5       135.000000      NaN
6       148.000000      NaN
7       148.000000      NaN
8       136.000000      NaN
9       119.000000      NaN
10      104.000000      NaN
11      118.000000      NaN
12      115.000000      NaN
```

```
13       126.000000      5.000000
14       141.000000      1.000000
15       135.000000      -3.000000
16       125.000000      -2.000000
17       149.000000      10.000000
18       170.000000      8.000000
19       170.000000      0.000000
20       158.000000      0.000000
21       133.000000      -8.000000
22       114.000000      -4.000000
23       140.000000      12.000000
24       145.000000      8.000000
25       150.000000      -6.000000
26       178.000000      13.000000
27       163.000000      -9.000000
28       172.000000      19.000000
29       178.000000      -18.000000
30       199.000000      0.000000
31       199.000000      0.000000
32       184.000000      -3.000000
33       162.000000      3.000000
34       146.000000      3.000000
35       166.000000      -6.000000
36       171.000000      0.000000
37       180.000000      4.000000
38       193.000000      -15.000000
39       181.000000      3.000000
40       183.000000      -7.000000
41       218.000000      29.000000
42       230.000000      -9.000000
43       242.000000      12.000000
44       209.000000      -18.000000
45       191.000000      4.000000
46       172.000000      -3.000000
47       194.000000      2.000000
48       196.000000      -3.000000
49       196.000000      -9.000000
50       236.000000      27.000000
51       235.000000      11.000000
52       229.000000      -8.000000
53       243.000000      -21.000000
54       264.000000      9.000000
55       272.000000      -4.000000
56       237.000000      -2.000000
57       211.000000      -8.000000
58       180.000000      -12.000000
59       201.000000      -1.000000
60       204.000000      1.000000
61       188.000000      -16.000000
62       235.000000      7.000000
63       227.000000      -7.000000
64       234.000000      13.000000
65       264.000000      16.000000
66       302.000000      17.000000
67       293.000000      -17.000000
```

```
68       259.000000      1.000000
69       229.000000      -4.000000
70       203.000000      5.000000
71       229.000000      5.000000
72       242.000000      10.000000
73       233.000000      7.000000
74       267.000000      -13.000000
75       269.000000      10.000000
76       270.000000      -6.000000
77       315.000000      15.000000
78       364.000000      11.000000
79       347.000000      -8.000000
80       312.000000      -1.000000
81       274.000000      -8.000000
82       237.000000      -11.000000
83       278.000000      15.000000
84       284.000000      -7.000000
85       277.000000      2.000000
86       317.000000      6.000000
87       313.000000      -6.000000
88       318.000000      4.000000
89       374.000000      11.000000
90       413.000000      -10.000000
91       405.000000      9.000000
92       355.000000      -15.000000
93       306.000000      -11.000000
94       271.000000      2.000000
95       306.000000      -6.000000
96       315.000000      3.000000
97       301.000000      -7.000000
98       356.000000      15.000000
99       348.000000      -4.000000
100      355.000000      2.000000
101      422.000000      11.000000
102      465.000000      4.000000
103      467.000000      10.000000
104      404.000000      -13.000000
105      347.000000      -8.000000
106      305.000000      -7.000000
107      336.000000      -4.000000
108      340.000000      -5.000000
109      318.000000      -8.000000
110      362.000000      -11.000000
111      348.000000      -6.000000
112      363.000000      8.000000
113      435.000000      5.000000
114      491.000000      13.000000
115      505.000000      12.000000
116      404.000000      -38.000000
117      359.000000      12.000000
118      310.000000      -7.000000
119      337.000000      -4.000000
120      360.000000      19.000000
121      342.000000      4.000000
122      406.000000      20.000000
```

```
123     396.000000      4.000000
124     420.000000      9.000000
125     472.000000      -20.000000
126     548.000000      20.000000
127     559.000000      -3.000000
128     463.000000      5.000000
129     407.000000      -11.000000
130     362.000000      4.000000
131     405.000000      16.000000
132     417.000000      -11.000000
133     391.000000      -8.000000
134     419.000000      -36.000000
135     461.000000      52.000000
136     472.000000      -13.000000
137     535.000000      11.000000
138     622.000000      11.000000
139     606.000000      -27.000000
140     508.000000      -2.000000
141     461.000000      9.000000
142     390.000000      -26.000000
143     432.000000      -1.000000
```

box_cox_transform

Performs a forward or an inverse Box-Cox (power) transformation.

Synopsis

#include `<imsls.h>`

float `*imsls_f_box_cox_transform` (*int* `n_observations`, *float* `z[]`, *float* `power`, ..., 0)

The type *double* function is `imsls_d_box_cox_transform`.

Required Arguments

int `n_observations` (Input)
Number of observations in `z`.

float `z[]` (Input)
Array of length `n_observations` containing the observations.

float `power` (Input)
Exponent parameter in the Box-Cox (power) transformation.

Return Value

Pointer to an internally allocated array of length `n_observations` containing the transformed data. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_box_cox_transform` (*int* `n_observations`, *float* `z[]`, *float*
`power`,
`IMSLS_SHIFT`, *float* `shift`,
`IMSLS_INVERSE_TRANSFORM`,
`IMSLS_RETURN_USER`, *float* `x[]`
0)

Optional Arguments

`IMSLS_SHIFT`, *float* `shift` (Input)
Shift parameter in the Box-Cox (power) transformation. Parameter shift must satisfy the relation min ($z(i)$) + `shift` > 0.
Default: `shift` = 0.0.

`IMSLS_INVERSE_TRANSFORM`
If `IMSLS_INVERSE_TRANSFORM` is specified, the inverse transform is performed.

`IMSLS_RETURN_USER`, *float* `x[]` (Output)
User-allocated array of length `n_observations` containing the transformed data.

Description

Function `imsls_f_box_cox_transform` performs a forward or an inverse Box-Cox (power) transformation of n = `n_observations` observations $\{Z_t\}$ for $t = 1, 2, ..., n$.

The forward transformation is useful in the analysis of linear models or models with nonnormal errors or nonconstant variance (Draper and Smith 1981, p. 222). In the time series setting, application of the appropriate transformation and subsequent differencing of a series can enable model identification and parameter estimation in the class of homogeneous stationary autoregressive-moving average models. The inverse transformation can later be applied to certain results of the analysis, such as forecasts and prediction limits of forecasts, in order to express the results in the scale of the original data. A brief note concerning the choice of transformations in the time series models is given in Box and Jenkins (1976, p. 328).

The class of power transformations discussed by Box and Cox (1964) is defined by

$$X_t = \begin{cases} \dfrac{(Z_t + \xi)^\lambda - 1}{\lambda} & \lambda \ne 0 \\ \ln(Z_t + \xi) & \lambda = 0 \end{cases}$$

where $Z_t + \xi > 0$ for all t. Since

$$\lim_{\lambda \to 0} \frac{(Z_t + \xi)^\lambda - 1}{\lambda} = \ln(Z_t + \xi)$$

the family of power transformations is continuous.

Let λ = `power` and ξ = `shift`; then, the computational formula used by `imsls_f_box_cox_transform` is given by

$$X_t = \begin{cases} (Z_t + \xi)^{\lambda} & \lambda \neq 0 \\ \ln(Z_t + \xi) & \lambda = 0 \end{cases}$$

where $Z_t + \xi > 0$ for all t. The computational and Box-Cox formulas differ only in the scale and origin of the transformed data. Consequently, the general analysis of the data is unaffected (Draper and Smith 1981, p. 225).

The inverse transformation is computed by

$$X_t = \begin{cases} Z_t^{1/\lambda} - \xi & \lambda \neq 0 \\ exp(Z_t) - \xi & \lambda = 0 \end{cases}$$

where $\{Z_t\}$ now represents the result computed by `imsls_f_box_cox_transform` for a forward transformation of the original data using parameters λ and ξ.

Examples

Example 1

The following example performs a Box-Cox transformation with `power` = 2.0 on 10 data points.

```
#include <imsls.h>

void main() {
    int n_observations = 10;
    float power = 2.0;
    float *x;
    static float z[10] ={
        1.0, 2.0, 3.0, 4.0, 5.0, 5.5, 6.5, 7.5, 8.0, 10.0};

    /* Transform Data using Box Cox Transform */
    x = imsls_f_box_cox_transform(n_observations, z, power, 0);

    imsls_f_write_matrix("Transformed Data", 1, n_observations, x, 0);

    free(x);
}
```

Output

```
                              Transformed Data
         1            2            3            4            5            6
       1.0          4.0          9.0         16.0         25.0         30.2

         7            8            9           10
      42.2         56.2         64.0        100.0
```

Example 2

This example extends the first example—an inverse transformation is applied to the transformed data to return to the orignal data values.

```
#include <imsls.h>

void main() {
    int n_observations = 10;
    float power = 2.0;
    float *x, *y;
    static float z[10] ={
        1.0, 2.0, 3.0, 4.0, 5.0, 5.5, 6.5, 7.5, 8.0, 10.0};

    /* Transform Data using Box Cox Transform */
    x = imsls_f_box_cox_transform(n_observations, z, power, 0);

    imsls_f_write_matrix("Transformed Data", 1, n_observations, x, 0);

    /* Perform an Inverse Transform on the Transformed Data */
    y = imsls_f_box_cox_transform(n_observations, x, power,
            IMSLS_INVERSE_TRANSFORM, 0);

    imsls_f_write_matrix("Inverse Transformed Data", 1, n_observations, y,
0);

    free(x);
    free(y);
}
```

Output

```
                    Transformed Data
         1           2           3           4           5           6
       1.0         4.0         9.0        16.0        25.0        30.2

         7           8           9          10
      42.2        56.2        64.0       100.0

                Inverse Transformed Data
         1           2           3           4           5           6
       1.0         2.0         3.0         4.0         5.0         5.5

         7           8           9          10
       6.5         7.5         8.0        10.0
```

Fatal Errors

`IMSLS_ILLEGAL_SHIFT`	“shift” = # and the smallest element of “z” is “z[#]” = #. “shift” plus “z[#]” = #. “shift” + “z[i]” must be greater than 0 for *i* = 1, ..., “n_observations”. “n_observations” = #.
`IMSLS_BCTR_CONTAINS_NAN`	One or more elements of “z” is equal to NaN (Not a number). No missing values are allowed. The

	smallest index of an element of “z” that is equal to NaN is #.
`IMSLS_BCTR_F_UNDERFLOW`	Forward transform. “power” = #. “shift” = #. The minimum element of “z” is “z[#]” = #. (“z[#]”+ “shift”) ^ “power” will underflow.
`IMSLS_BCTR_F_OVERFLOW`	Forward transformation. “power” = #. “shift” = #. The maximum element of “z” is “z[#]” = #. (“z[#]” + “shift”) ^ “power” will overflow.
`IMSLS_BCTR_I_UNDERFLOW`	Inverse transformation. “power” = #. The minimum element of “z” is “z[#]” = #. exp(“z[#]”) will underflow.
`IMSLS_BCTR_I_OVERFLOW`	Inverse transformation. “power” = #. The maximum element of “z[#]” = #. exp(“z[#]”) will overflow.
`IMSLS_BCTR_I_ABS_UNDERFLOW`	Inverse transformation. “power” = #. The element of “z” with the smallest absolute value is “z[#]” = #. “z[#]” ^ (1/ “power”) will underflow.
`IMSLS_BCTR_I_ABS_OVERFLOW`	Inverse transformation. “power” = #. The element of “z” with the largest absolute value is “z[#]” = #. “z[#]” ^ (1/ “power”) will overflow.

autocorrelation

Computes the sample autocorrelation function of a stationary time series.

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_autocorrelation`.

Required Arguments

int `n_observations` (Input)
    Number of observations in the time series `x`. `n_observations` must be greater than or equal to 2.

float `x[]` (Input)
    Array of length `n_observations` containing the time series.

int `lagmax` (Input)
    Maximum lag of autocovariance, autocorrelations, and standard errors of autocorrelations to be computed. `lagmax` must be greater than or equal to 1 and less than `n_observations`.

Return Value

Pointer to an array of length `lagmax` + 1 containing the autocorrelations of the time series `x`. The *0*-th element of this array is 1. The *k*-th element of this array contains the autocorrelation of lag *k* where *k* = 1, ..., `lagmax`.

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_autocorrelation (int n_observations, float x[],
        int lagmax,
        IMSLS_RETURN_USER, float autocorrelations[],
        IMSLS_PRINT_LEVEL, int iprint,
        IMSLS_ACV, float **autocovariances,
        IMSLS_ACV_USER, float autocovariances[],
        IMSLS_SEAC, float **standard_errors,
        int se_option,
        IMSLS_SEAC_USER, float standard_errors[],
        int se_option,
        IMSLS_X_MEAN_IN, float x_mean_in,
        IMSLS_X_MEAN_OUT, float *x_mean_out,
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `autocorrelations[]` (Output)
: If specified, `autocorrelations` is an array of length `lagmax + 1` containing the `autocorrelations` of the time series `x`. The *o*th element of this array is 1. The *k*th element of this array contains the `autocorrelation` of lag *k* where *k* = 1, . . . , `lagmax`.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
: Printing option.
Default = 0.

Iprint	**Action**
0	No printing is performed.
1	Prints the mean and variance.
2	Prints the mean, variance, and autocovariances.
3	Prints the mean, variance, autocovariances, autocorrelations, and standard errors of autocorrelations.

`IMSLS_ACV`, *float* `**autocovariances` (Output)
: Address of a pointer to an array of length `lagmax` + 1 containing the variance and autocovariances of the time series `x`. The *0*-th element of this array is the variance of the time series `x`. The *k*th element contains the autocovariance of lag *k* where *k* = 1, ..., `lagmax`.

`IMSLS_ACV_USER`, *float* `autocovariances[]` (Output)
: If specified, `autocovariances` is an array of length `lagmax` + 1 containing the variance and autocovariances of the time series `x`.
See `IMSLS_ACV`.

`IMSLS_SEAC`, *float* `**standard_errors`, *int* `se_option` (Output)
: Address of a pointer to an array of length `lagmax` containing the standard errors of the autocorrelations of the time series `x`.
Method of computation for standard errors of the autocorrelations is chosen by `se_option`.

se_option	**Action**
1	Compute the standard errors of `autocorrelations` using Barlett's formula.
2	Compute the standard errors of autocorrelations using Moran's formula.

`IMSLS_SEAC_USER`, *float* `standard_errors[]`, *int* `se_option` (Output)
: If specified, `autocovariances` is an array of length `lagmax` containing the standard errors of the autocorrelations of the time series `x`.
See `IMSLS_SEAC`.

`IMSLS_X_MEAN_IN`, *float* `x_mean_in` (Input)
: User input the estimate of the time series `x`.

`IMSLS_X_MEAN_OUT`, *float* `*x_mean_out` (Output)
: If specified, `x_mean_out` is the estimate of the mean of the time series `x`.

Description

Function `imsls_f_autocorrelation` estimates the autocorrelation function of a stationary time series given a sample of n = `n_observations` observations $\{X_t\}$ for $t = 1, 2, \ldots, n$.

Let

$$\hat{\mu} = \texttt{x\_mean}$$

be the estimate of the mean μ of the time series $\{X_t\}$ where

$$\hat{\mu} = \begin{cases} \mu, & \mu \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_t & \mu \text{ unknown} \end{cases}$$

The autocovariance function σ(k) is estimated by

$$\hat{\sigma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu})(X_{t+k} - \hat{\mu}), \quad k = 0,1,\ldots,K$$

where K = `lagmax`. Note that

$$\hat{\sigma}(0)$$

is an estimate of the sample variance. The autocorrelation function ρ(k) is estimated by

$$\hat{\rho}(k) = \frac{\hat{\sigma}(k)}{\hat{\sigma}(0)}, \qquad k = 0,1,\ldots,K$$

Note that

$$\hat{\rho}(0) \equiv 1$$

by definition.

The standard errors of the sample autocorrelations may be optionally computed according to argument `se_option` for the optional argument `IMSLS_SEAC`. One method (Bartlett 1946) is based on a general asymptotic expression for the variance of the sample autocorrelation coefficient of a stationary time series with independent, identically distributed normal errors. The theoretical formula is

$$\operatorname{var}\{\hat{\rho}(\mathrm{k})\} = \frac{1}{n}\sum_{i=-\infty}^{\infty}\left[\rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k)\right]$$

where

$$\hat{\rho}(k)$$

assumes μ is unknown. For computational purposes, the autocorrelations r(k) are replaced by their estimates

$$\hat{\rho}(k)$$

for |k| ≤ K, and the limits of summation are bounded because of the assumption that r(k) = 0 for all k such that |k| > K.

A second method (Moran 1947) utilizes an exact formula for the variance of the sample autocorrelation coefficient of a random process with independent, identically distributed normal errors. The theoretical formula is

$$\operatorname{var}\{\hat{\rho}(k)\} = \frac{n-k}{n(n+2)}$$

where μ is assumed to be equal to zero. Note that this formula does not depend on the autocorrelation function.

Example

Consider the Wolfer Sunspot Data (Anderson 1971, page 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Function imsls_f_autocorrelation with optional arguments computes the estimated autocovariances, estimated autocorrelations, and estimated standard errors of the autocorrelations.

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

void main()
{
   float *result=NULL, data[176][2], x[100], xmean;
   int i, nobs = 100, lagmax = 20;
   float *acv=NULL, *seac=NULL;


   imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
   for  (i=0;i<nobs;i++) x[i] = data[21+i][1];

   result = imsls_f_autocorrelation(nobs, x, lagmax,
                                IMSLS_X_MEAN_OUT, &xmean,
                                IMSLS_ACV, &acv,
                                IMSLS_SEAC, &seac, 1,
                                0);
   printf("Mean     = %8.3f\n", xmean);
   printf("Variance = %8.1f\n", acv[0]);
   printf("\nLag\t   ACV\t\t   AC\t\t   SEAC\n");
   printf("%2d\t%8.1f\t%8.5f\n", 0, acv[0], result[0]);
   for(i=1; i<21; i++)
      printf("%2d\t%8.1f\t%8.5f\t%8.5f\n", i, acv[i], result[i],
      seac[i-1]);

}
```

Output

```
Mean     =     46.976
Variance =     1382.9

Lag         ACV            AC           SEAC

 0         1382.9      1.00000
 1         1115.0      0.80629      0.03478
 2          592.0      0.42809      0.09624
 3           95.3      0.06891      0.15678
 4         -236.0     -0.17062      0.20577
 5         -370.0     -0.26756      0.23096
 6         -294.3     -0.21278      0.22899
 7          -60.4     -0.04371      0.20862
 8           227.6      0.16460      0.17848
 9           458.4      0.33146      0.14573
10           567.8      0.41061      0.13441
11           546.1      0.39491      0.15068
12           398.9      0.28848      0.17435
13           197.8      0.14300      0.19062
14            26.9      0.01945      0.19549
15           -77.3     -0.05588      0.19589
16          -143.7     -0.10394      0.19629
17          -202.0     -0.14610      0.19602
18          -245.4     -0.17743      0.19872
19          -230.8     -0.16691      0.20536
20          -142.9     -0.10332      0.20939
```

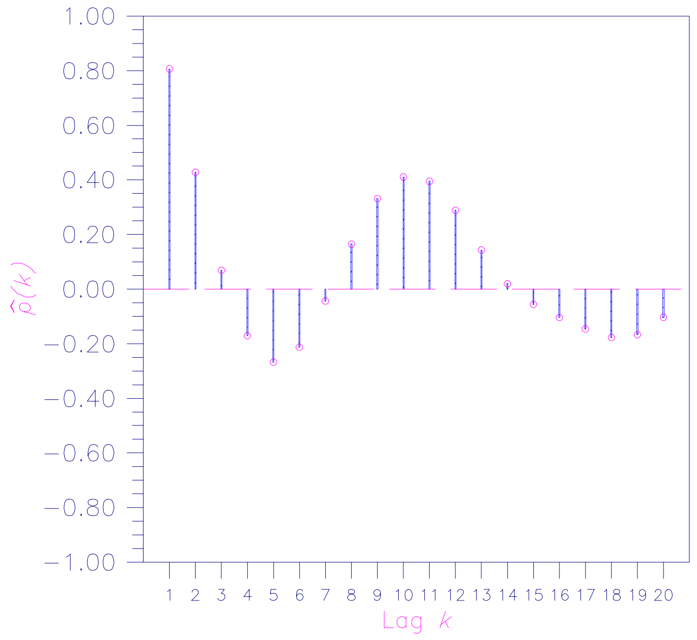


Figure 8-1 Sample Autocorrelation Function

crosscorrelation

Computes the sample cross-correlation function of two stationary time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_crosscorrelation` (*int* `n_observations`, *float* `x[]`,
    float `y[]`, *int* `lagmax`, ..., 0)

The type *double* function is `imsls_d_crosscorrelation`.

Required Arguments

int `n_observations` (Input)
: Number of observations in each time series. `n_observations` must be greater than or equal to 2.

float `x[]` (Input)
: Array of length `n_observations` containing the first time series.

float `y[]` (Input)
: Array of length `n_observations` containing the second time series.

int `lagmax` (Input)
: Maximum lag of cross-covariances and cross-correlations to be computed. `lagmax` must be greater than or equal to 1 and less than `n_observations`.

Return Value

Pointer to an array of length 2*`lagmax` + 1 containing the cross-correlations between the time series `x` and `y`. The *k*th element of this array contains the cross-correlation between `x` and `y` at lag (*k*-`lagmax`) where *k* = 0, 1, …, 2*`lagmax`. To release this space, use `free`. If no solution can be computed, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_crosscorrelation` (*int* `n_observations`, *float* `x[]`, *float* `y[]`, *int* `lagmax`,
`IMSLS_RETURN_USER`, *float* `crosscorrelations[]`,
`IMSLS_PRINT_LEVEL`, *int* `iprint`,
`IMSLS_VARIANCES`, *float* `*x_variance`, *float* `*y_variance`
`IMSLS_SE_CCF`, *float* `**standard_errors`, *int* `se_option`,
`IMSLS_SE_CCF_USER`, *float* `standard_errors[]`, *int* `se_option`,
`IMSLS_CROSS_COVARIANCES`, *float* `**cross_covariances`,
`IMSLS_CROSS_COVARIANCES_USER`, *float* `cross_covariances[]`,
`IMSLS_INPUT_MEANS`, *float* `x_mean_in`, *float* `y_mean_in`,
`IMSLS_OUTPUT_MEANS`, *float* `*x_mean_out`, *float* `*y_mean_out`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `crosscorrelations[]` (Output)
: If specified, `crosscorrelations` is an array of length 2*`lagmax + 1` containing the cross-correlations between the time series `x` and `y`. The *k*th element of this array contains the cross-correlation between `x` and `y` at lag (`k-lagmax`) where `k` = 0, 1, …, 2*`lagmax`.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
: Printing option. Default = 0.

iprint	**Action**
0	No printing is performed.
1	Prints the means and variances.

iprint	**Action**
2	Prints the means, variances, and cross-covariances.
3	Prints the means, variances, cross-covariances, cross-correlations, and standard errors of cross-correlations.

`IMSLS_VARIANCES`, *float* `*x_variance`, *float* `*y_variance` (Output)
If specified, `x_variance` is variance of the time series `x` and `y_variance` is variance of the time series `y`.

`IMSLS_SE_CCF`, *float* `**standard_errors`, *int* `se_option` (Output)
Address of a pointer to an array of length 2*`lagmax` + 1containing the standard errors of the cross-correlations between the time series `x` and `y`. Method of computation for standard errors of the cross-correlations is chosen by `se_option`.

se_option	**Action**
1	Compute standard errors of cross-correlations using Bartlett's formula.
2	Compute standard errors of cross-correlations using Bartlett's formula with the assumption of no cross-correlation.

`IMSLS_SE_CCF_USER`, *float* `standard_errors[]`, *int* `se_option` (Output)
If specified, `standard_errors` is an array of length 2*`lagmax` + 1 containing the standard errors of the cross-correlations between the time series `x` and `y`. See `IMSLS_SE_CC`.

`IMSLS_CROSS_COVARIANCES`, *float* `**cross_covariances` (Output)
Address of a pointer to an array of length 2*`lagmax` + 1 containing the cross-covariances between the time series `x` and `y`. The *k*th element of this array contains the cross-covariances between `x` and `y` at lag
(*k*-`lagmax`) where *k* = 0, 1, …, 2*`lagmax`.

`IMSLS_CROSS_COVARIANCES_USER`, *float* `cross_covariances[]` (Output)
If specified, `cross_covariances` is an array of length 2*`lagmax` + 1 the cross-covariances between the time series `x` and `y`. See `IMSLS_CROSS_COVARIANCES`.

`IMSLS_INPUT_MEANS`, *float* `x_mean_in`, *float* `y_mean_in` (Input)
If specified, `x_mean_in` is the user input of the estimate of the mean of the time series `x` and `y_mean_in` is the user input of the estimate of the mean of the time series `y`.

`IMSLS_OUTPUT_MEANS`, *float* `*x_mean_out`, *float* `*y_mean_out` (Output)
If specified, `x_mean_out` is the mean of the time series `x` and `y_mean_out` is the mean of the time series `y`.

Description

Function imsls_f_crosscorrelation estimates the cross-correlation function of two jointly stationary time series given a sample of n = `n_observations` observations $\{X_t\}$ and $\{Y_t\}$ for $t = 1, 2, \ldots, n$.

Let

$$\hat{\mu}_x = \texttt{x\_mean}$$

be the estimate of the mean μ_X of the time series $\{X_t\}$ where

$$\hat{\mu}_X = \begin{cases} \mu_X & \mu_X \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_t & \mu_X \text{ unknown} \end{cases}$$

The autocovariance function of $\{X_t\}$, $\sigma_X(k)$, is estimated by

$$\hat{\sigma}_X(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(X_{t+k} - \hat{\mu}_X), \quad k = 0, 1, \ldots, K$$

where K = `lagmax`. Note that

$$\hat{\sigma}_X(0)$$

is equivalent to the sample variance `x_variance`. The autocorrelation function $\rho_X(k)$ is estimated by

$$\hat{\rho}_X(k) = \frac{\hat{\sigma}_X(k)}{\hat{\sigma}_X(0)} \quad k = 0, 1, \ldots, K$$

Note that

$$\hat{\rho}_X(0) \equiv 1$$

by definition. Let

$$\hat{\mu}_Y = \text{y\_mean}, \hat{\sigma}_Y(k), \text{and } \hat{\rho}_Y(k)$$

be similarly defined.

The cross-covariance function $\sigma_{XY}(k)$ is estimated by

$$\hat{\sigma}_{XY}(k) = \begin{cases} \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \ldots, K \\ \frac{1}{n}\sum_{t=1-k}^{n}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \ldots, -K \end{cases}$$

The cross-correlation function $\rho_{XY}(k)$ is estimated by

$$\hat{\rho}_{XY}(k) = \frac{\hat{\sigma}_{XY}(k)}{\left[\hat{\sigma}_X(0)\hat{\sigma}_Y(0)\right]^{1/2}} \qquad k = 0, \pm 1, \ldots, \pm K$$

The standard errors of the sample cross-correlations may be optionally computed according to argument `se_option` for the optional argument `IMSLS_SE_CCF`. One method is based on a general asymptotic expression for the variance of the sample cross-correlation coefficient of two jointly stationary time series with independent, identically distributed normal errors given by Bartlett (1978, page 352). The theoretical formula is

$$\begin{aligned}\operatorname{var}\left\{\hat{\rho}_{\mathrm{XY}}(k)\right\} &= \frac{1}{n-k}\sum_{i=-\infty}^{\infty}\left[\rho_X(i)\rho_Y(i) + \rho_{XY}(i-k)\rho_{XY}(i+k)\right. \\ &\quad -2\rho_{XY}(k)\left\{\rho_X(i)\rho_{XY}(i+k) + \rho_{XY}(-i)\rho_Y(i+k)\right\} \\ &\quad \left. +\rho_{XY}^2(k)\left\{\rho_X(i) + \frac{1}{2}\rho_X^2(i) + \frac{1}{2}\rho_Y^2(i)\right\}\right]\end{aligned}$$

For computational purposes, the autocorrelations $\rho_X(k)$ and $\rho_Y(k)$ and the cross-correlations $\rho_{XY}(k)$ are replaced by their corresponding estimates for $|k| \le K$, and the limits of summation are equal to zero for all k such that $|k| > K$.

A second method evaluates Bartlett's formula under the additional assumption that the two series have no cross-correlation. The theoretical formula is

$$\operatorname{var}\left\{\hat{\rho}_{\mathrm{XY}}(k)\right\} = \frac{1}{n-k}\sum_{i=-\infty}^{\infty}\rho_X(i)\rho_Y(i) \qquad k \ge 0$$

For additional special cases of Bartlett's formula, see Box and Jenkins (1976, page 377).

An important property of the cross-covariance coefficient is $\sigma_{XY}(k) = \sigma_{YX}(-k)$ for $k \ge 0$. This result is used in the computation of the standard error of the sample cross-correlation for lag $k < 0$. In general, the cross-covariance function is not symmetric about zero so both positive and negative lags are of interest.

Example

Consider the Gas Furnace Data (Box and Jenkins 1976, pages 532–533) where X is the input gas rate in cubic feet/minute and Y is the percent CO_2 in the outlet gas. Function `imsls_f_crosscorrelation` is used to compute the cross-covariances and cross-correlations between time series X and Y with lags from −`lagmax` = −10 through lag `lagmax` = 10. In addition, the estimated standard errors of the estimated cross-correlations are computed. The standard errors are based on the additional assumption that all cross-correlations for X and Y are zero.

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

#define nobs 296
#define lagmax 10

void main ()
{
  int i;
  float data[nobs][2], x[nobs], y[nobs];
  float *secc = NULL, *ccv = NULL, *cc = NULL;
  float xmean, ymean, xvar, yvar;

  imsls_f_data_sets (7, IMSLS_X_COL_DIM, 2, IMSLS_RETURN_USER, data, 0);

  for (i = 0; i < nobs; i++)
    {
      x[i] = data[i][0];
      y[i] = data[i][1];
    }

  cc = imsls_f_crosscorrelation (nobs, x, y, lagmax,
                            IMSLS_OUTPUT_MEANS, &xmean, &ymean,
                            IMSLS_VARIANCES, &xvar, &yvar,
                            IMSLS_SE_CCF, &secc, 2,
                            IMSLS_CROSS_COVARIANCES, &ccv, 0);

  printf ("Mean of series X     = %g\n", xmean);
  printf ("Variance of series X = %g\n\n", xvar);
  printf ("Mean of series Y     = %g\n", ymean);
  printf ("Variance of series Y = %g\n\n", yvar);

  printf ("Lag            CCV           CC         SECC\n\n");
  for (i = 0; i < 2 * lagmax + 1; i++)
    printf ("%-5d%13g%13g%13g\n", i - lagmax, ccv[i], cc[i], secc[i]);
}
```

Output

```
Mean of series X     = -0.0568344
Variance of series X = 1.14694

Mean of series Y     = 53.5091
Variance of series Y = 10.2189

Lag            CCV           CC         SECC

-10      -0.404502    -0.118154     0.162754
-9       -0.508491    -0.148529      0.16247
-8        -0.61437    -0.179456     0.162188
-7       -0.705476    -0.206067     0.161907
-6       -0.776167    -0.226716     0.161627
-5       -0.831474    -0.242871     0.161349
-4       -0.891316    -0.260351     0.161073
-3       -0.980605    -0.286432     0.160798
-2        -1.12477    -0.328542     0.160524
```

```
-1         -1.34704    -0.393467     0.160252
0          -1.65853    -0.484451     0.159981
1          -2.04865    -0.598405     0.160252
2          -2.48217    -0.725033     0.160524
3          -2.88541     -0.84282     0.160798
4          -3.16536    -0.924592     0.161073
5          -3.25344    -0.950319     0.161349
6          -3.13113    -0.914593     0.161627
7          -2.83919     -0.82932     0.161907
8          -2.45302    -0.716521     0.162188
9          -2.05269    -0.599584      0.16247
10         -1.69466    -0.495004     0.162754
```

multi_crosscorrelation

Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_multi_crosscorrelation` (*int* `n_observations_x`,
int `n_channel_x`, *float* `x[]`, *int* `n_observations_y`,
int `n_channel_y`, *float* `y[]`, *int* `lagmax`, ..., 0)

The type *double* function is `imsls_d_multi_crosscorrelation`.

Required Arguments

int `n_observations_x` (Input)
Number of observations in each channel of the first time series `x`. `n_observations_x` must be greater than or equal to two.

int `n_channel_x` (Input)
Number of channels in the first time series `x`. `n_channel_x` must be greater than or equal to one.

float `x[]` (Input)
Array of length `n_observations_x` by `n_channel_x` containing the first time series.

int `n_observations_y` (Input)
Number of observations in each channel of the second time series `y`. `n_observations_y` must be greater than or equal to two.

int `n_channel_y` (Input)
Number of channels in the second time series `y`. `n_channel_y` must be greater than or equal to one.

float `y[]` (Input)
Array of length `n_observations_y` by `n_channel_y` containing the second time series.

int `lagmax` (Input)

Maximum lag of cross-covariances and cross-correlations to be computed. `lagmax` must be greater than or equal to one and less than the minimum of `n_observations_x` and `n_observations_y`.

Return Value

Pointer to an array of length `n_channel_x` * `n_channel_y` * (`2` * `lagmax + 1)` containing the cross-correlations between the channels of `x` and `y`. The *m*th element of this array contains the cross-correlation between channel *i* of the `x` series and channel *j* of the `y` series at lag (*k*-`lagmax`) where

i = 1, …, `n_channel_x`
j = 1, …, `n_channel_y`
k = 0, 1, …, 2*`lagmax`, and
m = (`n_channel_x`*`n_channel_y`**k* + (*i**`n_channel_x`+*j*))

To release this space, use `free`. If no solution can be computed, `NULL` is return.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_multi_crosscorrelation (`*int* `n_observations_x`,
int `n_channel_x`, *float* `x[]`, *int* `n_observations_y`,
int `n_channel_y`, *float* `y[]`, *int* `lagmax`,
`IMSLS_RETURN_USER`, *float* `crosscorrelations[]`,
`IMSLS_PRINT_LEVEL`, *int* `iprint`,
`IMSLS_VARIANCES`, *float* `**x_variance`, *float* `**y_variance`,
`IMSLS_VARIANCES_USER`, *float* `x_variance[]`,
float `y_variance[]`,
`IMSLS_CROSS_COVARIANCES`, *float* `**cross_covariances`,
`IMSLS_CROSS_COVARIANCES_USER`,
float `cross_covariances[]`,
`IMSLS_INPUT_MEANS`, *float* `*x_mean_in`, *float* `*y_mean_in`,
`IMSLS_OUTPUT_MEANS`, *float* `**x_mean_out`,
float `**y_mean_out`,
`IMSLS_OUTPUT_MEANS_USER`, *float* `x_mean_out[]`,
float `y_mean_out[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `crosscorrelations[]` (Output)

If specified, `crosscorrelations` is a user-specified array of `length n_channel_x * n_channel_y * (2*lagmax + 1)` containing the cross-correlations between the channels of `x` and `y`. See Return Value.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)

Printing option. Default = 0.

iprint	**Action**
0	No printing is performed.
1	Prints the means and variances.
2	Prints the means, variances, and cross-covariances.
3	Prints the means, variances, cross-covariances, and cross-correlations.

`IMSLS_VARIANCES`, *float* `**x_variance`, *float* `**y_variance` (Output)
If specified, `x_variance` is the address of a pointer to an array of length `n_channel_x` containing the variances of the channels of `x` and `y_variance` is the address of a pointer to an array of length `n_channel_y` containing the variances of the channels of `y`.

`IMSLS_VARIANCES_USER`, *float* `x_variance[]`, *float* `y_variance[]` (Output)
If specified, `x_variance` is an array of length `n_channel_x` containing the variances of the channels of `x` and `y_variance` is an array of length `n_channel_y` containing the variances of the channels of `y`. See `IMSLS_VARIANCES`.

`IMSLS_CROSS_COVARIANCES`, *float* `**cross_covariances` (Output)
Address of a pointer to an array of length `n_channel_x` * `n_channel_y` * (`2*lagmax` + 1) containing the cross-covariances between the channels of `x` and `y`. The *m*th element of this array contains the cross-covariance between channel *i* of the `x` series and channel *j* of the `y` series at lag (*k*-`lagmax`) where

i = 1, …, `n_channel_x`
j = 1, …, `n_channel_y`
k = 0, 1, …, 2*`lagmax`, and
m = (`n_channel_x`*`n_channel_y`**k* + (*i**`n_channel_x`+ *j*)).

`IMSLS_CROSS_COVARIANCES_USER`, *float* `cross_covariances[]` (Output)
If specified, `cross_covariances` is an array of length `n_channel_x` * `n_channel_y` * (`2*lagmax` + 1) containing the cross-covariances between the channels of `x` and `y`. See `IMSLS_CROSS_COVARIANCES`.

`IMSLS_INPUT_MEANS`, *float* `*x_mean_in`, *float* `*y_mean_in` (Input)
If specified, `x_mean_in` is an array of length `n_channel_x` containing the user input of the estimate of the means of the channels of `x` and `y_mean_in` is an array of length `n_channel_y` containing the user input of the estimate of the means of the channels of `y`.

`IMSLS_OUTPUT_MEANS`, *float* `**x_mean_out`, *float* `**y_mean_out` (Output)
If specified, `x_mean_out` is the address of a pointer to an array of length `n_channel_x` containing the means of the channels of `x` and `y_mean_out` is the address of a pointer to an array of length `n_channel_y` containing the means of the channels of `y`.

`IMSLS_OUTPUT_MEANS_USER`, *float* `x_mean_out[]`, *float* `y_mean_out[]` (Output)
If specified, `x_mean_out` is an array of length `n_channel_x` containing the means of the channels of `x` and `y_mean_out` is an array of length

`n_channel_y` containing the means of the channels of `y`. See `IMSLS_OUTPUT_MEANS`.

Description

Function `imsls_f_multi_crosscorrelation` estimates the multichannel cross-correlation function of two mutually stationary multichannel time series. Define the multichannel time series X by

$$X = (X_1, X_2, \ldots, X_p)$$

where

$$X_j = (X_{1j}, X_{2j}, \ldots, X_{nj})^T, \quad j = 1, 2, \ldots, p$$

with n = `n_observations_x` and p = `n_channel_x`. Similarly, define the multichannel time series Y by

$$Y = (Y_1, Y_2, \ldots, Y_q)$$

where

$$Y_j = (Y_{1j}, Y_{2j}, \ldots, Y_{mj})^T, \quad j = 1, 2, \ldots, q$$

with m = `n_observations_y` and q = `n_channel_y`. The columns of X and Y correspond to individual channels of multichannel time series and may be examined from a univariate perspective. The rows of X and Y correspond to observations of p-variate and q-variate time series, respectively, and may be examined from a multivariate perspective. Note that an alternative characterization of a multivariate time series X considers the columns to be observations of the multivariate time series while the rows contain univariate time series. For example, see Priestley (1981, page 692) and Fuller (1976, page 14).

Let

$$\hat{\mu}_X = \text{x\_mean}$$

be the row vector containing the means of the channels of X. In particular,

$$\hat{\mu}_X = \left(\hat{\mu}_{X_1}, \hat{\mu}_{X_2}, \ldots, \hat{\mu}_{X_p}\right)$$

where for $j = 1, 2, \ldots, p$

$$\hat{\mu}_{X_j} = \begin{cases} \mu_{X_j} & \mu_{X_j} \text{ known} \\ \dfrac{1}{n}\sum_{t=1}^{n} X_{tj} & \mu_{X_j} \text{ unknown} \end{cases}$$

Let

$$\hat{\mu}_Y = y\_\text{mean}$$

be similarly defined. The cross-covariance of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\sigma}_{X_iY_j}(k) = \begin{cases} \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = 0, 1, \ldots, K \\ \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = -1, -2, \ldots, -K \end{cases}$$

where $i = 1, \ldots, p, j = 1, \ldots, q$, and K = `lagmax`. The summation on t extends over all possible cross-products with N equal to the number of cross-products in the sum

Let

$$\hat{\sigma}_X(0) = \text{x\_variance}$$

be the row vector consisting of the estimated variances of the channels of X. In particular,

$$\hat{\sigma}_X(0) = (\hat{\sigma}_{X_1}(0), \hat{\sigma}_{X_2}(0), \ldots, \hat{\sigma}_{X_p}(0))$$

where

$$\hat{\sigma}_{X_j}(0) = \frac{1}{n}\sum_{t=1}^{n} X_{tj} - \hat{\mu}_{X_j})^2 \quad j = 1, 2, \ldots, p$$

Let

$$\hat{\sigma}_Y(0) = \text{y\_variance}$$

be similarly defined. The cross-correlation of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\rho}_{X_iY_j}(k) = \frac{\hat{\sigma}_{X_iY_j(k)}}{\left[\hat{\sigma}_{X_i}(0)\hat{\sigma}_{Y_j}(0)\right]^{1/2}} \quad k = 0, \pm 1, \ldots, \pm K$$

Example

Consider the Wolfer Sunspot Data (Y) (Box and Jenkins 1976, page 530) along with data on northern light activity (X_1) and earthquake activity (X_2) (Robinson 1967, page 204) to be a three-channel time series. Function `imsls_f_multi_crosscorrelation` is used to compute the cross-covariances and cross-correlations between X_1 and Y and between X_2 and Y with lags from −`lagmax` = −10 through lag `lagmax` = 10.

```
#include "imsls.h"

void main () {
  int i, lagmax, nobsx, nchanx, nobsy, nchany;
  float x[100 * 2], y[100], *result = NULL, *xvar = NULL, *yvar = NULL,
    *xmean = NULL, *ymean = NULL, *ccv = NULL;
  float data[100][4];
  char line[20];

  nobsx = nobsy = 100;
  nchanx = 2;
  nchany = 1;
  lagmax = 10;

  imsls_f_data_sets (8, IMSLS_X_COL_DIM, 4, IMSLS_RETURN_USER, data, 0);
  for (i = 0; i < 100; i++)
    {
      y[i] = data[i][1];
      x[i * 2] = data[i][2];
      x[i * 2 + 1] = data[i][3];
    }

  result =
    imsls_f_multi_crosscorrelation (nobsx, nchanx, &x[0], nobsy, nchany,
                                &y[0], lagmax, IMSLS_VARIANCES, &xvar,
                                &yvar, IMSLS_OUTPUT_MEANS, &xmean, &ymean,
                                IMSLS_CROSS_COVARIANCES, &ccv, 0);

  imsls_f_write_matrix ("Channel means of x", 1, nchanx, xmean, 0);
  imsls_f_write_matrix ("Channel variances of x", 1, nchanx, xvar, 0);
  imsls_f_write_matrix ("Channel means of y", 1, nchany, ymean, 0);
  imsls_f_write_matrix ("Channel variances of y", 1, nchany, yvar, 0);

  printf ("\nMultichannel cross-covariance between x and y\n");
  for (i = 0; i < (2 * lagmax + 1); i++)
    {
      sprintf (line, "Lag K = %d", i - lagmax);
      imsls_f_write_matrix (line, nchanx, nchany,
                          &ccv[nchanx * nchany * i], 0);
    }

  printf ("\nMultichannel cross-correlation between x and y\n");
  for (i = 0; i < (2 * lagmax + 1); i++)
    {
      sprintf (line, "Lag K = %d", i - lagmax);
      imsls_f_write_matrix (line, nchanx, nchany,
                          &result[nchanx * nchany * i], 0);
    }
}
```

Output

```
   Channel means of x
```

```
         1            2
     63.43        97.97

 Channel variances of x
         1            2
      2644         1978

Channel means of y
          46.94

Channel variances of y
             1384

Multichannel cross-covariance between x and y

  Lag K = -10
1       -20.51
2        70.71

  Lag K = -9
1        65.02
2        38.14

  Lag K = -8
1        216.6
2        135.6

  Lag K = -7
1        246.8
2        100.4

  Lag K = -6
1        142.1
2         45.0

  Lag K = -5
1        50.70
2       -11.81

  Lag K = -4
1        72.68
2        32.69

  Lag K = -3
1        217.9
2        -40.1

  Lag K = -2
1        355.8
2       -152.6

  Lag K = -1
1        579.7
2       -213.0
```

```
   Lag K = 0
1        821.6
2       -104.8

   Lag K = 1
1        810.1
2         55.2

   Lag K = 2
1        628.4
2         84.8

   Lag K = 3
1        438.3
2         76.0

   Lag K = 4
1        238.8
2        200.4

   Lag K = 5
1        143.6
2        283.0

   Lag K = 6
1        253.0
2        234.4

   Lag K = 7
1        479.5
2        223.0

   Lag K = 8
1        724.9
2        124.5

   Lag K = 9
1        925.0
2        -79.5

  Lag K = 10
1        922.8
2       -279.3

Multichannel cross-correlation between x and y

  Lag K = -10
1     -0.01072
2      0.04274

  Lag K = -9
1      0.03400
2      0.02305

  Lag K = -8
```

```
1       0.1133
2       0.0819

  Lag K = -7
1       0.1290
2       0.0607

  Lag K = -6
1      0.07431
2      0.02718

  Lag K = -5
1      0.02651
2     -0.00714

  Lag K = -4
1      0.03800
2      0.01976

  Lag K = -3
1       0.1139
2      -0.0242

  Lag K = -2
1       0.1860
2      -0.0923

  Lag K = -1
1       0.3031
2      -0.1287

   Lag K = 0
1       0.4296
2      -0.0633

   Lag K = 1
1       0.4236
2       0.0333

   Lag K = 2
1       0.3285
2       0.0512

   Lag K = 3
1       0.2291
2       0.0459

   Lag K = 4
1       0.1248
2       0.1211

   Lag K = 5
1       0.0751
2       0.1710
```

```
   Lag K = 6
1       0.1323
2       0.1417

   Lag K = 7
1       0.2507
2       0.1348

   Lag K = 8
1       0.3790
2       0.0752

   Lag K = 9
1       0.4836
2      -0.0481

  Lag K = 10
1       0.4825
2      -0.1688
```

partial_autocorrelation

Computes the sample partial autocorrelation function of a stationary time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_partial_autocorrelation` (*int* `lagmax`, *int* `cf[]`, …, 0)

The type *double* function is `imsls_d_partial_autocorrelation`.

Required Arguments

int `lagmax` (Input)
Maximum lag of partial autocorrelations to be computed.

float `cf[]` (Input)
Array of length `lagmax + 1` containing the autocorrelations of the time series `x`.

Return Value

Pointer to an array of length `lagmax` containing the partial autocorrelations of the time series `x`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_partial_autocorrelation` (*int* `lagmax`, *float* `cf[]`,
`IMSLS_RETURN_USER`, *float* `partial_autocorrelations[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `partial_autocorrelations[]` (Output)
: If specified, the partial autocorrelations are stored in an array of length `lagmax` provided by the user.

Description

Function `imsls_f_partial_autocorrelation` estimates the partial autocorrelations of a stationary time series given the K = `lagmax` sample autocorrelations

$$\hat{\rho}(k)$$

for $k = 0, 1, \ldots, K$. Consider the AR(k) process defined by

$$X_t = \phi_{k1} X_{t-1} + \phi_{k2} X_{t-2} + \ldots + \phi_{kk} X_{t-k} + A_t$$

where ϕ_{kj} denotes the j-th coefficient in the process. The set of estimates

$$\left\{ \hat{\phi}_{kk} \right\}$$

for $k = 1, \ldots, K$ is the sample partial autocorrelation function. The autoregressive parameters

$$\left\{ \hat{\phi}_{kj} \right\}$$

for $j = 1, \ldots, k$ are approximated by Yule-Walker estimates for successive AR(k) models where $k = 1, \ldots, K$. Based on the sample Yule-Walker equations

$$\hat{\rho}(j) = \hat{\phi}_{k1} \hat{\rho}(j-1) + \hat{\phi}_{k2} \hat{\rho}(j-2) + \ldots + \hat{\phi}_{kk} \hat{\rho}(j-k), \quad j = 1, 2, \ldots, k$$

a recursive relationship for $k = 1, \ldots, K$ was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & k = 1 \\ \dfrac{\hat{\rho}(k) - \Sigma_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(k-j)}{1 - \Sigma_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(j)} & k = 2, \ldots, K \end{cases}$$

and

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & k = 1 \\ \dfrac{\hat{\rho}(k) - \Sigma_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(k-j)}{1 - \Sigma_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(j)} & k = 2, \ldots, K \end{cases}$$

This procedure is sensitive to rounding error and should not be used if the parameters are near the nonstationarity boundary. A possible alternative would be to estimate $\{\phi_{kk}\}$ for successive AR(k) models using least or maximum likelihood. Based on the hypothesis that the true process is AR(p), Box and Jenkins (1976, page 65) note

$$\text{var}\{\hat{\phi}_{kk}\} \simeq \frac{1}{n} \qquad k \geq p+1$$

See Box and Jenkins (1976, pages 82–84) for more information concerning the partial autocorrelation function.

Example

Consider the Wolfer Sunspot Data (Anderson 1971, page 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Routine imsls_f_partial_autocorrelation is used to compute the estimated partial autocorrelations.

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

void main()
{
   float *partial=NULL, data[176][2], x[100];
   int i, nobs = 100, lagmax = 20;
   float *ac;

   imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
   for (i=0;i<nobs;i++) x[i] = data[21+i][1];

   ac = imsls_f_autocorrelation(100, x, lagmax, 0);
   partial = imsls_f_partial_autocorrelation(lagmax, ac, 0);
   imsls_f_write_matrix("Lag      PACF", 20, 1, partial, 0);
}
```

Output

```
 Lag      PACF
 1     0.806
 2    -0.635
 3     0.078
 4    -0.059
 5    -0.001
 6     0.172
 7     0.109
 8     0.110
 9     0.079
10     0.079
11     0.069
12    -0.038
13     0.081
14     0.033
```

```
15    -0.035
16    -0.131
17    -0.155
18    -0.119
19    -0.016
20    -0.004
```

lack_of_fit

Performs lack-of-fit test for a univariate time series or transfer function given the appropriate correlation function.

Synopsis

#include `<imsls.h>`

float `imsls_lack_of_fit` (*int* `n_observations`, *float* `cf[]`,
int `lagmax`, *int* `npfree`,..., 0)

Required Arguments

int `n_observations` (Input)
Number of observations of the stationary time series.

float `cf[]` (Input)
Array of length `lagmax`+1 containing the correlation function.

int `lagmax` (Input)
Maximum lag of the correlation function.

int `npfree` (Input)
Number of free parameters in the formulation of the time series model. `npfree` must be greater than or equal to zero and less than `lagmax`. Woodfield (1990) recommends `npfree` = `p` + `q`.

Return Value

Pointer to an array of length 2 with the test statistic, Q, and its *p*-value, *p*. Under the null hypothesis, Q has an approximate chi-squared distribution with `lagmax-lagmin+1-npfree` degrees of freedom.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_lack_of_fit` (*int* `n_observations`, *float* `cf[]`, *int* `lagmax`,
int `npfree`,
`IMSLS_RETURN_USER`, *float* `stat[]`,
`IMSLS_LAGMIN`, *int* `lagmin`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `stat[]` (Input)
User defined array for storage of lack-of-fit statistics.

`IMSLS_LAGMIN`, *int* `lagmin` (Input)
Minimum lag of the correlation function. `lagmin` corresponds to the lower bound of summation in the lack of fit test statistic. Default value is 1.

Description

Routine `imsls_f_lack_of_fit` may be used to diagnose lack of fit in both ARMA and transfer function models. Typical arguments for these situations are:

Model	LAGMIN	LAGMAX	NPFREE
ARMA (*p*, *q*)	1	$\sqrt{\text{NOBS}}$	$p + q$
Transfer function	0	$\sqrt{\text{NOBS}}$	$R + s$

Function `imsls_f_lack_of_fit` performs a portmanteau lack of fit test for a time series or transfer function containing `n` observations given the appropriate sample correlation function

$$\hat{\rho}(k)$$

for $k = L, L + 1, \ldots, K$ where $L =$ `lagmin` and $K =$ `lagmax`.

The basic form of the test statistic Q is

$$Q = n(n+2)\sum_{k=L}^{K}(n-k)^{-1}\hat{\rho}(k)$$

with $L = 1$ if

$$\hat{\rho}(k)$$

is an autocorrelation function. Given that the model is adequate, Q has a chi-squared distribution with $K - L + 1 - m$ degrees of freedom where $m =$ `npfree` is the number of parameters estimated in the model. If the mean of the time series is estimated, Woodfield (1990) recommends not including this in the count of the parameters estimated in the model. Thus, for an ARMA(p, q) model set `npfree`$= p + q$ regardless of whether the mean is estimated or not. The original derivation for time series models is due to Box and Pierce (1970) with the above modified version discussed by Ljung and Box (1978). The extension of the test to transfer function models is discussed by Box and Jenkins (1976, pages 394–395).

Example

Consider the Wölfer Sunspot Data (Anderson 1971, page 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. An ARMA(2,1) with nonzero mean is fitted using routine `imsls_f_arma`. The autocorrelations of the residuals are estimated using routine imsls_f_autocorrelation. A portmanteau lack of fit test is computed using 10 lags with imsls_f_lack_of_fit.

The warning message from imsls_f_arma in the output can be ignored. (See the example for routine imsls_f_arma for a full explanation of the warning message.)

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

void main()
{
  int   p = 2;
  int   q = 1;
  int   i;
  int   n_observations = 100;
  int   max_itereations = 0;
  int   lagmin = 1;
  int   lagmax = 10;
  int   npfree = 4;
  float data[176][2], x[100];
  float *parameters;
  float *correlations;
  float *residuals;
  float tolerance = 0.125;
  float *result;

  /* Get sunspot data for 1770 through 1869, store it in x[].      */
  imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
  for (i=0;i<n_observations;i++) x[i] = data[21+i][1];

  /* Get residuals from ARMA(2,1) for autocorrelation/lack of fit  */
  parameters = imsls_f_arma(n_observations, x, p, q,
                            IMSLS_LEAST_SQUARES,
                            IMSLS_CONVERGENCE_TOLERANCE, tolerance,
                            IMSLS_RESIDUAL, &residuals,
                            0);
  /* Get autocorrelations from residuals for lack of fit test      */
  /*     NOTE:  number of OBS is equal to number of residuals      */

correlations = imsls_f_autocorrelation(n_observations-p+lagmax,
   residuals, lagmax,
                                       0);

  /*  Get lack of fit test statistic and p-value                   */
  /*     NOTE:  number of OBS is equal to original number of data  */

   result = imsls_f_lack_of_fit(n_observations,  correlations, lagmax,
  npfree, 0);

  /*  Print parameter estimates, test statistic, and p-value       */
  /*     NOTE: Test Statistic Q follows a Chi-squared dist.        */

 printf("Lack of Fit Statistic,  Q = \t%3.5f\n             P-value of Q
          = \t %1.5f\n\n",result[0], result[1]);
}
```

Output

```
***WARNING  ERROR  IMSLS_LEAST_SQUARES_FAILED from imsls_f_arma.  Least
***         squares estimation of the parameters has failed to converge.
***         Increase "length" and/or "tolerence" and/or
***         "convergence_tolerence".  The estimates of the parameters at
***         the last iteration may be used as new starting values.

Lack of Fit statistic (Q) =       14.572

         P-value (PVALUE) =       0.9761
```

estimate_missing

Estimates missing values in a time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_estimate_missing`(*int* `n_obs`, *int* `tpoints[]`,
float `z[]`,…,0)

The type *double* function is `imsls_d_estimate_missing`.

Required Arguments

int `n_obs` (Input)
Number of non-missing observations in the time series. The time series must not contain gaps with more than 3 missing values.

int `tpoints[]` (Input)
Vector of length `n_obs` containing the time points $t_1,\ldots,t_{n\_obs}$ at which the time series values were observed. The time points must be in strictly increasing order. Time points for missing values must lie in the open interval (t_1, t_{n_obs}).

float `z[]` (Input)
Vector of length `n_obs` containing the time series values. The values must be ordered in accordance with the values in vector `tpoints`. It is assumed that the time series after estimation of missing values contains values at equidistant time points where the distance between two consecutive time points is one. If the non-missing time series values are observed at time points $t_1,\ldots,t_{n\_obs}$, then missing values between t_i and t_{i+1}, $i = 1,\ldots,\texttt{n\_obs}-1$, exist if $t_{i+1} - t_i > 1$. The size of the gap between t_i and t_{i+1} is then $t_{i+1} - t_i - 1$. The total length of the time series with non-missing and estimated missing values is $t_{n\_obs} - t_1 + 1$, or `tpoints[n_obs-1]-tpoints[0]`+1.

Return Value

Pointer to an array of length (`tpoints[n_obs-1]-tpoints[0]+1`) containing the time series together with estimates for the missing values. If an error occurred, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_estimate_missing` (*int* `n_obs`, *int* `tpoints[]`, *float* `z[]`,
`IMSLS_METHOD`, *int* `method`,
`IMSLS_MAX_LAG`, *int* `maxlag`,
`IMSLS_NTIMES`, *int* `*ntimes`,
`IMSLS_MEAN_ESTIMATE`, *float* `mean_estimate`,
`IMSLS_CONVERGENCE_TOLERANCE`, *float* `convergence_tolerance`,
`IMSLS_RELATIVE_ERROR`, *float* `relative_error`,
`IMSLS_MAX_ITERATIONS`, *int* `max_iterations`,
`IMSLS_TIMES_ARRAY`, *int* `**times`,
`IMSLS_TIMES_ARRAY_USER`, *int* `times[]`,
`IMSLS_MISSING_INDEX`, *int* `**missing_index`,
`IMSLS_MISSING_INDEX_USER`, *int* `missing_index[]`,
`IMSLS_RETURN_USER`, *float* `u_z[]`,
0)

Optional Arguments

`IMSLS_METHOD`, *int* `method` (Input)
The method used for estimating the missing values:
0 — Use median.
1 — Use cubic spline interpolation.
2 — Use one-step-ahead forecasts from an AR(1) model.
3 — Use one-step-ahead forecasts from an AR(p) model.
Default: `method` = 3
If `method` = 2 is chosen, then all values of gaps beginning at time points t_1+1 or t_1+2 are estimated by `method` 0. If `method` = 3 is chosen and the first gap starts at t_1+1, then the values of this gap are also estimated by `method` 0. If the length of the series before a gap, denoted `len`, is greater than 1 and less than $2\cdot$`maxlag`, then maxlag is reduced to `len/2` for the computation of the missing values within this gap.

`IMSLS_MAX_LAG`, *int* `maxlag` (Input)
Maximum lag number when `method` = 3 was chosen.
Default: `maxlag` = 10

`IMSLS_NTIMES`, *int* `*ntimes` (Output)
Number of elements in the time series with estimated missing values. Note that `ntimes = tpoints[n_obs-1]-tpoints[0]+1`.

`IMSLS_MEAN_ESTIMATE`, *float* `mean_estimate` (Input)
Estimate of the mean of the time series.

`IMSLS_CONVERGENCE_TOLERANCE`, *float* `convergence_tolerance` (Input)
Tolerance level used to determine convergence of the nonlinear least squares algorithm used in method 2. Argument `convergence_tolerance` represents

the minimum relative decrease in the sum of squares between two iterations required to determine convergence. Hence, `convergence_tolerance` must be greater than or equal to 0.
Default: $\max\{10^{-10}, \text{eps}^{2/3}\}$ for single precision and $\max\{10^{-20}, \text{eps}^{2/3}\}$ for double precision, where eps =`imsls_f_machine(4)` for single precision and eps =`imsls_d_machine(4)` for double precision.

`IMSLS_RELATIVE_ERROR`, *float* `relative_error` (Input)
Stopping criterion for use in the nonlinear equation solver used by method 2.
Default: `relative_error` = 100 × `imsls_f_machine(4)` for single precision, `relative_error` = 100 × `imsls_d_machine(4)` for double precision..

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations allowed in the nonlinear equations solver used by method 2.
Default: `max_iterations` = 200.

`IMSLS_TIMES_ARRAY`, *int* `**times` (Output)
Address of a pointer to an internally allocated array of length `ntimes` = `tpoints[n_obs-1]-tpoints[0]+1` containing the time points of the time series with estimates for the missing values.

`IMSLS_TIMES_ARRAY_USER`, *int* `times[]` (Output)
Storage for array times is provided by the user. See `IMSLS_TIMES_ARRAY`.

`IMSLS_MISSING_INDEX`, *int* `**missing_index` (Output)
Address of a pointer to an internally allocated array of length (`ntimes-n_obs`) containing the indices for the missing values in array times. If `ntimes-n_obs` = 0, then no missing value could be found and `NULL` is returned.

`IMSLS_MISSING_INDEX_USER`, *int* `missing_index[]` (Output)
Storage for array `missing_index` is provided by the user. See `IMSLS_MISSING_INDEX`.

`IMSLS_RETURN_USER`, *float* `u_z[]` (Output)
If specified, `u_z` is a vector of length `tpoints[n_obs-1]-tpoints[0]+1` containing the time series values together with estimates for missing values.

Description

Traditional time series analysis as described by Box, Jenkins and Reinsel (1994) requires the observations made at equidistant time points $t_1, t_1+1, t_1+2, \ldots, t_n$. When observations are missing, the problem occurs to determine suitable estimates. Function `imsls_f_estimate_missing` offers 4 estimation methods:
Method 0 estimates the missing observations in a gap by the median of the last four time series values before and the first four values after the gap. If not enough values are available before or after the gap then the number is reduced accordingly. This method is very fast and simple, but its use is limited to stationary ergodic series without outliers and level shifts.

Method 1 uses a cubic spline interpolation method to estimate missing values. Here the interpolation is again done over the last four time series values before and the first four values after the gap. The missing values are estimated by the resulting interpolant. This method gives smooth transitions across missing values.

Method 2 assumes that the time series before the gap can be well described by an AR(1) process. If the last observation prior to the gap is made at time point t_m then it uses the time series values at $t_1, t_1+1, \ldots, t_m$ to compute the one-step-ahead forecast at origin t_m. This value is taken as an estimate for the missing value at time point t_m+1. If the value at t_m+2 is also missing then the values at time points $t_1, t_1+1, \ldots, t_m+1$ are used to recompute the AR(1) model, estimate the value at t_m+2 and so on. The coefficient ϕ_1 in the AR(1) model is computed internally by the method of least squares from routine imsls_f_arma.

Finally, method 3 uses an AR(p) model to estimate missing values by a one-step-ahead forecast . First, function imsls_f_auto_uni_ar, applied to the time series prior to the missing values, is used to determine the optimum `p` from the set {0, 1, …, `maxlag`} of possible values and to compute the parameters $\phi_1, \ldots, \phi_p$ of the resulting AR(p) model. The parameters are estimated by the least squares method based on Householder transformations as described in Kitagawa and Akaike (1978). Denoting the mean of the series $y_{t_1}, y_{t_1+1}, \ldots, y_{t_m}$ by μ the one-step-ahead forecast at origin t_m, $\hat{y}_{t_m}(1)$, can be computed by the formula

$$\hat{y}_{t_m}(1) = \mu\left(1 - \sum\nolimits_{j=1}^{p} \phi_j\right) + \sum\nolimits_{j=1}^{p} \phi_j y_{t_m+1-j} \ .$$

This value is used as an estimate for the missing value. The procedure starting with imsls_f_auto_uni_ar is then repeated for every further missing value in the gap. All four estimation methods treat gaps of missing values in increasing time order.

Example

Consider the AR(1) process

$$Y_t = \phi_1 Y_{t-1} + a_t, \quad t = 1, 2, 3, \ldots$$

We assume that $\{a_t\}$ is a Gaussian white noise process, $a_t \sim N(0, \sigma^2)$. Then, $E[Y_t] = 0$ and $VAR[Y_t] = \sigma^2 / (1 - \phi_1^2)$ (see Anderson, p. 174).

The time series in the code below was artificially generated from an AR(1) process characterized by $\phi_1 = -0.7$ and $\sigma^2 = 1 - \phi_1^2 = 0.51$. This process is stationary with $VAR[Y_t] = 1$. As initial value, $Y_0 := a_0$ was taken. The sequence $\{a_t\}$ was generated by a random number generator.

From the original series, we remove the observations at time points *t*=130, *t*=140, *t*=141, *t*=160, *t*=175, *t*=176. Then, imsls_f_estimate_missing is used to compute estimates for the missing values by all 4 estimation methods available. The estimated values are compared with the actual values.

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

void main()
{
  int i, j, k;
  int maxlag = 20;
  int times_1[200], times_2[200];
  float x_1[200], x_2[200];
  int ntemp;
  int n_obs, n_miss;
  int ntimes;
  float *result = NULL;
  int *times = NULL, *missing_index = NULL;
  int miss_ind;


  float   y[200] = {
       1.30540,-1.37166,1.47905,-0.91059,1.36191,-2.16966,3.11254,
      -1.99536,2.29740,-1.82474,-0.25445,0.33519,-0.25480,-0.50574,
      -0.21429,-0.45932,-0.63813,0.25646,-0.46243,-0.44104,0.42733,
       0.61102,-0.82417,1.48537,-1.57733,-0.09846,0.46311,0.49156,
      -1.66090,2.02808,-1.45768,1.36115,-0.65973,1.13332,-0.86285,
       1.23848,-0.57301,-0.28210,0.20195,0.06981,0.28454,0.19745,
      -0.16490,-1.05019,0.78652,-0.40447,0.71514,-0.90003,1.83604,
      -2.51205,1.00526,-1.01683,1.70691,-1.86564,1.84912,-1.33120,
       2.35105,-0.45579,-0.57773,-0.55226,0.88371,0.23138,0.59984,
       0.31971,0.59849,0.41873,-0.46955,0.53003,-1.17203,1.52937,
      -0.48017,-0.93830,1.00651,-1.41493,-0.42188,-0.67010,0.58079,
      -0.96193,0.22763,-0.92214,1.35697,-1.47008,2.47841,-1.50522,
       0.41650,-0.21669,-0.90297,0.00274,-1.04863,0.66192,-0.39143,
       0.40779,-0.68174,-0.04700,-0.84469,0.30735,-0.68412,0.25888,
      -1.08642,0.52928,0.72168,-0.18199,-0.09499,0.67610,0.14636,
       0.46846,-0.13989,0.50856,-0.22268,0.92756,0.73069,0.78998,
      -1.01650,1.25637,-2.36179,1.99616,-1.54326,1.38220,0.19674,
      -0.85241,0.40463,0.39523,-0.60721,0.25041,-1.24967,0.26727,
       1.40042,-0.66963,1.26049,-0.92074,0.05909,-0.61926,1.41550,
       0.25537,-0.13240,-0.07543,0.10413,1.42445,-1.37379,0.44382,
      -1.57210,2.04702,-2.22450,1.27698,0.01073,-0.88459,0.88194,
      -0.25019,0.70224,-0.41855,0.93850,0.36007,-0.46043,0.18645,
       0.06337,0.29414,-0.20054,0.83078,-1.62530,2.64925,-1.25355,
       1.59094,-1.00684,1.03196,-1.58045,2.04295,-2.38264,1.65095,
      -0.33273,-1.29092,0.14020,-0.11434,0.04392,0.05293,-0.42277,
       0.59143,-0.03347,-0.58457,0.87030,0.19985,-0.73500,0.73640,
       0.29531,0.22325,-0.60035,1.42253,-1.11278,1.30468,-0.41923,
      -0.38019,0.50937,0.23051,0.46496,0.02459,-0.68478,0.25821,
       1.17655,-2.26629,1.41173,-0.68331
  };

    int tpoints[200] = {
     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
     25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
     46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,
     67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,
     88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,
     107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,
     123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,
     139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,
     155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,
     171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,
     187,188,189,190,191,192,193,194,195,196,197,198,199,200
  };


    n_miss = 0;
    times_1[0] = times_2[0] = tpoints[0];
    x_1[0] = x_2[0] = y[0];
    k = 0;

    for (i=1; i<200;i++)
    {
       times_1[i] = tpoints[i];
       x_1[i] = y[i];

       /* Generate series with missing values  */
       if ( i!=129 && i!= 139 && i!=140 && i!=159 && i!=174 && i!=175 )
       {
          k += 1;
          times_2[k] = times_1[i];
          x_2[k] = x_1[i];
       }
    }

    n_obs = k + 1;

    for (j=0;j<=3;j++)
    {
       if (j <= 2)
         result = imsls_f_estimate_missing(n_obs, times_2, x_2,
                                          IMSLS_METHOD, j,
                                          IMSLS_NTIMES, &ntimes,
                                          IMSLS_TIMES_ARRAY, &times,
                                          IMSLS_MISSING_INDEX,
&missing_index,
                                          0);
       else
         result = imsls_f_estimate_missing(n_obs, times_2, x_2,
                                          IMSLS_METHOD, j,
                                          IMSLS_NTIMES, &ntimes,
                                          IMSLS_MAX_LAG, 20,
                                          IMSLS_TIMES_ARRAY, &times,
                                          IMSLS_MISSING_INDEX,
&missing_index,
                                          0);


       if (!result)
       {
          if (times)
          {
             free(times);
             times = NULL;
          }
          if (missing_index)
          {
             free(missing_index);
             missing_index = NULL;
          }

          return;
       }

       if (j == 0) printf("\nMethod: Median\n");
       if (j == 1) printf("\nMethod: Cubic Spline Interpolation\n");
       if (j == 2) printf("\nMethod: AR(1) Forecast\n");
       if (j == 3) printf("\nMethod: AR(p) Forecast\n");

       printf("ntimes = %d\n", ntimes);
       printf("time\tactual\tpredicted\tdifference\n");

       n_miss = ntimes-n_obs;

       for (i = 0; i < n_miss; i++)
         {
          miss_ind = missing_index[i];
          printf("%d, %10.5f, %10.5f, %18.6f\n", times[miss_ind],
                      x_1[miss_ind], result[miss_ind],
                      fabs(x_1[miss_ind]-result[miss_ind]));
         }

       if (result)
       {
         free(result);
         result = NULL;
       }
       if (times)
       {
         free(times);
         times = NULL;
       }
       if (missing_index)
       {
         free(missing_index);
         missing_index = NULL;
       }
  }

  return;
}
```

Output

```
Method: Median
ntimes = 200
```

```
time    actual      predicted        difference
130,   -0.92074,    0.26132,           1.182060
140,    0.44382,    0.05743,           0.386390
141,   -1.57210,    0.05743,           1.629530
160,    2.64925,    0.04680,           2.602450
175,   -0.42277,    0.04843,           0.471195
176,    0.59143,    0.04843,           0.543005

Method: Cubic Spline Interpolation
ntimes = 200
time    actual      predicted        difference
130,   -0.92074,    1.54109,           2.461829
140,    0.44382,   -0.40730,           0.851119
141,   -1.57210,    2.49709,           4.069194
160,    2.64925,   -2.94712,           5.596371
175,   -0.42277,    0.25066,           0.673430
176,    0.59143,    0.38032,           0.211107

Method: AR(1) Forecast
ntimes = 200
time    actual     predicted           difference
130,   -0.92074,   -0.92971,           0.008968
140,    0.44382,    1.02824,           0.584424
141,   -1.57210,   -0.74527,           0.826832
160,    2.64925,    1.22880,           1.420454
175,   -0.42277,    0.01049,           0.433259
176,    0.59143,    0.03683,           0.554601

Method: AR(p) Forecast
ntimes = 200
time    actual      predicted        difference
130,   -0.92074,   -0.86385,           0.056894
140,    0.44382,    0.98098,           0.537164
141,   -1.57210,   -0.64489,           0.927206
160,    2.64925,    1.18966,           1.459592
175,   -0.42277,   -0.00105,           0.421722
176,    0.59143,    0.03773,           0.553705
```

garch

Computes estimates of the parameters of a GARCH(p,q) model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_garch` (*int* `p`, *int* `q`, *int* `m`, *float* `y[]`, *float* `xguess[]`, …, 0)

The type *double* function is `imsls_d_garch`.

Required Arguments

int `p` (Input)
Number of GARCH parameters.

int `q` (Input)
    Number of ARCH parameters.

int `m` (Input)
    Length of the observed time series.

float `y[]` (Input)
    Array of length `m` containing the observed time series data.

float `xguess[]` (Input)
    Array of length `p` + `q` + 1 containing the initial values for the parameter array `x[]`.

Return Value

Pointer to the parameter array `x[]` of length `p` + `q` + 1 containing the estimated values of sigma squared, followed by the `q` ARCH parameters, and the `p` GARCH parameters.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_garch` (*int* `p`, *int* `q`, *int* `m`, *float* `y[]`, *float* `xguess[]`,
    `IMSLS_MAX_SIGMA`, *float* `max_sigma`,
    `IMSLS_A`, *float* `*a`,
    `IMSLS_AIC`, *float* `*aic`,
    `IMSLS_VAR`, *float* `*var`,
    `IMSLS_VAR_USER`, *float* `var[]`,
    `IMSLS_VAR_COL_DIM`, *int* `var_col_dim`,
    `IMSLS_RETURN_USER`, *float* `x[]`,
    0)

Optional Arguments

`IMSLS_MAX_SIGMA`, *float* `max_sigma`, (Input)
    Value of the upperbound on the first element (sigma) of the array of returned estimated coefficients. Default = 10.

`IMSLS_A`, *float* `*a`, (Output)
    Value of Log-likelihood function evaluated at the estimated parameter array `x`.

`IMSLS_AIC`, *float* `*aic`, (Output)
    Value of Akaike Information Criterion evaluated at the estimated parameter array `x`.

`IMSLS_VAR`, *float* `*var`, (Output)
    Array of size $(p+q+1)$x$(p+q+1)$ containing the variance-covariance matrix.

`IMSLS_VAR_USER`, *float* `var[]`, (Output)
    Storage for array `var` is provided by the user.
    See `IMSLS_VAR`.

`IMSLS_VAR_COL_DIM`, *int* `var_col_dim`, (Input)
Column dimension `(p+q+1)` of the variance-covariance matrix.

`IMSLS_RETURN_USER`, *float* `x[]`, (Output)
If specified, `x` returns an array of length $p + q + 1$ containing the estimated values of sigma squared, followed by the q ARCH parameters, and the p GARCH parameters. Storage for estimated parameter array `x` is provided by the user.

Description

The Generalized Autoregressive Conditional Heteroskedastic (GARCH) model for a time series $\{w_t\}$ is defined as

$$w_t = z_t \sigma_t$$

$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i w_{t-i}^2,$$

where z_t's are independent and identically distributed standard normal random variables,

$$0 < \sigma^2 < \texttt{max\_sigma}, \ \beta_i \ge 0, \ \alpha_i \ge 0 \ \text{ and}$$

$$\sum_{i=2}^{p+q+1} x(i) = \sum_{i=1}^{p} \beta_i + \sum_{i=1}^{q} \alpha_i < 1.$$

The above model is denoted as GARCH(p,q). The β_i and α_i coeffecients will be referred to as GARCH and ARCH coefficents, respectively. When $\beta_i = 0$, $i = 1,2,\ldots,p$, the above model reduces to ARCH(q) which was proposed by Engle (1982). The nonnegativity conditions on the parameters imply a nonnegative variance and the condition on the sum of the β_i's and α_i's is required for wide sense stationarity.

In the empirical analysis of observed data, GARCH(1,1) or GARCH(1,2) models have often found to appropriately account for conditional heteroskedasticity (Palm 1996). This finding is similar to linear time series analysis based on ARMA models.

It is important to notice that for the above models positive and negative past values have a symmetric impact on the conditional variance. In practice, many series may have strong asymmetric influence on the conditional variance. To take into account this phenomena, Nelson (1991) put forward Exponential GARCH (EGARCH). Lai (1998) proposed and studied some properties of a general class of models that extended linear relationship of the conditional variance in ARCH and GARCH into nonlinear fashion.

The maximum likelihood method is used in estimating the parameters in GARCH(p,q). The log-likelihood of the model for the observed series $\{w_t\}$ with length m = `nobs` is

$$\log(L) = -\frac{m}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{m} y_t^2 / \sigma_t^2 - \frac{1}{2}\sum_{t=1}^{m}\log\sigma_t^2,$$

$$\textit{where } \sigma_t^2 = \sigma^2 + \sum_{i=1}^{p}\beta_i\sigma_{t-i}^2 + \sum_{i=1}^{q}\alpha_i w_{t-i}^2.$$

Thus log(L) is maximized subject to the constraints on the α_i, β_i, and σ.

In this model, if $q = 0$, the GARCH model is singular since the estimated Hessian matrix is singular.

The initial values of the parameter vector `x` entered in vector `xguess` must satisfy certain constraints. The first element of `xguess` refers to σ^2 and must be greater than zero and less than `max_sigma`. The remaining p+q initial values must each be greater than or equal to zero and sum to a value less than one.

To guarantee stationarity in model fitting,

$$\sum_{i=2}^{p+q+1} x(i) = \sum_{i=1}^{p}\beta_i + \sum_{i=1}^{q}\alpha_i < 1$$

is checked internally. The initial values should selected from values between zero and one.

`AIC` is computed by

$$-2\log(L) + 2(p+q+1),$$

where log(L) is the value of the log-likelihood function.

Statistical inferences can be performed outside the routine `GARCH` based on the output of the log-likelihood function `(A)`, the Akaike Information Criterion (AIC), and the variance-covariance matrix `(VAR)`.

Example

The data for this example are generated to follow a GARCH(p,q) process by using a random number generation function `sgarch`. The data set is analyzed and estimates of sigma, the ARCH parameters, and the GARCH parameters are returned. The values of the Log-likelihood function and the Akaike Information Criterion are returned from the optional arguments `IMSLS_A and IMSLS_AIC`.

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

static void  sgarch (int p, int q, int m, float x[],
                float y[], float z[], float y0[], float sigma[]);
#define      M      1000
#define      N      (P + Q + 1)
#define      P      2
#define      Q      1

void main ()
```

```
{
    int        n, p, q, m;
    float      a, aic, wk1[M + 1000], wk2[M + 1000],
                wk3[M + 1000], x[N], xguess[N],  y[M];
    float      *result;

    imsls_random_seed_set (182198625);
    m = M;
    p = P;
    q = Q;
    n = p+q+1;
    x[0] = 1.3;
    x[1] = .2;
    x[2] = .3;
    x[3] = .4;
    xguess[0] = 1.0;
    xguess[1] = .1;
    xguess[2] = .2;
    xguess[3] = .3;
    sgarch (p, q, m, x, y, wk1, wk2, wk3);
    result = imsls_f_garch(p, q, m, y, xguess,
                    IMSLS_A, &a,
                    IMSLS_AIC, &aic,
                    0);
    printf("Sigma estimate is\t%11.4f\n", result[0]);
    printf("ARCH(1) estimate is\t%11.4f\n", result[1]);
    printf("GARCH(1) estimate is\t%11.4f\n", result[2]);
    printf("GARCH(2) estimate is\t%11.4f\n", result[3]);
    printf("\nLog-likelihood function value is\t%11.4f\n", a);
    printf("Akaike Information Criterion value is\t%11.4f\n", aic);
    return;
}

static void sgarch (int p, int q, int m, float x[],
                float y[], float z[], float y0[], float sigma[])
{
    int        i, j, l;
    float      s1, s2, s3;

   imsls_f_random_normal ( m + 1000, IMSLS_RETURN_USER, z, 0);

    l = imsls_i_max (p, q);
    l = imsls_i_max (l, 1);
    for (i = 0; i < l; i++) y0[i] = z[i] * x[0];

    /* COMPUTE THE INITIAL VALUE OF SIGMA */
    s3 = 0.0;
    if (imsls_i_max (p, q) >= 1) {
       for (i = 1; i < (p + q + 1); i++) s3 += x[i];
    }
    for (i = 0; i < l; i++) sigma[i] = x[0] / (1.0 - s3);

    for (i = l; i < (m + 1000); i++) {
       s1 = 0.0;
       s2 = 0.0;
```

```
        if (q >= 1) {
            for (j = 0; j < q; j++)
              s1 += x[j + 1] * y0[i - j - 1] * y0[i - j - 1];
        }
        if (p >= 1) {
            for (j = 0; j < p; j++)
              s2 += x[q + 1 + j] * sigma[i - j - 1];
        }
        sigma[i] = x[0] + s1 + s2;
        y0[i] = z[i] * sqrt (sigma[i]);
    }
    /*
     * DISCARD THE FIRST 1000 SIMULATED OBSERVATIONS
     */
    for (i = 0; i < m; i++) y[i] = y0[1000 + i];
    return;
}                          /* end of function */
```

```
Output
Sigma estimate is   1.6480
ARCH(1) estimate is 0.2427
GARCH(1) estimate is       0.3175
GARCH(2) estimate is       0.3335

Log-likelihood function value is -2707.0903
Akaike Information Criterion value is 5422.1807
```

kalman

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

Synopsis

#include `<imsls.h>`

void `imsls_f_kalman` (*int* `nb`, *float* `nb[]`, *float* `covb[]`, *int* `*n`, *float* `*ss`, *float* `*alndet`, ..., 0)

The type *double* function is `imsls_d_kalman`.

Required Arguments

int `nb` (Input)
Number of elements in the state vector.

float `b[]` (Input/Output)
Array of length `nb` containing the estimated state vector. The input is the estimated state vector at time k given the observations through time $k - 1$. The output is the estimated state vector at time $k + 1$ given the observations through time k. On the first call to `imsls_f_kalman`, the input `b` must be the prior mean of the state vector at time 1.

float `covb[]` (Input/Output)
Array of size `nb` by `nb` such that `covb*` σ^2 is the mean squared error matrix for `b`.
Before the first call to `imsls_f_kalman`, `covb *` σ^2 must equal the variance-covariance matrix of the state vector.

int `*n` (Input/Output)
Pointer to the rank of the variance-covariance matrix for all the observations. `n` must be initialized to zero before the first call to `imsls_f_kalman`. In the usual case when the variance-covariance matrix is nonsingular, `n` equals the sum of the `ny`'s from the invocations to `imsls_f_kalman`. See optional argument `IMSLS_UPDATE` below for the definition of `ny`.

float `*ss` (Input/Output)
Pointer to the generalized sum of squares.
`ss` must be initialized to zero before the first call to `imsls_f_kalman`. The estimate of σ^2 is given by $\frac{\mathtt{ss}}{\mathtt{n}}$.

float `*alndet` (Input/Output)
Pointer to the natural log of the product of the nonzero eigenvalues of *P* where *P* `*` σ^2 is the variance-covariance matrix of the observations. Although `alndet` is computed, `imsls_f_kalman` avoids the explicit computation of *P*. `alndet` must be initialized to zero before the first call to `imsls_f_kalman`. In the usual case when *P* is nonsingular, `alndet` is the natural log of the determinant of *P*.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
voidt *imsls_f_random_sample (int nb,float nb[], float covb[],
         int *n, float *ss, float *alndet,
        IMSLS_UPDATE, int ny, float *y, float *z, float *r,
        IMSLS_Z_COL_DIM, int z_col_dim,
        IMSLS_R_COL_DIM, int r_col_dim,
        IMSLS_T, float *t,
        IMSLS_T_COL_DIM, int t_col_dim,
        IMSLS_Q, float *q,
        IMSLS_Q_COL_DIM, int t_col_dim,
        IMSLS_TOLERANCE, float tolerance,
        IMSLS_V, float **v,
        IMSLS_V_USER, float v[],
        IMSLS_COVV, float **v,
        IMSLS_COVV_USER, float v[],
         0)
```

Optional Arguments

`IMSLS_UPDATE`, *int* `ny`, *float* `*y`, *float* `*z`, *float* `*r` (Input)
Perform computation of the *update equations.*
`ny`: Number of observations for current update.

y: Array of length ny containing the observations.

z: ny by nb array containing the matrix relating the observations to the state vector in the observation equation.

r: ny by ny array containing the matrix such that r * σ^2 is the variance-covariance matrix of errors in the observation equation.
σ^2 is a positive unknown scalar. Only elements in the upper triangle of r are referenced.

IMSLS_Z_COL_DIM, *int* z_col_dim (Input)
Column dimension of the matrix z.
Default: z_col_dim = nb

IMSLS_R_COL_DIM, *int* r_col_dim (Input)
Column dimension of the matrix r.
Default: r_col_dim = ny

IMSLS_T, *float* *t (Input)
nb by nb transition matrix in the state equation
Default: t = identity matrix

IMSLS_T_COL_DIM, *int* r_col_dim (Input)
Column dimension of the matrix t.
Default: t_col_dim = nb

IMSLS_Q, *float* *q (Input)
nb by nb matrix such that q * σ^2 is the variance-covariance matrix of the error vector in the state equation.
Default: There is no error term in the state equation.

IMSLS_Q_COL_DIM, *int* q_col_dim (Input)
Column dimension of the matrix q.
Default: q_col_dim = nb

IMSLS_TOLERANCE, *float* tolerance (Input)
Tolerance used in determining linear dependence.
Default: tolerance = 100.0*imsls_f_machine(4)

IMSLS_V, *float* **v (Output)
Address to a pointer v to an array of length ny containing the one-step-ahead prediction error.

IMSLS_V_USER, *float* v[] (Output)
Storage for v is provided by the user. See IMSLS_V.

IMSLS_COVV, *float* **covv (Output)
The address to a pointer of size ny by ny containing a matrix such that covv * σ^2 is the variance-covariance matrix of v.

IMSLS_COVV_USER, *float* covv[] (Output)
Storage for covv is provided by the user. See IMSLS_COVV.

Description

Routine imsls_f_kalman is based on a recursive algorithm given by Kalman (1960), which has come to be known as the Kalman filter. The underlying model is known as the state-space model. The model is specified stage by stage where the stages generally correspond to time points at which the observations become available. The routine imsls_f_kalman avoids many of the computations and storage requirements that would be necessary if one were to process all the data at the end of each stage in order to estimate the state vector. This is accomplished by using previous computations and retaining in storage only those items essential for processing of future observations.

The notation used here follows that of Sallas and Harville (1981). Let y_k (input in y using optional argument IMSLS_UPDATE) be the $n_k \times 1$ vector of observations that become available at time k. The subscript k is used here rather than t, which is more customary in time series, to emphasize that the model is expressed in stages $k = 1, 2, \ldots$ and that these stages need not correspond to equally spaced time points. In fact, they need not correspond to time points of any kind. The *observation equation* for the state-space model is

$$y_k = Z_k b_k + e_k \quad k = 1, 2, \ldots$$

Here, Z_k (input in z using optional argument IMSLS_UPDATE) is an $n_k \times q$ known matrix and b_k is the $q \times 1$ state vector. The state vector b_k is allowed to change with time in accordance with the *state equation*

$$b_{k+1} = T_{k+1} b_k + w_{k+1} \qquad k = 1, 2, \ldots$$

starting with $b_1 = \mu_1 + w_1$.

The change in the state vector from time k to $k + 1$ is explained in part by the *transition matrix* T_{k+1} (the identity matrix by default, or optionally input using IMSLS_T), which is assumed known. It is assumed that the q-dimensional w_ks
($k = 1, 2, \ldots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 Q_k$, that the n_k-dimensional e_ks ($k = 1, 2, \ldots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 R_k$, and that the w_ks and e_ks are independent of each other. Here, μ_1 is the mean of b_1 and is assumed known, σ^2 is an unknown positive scalar. Q_{k+1}(input in Q) and R_k (input in R) are assumed known.

Denote the estimator of the realization of the state vector b_k given the observations y_1, $y_2, \ldots, y_j$ by

$$\hat{\beta}_{k|j}$$

By definition, the mean squared error matrix for

$$\hat{\beta}_{k|j}$$

is

$$\sigma^2 C_{k|j} = E(\hat{\beta}_{k|j} - b_k)(\hat{\beta}_{k|j} - b_k)^T$$

At the time of the *k*-th invocation, we have

$$\hat{\beta}_{k|k-1}$$

and

$C_{k|k-1}$, which were computed from the (k–1)-st invocation, input in `b` and `covb`, respectively. During the *k*-th invocation, function `imsls_f_kalman` computes the filtered estimate

$$\hat{\beta}_{k|k}$$

along with $C_{k|k}$. These quantities are given by the *update equations*:

$$\hat{\beta}_{k|k} = \hat{\beta}_{k|k-1} + C_{k|k-1} Z_k^T H_k^{-1} v_k$$
$$C_{k|k} = C_{k|k-1} - C_{k|k-1} Z_k^T H_k^{-1} Z_k C_{k|k-1}$$

where

$$v_k = y_k - Z_k \hat{\beta}_{k|k-1}$$

and where

$$H_k = R_k + Z_k C_{k|k-1} Z_k^T$$

Here, v_k (stored in `v`) is the one-step-ahead prediction error, and $\sigma^2 H_k$ is the variance-covariance matrix for v_k. H_k is stored in `covv`. The "start-up values" needed on the first invocation of `imsls_f_kalman` are

$$\hat{\beta}_{1|0} = \mu_1$$

and $C_{1|0} = Q_1$ input via `b` and `covb`, respectively. Computations for the *k*-th invocation are completed by `imsls_f_kalman` computing the one-step-ahead estimate

$\hat{\beta}_{k+1|k}$

along with $C_{k+1|k}$ given by the *prediction equations*:

$$\hat{\beta}_{k+1|k} = T_{k+1}\hat{\beta}_{k|k}$$
$$C_{k+1|k} = T_{k+1}C_{k|k}T_{k+1}^T + Q_{k+1}$$

If both the filtered estimates and one-step-ahead estimates are needed by the user at each time point, imsls_f_kalman can be invoked twice for each time point—first without `IMSLS_T` and `IMSLS_Q` to produce

$$\hat{\beta}_{k|k}$$

and $C_{k|k}$, and second without `IMSLS_UPDATE` to produce

$$\hat{\beta}_{k+1|k}$$

and $C_{k+1|k}$ (Without `IMSLS_T` and `IMSLS_Q`, the prediction equations are skipped. Without `IMSLS_UPDATE`, the update equations are skipped.).

Often, one desires the estimate of the state vector more than one-step-ahead, i.e., an estimate of

$$\hat{\beta}_{k|j}$$

is needed where $k > j + 1$. At time j, `imsls_f_kalman` is invoked with `IMSLS_UPDATE` to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of `imsls_f_kalman` without `IMSLS_UPDATE` can compute

$$\hat{\beta}_{j+2|j}, \hat{\beta}_{j+3|j}, \ldots, \hat{\beta}_{k|j}$$

Computations for

$$\hat{\beta}_{k|j}$$

and $C_{k|j}$ assume the variance-covariance matrices of the errors in the observation equation and state equation are known up to an unknown positive scalar multiplier, σ^2. The maximum likelihood estimate of σ^2 based on the observations $y_1, y_2, \ldots, y_m$, is given by

$$\hat{\sigma}^2 = SS / N$$

where

$$N = \sum_{k=1}^{m} n_k \text{ and } SS = \sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

N and SS are the input/output arguments `n` and `ss`.

If σ^2 is known, the R_ks and Q_ks can be input as the variance-covariance matrices exactly. The earlier discussion is then simplified by letting $\sigma^2 = 1$.

In practice, the matrices T_k, Q_k, and R_k are generally not completely known. They may be known functions of an unknown parameter vector θ. In this case, `imsls_f_kalman` can be used in conjunction with an optimization program (see routine `imsl_f_min_uncon_multivar`, IMSL C/Math/Library, Chapter 8, "Optimization") to obtain a maximum likelihood estimate of θ. The natural logarithm of the likelihood function for $y_1, y_2, \ldots, y_m$ differs by no more than an additive constant from

$$L(\theta,\sigma^2; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln \sigma^2 - \frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)] - \frac{1}{2}\sigma^{-2}\sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

(Harvey 1981, page 14, equation 2.21).

Here,

$$\sum_{k=1}^{m} \ln[\det(H_k)]$$

(stored in `alndet`) is the natural logarithm of the determinant of V where $\sigma^2 V$ is the variance-covariance matrix of the observations.

Minimization of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ over all θ and σ^2 produces maximum likelihood estimates. Equivalently, minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ where

$$L_c(\theta; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln\left(\frac{SS}{N}\right) - \frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)]$$

produces maximum likelihood estimates

$$\hat{\theta} \text{ and } \hat{\sigma}^2 = SS/N$$

The minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ instead of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$, reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

$$\hat{\sigma}^2(\theta) = SS(\theta)/N$$

minimizes $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ for all θ, consequently,

$$\hat{\sigma}^2(\theta)$$

can be substituted for σ^2 in $L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ to give a function that differs by no more than an additive constant from $L_c(\theta; y_1, y_2, \ldots, y_m)$.

The earlier discussion assumed H_k to be nonsingular. If H_k is singular, a modification for singular distributions described by Rao (1973, pages 527–528) is used. The necessary changes in the preceding discussion are as follows:

1. Replace

$$H_k^{-1}$$

   by a generalized inverse.

2. Replace $\det(H_k)$ by the product of the nonzero eigenvalues of H_k.
3. Replace N by

$$\sum_{k=1}^{m} \operatorname{rank}\left(H_k\right)$$

Maximum likelihood estimation of parameters in the Kalman filter is discussed by Sallas and Harville (1988) and Harvey (1981, pages 111–113).

Example 1

Function `imsls_f_kalman` is used to compute the filtered estimates and one-step-ahead estimates for a scalar problem discussed by Harvey (1981, pages 116–117). The observation equation and state equation are given by

$$\begin{aligned} y_k &= b_k + e_k \\ b_{k+1} &= b_k + w_{k+1} \qquad k = 1, 2, 3, 4 \end{aligned}$$

where the e_ks are identically and independently distributed normal with mean 0 and variance σ^2, the w_ks are identically and independently distributed normal with mean 0 and variance $4\sigma^2$, and b_1 is distributed normal with mean 4 and variance $16\sigma^2$. Two invocations of `imsls_f_kalman` are needed for each time point in order to compute the filtered estimate and the one-step-ahead estimate. The first invocation does not use the optional arguments `IMSLS_T` and `IMSLS_Q` so that the prediction equations are skipped in the computations. The update equations are skipped in the computations in the second invocation.

This example also computes the one-step-ahead prediction errors. Harvey (1981, page 117) contains a misprint for the value v_4 that he gives as 1.197. The correct value of $v_4 = 1.003$ is computed by `imsls_f_kalman`.

.

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

```
#define NB 1
#define NOBS 4
#define NY 1

void main()
{
    int         nb = NB, nobs = NOBS, ny = NY;
    int         ldcovb, ldcovv, ldq, ldr, ldt, ldz;
    int         i, iq, it, n, nout;
    float       alndet, b[NB], covb[NB][NB], covv[NY][NY],
                q[NB][NB], r[NY][NY], ss,
                t[NB][NB], tol, v[NY], y[NY], z[NY][NB];
    float       ydata[] = {4.4, 4.0, 3.5, 4.6};

    z[0][0] = 1.0;
    r[0][0] = 1.0;
    q[0][0] = 4.0;
    t[0][0] = 1.0;
    b[0] = 4.0;
    covb[0][0] = 16.0;

    /* Initialize arguments for initial call to imsls_f_kalman. */
    n = 0;
    ss = 0.0;
    alndet = 0.0;
    printf("k/j      b       covb n     ss      alndet     v       covv\n");

    for (i = 0; i < nobs; i++) {
      /* Update */
      y[0] = ydata[i];
      imsls_f_kalman(nb, b, (float*)covb, &n, &ss, &alndet,
                  IMSLS_UPDATE, ny, y, z, r,
                  IMSLS_V_USER, v,
                  IMSLS_COVV_USER, covv,
                  0);

      printf("%d/%d %8.3f %8.3f %d %8.3f %8.3f %8.3f %8.3f\n",
            i, i, b[0], covb[0][0], n, ss, alndet, v[0], covv[0][0]);

      /* Prediction */
      imsls_f_kalman(nb, b, (float*)covb, &n, &ss, &alndet,
                  IMSLS_T, t,
                  IMSLS_Q, q,
                  0);

      printf("%d/%d %8.3f %8.3f %d %8.3f %8.3f %8.3f %8.3f\n",
            i+1, i, b[0], covb[0][0], n, ss, alndet, v[0], covv[0][0]);
    }

}
```

Output

```
k/j      b       covb n     ss      alndet     v       covv
0/0    4.376    0.941 1    0.009    2.833    0.400   17.000
```

```
1/0    4.376    4.941 1    0.009    2.833    0.400   17.000
1/1    4.063    0.832 2    0.033    4.615   -0.376    5.941
2/1    4.063    4.832 2    0.033    4.615   -0.376    5.941
2/2    3.597    0.829 3    0.088    6.378   -0.563    5.832
3/2    3.597    4.829 3    0.088    6.378   -0.563    5.832
3/3    4.428    0.828 4    0.260    8.141    1.003    5.829
4/3    4.428    4.828 4    0.260    8.141    1.003    5.829
```

Example 2

Function imsls_f_kalman is used with routine imsl_f_min_uncon_multivar, (see IMSL C/Math/Library, Chapter 8, “Optimization”) to find a maximum likelihood estimate of the parameter θ in a MA(1) time series represented by $y_k = \varepsilon_k - \theta\varepsilon_{k-1}$. Function imsls_f_random_arma (see IMSL C/Stat/Library, Chapter 12, “Random Number Generation”) is used to generate 200 random observations from an MA(1) time series with $\theta = 0.5$ and $\sigma^2 = 1$.

The MA(1) time series is cast as a state-space model of the following form (see Harvey 1981, pages 103–104, 112):

$$y_k = \begin{pmatrix} 1 & 0 \end{pmatrix} b_k$$

$$b_k = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} b_{k-1} + w_k$$

where the two-dimensional w_ks are independently distributed bivariate normal with mean 0 and variance $\sigma^2 Q_k$ and

$$Q_1 = \begin{pmatrix} 1+\theta^2 & -\theta \\ -\theta & \theta^2 \end{pmatrix}$$

$$Q_k = \begin{pmatrix} 1 & -\theta \\ -\theta & \theta^2 \end{pmatrix} \qquad k = 2, 3, \ldots, 200$$

The warning error that is printed as part of the output is not serious and indicates that imsl_f_min_uncon_multivar (See Chapter 8, “Optimization” in the math manual) is generally used for multi-parameter minimization.

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

#define NOBS 200
#define NTHETA 1
#define NB 2
#define NY 1

float fcn(int ntheta, float theta[]);
float *ydata;
```

```
void main ()
{
    int  lagma[1];
    float pma[1];
    float *theta;

    imsls_random_seed_set(123457);
    pma[0] = 0.5;
    lagma[0] = 1;
    ydata = imsls_f_random_arma(200, 0, NULL, 1, pma,
                            IMSLS_ACCEPT_REJECT_METHOD,
                            IMSLS_NONZERO_MALAGS, lagma,
                            0);

    theta = imsl_f_min_uncon_multivar(fcn, NTHETA, 0);

    printf("* * * Final Estimate for THETA * * *\n");
    printf("Maximum likelihood estimate, THETA = %f\n", theta[0]);

}

float fcn(int ntheta, float theta[])
{
  int i, n;
  float res, ss, alndet;
  float t[] = {0.0, 1.0, 0.0, 0.0};
  float z[] = {1.0, 0.0};
  float q[NB][NB], r[NY][NY], b[NB], covb[NB][NB], y[NY];
  if (fabs(theta[0]) > 1.0) {
    res = 1.0e10;
  } else {
    q[0][0] = 1.0;
    q[0][1] = -theta[0];
    q[1][0] = -theta[0];
    q[1][1] = theta[0]*theta[0];

    r[0][0] = 0.0;

    b[0] = 0.0;
    b[1] = 0.0;

    covb[0][0] = 1.0 + theta[0]*theta[0];
    covb[0][1] = -theta[0];
    covb[1][0] = -theta[0];
    covb[1][1] = theta[0]*theta[0];

    n = 0;
    ss = 0.0;
    alndet = 0.0;

    for (i = 0; i<NOBS; i++) {
      y[0] = ydata[i];
      imsls_f_kalman(NB, b, (float*)covb, &n, &ss, &alndet,
                  IMSLS_UPDATE, NY, y, z, r,
                  IMSLS_Q, q,
                  IMSLS_T, t,
                  0);
    }
    res = n*log(ss/n) + alndet;
  }
  return(res);
}
```

Output

```
*** WARNING_IMMEDIATE Error from imsl_f_min_uncon_multivar.  This routine
***          may be inefficient for a problem of size "n" = 1.


*** WARNING_IMMEDIATE Error from imsl_f_min_uncon_multivar.  The last global
***          step failed to locate a lower point than the current X value.
***          The current X may be an approximate local minimizer and no more
***          accuracy is possible or the step tolerance may be too large
***          where "step_tol" = 2.422181e-05 is given.

* * * Final Estimate for THETA * * *
Maximum likelihood estimate, THETA = 0.453256
```

Chapter 9: Multivariate Analysis

Routines

Hierarchical Cluster Analysis

Computes matrix of dissimilarities or similarities	dissimilarities	641
Hierarchical cluster analysis	cluster_hierarchical	645
Retrieves cluster numbers in hierarchical cluster analysis	cluster_number	649

K-means Cluster Analysis

Performs a *K*-means (centroid) cluster analysis	cluster_k_means	653

Principal Component Analysis

Computes principal components	principal_components	657

Factor Analysis

Extracts factor-loading estimates	factor_analysis	663
Performs discriminant function analysis	discriminant_analysis	682

Usage Notes

Cluster Analysis

Function `imsls_f_cluster_k_means` performs a *K*-means cluster analysis. Basic *K*-means clustering attempts to find a clustering that minimizes the within-cluster sums-of-squares. In this method of clustering the data, matrix *X* is grouped so that each observation (row in *X*) is assigned to one of a fixed number, *K*, of clusters. The sum of the squared difference of each observation about its assigned cluster's mean is used as the criterion for assignment. In the basic algorithm, observations are transferred from one cluster or another when doing so decreases the within-cluster sums-of-squared differences. When no transfer occurs in a pass through the entire data set, the algorithm stops. Function `imsls_f_cluster_k_means` is one implementation of the basic algorithm.

The usual course of events in *K*-means cluster analysis is to use `imsls_f_cluster_k_means` to obtain the optimal clustering. The clustering is then evaluated by functions described in Chapter 1, "Basic Statistics," and/or other chapters in this manual. Often, *K*-means clustering with more than one value of *K* is performed, and the value of *K* that best fits the data is used.

Clustering can be performed either on observations or variables. The discussion of the function `imsls_f_cluster_k_means` assumes the clustering is to be performed on the observations, which correspond to the rows of the input
data matrix. If variables, rather than observations, are to be clustered, the
data matrix should first be transposed. In the documentation for
`imsls_f_cluster_k_means`, the words "observation" and "variable" are interchangeable.

Principal Components

The idea in principal components is to find a small number of linear combinations of the original variables that maximize the variance accounted for in the original data. This amounts to an eigensystem analysis of the covariance (or correlation) matrix. In addition to the eigensystem analysis, `imsls_f_principal_components` computes standard errors for the eigenvalues. Correlations of the original variables with the principal component scores also are computed.

Factor Analysis

Factor analysis and principal component analysis, while quite different in assumptions, often serve the same ends. Unlike principal components in which linear combinations yielding the highest possible variances are obtained, factor analysis generally obtains linear combinations of the observed variables according to a model relating the observed variable to hypothesized underlying factors, plus a random error term called the unique error or uniqueness. In factor analysis, the unique errors associated with each variable are usually assumed to be independent of the factors. Additionally, in the common factor model, the unique errors are assumed to be mutually independent. The factor analysis model is expressed in the following equation:

$$x - \mu = \Lambda f + e$$

where x is the p vector of observed values, μ is the p vector of variable means, Λ is the $p \times k$ matrix of factor loadings, f is the k vector of hypothesized underlying random factors, e is the p vector of hypothesized unique random errors, p is the number of variables in the observed variables, and k is the number of factors.

Because much of the computation in factor analysis was originally done by hand or was expensive on early computers, quick (but dirty) algorithms that made the calculations possible were developed. One result is the many factor extraction methods available today. Generally speaking, in the exploratory or model building phase of a factor analysis, a method of factor extraction that is not computationally intensive (such as principal components, principal factor, or image analysis) is used. If desired, a computationally intensive method is then used to obtain the final factors.

In exploratory factor analysis, the unrotated factor loadings obtained from the factor extraction are generally transformed (rotated) to simplify the interpretation of the factors. Rotation is possible because of the overparameterization in the factor analysis model. The method used for rotation may result in factors that are independent (orthogonal rotations) or correlated (oblique rotations). Prior information may be available (or hypothesized) in which case a Procrustes rotation could be used. When no prior information is available, an analytic rotation can be performed.

The steps generally used in a factor analysis are summarized as follows:

Steps in a Factor Analysis

Step 1

Calculate Covariance (Correlation) Matrix
IMSL routine `imsls_f_covariances`
(see Chapter 3, "Correlation and Covariance")

Step 2

Initial Factor Extraction
`imsls_f_factor_analysis`

Step 3

Factor Rotation using `imsls_f_factor_analysis`' optional arguments	
Orthogonal	**Oblique**

No Prior Info.
`IMSLS_ORTHOMAX_ROTATION`,
	No Prior Info.
	`IMSLS_OBLIQUE_PROMAX_ROTATION`
	`IMSLS_DIRECT_OBLIMIN_ROTATION`
	`IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`

Prior Info. `IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION`	Prior Info. `IMSLS_OBLIQUE_PROCRUSTES_ROTATION`

Step 4

Factor Structure and Variance
`imsls_f_factor_analysis`
optional argument
`IMSLS_FACTOR_STRUCTURE`

dissimilarities

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_dissimilarities` (*int* `nrow`, *int* `ncol`, *float* `*x`, …, 0)

The type *double* function is `imsls_d_dissimilarities`.

Required Arguments

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

int `ncol` (Input)
Number of columns in the matrix.

float `*x` (Input)
Array of size `nrow` by `ncol` containing the matrix.

Return Value

An array of size *m* by *m* containing the computed dissimilarities or similarities, where *m* = `nrow` if optional argument `IMSLS_ROWS` is used, and *m* = `ncol` otherwise.

Synopsis with Optional Arugments

#include `<imsls.h>`

float `*imsls_f_dissimilarities` (*int* `nrow`, *int* `ncol`, *float* `*x`,
`IMSLS_ROWS`, or `IMSLS_COLUMNS`,
`IMSLS_INDEX`, *int* `ndstm`, *int* `ind[]`,
`IMSLS_METHOD`, *int* `imeth`,
`IMSLS_SCALE`, *int* `iscale`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_RETURN_USER`, *float* `dist[]`,
0)

Optional Arguments

`IMSLS_ROWS`,
or

`IMSLS_COLUMNS`, (Input)
Exactly one of these options can be present to indicate whether distances are computed between rows or columns of `x`.
Default: Distances are computed between rows.

`IMSLS_INDEX`, *int* `ndstm`, *int* `ind[]`, (Input)
Argument `ind` is an array of length `ndstm` containing the indices of the rows (columns if `IMSLS_ROWS` is used) to be used in computing the distance measure.
Default: All rows(columns) are used.

`IMSLS_METHOD`, *int* `imeth` (Input)
Method to be used in computing the dissimilarities or similarities.
Default: `imeth` = 0.

imeth	**Method**
0	Euclidean distance (L_2 norm)
1	Sum of the absolute differences (L_1 norm)
2	Maximum difference (L_∞ norm)
3	Mahalanobis distance
4	Absolute value of the cosine of the angle between the vectors

imeth	**Method**
5	Angle in radians (0, π) between the lines through the origin defined by the vectors
6	Correlation coefficient
7	Absolute value of the correlation coefficient
8	Number of exact matches

See the Description section for a more detailed description of each measure.

`IMSLS_SCALE`, *int* iscale (Input)
: Scaling option.   (Input)
`iscale` is not used for methods 3 through 8.
Default: `iscale` = 0.

iscale	**Scaling Performed**
0	No scaling is performed.
1	Scale each column (row, if `IMSLS_ROWS` is used) by the standard deviation of the column (row).
2	Scale each column (row, if `IMSLS_ROWS` is used) by the range of the column (row).

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

`IMSLS_RETURN_USER`, *float* `dist[]` (Output)
: User allocated array of size *m* by *m* containing the computed dissimilarities or similarities, where *m* = `nrow` if `IMSLS_ROWS` is used, and *m* = `ncol` otherwise.

Description

Function `imsls_f_dissimilarities` computes an upper triangular matrix (excluding the diagonal) of dissimilarities (or similarities) between the columns or rows of a matrix. Nine different distance measures can be computed. For the first three measures, three different scaling options can be employed. Output from `imsls_f_dissimilarities` is generally used as input to clustering or multidimensional scaling functions.

The following discussion assumes that the distance measure is being computed between the columns of the matrix, i.e., that `IMSLS_COLUMNS` is used. If distances between the rows of the matrix are desired, use optional argument `IMSLS_ROWS`.

For `imeth` = 0 to 2, each row of `x` is first scaled according to the value of `iscale`. The scaling parameters are obtained from the values in the row scaled as either the standard deviation of the row or the row range; the standard deviation is computed from the unbiased estimate of the variance. If `iscale` is 0, no scaling is performed, and the parameters in the following discussion are all 1.0. Once the scaling value (if any) has been computed, the distance between column i and column j is computed via the difference vector $z_k = (x_k - y_k)/s_k$, $i = 1, \ldots,$ `ndstm`, where x_k denotes the k-th element in the i-th column, and y_k denotes the corresponding element in the j-th column. For given z_i, the metrics 0 to 2 are defined as:

`imeth`		**Metric**
0	$\sqrt{\left(\sum_{i=1}^{\text{ndstm}} z_i^2\right)}$	Euclidean distance
1	$\sum_{i=1}^{\text{ndstm}} \lvert z_i \rvert$	L_1 norm
2	$\max_i \lvert z_i \rvert$	L_∞ norm

Distance measures corresponding to `imeth` = 3 to 8 do not allow for scaling. These measures are defined via the column vectors $X = (x_i)$, $Y = (y_i)$, and $Z = (x_i - y_i)$ as follows:

`imeth`	**Scaling Performed**
3	$Z'\hat{\Sigma}^{-1}Z =$ Mahalanobis distance, where $\hat{\Sigma}$ is the usual unbiased sample estimate of the covariance matrix of the rows.
4	$\cos(\theta) = X^T Y / \left(\sqrt{X^T X}\sqrt{Y^T Y}\right) =$ the dot product of X and Y divided by the length of X times the length of Y.
5	θ, where θ is defined in 4.
6	ρ = the usual (centered) estimate of the correlation between X and Y.
7	The absolute value of ρ (where ρ is defined in 6).
8	The number of times $x_i = y_i$, where x_i and y_i are elements of X and Y.

For the Mahalanobis distance, any variable used in computing the distance measure that is (numerically) linearly dependent upon the previous variables in the `ind` vector is omitted from the distance measure.

Example

The following example illustrates the use of `imsls_f_dissimilarities` for computing the Euclidean distance between the rows of a matrix.

```
#include "imsls.h"

void main()
{
  int ncol=2, nrow = 4;
  float x [4][2] = {1., 1.,
                  1., 0.,
                  1.,-1.,
                  1., 2.};
  float *dist;

  dist = imsls_f_dissimilarities(nrow, ncol, (float*)x, 0);
  imsls_f_write_matrix("dist", 4, 4, dist, 0);
}
```

Output

```
                     dist
            1           2           3           4
1           0           1           2           1
2           0           0           1           2
3           0           0           0           3
4           0           0           0           0
```

cluster_hierarchical

Performs a hierarchical cluster analysis given a distance matrix.

Synopsis

#include `<imsls.h>`

void `imsls_f_cluster_hierarchical` (*int* `npt`, *float* `*dist`, …, 0)

The type *double* function is `imsls_d_cluster_hierarchical`.

Required Arguments

int `npt` (Input)
Number of data points to be clustered.

float `*dist` (Input/Ouput)
An `npt` by `npt` symmetric matrix containing the distance (or similarity) matrix.
`dist` is a symmetric matrix. On input, only the upper triangular part needs to be present. The function `imsls_f_cluster_hierarchical` saves the upper triangular part of `dist` in the lower triangle. On return from `imsls_f_cluster_hierarchical`, the upper triangular part of `dist` is restored, and the matrix is made symmetric.

Synopsis with Optional Arugments

#include `<imsls.h>`

void `*imsls_f_cluster_hierarchical` (*int* `npt`, *float* `*dist`,
`IMSLS_METHOD`, *int* `imeth`,
`IMSLS_TRANSFORMATION`, *int* `itrans`,
`IMSLS_CLUSTERS`, *float* `**clevel`, *int* `**iclson`, *int* `**icrson`,
`IMSLS_CLUSTERS_USER`, *float* `clevel[]`, *int* `iclson[]`, *int* `icrson[]`,
0)

Optional Arguments

`IMSLS_METHOD`, *int* `imeth` (Input)

Option giving the clustering method to be used.
Default: `imeth` = 0.

imeth	**Method**
0	Single linkage (minimum distance)
1	Complete linkage (maximum distance)
2	Average distance within (average distance between objects within the merged cluster)
3	Average distance between (average distance between objects in the two clusters)
4	Ward's method (minimize the within-cluster sums of squares). For Ward's method, the elements of `dist` are assumed to be Euclidean distances.

`IMSLS_TRANSFORMATION`, *int* itrans (Input)

Option giving the method to be used for clustering.
Default: `itrans` = 0.

Imeth	**Method**
0	No transformation is required. The elements of `dist` are distances.
1	Convert similarities to distances by multiplication by −1.0.
2	Convert similarities (usually correlations) to distances by taking the reciprocal of the absolute value.

`IMSLS_CLUSTERS`, *float* `**clevel`, *int* `**iclson`, *int* `**icrson` (Output)

Argument `clevel` is the address of an array of length `npt` − 1 containing the level at which the clusters are joined. `clevel`[*k*-1] contains the distance (or similarity) level at which cluster `npt` + *k* was formed. If the original data in `dist` was transformed via the optional argument `IMSLS_TRANSFORMATION`, the inverse transformation is applied to the values in `clevel` prior to exit from `imsls_f_cluster_hierarchical`. Argument `iclson` is the address of an array of length `npt` − 1 containing the left sons of each merged cluster. Argument `icrson` is the address of an array of length `npt` − 1 containing the right sons of each merged cluster. Cluster
`npt` + *k* is formed by merging clusters `iclson`[*k*-1] and `icrson`[*k*-1].

`IMSLS_CLUSTERS_USER`, *float* `clevel[]`, *int* `iclson[]`, *int* `icrson[]` (Output)
Storage for arrays `clevel`, `iclson`, and `icrson` is provided by the user. See `IMSLS_CLUSTERS`.

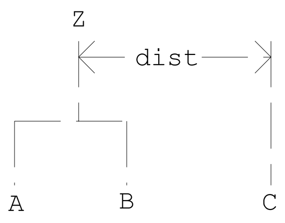
Description

Function `imsls_f_cluster_hierarchical` conducts a hierarchical cluster analysis based upon the distance matrix, or by appropriate use of the `IMSLS_TRANSFORMATION` optional argument, based upon a similarity matrix. Only the upper triangular part of the matrix `dist` is required as input to `imsls_f_cluster_hierarchical`.

Hierarchical clustering in `imsls_f_cluster_hierarchical` proceeds as follows. Initially, each data point is considered to be a cluster, numbered 1 to n = `npt`.

1. If the data matrix contains similarities, they are converted to distances by the method specified by `IMSLS_TRANSFORMATION`. Set $k = 1$.
2. A search is made of the distance matrix to find the two closest clusters. These clusters are merged to form a new cluster, numbered $n + k$. The cluster numbers of the two clusters joined at this stage are saved in `icrson` and `iclson`, and the distance measure between the two clusters is stored in `clevel`.
3. Based upon the method of clustering, updating of the distance measure in the row and column of `dist` corresponding to the new cluster is performed.
4. Set $k = k + 1$. If $k < n$, go to Step 2.

The five methods differ primarily in how the distance matrix is updated after two clusters have been joined. The `IMSLS_METHOD` optional argument specifies how the distance of the cluster just merged with each of the remaining clusters will be updated. Function `imsls_f_cluster_hierarchical` allows five methods for computing the distances. To understand these measures, suppose in the following discussion that clusters "*A*" and "*B*" have just been joined to form cluster "*Z*", and interest is in computing the distance of *Z* with another cluster called "*C*".



Imeth	**Method**
0	Single linkage method. The distance from *Z* to *C* is the minimum of the distances (*A* to *C*, *B* to *C*).
1	Complete linkage method. The distance from *Z* to *C* is the maximum of the distances (*A* to *C*, *B* to *C*).
2	Average-distance-within-clusters method. The distance from *Z* to

Imeth	Method
	C is the average distance of all objects that would be within the cluster formed by merging clusters *Z* and *C*. This average may be computed according to formulas given by Anderberg (1973, page 139).
3	Average-distance-between-clusters method. The distance from *Z* to *C* is the average distance of objects within cluster *Z* to objects within cluster *C*. This average may be computed according to methods given by Anderberg (1973, page 140).
4	Ward's method. Clusters are formed so as to minimize the increase in the within-cluster sums of squares. The distance between two clusters is the increase in these sums of squares if the two clusters were merged. A method for computing this distance from a squared Euclidean distance matrix is given by Anderberg (1973, pages 142–145).

In general, single linkage will yield long thin clusters while complete linkage will yield clusters that are more spherical. Average linkage and Ward's linkage tend to yield clusters that are similar to those obtained with complete linkage.

Function `imsls_f_cluster_hierarchical` produces a unique representation of the binary cluster tree via the following three conventions; the fact that the tree is unique should aid in interpreting the clusters. First, when two clusters are joined and each cluster contains two or more data points, the cluster that was initially formed with the smallest level (in `clevel`) becomes the left son. Second, when a cluster containing more than one data point is joined with a cluster containing a single data point, the cluster with the single data point becomes the right son. Finally, when two clusters containing only one object are joined, the cluster with the smallest cluster number becomes the right son.

Comments

1. The clusters corresponding to the original data points are numbered from 1 to `npt`. The `npt` – 1 clusters formed by merging clusters are numbered `npt` + 1 to `npt` + (`npt` – 1).
2. Raw correlations, if used as similarities, should be made positive and transformed to a distance measure. One such transformation can be performed by specifying optional argument `IMSLS_TRANSFORMATION`, with `itrans` = 2 in `imsls_f_cluster_hierarchical`.
3. The user may cluster either variables or observations in `imsls_f_cluster_hierarchical` since a dissimilarity matrix, not the original data, is used. Function `imsls_f_dissimilarities` may be used to compute the matrix `dist` for either the variables or observations.

Example

In the following example, the average distance within clusters method is used to perform a hierarchical cluster analysis of the Fisher iris data. Function imsls_f_data_sets (see Chapter 15, "Utilities") is first used to obtain the Fisher iris data. The example is typical in that after the program obtains the data, function imsls_f_dissimilarities computes the distance matrix (dist) prior to calling imsls_f_cluster_hierarchical.

```
#include "imsls.h"

void main()
{
  int  iscale=1, ncol=5, nrow=150, nvar=4, npt = 150;
  int i, iclson[149], icrson[149], ind[4] = {1, 2, 3, 4};
  float clevel[149], *dist, *x;

  x = imsls_f_data_sets(3, 0);

  dist = imsls_f_dissimilarities(nrow, ncol, x,
                           IMSLS_INDEX, nvar, ind,
                           IMSLS_SCALE, iscale,
                           0);
  imsls_f_cluster_hierarchical(npt, dist,
              IMSLS_CLUSTERS_USER, clevel, iclson, icrson,
              IMSLS_METHOD, 2,
              0);

  for (i=0;i<149;i+=15) printf("%6.2f\t", clevel[i]);
  printf("\n");
  for (i=0;i<149;i+=15) printf("%6d\t", iclson[i]);
  printf("\n");
  for (i=0;i<149;i+=15) printf("%6d\t", icrson[i]);
  printf("\n");
}
```

Output

```
  0.00    0.17    0.23    0.27    0.31    0.37    0.41    0.48    0.60    0.78
   143     153      17     140      53     198     186     218     261     249
   102      29       6     113      51      91     212     243     266     262
```

cluster_number

Computes cluster membership for a hierarchical cluster tree.

Synopsis

#include <imsls.h>

int *imsls_cluster_number (*int* npt, *int* *iclson, *int* *icrson, *int* k, …, 0)

Required Arguments

int npt (Input)
Number of data points to be clustered.

int `*iclson` (Input)
Vector of length `npt` – 1 containing the left son cluster numbers. Cluster `npt` + `i` is formed by merging clusters `iclson[i-1]` and `icrson[i-1]`.

int `*icrson` (Input)
Vector of length `npt` – 1 containing the left son cluster numbers. Cluster `npt` + `i` is formed by merging clusters `iclson[i-1]` and `icrson[i-1]`.

int `k` (Input)
Desired number of clusters.

Return Value

Vector of length `npt` containing the cluster membership of each observation.

Synopsis with Optional Arugments

#include `<imsls.h>`

int `*imsls_cluster_number` (*int* `npt`, *int* `*iclson`, *int* `*icrson`, *int* `k`,
`IMSLS_OBS_PER_CLUSTERS`, *int* `**nclus`,
`IMSLS_OBS_PER_CLUSTERS_USER`, *int* `nclus[]`,
`IMSLS_RETURN_USER`, *int* `iclus[]`,
0)

Optional Arguments

`IMSLS_OBS_PER_CLUSTERS`, *int* `**nclus` (Output)
Address of a pointer to an internally allocated array of length `k` containing the number of observations in each cluster.

`IMSLS_OBS_PER_CLUSTERS_USER`, *int* `nclus[]` (Output)
Storage for array `nclus` is provided by the user. See `IMSLS_OBS_PER_CLUSTERS`.

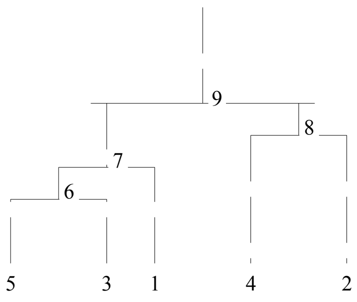
`IMSLS_RETURN_USER`, *float* `iclus[]` (Output)
User allocated array of length `npt` containing the cluster membership of each observation.

Description

Given a fixed number of clusters (K) and the cluster tree (vectors `icrson` and `iclson`) produced by the hierarchical clustering algorithm (see function `imsls_f_cluster_hierarchical`, function `imsls_cluster_number` determines the cluster membership of each observation. The function `imsls_cluster_number` first determines the root nodes for the K distinct subtrees forming the K clusters and then traverses each subtree to determine the cluster membership of each observation. The function `imsls_cluster_number` also returns the number of observations found in each cluster.

Example 1

In the following example, cluster membership for $K = 2$ clusters is found for the displayed cluster tree. The output vector `iclus` contains the cluster numbers for each observation.



```
#include "imsls.h"

void main()
{
  int  k = 2, npt = 5, *iclus;
  int iclson[] = {5, 6, 4, 7};
  int icrson[] = {3, 1, 2, 8};

  iclus = imsls_cluster_number(npt, iclson, icrson, k, 0);
  imsls_i_write_matrix("iclus", 1, 5, iclus, 0);
}
```

Output

```
        iclus
 1   2   3   4   5
 1   2   1   2   1
```

Example 2

This example illustrates the typical usage of `imsls_cluster_number`. The Fisher iris data (see function `imsls_f_data_sets`, see Chapter 15, "Utilities") is clustered. First the distance between the irises are computed using function `imsls_f_dissimilarities`. The resulting distance matrix is then clustered using function `imsls_f_cluster_hierarchical`. The cluster membership for 5 clusters is then obtained via function `imsls_cluster_number` using the output from `imsls_f_cluster_hierarchical`. The need for 5 clusters can be obtained either by theoretical means or by examining a cluster tree. The cluster membership for each of the iris observations is printed.

```
#include "imsls.h"
#define MAX(A,B) ((A)>(B)?(A): (B))


void main()
```

```
{
  int ncol = 5, nrow = 150, nvar = 4, npt = 150, k = 5;
  int i, j, *iclson, *icrson, *iclus, *nclus;
  int ind[4] = {1, 2, 3, 4};
  float *clevel, dist[150][150], *x, f_rand;
  int *p_iclus = NULL, *p_nclus = NULL;

  x = imsls_f_data_sets (3, 0);
  imsls_f_dissimilarities(nrow, ncol, x,
                      IMSLS_INDEX, nvar, ind,
                      IMSLS_RETURN_USER, dist,
                      0);

  imsls_random_seed_set (4);
  for (i = 0; i < npt; i++)
    {
      for (j = i + 1; j < npt; j++)
       {
         imsls_f_random_uniform (1, IMSLS_RETURN_USER, &f_rand, 0);
         dist[i][j] = MAX (0.0, dist[i][j] + .001 * f_rand);
         dist[j][i] = dist[i][j];
       }
      dist[i][i] = 0.;
    }
  imsls_f_cluster_hierarchical (npt, (float*)dist,
               IMSLS_CLUSTERS, &clevel, &iclson, &icrson,
               0);

  iclus = imsls_cluster_number (npt, iclson, icrson, k,
                          IMSLS_OBS_PER_CLUSTER, &nclus,
                          0);

  imsls_i_write_matrix ("iclus", 25, 5, iclus, 0);
  imsls_i_write_matrix ("nclus", 1, 5, nclus, 0); }
```

Output

```
         iclus
     1   2   3   4   5
 1   5   5   5   5   5
 2   5   5   5   5   5
 3   5   5   5   5   5
 4   5   5   5   5   5
 5   5   5   5   5   5
 6   5   5   5   5   5
 7   5   5   5   5   5
 8   5   5   5   5   5
 9   5   5   5   5   5
10   5   5   5   5   5
11   2   2   2   2   2
12   2   2   1   2   2
13   1   2   2   2   2
14   2   2   2   2   2
15   2   2   2   2   2
16   2   2   2   2   2
17   2   2   2   2   2
18   2   2   2   2   2
```

```
19   2   2   2   1   2
20   2   2   2   1   2
21   2   2   2   2   2
22   2   3   2   2   2
23   2   2   2   2   2
24   2   2   4   2   2
25   2   2   2   2   2

          nclus
  1    2    3    4    5
  4   93    1    2   50
```

cluster_k_means

Performs a *K*-means (centroid) cluster analysis.

Synopsis

#include `<imsls.h>`

int `*imsls_f_cluster_k_means` (*int* `n_observations`, *int* `n_variables`,
    float `x[]`, *int* `n_clusters`, *float* `cluster_seeds`, …, 0)

The type *double* function is `imsls_d_cluster_k_means`.

Required Arguments

int `n_observations` (Input)
    Number of observations.

int `n_variables` (Input)
    Number of variables to be used in computing the metric.

float `x[]` (Input)
    Array of length `n_observations` × `n_variables` containing the observations to be clustered.

int `n_clusters` (Input)
    Number of clusters.

float `cluster_seeds[]` (Input)
    Array of length `n_clusters` × `n_variables` containing the cluster seeds, i.e., estimates for the cluster centers.

Return Value

The cluster membership for each observation is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_cluster_k_means` (*int* `n_observations`, *int* `n_variables`,
    float `x[]`, *int* `n_clusters`, *float* `cluster_seeds`,
    `IMSLS_WEIGHTS`, *float* `weights[]`,
    `IMSLS_FREQUENCIES`, *float* `frequencies[]`,

IMSLS_MAX_ITERATIONS, *int* max_iterations,
IMSLS_CLUSTER_MEANS, *float* **cluster_means,
IMSLS_CLUSTER_MEANS_USER, *float* cluster_means[],
IMSLS_CLUSTER_SSQ, *float* **cluster_ssq,
IMSLS_CLUSTER_SSQ_USER, *float* cluster_ssq[],
IMSLS_X_COL_DIM, *int* x_col_dim,
IMSLS_CLUSTER_MEANS_COL_DIM, *int* cluster_means_col_dim,
IMSLS_CLUSTER_SEEDS_COL_DIM, *int* cluster_seeds_col_dim,
IMSLS_CLUSTER_COUNTS, *int* **cluster_counts,
IMSLS_CLUSTER_COUNTS_USER, *int* cluster_counts[],
IMSLS_CLUSTER_VARIABLE_COLUMNS, *int* cluster_variables[],
IMSLS_RETURN_USER, *int* cluster_group[],
0)

Optional Arguments

IMSLS_WEIGHTS, *float* weights[] (Input)
Array of length n_observations containing the weight of each observation of matrix x.
Default: weights [ ] = **1**

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

IMSLS_MAX_ITERATIONS, *int* max_iterations (Input)
Maximum number of iterations.
Default: max_iterations = 30

IMSLS_CLUSTER_MEANS, *float* **cluster_means (Output)
The address of a pointer to an internally allocated array of length n_clusters × n_variables containing the cluster means.

IMSLS_CLUSTER_MEANS_USER, *float* cluster_means[] (Output)
Storage for array cluster_means is provided by the user. See IMSLS_CLUSTER_MEANS.

IMSLS_CLUSTER_SSQ, *float* **cluster_ssq (Output)
The address of a pointer to internally allocated array of length n_clusters containing the within sum-of-squares for each cluster.

IMSLS_CLUSTER_SSQ_USER, *float* cluster_ssq[] (Output)
Storage for array cluster_ssq is provided by the user. See IMSLS_CLUSTER_SSQ.

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

`IMSLS_CLUSTER_MEANS_COL_DIM`, *int* `cluster_means_col_dim` (Input)
Column dimension for the vector `cluster_means`.
Default: `cluster_means_col_dim = n_variables`

`IMSLS_CLUSTER_SEEDS_COL_DIM`, *int* `cluster_seeds_col_dim` (Input)
Column dimension for the vector `cluster_seeds`.
Default: `cluster_seeds_col_dim = n_variables`

`IMSLS_CLUSTER_COUNTS`, *int* `**cluster_counts` (Output)
The address of a pointer to an internally allocated array of length `n_clusters` containing the number of observations in each cluster.

`IMSLS_CLUSTER_COUNTS_USER`, *int* `cluster_counts[]` (Output)
Storage for array `cluster_counts` is provided by the user. See `IMSLS_CLUSTER_COUNTS`.

`IMSLS_CLUSTER_VARIABLE_COLUMNS`, *int* `cluster_variables[]` (Input)
Vector of length `n_variables` containing the columns of `x` to be used in computing the metric. Columns are numbered 0, 1, 2, ..., `n_variables`
Default: `cluster_variables` [ ] = 0, 1, 2, …, `n_variables`

`IMSLS_RETURN_USER`, *int* `cluster_group[]` (Output)
User-allocated array of length `n_observations` containing the cluster membership for each observation.

Description

Function `imsls_f_cluster_k_means` is an implementation of Algorithm AS 136 by Hartigan and Wong (1979). It computes *K*-means (centroid) Euclidean metric clusters for an input matrix starting with initial estimates of the *K*-cluster means. The function allows for missing values coded as NaN (Not a Number) and for weights and frequencies.

Let p = `n_variables` be the number of variables to be used in computing the Euclidean distance between observations. The idea in *K*-means cluster analysis is to find a clustering (or grouping) of the observations so as to minimize the total within-cluster sums-of-squares. In this case, the total sums-of-squares within each cluster is computed as the sum of the centered sum-of-squares over all nonmissing values of each variable. That is,

$$\phi = \sum_{i=1}^{K} \sum_{j=1}^{p} \sum_{m=1}^{n_i} f_{\nu_{im}} w_{\nu_{im}} \delta_{\nu_{im},j} \left(x_{\nu_{im},j} - \bar{x}_{ij} \right)^2$$

where ν_{im} denotes the row index of the m-th observation in the i-th cluster in the matrix X; n_i is the number of rows of X assigned to group i; f denotes the frequency of the observation; w denotes its weight; δ is 0 if the j-th variable on observation ν_{im} is missing, otherwise δ is 1; and

$$\bar{x}_{ij}$$

is the average of the nonmissing observations for variable *j* in group *i*. This method sequentially processes each observation and reassigns it to another cluster if doing so results in a decrease of the total within-cluster sums-of-squares. See Hartigan and Wong (1979) or Hartigan (1975) for details.

Example

This example performs *K*-means cluster analysis on Fisher's iris data, which is obtained by function `imsls_f_data_sets` (see Chapter 15, "Utilities"). The initial cluster seed for each iris type is an observation known to be in the iris type.

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

main()
{
#define N_OBSERVATIONS 150
#define N_VARIABLES    4
#define N_CLUSTERS     3
    float        x[N_OBSERVATIONS][5];
    float        cluster_seeds[N_CLUSTERS][N_VARIABLES];
    float        cluster_means[N_CLUSTERS][N_VARIABLES];
    float        cluster_ssq[N_CLUSTERS];
    int          cluster_variables[N_VARIABLES] = {1, 2, 3, 4};
    int          cluster_counts[N_CLUSTERS];
    int          cluster_group[N_OBSERVATIONS];
    int          i;

                 /* Retrieve the data set */
    imsls_f_data_sets(3, IMSLS_RETURN_USER, x, 0);
                 /* Assign initial cluster seeds */
    for (i=0; i<N_VARIABLES; i++) {
        cluster_seeds[0][i] = x[0][i+1];
        cluster_seeds[1][i] = x[50][i+1];
        cluster_seeds[2][i] = x[100][i+1];
    }

                 /* Perform the analysis */
    imsls_f_cluster_k_means(N_OBSERVATIONS, N_VARIABLES, (float*)x,
        N_CLUSTERS, (float*)cluster_seeds,
        IMSLS_X_COL_DIM,           5,
        IMSLS_CLUSTER_VARIABLE_COLUMNS,  cluster_variables,
        IMSLS_CLUSTER_COUNTS_USER,     cluster_counts,
        IMSLS_CLUSTER_MEANS_USER, cluster_means,
        IMSLS_CLUSTER_SSQ_USER,   cluster_ssq,
        IMSLS_RETURN_USER,        cluster_group,
        0);
                /* Print results */
    imsls_i_write_matrix("Cluster Membership", 1, N_OBSERVATIONS,
        cluster_group, 0);
    imsls_f_write_matrix("Cluster Means", N_CLUSTERS, N_VARIABLES,
        (float*)cluster_means, 0);
    imsls_f_write_matrix("Cluster Sum of Squares", 1, N_CLUSTERS,
        cluster_ssq, 0);
    imsls_i_write_matrix("# Observations in Each Cluster", 1,
        N_CLUSTERS, cluster_counts, 0);
}

                              Cluster Membership
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
 1  1  1  1  1  1  1  1  1   1   1   1   1   1   1   1   1   1   1   1

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 1  1  1  1  1  1  1  1  1  1  2  2  3  2  2  2  2  2  2  2

61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  3  2  2

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   2   3   3   3   3   2   3   3   3   3   3   3   2   2

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   2   3   2   3   2   3   3   2   2   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   2   3   3   3   2   3   3   3   2

148 149 150
  3   3   2

                 Cluster Means
            1           2           3           4
1       5.006       3.428       1.462       0.246
2       5.902       2.748       4.394       1.434
3       6.850       3.074       5.742       2.071

      Cluster Sum of Squares
          1           2           3
     15.15       39.82       23.88

# Observations in Each Cluster
           1    2    3
          50   62   38
```

Warning Errors

`IMSLS_NO_CONVERGENCE`	Convergence did not occur.

principal_components

Computes principal components.

Synopsis

#include `<imsls.h>`

float `*imsls_f_principal_components` (*int* `n_variables`,
float `covariances[]`, ..., 0)

The type *double* function is `imsls_d_principal_components`.

Required Arguments

int `n_variables` (Input)
Order of the covariance matrix.

float `covariances[]` (Input)
Array of length `n_variables` × `n_variables` containing the covariance or correlation matrix.

Return Value

An array of length `n_variables` containing the eigenvalues of the matrix `covariances` ordered from largest to smallest.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float *imsls_f_principal_components (int n_variables,
        float covariances[],
        IMSLS_COVARIANCE_MATRIX, or
        IMSLS_CORRELATION_MATRIX,
        IMSLS_CUM_PERCENT, float **cum_percent,
        IMSLS_CUM_PERCENT_USER, float cum_percent[],
        IMSLS_EIGENVECTORS, float **eigenvectors,
        IMSLS_EIGENVECTORS_USER, float eigenvectors[],
        IMSLS_CORRELATIONS, float **correlations,
        IMSLS_CORRELATIONS_USER, float correlations[],
        IMSLS_STD_DEV, int n_degrees_freedom, float **std_dev,
        IMSLS_STD_DEV_USER, int n_degrees_freedom,     float std_dev[],
        IMSLS_COV_COL_DIM, int cov_col_dim,
        IMSLS_RETURN_USER, float eigenvalues[],
        0)
```

Optional Arguments

`IMSLS_COVARIANCE_MATRIX`
Treat the input vector `covariances` as a covariance matrix. This option is the default.
or

`IMSLS_CORRELATION_MATRIX`
Treat the input vector `covariances` as a correlation matrix.

`IMSLS_CUM_PERCENT`, *float* `**cum_percent` (Output)
The address of a pointer to an internally allocated array of length `n_variables` containing the cumulative percent of the total variances explained by each principal component.

`IMSLS_CUM_PERCENT_USER`, *float* `cum_percent[]` (Output)
: Storage for array `cum_percent` is provided by the user. See `IMSLS_CUM_PERCENT`.

`IMSLS_EIGENVECTORS`, *float* `**eigenvectors` (Output)
: The address of a pointer to an internally allocated array of length `n_variables` × `n_variables` containing the eigenvectors of `covariances`, stored columnwise. Each vector is normalized to have Euclidean length equal to the value one. Also, the sign of each vector is set so that the largest component in magnitude (the first of the largest if there are ties) is made positive.

`IMSLS_EIGENVECTORS_USER`, *float* `eigenvectors[]` (Output)
: Storage for array `eigenvectors` is provided by the user. See `IMSLS_EIGENVECTORS`.

`IMSLS_CORRELATIONS`, *float* `**correlations` (Output)
: The address of a pointer to an internally allocated array of length `n_variables * n_variables` containing the correlations of the principal components (the columns) with the observed/standardized variables (the rows). If `IMSLS_COVARIANCE_MATRIX` is specified, then the correlations are with the observed variables. Otherwise, the correlations are with the standardized (to a variance of 1.0) variables. In the principal component model for factor analysis, matrix `correlations` is the matrix of unrotated factor loadings.

`IMSLS_CORRELATIONS_USER`, *float* `correlations[]` (Output)
: Storage for array `correlations` is provided by the user. See `IMSLS_CORRELATIONS`.

`IMSLS_STD_DEV`, *int* `n_degrees_freedom`, *float* `**std_dev` (Input/Output)
: Argument `n_degrees_freedom` contains the number of degrees of freedom in `covariances`. Argument `std_dev` is the address of a pointer to an internally allocated array of length `n_variables` containing the estimated asymptotic standard errors of the eigenvalues.

`IMSLS_STD_DEV_USER`, *int* `n_degrees_freedom`, *float* `std_dev[]`
: (Input/Output)
: Storage for array `std_dev` is provided by the user. See `IMSLS_STD_DEV`.

`IMSLS_COV_COL_DIM` *int* `cov_col_dim` (Input)
: Column dimension of covariances.
: Default: `cov_col_dim = n_variables`

`IMSLS_RETURN_USER`, *float* `eigenvalues[]` (Output)
: User-supplied array of length `n_variables` containing the eigenvalues of covariances ordered from largest to smallest.

Description

Function `imsls_f_principal_components` finds the principal components of a set of variables from a sample covariance or correlation matrix. The characteristic roots, characteristic vectors, standard errors for the characteristic roots, and the correlations of the principal component scores with the original variables are computed. Principal components obtained from correlation matrices are the same as principal components obtained from standardized (to unit variance) variables.

The principal component scores are the elements of the vector $y = \Gamma^T x$, where Γ is the matrix whose columns are the characteristic vectors (eigenvectors) of the sample covariance (or correlation) matrix and x is the vector of observed (or standardized) random variables. The variances of the principal component scores are the characteristic roots (eigenvalues) of the covariance (correlation) matrix.

Asymptotic variances for the characteristic roots were first obtained by Girschick (1939) and are given more recently by Kendall et al. (1983, p. 331). These variances are computed either for covariance matrices or for correlation matrices.

The correlations of the principal components with the observed (or standardized) variables are given in the matrix `correlations`. When the principal components are obtained from a correlation matrix, `correlations` is the same as the matrix of unrotated factor loadings obtained for the principal components model for factor analysis.

Examples

Example 1

In this example, eigenvalues of the covariance matrix are computed.

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

main()
{
#define N_VARIABLES 9

    float  *values;
    static float covariances[N_VARIABLES][N_VARIABLES] = {
        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,   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};

                    /* Perform analysis */
    values = imsls_f_principal_components(N_VARIABLES, covariances, 0);

                    /* Print results. */
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, values, 0);

                    /* Free allocated memory. */
    free(values);
}
```

Output

```
                             Eigenvalues
          1            2            3            4            5            6
      4.677        1.264        0.844        0.555        0.447        0.429

          7            8            9
      0.310        0.277        0.196
```

Example 2

In this example, principal components are computed for a nine-variable correlation matrix.

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

main()
{
#define N_VARIABLES 9

    float  *values, *eigenvectors, *std_dev, *cum_percent, *a;
    static float covariances[N_VARIABLES][N_VARIABLES] = {
        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,   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};

                    /* Perform analysis */
    values = imsls_f_principal_components(N_VARIABLES, covariances,
        IMSLS_CORRELATION_MATRIX,
        IMSLS_EIGENVECTORS,                    &eigenvectors,
        IMSLS_STD_DEV,                         100, &std_dev,
        IMSLS_CUM_PERCENT,                     &cum_percent,
        IMSLS_CORRELATIONS, &a,
        0);

                    /* Print results */
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, values, 0);
    imsls_f_write_matrix("Eigenvectors", N_VARIABLES, N_VARIABLES,
        eigenvectors, 0);
    imsls_f_write_matrix("STD", 1, N_VARIABLES, std_dev, 0);
    imsls_f_write_matrix("PCT", 1, N_VARIABLES, cum_percent, 0);
    imsls_f_write_matrix("A", N_VARIABLES, N_VARIABLES, a, 0);

                   /* Free allocated memory */
    free(values);
    free(eigenvectors);
    free (cum_percent);
    free (std_dev);
    free(a);
}
```

Output

```
                              Eigenvalues
          1           2           3           4           5           6
      4.677       1.264       0.844       0.555       0.447       0.429

          7           8           9
      0.310       0.277       0.196


                              Eigenvectors
             1           2           3           4           5           6
1       0.3462     -0.2354      0.1386     -0.3317     -0.1088      0.7974
2       0.3526     -0.1108     -0.2795     -0.2161      0.7664     -0.2002
3       0.2754     -0.2697     -0.5585      0.6939     -0.1531      0.1511
4       0.3664      0.4031      0.0406      0.1196      0.0017      0.1152
5       0.3144      0.5022     -0.0733     -0.0207     -0.2804     -0.1796
6       0.3455      0.4553      0.1825      0.1114      0.1202      0.0697
7       0.3487     -0.2714     -0.0725     -0.3545     -0.5242     -0.4355
8       0.2407     -0.3159      0.7383      0.4329      0.0861     -0.1969
9       0.3847     -0.2533     -0.0078     -0.1468      0.0459     -0.1498

             7           8           9
1       0.1735     -0.1240     -0.0488
2       0.1386     -0.3032     -0.0079
3       0.0099     -0.0406     -0.0997
4      -0.4022     -0.1178      0.7060
5       0.7295      0.0075      0.0046
6      -0.3742      0.0925     -0.6780
7      -0.2854     -0.3408     -0.1089
8       0.1862     -0.1623      0.0505
9      -0.0251      0.8521      0.1225

                                  STD
          1           2           3           4           5           6
     0.6498      0.1771      0.0986      0.0879      0.0882      0.0890

          7           8           9
     0.0944      0.0994      0.1113

                                  PCT
          1           2           3           4           5           6
      0.520       0.660       0.754       0.816       0.865       0.913

          7           8           9
      0.947       0.978       1.000

                                    A
             1           2           3           4           5           6
1       0.7487     -0.2646      0.1274     -0.2471     -0.0728      0.5224
2       0.7625     -0.1245     -0.2568     -0.1610      0.5124     -0.1312
3       0.5956     -0.3032     -0.5133      0.5170     -0.1024      0.0990
```

```
4      0.7923      0.4532      0.0373      0.0891      0.0012      0.0755
5      0.6799      0.5646     -0.0674     -0.0154     -0.1875     -0.1177
6      0.7472      0.5119      0.1677      0.0830      0.0804      0.0456
7      0.7542     -0.3051     -0.0666     -0.2641     -0.3505     -0.2853
8      0.5206     -0.3552      0.6784      0.3225      0.0576     -0.1290
9      0.8319     -0.2848     -0.0071     -0.1094      0.0307     -0.0981

            7           8           9
1      0.0966     -0.0652     -0.0216
2      0.0772     -0.1596     -0.0035
3      0.0055     -0.0214     -0.0442
4     -0.2240     -0.0620      0.3127
5      0.4063      0.0039      0.0021
6     -0.2084      0.0487     -0.3003
7     -0.1589     -0.1794     -0.0482
8      0.1037     -0.0854      0.0224
9     -0.0140      0.4485      0.0543
```

Warning Errors

`IMSLS_100_DF`	Because the number of degrees of freedom in "covariances" and "n_degrees_freedom" is less than or equal to 0, 100 degrees of freedom will be used.
`IMSLS_COV_NOT_NONNEG_DEF`	"eigenvalues[#]" = #. One or more eigenvalues much less than zero are computed. The matrix "covariances" is not nonnegative definite. In order to continue computations of "eigenvalues" and "correlations," these eigenvalues are treated as 0.
`IMSLS_FAILED_TO_CONVERGE`	The iteration for the eigenvalue failed to converge in 100 iterations before deflating.

factor_analysis

Extracts initial factor-loading estimates in factor analysis with rotation options.

Synopsis

#include `<imsls.h>`

float `*imsls_f_factor_analysis` (*int* `n_variables`, *float* `covariances[]`,
    int `n_factors`, ..., 0)

The type *double* function is `imsls_d_factor_analysis`.

Required Arguments

int `n_variables` (Input)
    Number of variables.

float `covariances[]` (Input)

Array of length `n_variables*n_variables` containing the variance-covariance or correlation matrix.

int `n_factors` (Input)

Number of factors in the model.

Return Value

An array of length `n_variables*n_factors` containing the matrix of factor loadings.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_factor_analysis (int n_variables,
        float covariances[], int n_factors,
        IMSLS_MAXIMUM_LIKELIHOOD, int df_covariances, or
        IMSLS_PRINCIPAL_COMPONENT, or
        IMSLS_PRINCIPAL_FACTOR, or
        IMSLS_UNWEIGHTED_LEAST_SQUARES,or
        IMSLS_GENERALIZED_LEAST_SQUARES, int df_covariances, or
        IMSLS_IMAGE, or
        IMSLS_ALPHA, int df_covariances,
        IMSLS_UNIQUE_VARIANCES_INPUT,float unique_variances[],
        IMSLS_UNIQUE_VARIANCES_OUTPUT,
                float unique_variances[],
        IMSLS_MAX_ITERATIONS, int max_iterations,
        IMSLS_MAX_STEPS_LINE_SEARCH, int max_steps_line_search,
        IMSLS_CONVERGENCE_EPS,float convergence_eps,
        IMSLS_SWITCH_EXACT_HESSIAN,float switch_epsilon,
        IMSLS_EIGENVALUES,float **eigenvalues,
        IMSLS_EIGENVALUES_USER,float eigenvalues[],
        IMSLS_CHI_SQUARED_TEST, int *df,float *chi_squared,
                float *p_value,
        IMSLS_TUCKER_RELIABILITY_COEFFICIENT,float *coefficient,
        IMSLS_N_ITERATIONS, int *n_iterations,
        IMSLS_FUNCTION_MIN,float *function_min,
        IMSLS_LAST_STEP,float **last_step,
        IMSLS_LAST_STEP_USER,float last_step[],
        IMSLS_ORTHOMAX_ROTATION, float w, int norm, float **b,
                float **t,
        IMSLS_ORTHOMAX_ROTATION_USER, float w, int norm, float b[],
                float t[],
        IMSLS_ORTHOGONAL_PROCUSTES_ROTATION, float target[],
                float **b, float **t,
        IMSLS_ORTHOGONAL_PROCUSTES_ROTATION_USER,
                float target[], float b[], float t[],
        IMSLS_DIRECT_OBLIMIN_ROTATION, float w, int norm, float **b,
                *float* **t, *float* **factor_correlations,
        IMSLS_DIRECT_OBLIMIN_ROTATION_USER, *float* w, *int* norm,
                *float* b[], *float* t[], *float* factor_correlations[],
        IMSLS_OBLIQUE_PROMAX_ROTATION, *float* w, *float* power[],
                *int* norm, *float* **target, *float* **b, *float* **t,
                *float* **factor_correlations,
        IMSLS_OBLIQUE_PROMAX_ROTATION_USER, *float* w, *float* power[], *nt*
        norm, *float* target[], *float* b[], *float* t[],
                *loat* factor_correlations[],
        IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION, *float* w,
                *float* pivot[], *int* norm, *float* **target, *float* **b,
                *float* **t, *float* **factor_correlations,
        IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION_USER, *float* w,
                *loat* pivot[], *int* norm, *float* target[], *float* b[],
                *float* t[], *float* factor_correlations[],
        IMSLS_OBLIQUE_PROCRUSTES_ROTATION, *float* target[],
                *float* **b, *float* **t, *float* **factor_correlations,
        IMSLS_OBLIQUE_PROCRUSTES_ROTATION_USER, *float* target[],
                *float* b[], *float* t[], *float* factor_correlations[],
        IMSLS_FACTOR_STRUCTURE, *float* **s, *float* **fvar,
        IMSLS_FACTOR_STRUCTURE_USER, *float* s[], *float* fvar[],
        IMSLS_COV_COL_DIM, *int* cov_col_dim,
        IMSLS_RETURN_USER, *float* factor_loadings[],
        0)
```

Optional Arguments

`IMSLS_MAXIMUM_LIKELIHOOD`, *int* `df_covariances` (Input)
Maximum likelihood (common factor model) method used to obtain the estimates. Argument `df_covariances` is the number of degrees of freedom in covariances.
or

`IMSLS_PRINCIPAL_COMPONENT`
Principal component (principal component model) method used to obtain the estimates.
or

`IMSLS_PRINCIPAL_FACTOR`
Principal factor (common factor model) method used to obtain the estimates.
or

`IMSLS_UNWEIGHTED_LEAST_SQUARES`
Unweighted least-squares (common factor model) method used to obtain the estimates. This option is the default.
or

`IMSLS_GENERALIZED_LEAST_SQUARES`, *int* `df_covariances` (Input)
Generalized least-squares (common factor model) method used to obtain the

estimates.
or

`IMSLS_IMAGE`
Image-factor analysis (common factor model) method used to obtain the estimates.
or

`IMSLS_ALPHA`, *int* `df_covariances` (Input)
Alpha-factor analysis (common factor model) method used to obtain the estimates. Argument `df_covariances` is the number of degrees of freedom in covariances.

`IMSLS_UNIQUE_VARIANCES_INPUT`, *float* `unique_variances[]` (Input)
Array of length `n_variables` containing the initial estimates of the unique variances.
Default: Initial estimates are taken as the constant
1 − `n_factors`/2 `*` `n_variables` divided by the diagonal elements of the inverse of `covariances`.

`IMSLS_UNIQUE_VARIANCES_OUTPUT`, *float* `unique_variances[]` (Output)
User-allocated array of length `n_variables` containing the estimated unique variances.

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations in the iterative procedure.
Default: `max_iterations` = 60

`IMSLS_MAX_STEPS_LINE_SEARCH`, *int* `max_steps_line_search` (Input)
Maximum number of step halvings allowed during any one iteration.
Default: `max_steps_line_search` = 10

`IMSLS_CONVERGENCE_EPS`, *float* `convergence_eps` (Input)
Convergence criterion used to terminate the iterations. For the unweighted least squares, generalized least squares or maximum likelihood methods, convergence is assumed when the relative change in the criterion is less than `convergence_eps`. For alpha-factor analysis, convergence is assumed when the maximum change (relative to the variance) of a uniqueness is less than `convergence_eps`.
Default: `convergence_eps` = 0.0001

`IMSLS_SWITCH_EXACT_HESSIAN`, *float* `switch_epsilon` (Input)
Convergence criterion used to switch to exact second derivatives. When the largest relative change in the unique standard deviation vector is less than `switch_epsilon`, exact second derivative vectors are used. Argument `switch_epsilon` is not used with the principal component, principal factor, image-factor analysis, or alpha-factor analysis methods.
Default: `switch_epsilon` = 0.1

`IMSLS_EIGENVALUES`, *float* `**eigenvalues` (Output)
The address of a pointer to an internally allocated array of length

n_variables containing the eigenvalues of the matrix from which the factors were extracted.

IMSLS_EIGENVALUES_USER, *float* eigenvalues[] (Output)
Storage for array eigenvalues is provided by the user. See IMSLS_EIGENVALUES.

IMSLS_CHI_SQUARED_TEST, *int* *df, *float* *chi_squared, *float* *p_value (Output)
Number of degrees of freedom in chi-squared is df; chi_squared is the chi-squared test statistic for testing that n_factors common factors are adequate for the data; p_value is the probability of a greater chi-squared statistic.

IMSLS_TUCKER_RELIABILITY_COEFFICIENT, *float* *coefficient (Output)
Tucker reliability coefficient.

IMSLS_N_ITERATIONS, *int* *n_iterations (Output)
Number of iterations.

IMSLS_FUNCTION_MIN, *float* *function_min (Output)
Value of the function minimum.

IMSLS_LAST_STEP, *float* **last_step (Output)
Address of a pointer to an internally allocated array of length n_variables containing the updates of the unique variance estimates when convergence was reached (or the iterations terminated).

IMSLS_LAST_STEP_USER, *float* last_step[] (Output)
Storage for array last_step is provided by the user. See IMSLS_LAST_STEP.

IMSLS_ORTHOMAX_ROTATION, *float* w, *int* norm, *float* **b, *float* **t (Input/Output)
Nonnegative constant w defines the rotation. If norm =1, row normalization is performed. Otherwise, row normalization is not performed. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix. w = 0.0 results in quartimax rotations, w = 1.0 results in varimax rotations, and w = n_factors/2.0 results in equamax rotations. Other nonnegative values of w may also be used, but the best values for w are in the range (0.0, 5 * n_factors).

IMSLS_ORTHOMAX_ROTATION_USER, *float* w, *int* norm, *float* b[], *float* t[] (Input/Output)
Storage for b and t are provided by the user. See IMSLS_ORTHOMAX_ROTATION.

IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION, *float* target[], *float* **b, *float* **t (Input/Output)
If specified, the n_variables by n_factors target matrix target will be used to compute an orthogonal Procrustes rotation of the factor-loading

matrix. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix.

IMSLS_ORTHOGONAL_PROCRUTES_ROTATION_USER, *float* target[], *float* b[], *float* t[] (Input/Output)
Storage for b and t are provided by the user. See IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION.

IMSLS_DIRECT_OBLIMIN_ROTATION, *float* w , *int* norm, *float* **b, *float* **t, *float* **factor_correlations (Input/Output)
Computes a direct oblimin rotation. Nonpositive constant w defines the rotation. If norm =1, row normalization is performed. Otherwise, row normalization is not performed. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix. factor_correlations contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the factor correlations. The parameter w determines the type of direct oblimin rotation to be performed. In general w must be negative. w = 0.0 results in direct quartimin rotations. As w approaches negative infinity, the orthogonality among factors will increase.

IMSLS_DIRECT_OBLIMIN_ROTATION_USER, *float* w, *int* norm, *float* b[], *float* t[], *float* factor_correlations[] (Input/Output)
Storage for b, t and factor_correlations are provided by the user. See IMSLS_DIRECT_OBLIMIN_ROTATION.

IMSLS_OBLIQUE_PROMAX_ROTATION, *float* w, *float* power[], *int* norm, *float* **target, *float* **b, *float* **t, *float* **factor_correlations, (Input/Output)
Computes an oblique promax rotation of the factor loading matrix using a power vector. Nonnegative constant w defines the rotation. power, a vector of length n_factors containing the power vector. If norm =1, row (Kaiser) normalization is performed. Otherwise, row normalization is not performed. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix. factor_correlations contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the factor correlations. target contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the target matrix for rotation, derived from the orthomax rotation. w is used in the orthomax rotation, see the optional argument IMSLS_ORTHOMAX_ROTATION for common values of w.

All `power[j]` should be greater than 1.0, typically 4.0. Generally, the larger the values of `power [j]`, the more oblique the solution will be.

`IMSLS_OBLIQUE_PROMAX_ROTATION_USER`, *float* `w`, *float* `power[]`, *int* `norm`, *float* `target[]`, *float* `b[]`, *float* `t[]`, *float* `factor_correlations[]`, (Input/Output)
Storage for `b`, `t`, `factor_correlations`, and `target` are provided by the user. See `IMSLS_OBLIQUE_PROMAX_ROTATION`.

`IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`, *float* `w`, *float* `pivot[]`, *int* `norm`, *float* `**target` , *float* `**b`, *float* `**t`, *float* `**factor_correlations`, (Input/Output)
Computes an oblique pivotal promax rotation of the factor loading matrix using pivot constants. Nonnegative constant `w` defines the rotation. `pivot`, a vector of length `n_factors` containing the pivot constants. `pivot[j]` should be in the interval (0.0, 1.0). If `norm` =1, row (Kaiser) normalization is performed. Otherwise, row normalization is not performed. `b` contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the rotation transformation matrix. `factor_correlations` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the factor correlations. `target` contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the target matrix for rotation, derived from the orthomax rotation. `w` is used in the orthomax rotation, see the optional argument `IMSLS_ORTHOMAX_ROTATION` for common values of `w`.

`IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION_USER`, *float* `w`, *float* `pivot[]`, *int* `norm`, *float* `target[]`, *float* `b[]`, *float* `t[]`, *float* `factor_correlations[]`, (Input/Output)
Storage for `b`, `t`, `factor_correlations`, and `target` are provided by the user. See `IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`.

`IMSLS_OBLIQUE_PROCRUSTES_ROTATION`, *float* `**target`, *float* `**b`, *float* `**t`, *float* `**factor_correlations` (Input/Output)
Computes an oblique procrustes rotation of the factor loading matrix using a target matrix. `target` is a hypothesized rotated factor loading matrix based upon prior knowledge with loadings chosen to the enhance interpretability. A simple structure solution will have most of the weights `target[i][j]` either zero or large in magnitude. `b` contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the rotation transformation matrix. `factor_correlations` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the factor correlations.

IMSLS_OBLIQUE_PROCRUSTES_ROTATION_USER, *float* target[],
float b[], *float* t[], *float* factor_correlations[] (Input/Output)
Storage for b, t, and factor_correlations are provided by the user. See IMSLS_PROCRUSTES_ROTATION.

IMSLS_FACTOR_STRUCTURE,*float* **s,*float* **fvar, (Output)
Computes the factor structure and the variance explained by each factor. s contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the factor structure matrix. fvar contains the address of a pointer to the internally allocated array of length n_factors containing the variance accounted for by each of the n_factors rotated factors. A factor rotation matrix is used to compute the factor structure and the variance. One and only one rotation option argument can be specified.

IMSLS_FACTOR_STRUCTURE_USER, *float* s[],*float* fvar[], (Output)
Storage for s, and fvar are provided by the user.
See IMSLS_FACTOR_STRUCTURE.

IMSLS_COV_COL_DIM, *int* cov_col_dim (Input)
Column dimension of the matrix covariances.
Default: cov_col_dim = n_variables

IMSLS_RETURN_USER, *float* factor_loadings[] (Output)
User-allocated array of length n_variables*n_factors containing the unrotated factor loadings.

Description

Function imsls_f_factor_analysis computes factor loadings in exploratory factor analysis models. Models available in imsls_f_factor_analysis are the principal component model for factor analysis and the common factor model with additions to the common factor model in alpha-factor analysis and image analysis. Methods of estimation include principal components, principal factor, image analysis, unweighted least squares, generalized least squares, and maximum likelihood.

In the factor analysis model used for factor extraction, the basic model is given as $\Sigma = \Lambda\Lambda^T + \Psi$, where Σ is the $p \times p$ population covariance matrix, Λ is the $p \times k$ matrix of factor loadings relating the factors f to the observed variables x, and Ψ is the $p \times p$ matrix of covariances of the unique errors e. Here, p = n_variables and k = n_factors. The relationship between the factors, the unique errors, and the observed variables is given as $x = \Lambda f + e$, where in addition, the expected values of e, f, and x are assumed to be 0. (The sample means can be subtracted from x if the expected value of x is not 0.) It also is assumed that each factor has unit variance, the factors are independent of each other, and that the factors and the unique errors are mutually independent. In the common factor model, the elements of unique errors e also are assumed to be independent of one another so that the matrix Ψ is diagonal. This is not the case in the principal component model in which the errors may be correlated.

Further differences between the various methods concern the criterion that is optimized and the amount of computer effort required to obtain estimates. Generally speaking, the least-squares and maximum likelihood methods, which use iterative algorithms, require the most computer time with the principal factor, principal component and the image methods requiring much less time since the algorithms in these methods are not iterative. The algorithm in alpha-factor analysis is also iterative, but the estimates in this method generally require somewhat less computer effort than the least-squares and maximum likelihood estimates. In all methods, one eigensystem analysis is required on each iteration.

Principal Component and Principal Factor Methods

Both the principal component and principal factor methods compute the factor-loading estimates as

$$\hat{\Gamma}\hat{\Delta}^{-1/2}$$

where Γ and the diagonal matrix Δ are the eigenvectors and eigenvalues of a matrix. In the principal component model, the eigensystem analysis is performed on the sample covariance (correlation) matrix S, while in the principal factor model, the matrix $(S + \Psi)$ is used. If the unique error variances Ψ are not known in the principal factor mode, then `imsls_f_factor_analysis` obtains estimates for them.

The basic idea in the principal component method is to find factors that maximize the variance in the original data that is explained by the factors. Because this method allows the unique errors to be correlated, some factor analysts insist that the principal component method is not a factor analytic method. Usually, however, the estimates obtained by the principal component model and factor analysis model will be quite similar.

It should be noted that both the principal component and principal factor methods give different results when the correlation matrix is used in place of the covariance matrix. Indeed, any rescaling of the sample covariance matrix can lead to different estimates with either of these methods. A further difficulty with the principal factor method is the problem of estimating the unique error variances. Theoretically, these must be known in advance and be passed to `imsls_f_factor_analysis` using optional argument `IMSLS_UNIQUE_VARIANCES_INPUT`. In practice, the estimates of these parameters are produced by `imsls_f_factor_analysis` when `IMSLS_UNIQUE_VARIANCES_INPUT` is not specified. In either case, the resulting adjusted covariance (correlation) matrix

$$S - \hat{\psi}$$

may not yield the `n_factors` positive eigenvalues required for `n_factors` factors to be obtained. If this occurs, the user must either lower the number of factors to be estimated or give new unique error variance values.

Least-squares and Maximum Likelihood Methods

Unlike the previous two methods, the algorithm used to compute estimates in this section is iterative (see Jöreskog 1977). As with the principal factor model, the user may either initialize the unique error variances or allow `imsls_f_factor_analysis` to compute initial estimates. Unlike the principal factor method, `imsls_f_factor_analysis` optimizes the criterion function with respect to both Ψ

and Γ. (In the principal factor method, Ψ is assumed to be known. Given Ψ, estimates for Λ may be obtained.)

The major difference between the methods discussed in this section is in the criterion function that is optimized. Let S denote the sample covariance (correlation) matrix, and let Σ denote the covariance matrix that is to be estimated by the factor model. In the unweighted least-squares method, also called the iterated principal factor method or the minres method (see Harman 1976, p. 177), the function minimized is the sum-of-squared differences between S and Σ. This is written as $\Phi_{ul} = 0.5\ (\text{trace}\ (S - \Sigma)^2)$.

Generalized least-squares and maximum likelihood estimates are asymptotically equivalent methods. Maximum likelihood estimates maximize the (normal theory) likelihood $\{\Phi_{ml} = \text{trace}\ (\Sigma^{-1} S) - \log\ (|\Sigma^{-1} S|)\}$, while generalized least squares optimizes the function $\Phi_{gs} = \text{trace}\ (\Sigma S^{-1} - I)^2$.

In all three methods, a two-stage optimization procedure is used. This proceeds by first solving the likelihood equations for Λ in terms of Ψ and substituting the solution into the likelihood. This gives a criterion $\phi\ (\Psi, \Lambda\ (\Psi))$, which is optimized with respect to Ψ. In the second stage, the estimates $\hat{\Lambda}$ are obtained from the estimates for Ψ.

The generalized least-squares and maximum likelihood methods allow for the computation of a statistic (`IMSLS_CHI_SQUARED_TEST`) for testing that `n_factors` common factors are adequate to fit the model. This is a chi-squared test that all remaining parameters associated with additional factors are 0. If the probability of a larger chi-squared is so small that the null hypothesis is rejected, then additional factors are needed (although these factors may not be of any practical importance). Failure to reject does not legitimize the model. The statistic `IMSLS_CHI_SQUARED_TEST` is a likelihood ratio statistic in maximum likelihood estimation. As such, it asymptotically follows a chi-squared distribution with degrees of freedom given by `df`.

The Tucker and Lewis reliability coefficient, ρ, is returned by `IMSLS_TUCKER_RELIABILITY_COEFFICIENT` when the maximum likelihood or generalized least-squares methods are used. This coefficient is an estimate of the ratio of explained variation to the total variation in the data. It is computed as follows:

$$\rho = \frac{mM_0 - mM_k}{mM_0 - 1}$$

$$m = d - \frac{2p+5}{6} - \frac{2k}{6}$$

$$M_0 = \frac{-\ln\left(|S|\right)}{p\left(p-1\right)/2}$$

$$M_k = \frac{\phi}{\left(\left(p-k\right)^2 - p - k\right)/2}$$

where $|S|$ is the determinant of covariances, p = `n_variables`, k = `n_variables`, ϕ is the optimized criterion, and d = `df_covariances`.

Image Analysis Method

The term *image analysis* is used here to denote the noniterative image method of Kaiser (1963). It is not the image analysis discussed by Harman (1976, p. 226). The image method (as well as the alpha-factor analysis method) begins with the notion that only a finite number from an infinite number of possible variables have been measured. The image factor pattern is calculated under the assumption that the ratio of the number of factors to the number of observed variables is near 0, so that a very good estimate for the unique error variances (for standardized variables) is given as 1 minus the squared multiple correlation of the variable under consideration with all variables in the covariance matrix.

First, the matrix $D^2 = (\text{diag}\,(S^{-1}))^{-1}$ is computed where the operator "diag" results in a matrix consisting of the diagonal elements of its argument and S is the sample covariance (correlation) matrix. Then, the eigenvalues Λ and eigenvectors Γ of the matrix $D^{-1}SD^{-1}$ are computed. Finally, the unrotated image-factor pattern is computed as $D\Gamma\,[(\Lambda - I)^2\Lambda^{-1}]^{1/2}$.

Alpha-factor Analysis Method

The alpha-factor analysis method of Kaiser and Caffrey (1965) finds factor-loading estimates to maximize the correlation between the factors and the complete universe of variables of interest. The basic idea in this method is that only a finite number of variables out of a much larger set of possible variables is observed. The population factors are linearly related to this larger set, while the observed factors are linearly related to the observed variables. Let f denote the factors obtainable from a finite set of observed random variables, and let ξ denote the factors obtainable from the universe of observable variables. Then, the alpha method attempts to find factor-loading estimates so as to maximize the correlation between f and ξ. In order to obtain these estimates, the iterative algorithm of Kaiser and Caffrey (1965) is used.

Rotation Methods

The `IMSLS_ORTHOMAX_ROTATION` optional argument performs an orthogonal rotation according to an orthomax criterion. In this analytic method of rotation, the criterion function

$$Q = \sum_i \sum_r \lambda_{ir}^4 - \frac{\gamma}{p} \sum_r \left[\sum_i \lambda_{ir}^2 \right]^2$$

is minimized by finding an orthogonal rotation matrix T such that $(\lambda_{ij}) = \Lambda = AT$ where A is the matrix of unrotated factor loadings. Here, $\gamma \geq 0$ is a user-specified constant (W) yielding a family of rotations, and p is the number of variables.

Kaiser (row) normalization can be performed on the factor loadings prior to rotation by specifying the parameter `norm =1`. In Kaiser normalization, the rows of `A` are first "normalized" by dividing each row by the square root of the sum of its squared elements (Harman 1976). After the rotation is complete, each row of `b` is "denormalized" by multiplication by its initial normalizing constant.

The method for optimizing Q proceeds by accumulating simple rotations where a simple rotation is defined to be one in which Q is optimized for two columns in Λ and for which the requirement that T be orthogonal is satisfied. A single iteration is defined to be such that each of the `n_factors`(`n_factors` – 1)/2 possible simple rotations is performed where `n_factors` is the number of factors. When the relative change in Q from one iteration to the next is less than `EPS` (the user-specified convergence criterion), the algorithm stops. `eps` = 0.0001 is usually sufficient. Alternatively, the algorithm stops when the user-specified maximum number of iterations, `max_iterations`, is reached. `max_iterations` = 30 is usually sufficient.

The parameter in the rotation, γ, is used to provide a family of rotations. When $\gamma = 0.0$, a direct quartimax rotation results. Other values of γ yield other rotations.

The `IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION` optional argument performs orthogonal Procrustes rotation according to a method proposed by Schöneman (1966). Let k = `n_factors` denote the number of factors, p = `n_variables` denote the number of variables, A denote the $p \times k$ matrix of unrotated factor loadings, T denote the $k \times k$ orthogonal rotation matrix (orthogonality requires that $T^T T$ be a $k \times k$ identity matrix), and let X denote the target matrix. The basic idea in orthogonal Procrustes rotation is to find an orthogonal rotation matrix T such that $B = AT$ and T provides a least-squares fit between the target matrix X and the rotated loading matrix B. Schöneman's algorithm proceeds by finding the singular value decomposition of the matrix $A^T X = U\Sigma V^T$. The rotation matrix is computed as $T = UV^T$.

The `IMSLS_DIRECT_OBLIMIN_ROTATION` optional argument performs direct oblimin rotation. In this analytic method of rotation, the criterion function

$$Q = \sum_{r \neq s} \left[\sum_i \lambda_{ir}^2 \lambda_{is}^2 - \frac{\gamma}{p} \sum_i \lambda_{ir}^2 \sum_i \lambda_{is}^2 \right]$$

is minimized by finding a rotation matrix T such that $(\lambda_{ir}) = \Lambda = AT$ and $(T^T T)^{-1}$ is a correlation matrix. Here, $\gamma \le 0$ is a user-specified constant (`w`) yielding a family of rotations, and p is the number of variables. The rotation is said to be direct because it minimizes Q with respect to the factor loadings directly, ignoring the reference structure.

Kaiser normalization can be performed on the factor loadings prior to rotation via the parameter `norm`. In Kaiser normalization (see Harman 1976), the rows of the factor loading matrix are first "normalized" by dividing each row by the square root of the sum of its squared elements. After the rotation is complete, each row of `b` is "denormalized" by multiplication by its initial normalizing constant.

The method for optimizing Q is essentially the method first proposed by Jennrich and Sampson (1966). It proceeds by accumulating simple rotations where a simple rotation is defined to be one in which Q is optimized for a given factor in the plane of a second factor, and for which the requirement that $(T^T T)^{-1}$ be a correlation matrix is satisfied. An iteration is defined to be such that each of the `n_factors`[`n_factors` – 1] possible simple rotations is performed, where n_factors is the number of factors. When the relative change in Q from one iteration to the next is less than eps (the user-specified convergence criterion), the algorithm stops. eps = .0001 is usually sufficient. Alternatively, the algorithm stops when the user-specified maximum number of iterations, max_iterations, is reached. max_iterations = 30 is usually sufficient.

The parameter in the rotation, γ, is used to provide a family of rotations. Harman (1976) recommends that γ be strictly less than or equal to zero. When $\gamma = 0.0$, a direct quartimin rotation results. Other values of γ yield other rotations. Harman (1976) suggests that the direct quartimin rotations yield the most highly correlated factors while more orthogonal factors result as γ approaches $-\infty$.

IMSLS_OBLIQUE_PROMAX_ROTATION,
IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION,
IMSLS_OBLIQUE_PROCRUSTES_ROTATION, optional arguments performs oblique rotations using the Promax, pivotal Promax, or oblique Procrustes methods. In all of these methods, a target matrix X is first either computed or specified by the user. The differences in the methods relate to how the target matrix is first obtained.

Given a $p \times k$ target matrix, X, and a $p \times k$ orthogonal matrix of unrotated factor loadings, A, compute the rotation matrix T as follows: First regress each column of A on X yielding a $k \times k$ matrix β. Then, let $\gamma = \mathrm{diag}(\beta^T \beta)$ where diag denotes the diagonal matrix obtained from the diagonal of the square matrix. Standardize β to obtain
$T = \gamma^{-1/2} \beta$. The rotated loadings are computed as $B = AT$ while the factor correlations can be computed as the inverse of the $T^T T$ matrix.

In the Promax method, the unrotated factor loadings are first rotated according to an orthomax criterion via optional argument IMSLS_ORTHOMAX_ROTATION . The target matrix X is taken as the elements of the B raised to a power greater than one but retaining the same sign as the original loadings. The column i of the rotated matrix B is raised to the power power[*i*]. A power of four is commonly used. Generally, the larger the power, the more oblique the solution.

In the pivotal Promax method, the unrotated matrix is first rotated to an orthomax orthogonal solution as in the Promax case. Then, rather than raising the i-th column in B to the power pivot[*i*], the elements x_{ij} of X are obtained from the elements b_{ij} of B by raising the ij element of B to the power pivot[*i*]/b_{ij}. This has the effects of greatly increasing in X those elements in B that are greater in magnitude than the pivot elements pivot[*i*], and of greatly decreasing those elements that are less than pivot[*i*].

In the oblique Procrustes method, the elements of X are specified by the user as input to the routine via the target argument. No orthogonal rotation is performed in the oblique Procrustes method.

Factor Structure and Variance

The IMSLS_FACTOR_STRUCTURE optional argument computes the factor structure matrix (the matrix of correlations between the observed variables and the hypothesized factors) and the variance explained by each of the factors (for orthogonal rotations).

For oblique rotations, `IMSLS_FACTOR_STRUCTURE` computes a measure of the importance of the factors, the sum of the squared elements in each column.

Let Δ denote the diagonal matrix containing the elements of the variance of the original data along its diagonal. The estimated factor structure matrix S is computed as

$$S = \Delta^{-\frac{1}{2}} A (T^{-1})^T$$

while the elements of `fvar` are computed as the diagonal elements of

$$S^T \Delta^{\frac{1}{2}} A T$$

If the factors were obtained from a correlation matrix (or the factor variances for standardized variables are desired), then the variances should all be 1.0.

Comments

1. Function `imsls_f_factor_analysis` makes no attempt to solve for `n_factors`. In general, if `n_factors` is not known in advance, several different values of `n_factors` should be used and the most reasonable value kept in the final solution.
2. Iterative methods are generally thought to be superior from a theoretical point of view, but in practice, often lead to solutions that differ little from the noniterative methods. For this reason, it is usually suggested that a noniterative method be used in the initial stages of the factor analysis and that the iterative methods be used when issues such as the number of factors have been resolved.
3. Initial estimates for the unique variances can be input. If the iterative methods fail for these values, new initial estimates should be tried. These can be obtained by use of another factoring method. (Use the final estimates from the new method as the initial estimates in the old method.)

Examples

Example 1

In this example, factor analysis is performed for a nine-variable matrix using the default method of unweighted least squares.

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

main()
{
#define N_VARIABLES 9
#define N_FACTORS   3
    float *a;

    float covariances[N_VARIABLES][N_VARIABLES] = {
        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,   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};

                        /* Perform analysis */
    a = imsls_f_factor_analysis (9, covariances, 3, 0);

                        /* Print results */
    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
        a, 0);

    free(a);
}
```

Output

```
         Unrotated Loadings
            1           2           3
1      0.7018     -0.2316      0.0796
2      0.7200     -0.1372     -0.2082
3      0.5351     -0.2144     -0.2271
4      0.7907      0.4050      0.0070
5      0.6532      0.4221     -0.1046
6      0.7539      0.4842      0.1607
7      0.7127     -0.2819     -0.0701
8      0.4835     -0.2627      0.4620
9      0.8192     -0.3137     -0.0199
```

Example 2

The following data were originally analyzed by Emmett (1949). There are 211 observations on 9 variables. Following Lawley and Maxwell (1971), three factors are obtained by the method of maximum likelihood.

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

main()
{
#define N_VARIABLES 9
#define N_FACTORS   3
    float *a;
    float *evals;
    float chi_squared, p_value, reliability_coef, function_min;
    int   chi_squared_df, n_iterations;
    float uniq[N_VARIABLES];

    float covariances[N_VARIABLES][N_VARIABLES] = {
        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,   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};

                           /* Perform analysis */
    a = imsls_f_factor_analysis (9, covariances, 3,
        IMSLS_MAXIMUM_LIKELIHOOD,          210,
        IMSLS_SWITCH_EXACT_HESSIAN,        0.01,
        IMSLS_CONVERGENCE_EPS,             0.000001,
        IMSLS_MAX_ITERATIONS,              30,
        IMSLS_MAX_STEPS_LINE_SEARCH,       10,
        IMSLS_EIGENVALUES,                 &evals,
        IMSLS_UNIQUE_VARIANCES_OUTPUT,     uniq,
        IMSLS_CHI_SQUARED_TEST,
            &chi_squared_df,
            &chi_squared,
            &p_value,
        IMSLS_TUCKER_RELIABILITY_COEFFICIENT, &reliability_coef,
        IMSLS_N_ITERATIONS,                &n_iterations,
        IMSLS_FUNCTION_MIN,                &function_min,
        0);

                         /* Print results */
    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
        a, 0);
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, evals, 0);
    imsls_f_write_matrix("Unique Error Variances", 1, N_VARIABLES,
        uniq, 0);
    printf("\n\nchi_squared_df =    %d\n", chi_squared_df);
    printf("chi_squared =       %f\n", chi_squared);
    printf("p_value =           %f\n\n", p_value);
    printf("reliability_coef = %f\n", reliability_coef);
    printf("function_min =      %f\n", function_min);
    printf("n_iterations =      %d\n", n_iterations);

    free(evals);
    free(a);
}
```

Output

```
         Unrotated Loadings
           1           2           3
1      0.6642     -0.3209      0.0735
2      0.6888     -0.2471     -0.1933
3      0.4926     -0.3022     -0.2224
4      0.8372      0.2924     -0.0354
5      0.7050      0.3148     -0.1528
6      0.8187      0.3767      0.1045
7      0.6615     -0.3960     -0.0777
8      0.4579     -0.2955      0.4913
9      0.7657     -0.4274     -0.0117
```

```
                         Eigenvalues
         1           2           3           4           5           6
     0.063       0.229       0.541       0.865       0.894       0.974

         7           8           9
     1.080       1.117       1.140

                   Unique Error Variances
         1           2           3           4           5           6
    0.4505      0.4271      0.6166      0.2123      0.3805      0.1769

         7           8           9
    0.3995      0.4615      0.2309


chi_squared_df =    12
chi_squared =       7.149356
p_value =           0.847588

reliability_coef = 1.000000
function_min =      0.035017
n_iterations =      5
```

Example 3

This example is a continuation of example 1 and illustrates the use of the `IMSLS_FACTOR_STRUCTURE` optional argument when the structure and an index of factor importance for obliquely rotated loadings are desired. A direct oblimin rotation is used to compute the factors, derived from nine variables and using $\gamma = -1$. Note in this example that the elements of `fvar` are not variances since the rotation is oblique.

```
       #include <stdio.h>
       #include <imsls.h>
       #include <stdlib.h>
       void main()
       {
       #define N_VARIABLES 9
       #define N_FACTORS   3
           float *a;
           float w= -1.0;
           int   norm=1;
           float *b, *t, *fcor;
           float *s, *fvar;
           float covariances[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,   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};

                           /* Perform analysis */
    a = imsls_f_factor_analysis (9, (float *)covariances, 3,
        IMSLS_MAXIMUM_LIKELIHOOD,           210,
        IMSLS_SWITCH_EXACT_HESSIAN,         0.01,
        IMSLS_CONVERGENCE_EPS,              0.00001,
        IMSLS_MAX_ITERATIONS,               30,
        IMSLS_MAX_STEPS_LINE_SEARCH,        10,
        IMSLS_DIRECT_OBLIMIN_ROTATION, w, norm, &b, &t, &fcor,
        IMSLS_FACTOR_STRUCTURE, &s, &fvar,
        0);

                         /* Print results */

    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
        a, 0);
    imsls_f_write_matrix("Rotated Loadings", N_VARIABLES, N_FACTORS,
        b, 0);
    imsls_f_write_matrix("Transformation Matrix", N_FACTORS, N_FACTORS,
        t, 0);
    imsls_f_write_matrix("Factor Correlation Matrix", N_FACTORS, N_FACTORS,
        fcor, 0);
    imsls_f_write_matrix("Factor Structure",  N_VARIABLES,
        N_FACTORS,s,0);
    imsls_f_write_matrix("Factor Variance", 1, N_FACTORS, fvar, 0);
}
```

Output

```
         Unrotated Loadings
            1           2           3
1      0.6642     -0.3209      0.0735
2      0.6888     -0.2471     -0.1933
3      0.4926     -0.3022     -0.2224
4      0.8372      0.2924     -0.0354
5      0.7050      0.3148     -0.1528
6      0.8187      0.3767      0.1045
7      0.6615     -0.3960     -0.0777
8      0.4579     -0.2955      0.4913
9      0.7657     -0.4274     -0.0117

          Rotated Loadings
            1           2           3
1      0.1128     -0.5144      0.2917
2      0.1847     -0.6602     -0.0018
3      0.0128     -0.6354     -0.0585
4      0.7797     -0.1751      0.0598
5      0.7147     -0.1813     -0.0959
6      0.8520      0.0039      0.1820
7      0.0354     -0.6844      0.1510
8      0.0276     -0.0941      0.6824
```

```
9      0.0729     -0.7100      0.2493

        Transformation Matrix
            1           2           3
1        0.611      -0.462       0.203
2        0.923       0.813      -0.249
3        0.042       0.728       1.050

      Factor Correlation Matrix
            1           2           3
1        1.000      -0.427       0.217
2       -0.427       1.000      -0.411
3        0.217      -0.411       1.000

          Factor Structure
            1           2           3
1       0.3958     -0.6824      0.5275
2       0.4662     -0.7383      0.3094
3       0.2714     -0.6169      0.2052
4       0.8675     -0.5326      0.3011
5       0.7713     -0.4471      0.1339
6       0.8899     -0.4347      0.3656
7       0.3605     -0.7616      0.4398
8       0.2161     -0.3861      0.7271
9       0.4302     -0.8435      0.5568

          Factor Variance
         1           2           3
     2.170       2.560       0.914
```

Warning Errors

`IMSLS_VARIANCES_INPUT_IGNORED`	When using the `IMSLS_PRINCIPAL_COMPONENT` option, the unique variances are assumed to be zero. Input for `IMSLS_UNIQUE_VARIANCES_INPUT` is ignored.
`IMSLS_TOO_MANY_ITERATIONS`	Too many iterations. Convergence is assumed.
`IMSLS_NO_DEG_FREEDOM`	There are no degrees of freedom for the significance testing.
`IMSLS_TOO_MANY_HALVINGS`	Too many step halvings. Convergence is assumed.
`IMSLS_NO_ROTATION`	`n_factors = 1`. No rotation is possible.
`IMSLS_SVD_ERROR`	An error occurred in the singular value decomposition of tran(A)*X. The rotation matrix, T, may not be correct.

Fatal Errors

`IMSLS_HESSIAN_NOT_POS_DEF`	The approximate Hessian is not semi-definite on iteration #. The computations cannot proceed. Try using different initial estimates.
`IMSLS_FACTOR_EVAL_NOT_POS`	"eigenvalues[#]" = #. An eigenvalue corresponding to a factor is negative or zero. Either use different initial estimates for "unique_variances" or reduce the number of factors.
`IMSLS_COV_NOT_POS_DEF`	"covariances" is not positive semi-definite. The computations cannot proceed.
`IMSLS_COV_IS_SINGULAR`	The matrix "covariances" is singular. The computations cannot continue because variable # is linearly related to the remaining variables.
`IMSLS_COV_EVAL_ERROR`	An error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check "covariances."
`IMSLS_ALPHA_FACTOR_EVAL_NEG`	In alpha factor analysis on iteration #, eigenvalue # is #. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced or new initial estimates for "unique_variances" must be given.
`IMSLS_RANK_LESS_THAN`	The rank of TRAN(A)*`target` = #. This must be greater than or equal to `n_factors` = #.

discriminant_analysis

Performs a linear or a quadratic discriminant function analysis among several known groups.

Synopsis

#include `<imsls.h>`

void `imsls_f_discriminant_analysis` (*int* `n_rows`, *int* `n_variables`, *float* `*x`, *int* `n_groups`, ..., 0)

The type *double* function is `imsls_d_discriminant_analysis`.

Required Arguments

int `n_rows` (Input)
Number of rows of `x` to be processed.

int `n_variables` (Input)
Number of variables to be used in the discrimination.

float `*x` (Input)

Array of size `n_rows` by `n_variables` + 1 containing the data. The first `n_variables` columns correspond to the variables, and the last column (column `n_variables`) contains the group numbers. The groups must be numbered 1, 2, ..., `n_groups`.

int `n_groups` (Input)

Number of groups in the data.

Synopsis with Optional Arguments

```
#include <imsls.h>

void imsls_f_discriminant_analysis (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_METHOD, int method,
        IMSLS_IDO, int ido,
        IMSLS_ROWS_ADD,
        IMSLS_ROWS_DELETE,
        IMSLS_PRIOR_EQUAL,
        IMSLS_PRIOR_PROPORTIONAL,
        IMSLS_PRIOR_INPUT, float prior_input[],
        IMSLS_PRIOR_OUTPUT, float **prior_output
        IMSLS_PRIOR_OUTPUT_USER, float prior_output[]
        IMSLS_GROUP_COUNTS, int **gcounts,
        IMSLS_GROUP_COUNTS_USER, int gcounts[]
        IMSLS_MEANS, float **means,
        IMSLS_MEANS_USER, float means[],
        IMSLS_COV, float **covariances,
        IMSLS_COV_USER, float covariances[],
        IMSLS_COEF, float **coefficients
        IMSLS_COEF_USER, float coefficients[],
        IMSLS_CLASS_MEMBERSHIP, int **class_membership,
        IMSLS_CLASS_MEMBERSHIP_USER, int class_membership[],
        IMSLS_CLASS_TABLE, float **class_table,
        IMSLS_CLASS_TABLE_USER, float class_table[],
        IMSLS_PROB, float **prob,
        IMSLS_PROB_USER, float prob[],
        IMSLS_MAHALANOBIS, float **d2,
        IMSLS_MAHALANOBIS_USER, float d2[],
        IMSLS_STATS, float **stats,
        IMSLS_STATS_USER, float stats[],
        IMSLS_N_ROWS_MISSING, int *nrmiss,
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of array `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_METHOD`, *int* `method` (Input)
Method of discrimination. The method chosen determines whether linear or quadratic discrimination is used, whether the group covariance matrices are computed (the pooled covariance matrix is always computed), and whether the leaving-out-one or the reclassification method is used to classify each observation.

method	discrimination method	covariances computed	classification method
1	linear	pooled, group	Reclassification
2	quadratic	pooled, group	Reclassification
3	linear	pooled	Reclassification
4	linear	pooled, group	leaving-out-one
5	quadratic	pooled, group	leaving-out-one
6	linear	pooled	leaving-out-one

In the leaving-out-one method of classification, the posterior probabilities are adjusted so as to eliminate the effect of the observation from the sample statistics prior to its classification. In the classification method, the effect of the observation is not eliminated from the classification function.

When optional argument `IMSLS_IDO` is specified, the following rules for mixing methods apply; Methods 1, 2, 4, and 5 can be intermixed, as can methods 3 and 6. Methods 1, 2, 4, and 5 *cannot* be intermixed with methods 3 and 6.

Default: `method` = 1

`IMSLS_IDO`, *int* `ido` (Input)
Processing option. See Comments 3 and 4 for more information.

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 discriminant functions and other statistics are computed.
4	The discriminant functions are used to classify each of the `n_rows` observations of `x`.
5	The covariance matrices are computed, and workspace is released. No further call to `discriminant_analysis` with `ido` greater than 1 should be made without first calling `discriminant_analysis` with `ido` = 1.
6	Workspace is released. No further calls to `discriminant_analysis` with `ido` greater than 1 should be made without first calling `discriminant_analysis` with `ido` = 1. Invocation with this option is not required if a call has already been made with `ido` = 5.

Default: `ido` = 0

`IMSLS_ROWS_ADD`, *or*
`IMSLS_ROWS_DELETE` (Input)
By default (or if `IMSLS_ROWS_ADD` is specified), then 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_PRIOR_EQUAL`, *or*
`IMSLS_PRIOR_PROPORTIONAL`, *or*
`IMSLS_PRIOR_INPUT`, *float* `prior_input[]` (Input)
By default, (or if `IMSLS_PRIOR_EQUAL` is specified), equal prior probabilities are calculated as 1.0/`n_groups`.

If `IMSLS_PRIOR_PROPORTIONAL` is specified, prior probabilities are calculated to be proportional to the sample size in each group.

If `IMSLS_PRIOR_INPUT` is specified, then array `prior_input` is an array of length `n_groups` containing the prior probabilities for each group, such that the sum of all prior probabilities is equal to 1.0. Prior probabilities are not used if `ido` is equal to 1, 2, 5, or 6.

`IMSLS_PRIOR_OUTPUT`, *float* `**prior_output` (Output)
Address of a pointer to an array of length `n_groups` containing the most recently calculated or input prior probabilities. If `IMSLS_PRIOR_PROPORTIONAL` is specified, every element of `prior_output` is equal to −1 until a call is made with `ido` equal to 0 or 3, at which point the priors are calculated. Note that subsequent calls to `discriminant_analysis` with `IMSLS_PRIOR_PROPORTIONAL` specified, and ido not equal to 0 or 3 will result in the elements of `prior_output` being reset to −1.

`IMSLS_PRIOR_OUTPUT_USER`, *float* `prior_output[]` (Output)
Storage for array `prior_output` is provided by the user. See `IMSLS_PRIOR_OUTPUT`.

`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_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. Array means is updated when `ido` is equal to 0, 1, 2, or 5. The means are *unscaled* until a call is made with `ido` = 5. where the unscaled means are calculated as $\Sigma w_i f_i x_i$ and the scaled means as

$$\frac{\sum w_i f_i x_i}{\sum w_i f_i}$$

where x_i is the value of the *i*-th observation, w_i is the weight of the *i*-th observation, and f_i is the frequency of the *i*-th observation.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
Storage for array `means` is provided by the user. See `IMSLS_MEANS`.

`IMSLS_COV`, *float* `**covariances` (Output)
Address of a pointer to an array of size `g` by `n` variables by `n_variables` containing the within-group covariance matrices (`methods` 1, 2, 4, and 5 only) as the first *g*-1 matrices, and the pooled covariance matrix as the *g*-th matrix (that is, the first `n_variables` * `n_variables` elements comprise the group 1 covariance matrix, the next `n_variables` * `n_variables` elements comprise the group 2 covariance, ..., and the last `n_variables` * `n_variables` elements comprise the pooled covariance matrix). If `method` is 3 or 6 then *g* is equal to 1. Otherwise*, g* is equal to `n_groups` + 1. Argument `cov` is updated when `ido` is equal to 0, 1, 2, 3, or 5.

IMSLS_COV_USER, *float* covariances[] (Output)
Storage for array covariances is provided by the user. See IMSLS_COVARIANCES.

IMSLS_COEF, *float* **coefficients (Output)
Address of a pointer to an array of size n_groups by (n_variables + 1) containing the linear discriminant coefficients. The first column of coefficients contains the constant term, and the remaining columns contain the variable coefficients. Row $i - 1$ of coefficients corresponds to group i, for
$i = 1, 2, ...,$ n_variables + 1. Array coefficients are always computed as the linear discriminant function coefficients even when quadratic discrimination is specified.

Array coefficients is updated when ido is equal to 0 or 3.

IMSLS_COEF_USER, *float* coefficients[] (Output)
Storage for array coefficients is provided by the user. See IMSLS_COEFFICIENTS.

IMSLS_CLASS_MEMBERSHIP, *int* **class_membership (Output)
Address of a pointer to an integer array of length n_rows containing the group to which the observation was classified. Array class_membership is updated when ido is equal to 0 or 4.

If an observation has an invalid group number, frequency, or weight when the leaving-out-one method has been specified, then the observation is not classified and the corresponding elements of class_membership (and prob, see IMSLS_PROB) are set to zero.

IMSLS_CLASS_MEMBERSHIP_USER, *int* class_membership[] (Ouput)
Storage for array class_membership is provided by the user. See IMSLS_CLASS_MEMBERSHIP.

IMSLS_CLASS_TABLE, *float* **class_table (Output)
Address of a pointer to an array of size n_groups by n_groups containing the classification table. Array class_table is updated when ido is equal to 0, 1, or 4. Each observation that is classified and has a group number 1.0, 2.0, ..., n_groups is entered into the table. The rows of the table correspond to the known group membership. The columns refer to the group to which the observation was classified. Classification results accumulate with each call to imsls_f_discriminant_analysis with ido equal to 4. For example, if two calls with ido equal to 4 are made, the elements in class_table sum to the total number of valid observations in the two calls.

IMSLS_CLASS_TABLE_USER, *float* class_table[] (Output)
Storage for array class_table is provided by the user. See IMSLS_CLASS_TABLE.

IMSLS_PROB, *float* **prob (Output)
Address of a pointer to an array of size n_rows by n_groups containing the

posterior probabilities for each observation. Argument prob is updated when ido is equal to 0 or 4.

IMSLS_PROB_USER, *float* prob[] (Output)
Storage for array prob is provided by the user. See IMSLS_PROB.

IMSLS_MAHALANOBIS, *float* **d2 (Output)
Address of a pointer to an array of size n_groups by n_groups containing the Mahalanobis distances

$$D_{ij}^2$$

between the group means. Argument d2 is updated when ido is equal to 0 or 3.

For linear discrimination, the Mahalanobis distance is computed using the pooled covariance matrix. Otherwise, the Mahalanobis distance

$$D_{ij}^2$$

between group means i and j is computed using the within covariance matrix for group i in place of the pooled covariance matrix.

IMSLS_MAHALANOBIS_USER, *float* d2[] (Output)
Storage for array d2 is provided by the user. See IMSLS_MAHALANOBIS.

IMSLS_STATS, *float* **stats (Output)
Address of a pointer to an array of length 4 + 2 × (n_groups + 1) containing various statistics of interest. Array stats is updated when ido is equal to 0, 1, 3, or 5. The first element of stats is the sum of the degrees of freedom for the within-covariance matrices. The second, third, and fourth elements of stats correspond to the chi-squared statistic, its degrees of freedom, and the probability of a greater
chi-squared, respectively, of a test of the homogeneity of the within-covariance matrices (not computed if method is equal to 3 or 6). The fifth through 5 + n_groups elements of stats contain the log of the determinants of each group's covariance matrix (not computed if method is equal to 3 or 6) and of the pooled covariance matrix (element 4 + n_groups). Finally, the last n_groups + 1 elements of stats contain the sum of the weights within each group, and in the last position, the sum of the weights in all groups.

IMSLS_STATS_USER, *float* stats[] (Output)
Storage for array stats is provided by the user. See IMSLS_STATS_USER.

IMSLS_N_ROWS_MISSING, *int* *nrmiss (Output)
Number of rows of data encountered in calls to discriminant_analysis containing missing values (NaN) for the classification, group, weight, and/or frequency variables. If a row of data contains a missing value (NaN) for any of these variables, that row is excluded from the computations.

Array nrmiss is updated when ido is equal to 0, 1, 2, or 3.

Comments

1. Common choices for the Bayesian prior probabilities are given by:
   prior_input[i] = 1.0/n_groups (equal priors)
   prior_input[i] = gcounts/n_rows (proportional priors)
   prior_input[i] = Past history or subjective judgment.
   In all cases, the priors should sum to 1.0.

2. Two passes of the data are made. In the first pass, the statistics required to compute the discriminant functions are obtained (ido equal to 1, 2, and 3). In the second pass, the discriminant functions are used to classify the observations. When ido is equal to 0, all of the data are memory resident, and both passes are made in one call to imsls_f_discriminant_analysis. When ido > 0 (optional argument IMSLS_IDO is specified), a third call to imsls_f_discriminant_analysis involving no data is required with ido equal to 5 or 6.

3. Here are a few rules and guidelines for the correct value of ido in a series of calls:

   1 Calls with ido = 0 or ido = 1 may be made at any time, subject to rule 2. These calls indicate that a new analysis is to begin, and therefore allocate memory and destroy all statistics from previous calls.

   2 Each series of calls to imsls_f_discriminant_analysis which begins with ido = 1 must end with ido equal to 5 or 6 to ensure the proper release of workspace, subject to rule 3.

   3 ido may not be 4 or 5 before a call with ido = 3 has been made.

   4 ido may not be 2, 3, 4, 5, or 6
   a) Immediately after a call with ido = 0.
   b) Before a call with ido = 1 has been made.
   c) Immediately after a call with ido equal to 5 or 6 has been made.

The following is a valid sequence of ido's:

ido	Explanation
0	Data Set A: Perform a complete analysis. All data to be used in the analysis must be present in x. Since cleanup of workspace is automatic for ido = 0, no further calls are necessary.
1	Data Set B: Begin analysis. The n_rows observations in x are used for initialization.
2	Data Set B: Continue analysis. New observations placed in x are added to (or deleted from, see IMSLS_ROWS_DELETE) the analysis.

ido	Explanation
2	Data Set B: Continue analysis. `n_rows` new observations placed in `x` are added to (or deleted from, see `IMSLS_ROWS_DELETE`) the analysis.
3	Data Set B: Continue analysis. `n_rows` new observations are added (or deleted) and discriminant functions and other statistics are computed.
4	Data Set B: Classification of each of the `n_rows` observations in the current `x` matrix.
5	Data Set B: End analysis. Covariance matrices are computed and workspace is released. This analysis could also have been ended by choosing `ido` = 6
1	Data Set C: Begin analysis. Note that for this call to be valid the previous call must have been made with `ido` equal to 5 or 6.
3	Data Set C: Continue analysis.
4	Data Set C: Continue analysis.
3	Data Set C: Continue analysis.
6	Data Set C: End analysis.

4. Because of the internal workspace allocation and saved variables, function `imsls_f_discriminant_analysis` must complete the analysis of a data set before beginning processing of the next data set.

Return Value

The return value is void.

Description

Function `imsls_f_discriminant_analysis` performs discriminant function analysis using either linear or quadratic discrimination. The output includes a measure of distance between the groups, a table summarizing the classification results, a matrix containing the posterior probabilities of group membership for each observation, and the within-sample means and covariance matrices. The linear discriminant function coefficients are also computed.

By default (or if optional argument `IMSLS_IDO` is specified with `ido` = 0) all observations are input during one call, a method of operation that has the advantage of simplicity. Alternatively, one or more rows of observations can be input during separate calls. This method does not require that all observations be memory resident, a significant advantage with large data sets. Note, however, that the algorithm requires two passes of the data. During the first pass the discriminant functions are computed while in the second pass, the observations are classified. Thus, with the second method of operation, the data will usually need to be input twice.

Because both methods result in the same operations being performed, the algorithm is discussed as if only a few observations are input during each call. The operations performed during each call depend upon the `ido` parameter.

The `ido` = 1 step is the initialization step. "Private" internally allocated saved variables corresponding to `means`, `class_table`, and `covariances` are initialized to zero, and other program parameters are set (copies of these private variables are written to the corresponding output variables upon return from the function call, assuming `ido` values such that the results are to be returned). Parameters `n_rows`, `x`, and `method` can be changed from one call to the next *within* the two sets {1, 2, 4, 5} and {3, 6} but not *between* these sets when `ido` > 1. That is, do not specify `method` = 1 in one call and `method` = 3 in another call without first making a call with `ido` = 1.

After initialization has been performed in the `ido` = 1 step, the within-group means are updated for all valid observations in `x`. Observations with invalid group numbers are ignored, as are observation with missing values. The *LU* factorization of the covariance matrices are updated by adding (or deleting) observations via Givens rotations.

The `ido` = 2 step is used solely for adding or deleting observations from the model as in the above paragraph.

The `ido` = 3 step begins by adding all observations in `x` to the means and the factorizations of the covariance matrices. It continues by computing some statistics of interest: the linear discriminant functions, the prior probabilities (by default, or if `IMSLS_PROPORTIONAL_PRIORS` is specified), the log of the determinant of each of the covariance matrices, a test statistic for testing that all of the within-group covariance matrices are equal, and a matrix of Mahalanobis distances between the groups. The matrix of Mahalanobis distances is computed via the pooled covariance matrix when linear discrimination is specified; the row covariance matrix is used when the discrimination is quadratic.

Covariance matrices are defined as follows: Let N_i denote the sum of the frequencies of the observations in group *i* and M_i denote the number of observations in group *i*. Then, if S_i denotes the within-group *i* covariance matrix,

$$S_i = \frac{1}{N_i - 1} \sum_{j=1}^{M_i} w_j f_j \left(x_j - \overline{x}\right)\left(x_j - \overline{x}\right)^T$$

Where w_j is the weight of the *j*-th observation in group *i*, f_j is the frequency, x_j is the *j*-th observation column vector (in group *i*), and $\overline{x}$ denotes the mean vector of the observations in group *i*. The mean vectors are computed as

$$\overline{x} = (\frac{1}{W_i}) \sum_{j=1}^{M_i} w_j f_j x_j \qquad \text{where } W_i = \sum_{j=1}^{M_i} w_j f_j$$

Given the means and the covariance matrices, the linear discriminant function for group *i* is computed as:

$$z_i = \ln\left(p_i\right) - 0.5\overline{x}_i^T S_p^{-1} \overline{x}_i + x^T S_p^{-1} \overline{x}_i$$

where ln (p_i) is the natural log of the prior probability for the *i*-th group, *x* is the observation to be classified, and S_p denoted the pooled covariance matrix.

Let *S* denote either the pooled covariance matrix of one of the within-group covariance matrices S_i. (*S* will be the pooled covariance matrix in linear discrimination, and S_i otherwise.) The Mahalanobis distance between group *i* and group *j* is computed as:

$$D_{ij}^2 = \left(\bar{x}_i - \bar{x}_j\right)^T S^{-1} \left(\bar{x}_i - \bar{x}_j\right)$$

Finally, the asymptotic chi-squared test for the equality of covariance matrices is computed as follows (Morrison 1976, p. 252):

$$\gamma = C^{-1} \sum_{i=1}^{k} n_i \left\{ \ln\left(\left|S_p\right|\right) - \ln\left(\left|S_i\right|\right) \right\}$$

where n_i is the number of degrees of freedom in the i-th sample covariance matrix, k is the number of groups, and

$$C^{-1} = \frac{1 - 2p^2 + 3p - 1}{6\left(p+1\right)\left(k-1\right)} \left(\sum_{i=1}^{k} \frac{1}{n_i} - \frac{1}{\sum_j n_j} \right)$$

where p is the number of variables.

When `ido` = 4, the estimated posterior probability of each observation x belonging to group is computed using the prior probabilities and the sample mean vectors and estimated covariance matrices under a multivariate normal assumption. Under quadratic discrimination, the within-group covariance matrices are used to compute the estimated posterior probabilities. The estimated posterior probability of an observation x belonging to group i is

$$\hat{q}_i\left(x\right) = \frac{\exp\left(-0.5 D_i^2\left(x\right)\right)}{\sum_{j=1}^{k} \exp\left(-0.5 D_j^2\left(x\right)\right)}$$

where

$$D_i^2\left(x\right) = \begin{cases} \left(x - \bar{x}_i\right)^T S_i^{-1} \left(x - \bar{x}_i\right) + \ln\left|S_i\right| - 2\ln\left(p_i\right) & \text{METHOD = 1 or 2} \\ \left(x - \bar{x}_i\right)^T S_p^{-1} \left(x - \bar{x}_i\right) - 2\ln\left(p_i\right) & \text{METHOD = 3} \end{cases}$$

For the leaving-out-one method of classification (`method` equal to 4, 5 or 6), the sample mean vector and sample covariance matrices in the formula for

$$D_i^2$$

are adjusted so as to remove the observation x from their computation. For linear discrimination (`method` equal to 1, 3, 4, or 6), the linear discriminant function coefficients are actually used to compute the same posterior probabilities.

Using the posterior probabilities, each observation in x is classified into a group; the result is tabulated in the matrix `class_table` and saved in the vector `class_membership`. Matrix `class_table` is not altered at this stage if x[i][x_group] (by default, x_igrp = 0; see optional argument IMSLS_INDICES) contains a group number that is out of range. If the reclassification method is specified, then all observations with no missing values in the n_variables classification variables are classified. When the leaving-out-one method is used, observations with invalid group numbers, weights, frequencies, or classification variables are not classified. Regardless of the frequency, a 1 is added (or subtracted) from class_table for each row of x that is classified and contains a valid group number.

When method > 3, adjustment is made to the posterior probabilities to remove the effect of the observation in the classification rule. In this adjustment, each observation is presumed to have a weight of x[i][iwt] if
iwt > –1 (and a weight of 1.0 if iwt = –1), and a frequency of 1.0. See Lachenbruch (1975, p. 36) for the required adjustment.

Finally, when ido = 5, the covariance matrices are computed from their *LU* factorizations. Internally allocated and saved variables are cleaned up at this step (ido equal to 5 or 6).

Example 1

The following example uses liner discrimination with equal prior probabilities on Fisher's (1936) iris data. This example illustrates the execution of imsls_f_discriminant_analysis when one call is made (i.e. using the default of ido = 0).

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

main() {
    int   n_groups = 3;
    int   nrow, nvar, ncol, nrmiss;
    float *x, *xtemp;
    float *prior_out, *means, *cov, *coef;
    float *table, *d2, *stats, *prob;
    int   *counts, *cm;
    static int perm[5] = {1, 2, 3, 4, 0};

    /* Retrieve the Fisher Iris Data Set */
    xtemp = imsls_f_data_sets(3, IMSLS_N_OBSERVATIONS, &nrow,
        IMSLS_N_VARIABLES, &ncol, 0);
    nvar = ncol - 1;

    /* Move the group column to end of the the matrix */
    x = imsls_f_permute_matrix(nrow, ncol, xtemp, perm,
        IMSLS_PERMUTE_COLUMNS, 0);
    free(xtemp);

    imsls_f_discriminant_analysis (nrow, nvar, x, n_groups,
        IMSLS_METHOD, 3,
        IMSLS_GROUP_COUNTS, &counts,
        IMSLS_COEF, &coef,
        IMSLS_MEANS, &means,
        IMSLS_STATS, &stats,
        IMSLS_CLASS_MEMBERSHIP, &cm,
        IMSLS_CLASS_TABLE, &table,
        IMSLS_PROB, &prob,
        IMSLS_MAHALANOBIS, &d2,
        IMSLS_COV, &cov,
        IMSLS_PRIOR_OUTPUT, &prior_out,
        IMSLS_N_ROWS_MISSING, &nrmiss,
        IMSLS_PRIOR_EQUAL,
        IMSLS_METHOD, 3, 0);

    imsls_i_write_matrix("Counts", 1, n_groups, counts, 0);
    imsls_f_write_matrix("Coef", n_groups, nvar+1, coef, 0);
    imsls_f_write_matrix("Means", n_groups, nvar, means, 0);
    imsls_f_write_matrix("Stats", 12, 1, stats, 0);
    imsls_i_write_matrix("Membership", 1, nrow, cm, 0);
    imsls_f_write_matrix("Table", n_groups, n_groups, table, 0);
    imsls_f_write_matrix("Prob", nrow, n_groups, prob, 0);
    imsls_f_write_matrix("D2", n_groups, n_groups, d2, 0);
    imsls_f_write_matrix("Covariance", nvar, nvar, cov, 0);
    imsls_f_write_matrix("Prior OUT", 1, n_groups, prior_out, 0);
    printf("\nnrmiss = %3d\n", nrmiss);

    free(means);
    free(stats);
    free(counts);
    free(coef);
    free(cm);
    free(table);
    free(prob);
    free(d2);
    free(prior_out);
    free(cov);
}
```

Output

```
   Counts
  1    2    3
 50   50   50

                           Coef
           1           2           3           4           5
1      -86.3        23.5        23.6       -16.4       -17.4
2      -72.9        15.7         7.1         5.2         6.4
3     -104.4        12.4         3.7        12.8        21.1

                     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

     Stats
 1         147
 2  ..........
 3  ..........
 4  ..........
 5  ..........
```

```
  6  ..........
  7  ..........
  8         -10
  9          50
 10          50
 11          50
 12         150

                               Membership
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

 41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
  1   1   1   1   1   1   1   1   1   1   2   2   2   2   2   2   2   2   2   2

 61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
  2   2   2   2   2   2   2   2   2   2   3   2   2   2   2   2   2   2   2   2

 81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
  2   2   2   3   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   3   3   3   3   3   3   3   3   3

148  149  150
  3    3    3

                Table
            1           2           3
1          50           0           0
2           0          48           2
3           0           1          49

                Prob
              1           2           3
  1       1.000       0.000       0.000
  2       1.000       0.000       0.000
  3       1.000       0.000       0.000
  4       1.000       0.000       0.000
  5       1.000       0.000       0.000
  6       1.000       0.000       0.000
  7       1.000       0.000       0.000
  8       1.000       0.000       0.000
  9       1.000       0.000       0.000
 10       1.000       0.000       0.000
 11       1.000       0.000       0.000
 12       1.000       0.000       0.000
 13       1.000       0.000       0.000
 14       1.000       0.000       0.000
 15       1.000       0.000       0.000
 16       1.000       0.000       0.000
 17       1.000       0.000       0.000
 18       1.000       0.000       0.000
 19       1.000       0.000       0.000
 20       1.000       0.000       0.000
 21       1.000       0.000       0.000
 22       1.000       0.000       0.000
 23       1.000       0.000       0.000
 24       1.000       0.000       0.000
 25       1.000       0.000       0.000
 26       1.000       0.000       0.000
 27       1.000       0.000       0.000
 28       1.000       0.000       0.000
 29       1.000       0.000       0.000
 30       1.000       0.000       0.000
 31       1.000       0.000       0.000
 32       1.000       0.000       0.000
 33       1.000       0.000       0.000
 34       1.000       0.000       0.000
 35       1.000       0.000       0.000
 36       1.000       0.000       0.000
 37       1.000       0.000       0.000
 38       1.000       0.000       0.000
 39       1.000       0.000       0.000
 40       1.000       0.000       0.000
 41       1.000       0.000       0.000
 42       1.000       0.000       0.000
 43       1.000       0.000       0.000
 44       1.000       0.000       0.000
 45       1.000       0.000       0.000
 46       1.000       0.000       0.000
 47       1.000       0.000       0.000
 48       1.000       0.000       0.000
 49       1.000       0.000       0.000
 50       1.000       0.000       0.000
 51       0.000       1.000       0.000
 52       0.000       0.999       0.001
 53       0.000       0.996       0.004
 54       0.000       1.000       0.000
 55       0.000       0.996       0.004
 56       0.000       0.999       0.001
 57       0.000       0.986       0.014
 58       0.000       1.000       0.000
 59       0.000       1.000       0.000
 60       0.000       1.000       0.000
 61       0.000       1.000       0.000
 62       0.000       0.999       0.001
 63       0.000       1.000       0.000
 64       0.000       0.994       0.006
 65       0.000       1.000       0.000
 66       0.000       1.000       0.000
 67        0.000        0.981        0.019
 68        0.000        1.000        0.000
 69        0.000        0.960        0.040
 70        0.000        1.000        0.000
 71        0.000        0.253        0.747
 72        0.000        1.000        0.000
 73        0.000        0.816        0.184
 74        0.000        1.000        0.000
 75        0.000        1.000        0.000
 76        0.000        1.000        0.000
 77        0.000        0.998        0.002
 78        0.000        0.689        0.311
 79        0.000        0.993        0.007
 80        0.000        1.000        0.000
 81        0.000        1.000        0.000
 82        0.000        1.000        0.000
 83        0.000        1.000        0.000
 84        0.000        0.143        0.857
 85        0.000        0.964        0.036
 86        0.000        0.994        0.006
 87        0.000        0.998        0.002
 88        0.000        0.999        0.001
 89        0.000        1.000        0.000
 90        0.000        1.000        0.000
 91        0.000        0.999        0.001
 92        0.000        0.998        0.002
 93        0.000        1.000        0.000
 94        0.000        1.000        0.000
 95        0.000        1.000        0.000
 96        0.000        1.000        0.000
 97        0.000        1.000        0.000
 98        0.000        1.000        0.000
 99        0.000        1.000        0.000
100        0.000        1.000        0.000
101        0.000        0.000        1.000
102        0.000        0.001        0.999
103        0.000        0.000        1.000
104        0.000        0.001        0.999
105        0.000        0.000        1.000
106        0.000        0.000        1.000
107        0.000        0.049        0.951
108        0.000        0.000        1.000
109        0.000        0.000        1.000
110        0.000        0.000        1.000
111        0.000        0.013        0.987
112        0.000        0.002        0.998
113        0.000        0.000        1.000
114        0.000        0.000        1.000
115        0.000        0.000        1.000
116        0.000        0.000        1.000
117        0.000        0.006        0.994
118        0.000        0.000        1.000
119        0.000        0.000        1.000
120        0.000        0.221        0.779
121        0.000        0.000        1.000
```

```
122       0.000       0.001       0.999
123       0.000       0.000       1.000
124       0.000       0.097       0.903
125       0.000       0.000       1.000
126       0.000       0.003       0.997
127       0.000       0.188       0.812
128       0.000       0.134       0.866
129       0.000       0.000       1.000
130       0.000       0.104       0.896
131       0.000       0.000       1.000
132       0.000       0.001       0.999
133       0.000       0.000       1.000
134       0.000       0.729       0.271
135       0.000       0.066       0.934
136       0.000       0.000       1.000
137       0.000       0.000       1.000
138       0.000       0.006       0.994
139       0.000       0.193       0.807
140       0.000       0.001       0.999
141       0.000       0.000       1.000
142       0.000       0.000       1.000
143       0.000       0.001       0.999
144       0.000       0.000       1.000
145       0.000       0.000       1.000
146       0.000       0.000       1.000
147       0.000       0.006       0.994
148       0.000       0.003       0.997
149       0.000       0.000       1.000
150       0.000       0.018       0.982

                 D2
           1           2           3
1        0.0        89.9       179.4
2       89.9         0.0        17.2
3      179.4        17.2         0.0

                   Covariance
           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

                   Prior OUT
        1           2           3
    0.3333      0.3333      0.3333

nrmiss =   0
```

Example 2

Continuing with Fisher's iris data, the example below computes the quadratic discriminant functions using values of IDO greater than 0. In the first loop, all observations are added to the functions, one at a time. In the second loop, each of the observations is classified, one by one, using the leaving-out-one method.

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

main() {
    int   n_groups = 3;
    int   nrow, nvar, ncol, i, nrmiss;
    float *x, *xtemp;
    float *prior_out, *means, *cov, *coef;
    float *table, *d2, *stats, *prob;
    int   *counts, *cm;
    static int perm[5] = {1, 2, 3, 4, 0};

    /* Retrieve the Fisher Iris Data Set */
    xtemp = imsls_f_data_sets(3, IMSLS_N_OBSERVATIONS, &nrow,
        IMSLS_N_VARIABLES, &ncol, 0);
    nvar = ncol - 1;

    /* Move the group column to end of the the matrix */
    x = imsls_f_permute_matrix(nrow, ncol, xtemp, perm,
        IMSLS_PERMUTE_COLUMNS, 0);
    free(xtemp);

    prior_out = (float *) malloc(n_groups*sizeof(float));
    counts    = (int *)   malloc(n_groups*sizeof(int));
    means     = (float *) malloc(n_groups*nvar*sizeof(float));
    cov       = (float *) malloc(nvar*nvar*(ngroups+1)*sizeof(float));
    coef      = (float *) malloc(n_groups*(nvar+1)*sizeof(float));
    table     = (float *) malloc(n_groups*n_groups*sizeof(float));
    d2        = (float *) malloc(n_groups*n_groups*sizeof(float));
    stats     = (float *) malloc((4+2*(n_groups+1))*sizeof(float));
    cm        = (int *)   malloc(nrow*sizeof(int));
    prob      = (float *) malloc(nrow*n_groups*sizeof(float));

    /*Initialize Analysis*/
    imsls_f_discriminant_analysis (0, nvar, x, n_groups,
         IMSLS_IDO, 1,
         IMSLS_METHOD, 2, 0);

    /*Add In Each Observation*/
    for (i=0;i<nrow;i=i+1) {
      imsls_f_discriminant_analysis (1, nvar, (x+i*ncol), n_groups,
         IMSLS_IDO, 2, 0);
    }

    /*Remove observation 0 from the analysis */
    imsls_f_discriminant_analysis (1, nvar, (x+0), n_groups,
         IMSLS_ROWS_DELETE,
         IMSLS_IDO, 2, 0);

    /*Add observation 0 back into the analysis */
    imsls_f_discriminant_analysis (1, nvar, (x+0), n_groups,
         IMSLS_IDO, 2, 0);

    /*Compute statistics*/
    imsls_f_discriminant_analysis (0, nvar, x, n_groups,
         IMSLS_PRIOR_PROPORTIONAL,
         IMSLS_PRIOR_OUTPUT_USER, prior_out,
         IMSLS_IDO, 3, 0);

    imsls_f_write_matrix("Prior OUT", 1, n_groups, prior_out, 0);

    /*Classify One observation at a time, using proportional priors*/
    for (i=0;i<nrow;i=i+1) {
      imsls_f_discriminant_analysis (1, nvar, (x+i*ncol), n_groups,
         IMSLS_IDO, 4,
         IMSLS_CLASS_MEMBERSHIP_USER, (cm+i),
         IMSLS_PROB_USER, (prob+i*n_groups), 0);
    }

    /*Compute covariance matrices and release internal workspace*/
    imsls_f_discriminant_analysis (0, nvar, x, n_groups,
         IMSLS_IDO, 5,
         IMSLS_COV_USER, cov,
         IMSLS_GROUP_COUNTS_USER, counts,
         IMSLS_COEF_USER, coef,
         IMSLS_MEANS_USER, means,
         IMSLS_STATS_USER, stats,
         IMSLS_CLASS_TABLE_USER, table,
         IMSLS_MAHALANOBIS_USER, d2,
         IMSLS_N_ROWS_MISSING, &nrmiss, 0);

    imsls_i_write_matrix("Counts", 1, n_groups, counts, 0);
    imsls_f_write_matrix("Coef", n_groups, nvar+1, coef, 0);
    imsls_f_write_matrix("Means", n_groups, nvar, means, 0);
    imsls_f_write_matrix("Stats", 12, 1, stats, 0);
    imsls_i_write_matrix("Membership", 1, nrow, cm, 0);
    imsls_f_write_matrix("Table", n_groups, n_groups, table, 0);
    imsls_f_write_matrix("Prob", nrow, n_groups, prob, 0);
    imsls_f_write_matrix("D2", n_groups, n_groups, d2, 0);
    imsls_f_write_matrix("Covariance", nvar, nvar, cov, 0);
    printf("\nnrmiss = %3d\n", nrmiss);

    free(means);
    free(stats);
    free(counts);
    free(coef);
    free(cm);
    free(table);
    free(prob);
    free(d2);
    free(prior_out);
    free(cov);

}
```

Output

```
         Prior OUT
     1           2           3
0.3333      0.3333      0.3333
```

```
   Counts
  1    2    3
 50   50   50

                              Coef
            1            2            3            4            5
1       -86.3         23.5         23.6        -16.4        -17.4
2       -72.9         15.7          7.1          5.2          6.4
3      -104.4         12.4          3.7         12.8         21.1

                        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

     Stats
 1       147.0
 2       143.8
 3        20.0
 4         0.0
 5       -13.1
 6       -10.9
 7        -8.9
 8       -10.0
 9        50.0
10        50.0
11        50.0
12       150.0

                                Membership
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2

61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 2  2  2  2  2  2  2  2  2  2  3  2  2  2  2  2  2  2  2  2

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 2  2  2  3  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   3   3   3   3   3   3   3   3   3
```

```
148 149 150
  3   3   3

                Table
            1           2           3
1          50           0           0
2           0          48           2
3           0           1          49

                 Prob
              1           2           3
  1       1.000       0.000       0.000
  2       1.000       0.000       0.000
  3       1.000       0.000       0.000
  4       1.000       0.000       0.000
  5       1.000       0.000       0.000
  6       1.000       0.000       0.000
  7       1.000       0.000       0.000
  8       1.000       0.000       0.000
  9       1.000       0.000       0.000
 10       1.000       0.000       0.000
 11       1.000       0.000       0.000
 12       1.000       0.000       0.000
 13       1.000       0.000       0.000
 14       1.000       0.000       0.000
 15       1.000       0.000       0.000
 16       1.000       0.000       0.000
 17       1.000       0.000       0.000
 18       1.000       0.000       0.000
 19       1.000       0.000       0.000
 20       1.000       0.000       0.000
 21       1.000       0.000       0.000
 22       1.000       0.000       0.000
 23       1.000       0.000       0.000
 24       1.000       0.000       0.000
 25       1.000       0.000       0.000
 26       1.000       0.000       0.000
 27       1.000       0.000       0.000
 28       1.000       0.000       0.000
 29       1.000       0.000       0.000
 30       1.000       0.000       0.000
 31       1.000       0.000       0.000
 32       1.000       0.000       0.000
 33       1.000       0.000       0.000
 34       1.000       0.000       0.000
 35       1.000       0.000       0.000
 36       1.000       0.000       0.000
 37       1.000       0.000       0.000
 38       1.000       0.000       0.000
 39       1.000       0.000       0.000
 40       1.000       0.000       0.000
 41       1.000       0.000       0.000
 42       1.000       0.000       0.000
 43       1.000       0.000       0.000
 44       1.000       0.000       0.000
 45       1.000       0.000       0.000
 46       1.000       0.000       0.000
 47       1.000       0.000       0.000
 48       1.000       0.000       0.000
 49       1.000       0.000       0.000
 50       1.000       0.000       0.000
 51       0.000       1.000       0.000
 52       0.000       1.000       0.000
 53       0.000       0.998       0.002
 54       0.000       0.997       0.003
 55       0.000       0.997       0.003
 56       0.000       0.989       0.011
 57       0.000       0.995       0.005
 58       0.000       1.000       0.000
 59       0.000       1.000       0.000
 60       0.000       0.994       0.006
 61       0.000       1.000       0.000
 62       0.000       0.999       0.001
 63       0.000       1.000       0.000
 64       0.000       0.988       0.012
 65       0.000       1.000       0.000
 66       0.000       1.000       0.000
 67       0.000       0.973       0.027
 68       0.000       1.000       0.000
 69       0.000       0.813       0.187
 70       0.000       1.000       0.000
 71       0.000       0.336       0.664
 72       0.000       1.000       0.000
 73       0.000       0.699       0.301
 74       0.000       0.972       0.028
 75       0.000       1.000       0.000
 76       0.000       1.000       0.000
 77       0.000       0.998       0.002
 78       0.000       0.861       0.139
 79       0.000       0.992       0.008
 80       0.000       1.000       0.000
 81       0.000       1.000       0.000
 82       0.000       1.000       0.000
 83       0.000       1.000       0.000
 84       0.000       0.154       0.846
 85       0.000       0.943       0.057
 86       0.000       0.996       0.004
 87       0.000       0.999       0.001
 88       0.000       0.999       0.001
 89       0.000       1.000       0.000
 90       0.000       0.999       0.001
 91       0.000       0.981       0.019
 92       0.000       0.997       0.003
 93       0.000       1.000       0.000
 94       0.000       1.000       0.000
 95       0.000       0.999       0.001
 96       0.000       1.000       0.000
 97       0.000       1.000       0.000
 98       0.000       1.000       0.000
 99       0.000       1.000       0.000
100       0.000       1.000       0.000
101       0.000       0.000       1.000
102       0.000       0.000       1.000
103       0.000       0.000       1.000
104       0.000       0.006       0.994
105       0.000       0.000       1.000
106       0.000       0.000       1.000
107       0.000       0.004       0.996
108       0.000       0.000       1.000
109       0.000       0.000       1.000
110       0.000       0.000       1.000
111       0.000       0.006       0.994
112       0.000       0.001       0.999
113       0.000       0.000       1.000
114       0.000       0.000       1.000
115       0.000       0.000       1.000
116       0.000       0.000       1.000
117       0.000       0.033       0.967
118       0.000       0.000       1.000
119       0.000       0.000       1.000
120       0.000       0.041       0.959
121       0.000       0.000       1.000
122       0.000       0.000       1.000
123       0.000       0.000       1.000
124       0.000       0.028       0.972
125       0.000       0.001       0.999
126       0.000       0.007       0.993
127       0.000       0.057       0.943
128       0.000       0.151       0.849
129       0.000       0.000       1.000
130       0.000       0.020       0.980
131       0.000       0.000       1.000
132       0.000       0.009       0.991
133       0.000       0.000       1.000
134       0.000       0.605       0.395
135       0.000       0.000       1.000
136       0.000       0.000       1.000
137       0.000       0.000       1.000
138       0.000       0.050       0.950
139       0.000       0.141       0.859
140       0.000       0.000       1.000
141       0.000       0.000       1.000
142       0.000       0.000       1.000
143       0.000       0.000       1.000
144       0.000       0.000       1.000
145       0.000       0.000       1.000
146       0.000       0.000       1.000
147       0.000       0.000       1.000
148       0.000       0.001       0.999
149       0.000       0.000       1.000
150       0.000       0.061       0.939

                 D2
            1           2           3
```

```
1         0.0       323.1       706.1
2       103.2         0.0        17.9
3       168.8        13.8         0.0


                 Covariance
              1           2           3           4
1        0.1242      0.0992      0.0164      0.0103
2        0.0992      0.1437      0.0117      0.0093
3        0.0164      0.0117      0.0302      0.0061
4        0.0103      0.0093      0.0061      0.0111

nrmiss =    0
```

Warning Errors

`IMSLS_BAD_OBS_1`	In call #, row # of the data matrix, "x", has group number = #. The group number must be an integer between 1.0 and "n_groups" = #, inclusively. This observation will be ignored.
`IMSLS_BAD_OBS_2`	The leaving out one method is specified but this observation does not have a valid group number (Its group number is #.). This observation (row #) is ignored.
`IMSLS_BAD_OBS_3`	The leaving out one method is specified but this observation does not have a valid weight or it does not have a valid frequency. This observation (row #) is ignored.
`IMSLS_COV_SINGULAR_3`	The group # covariance matrix is singular. "stats[1]" cannot be computed. "stats[1]" and "stats[3]" are set to the missing value code (NaN).

Fatal Errors

`IMSLS_BAD_IDO_1`	"ido" = #. Initial allocations must be performed by making a call to discriminant_analysis with "ido" = 1.
`IMSLS_BAD_IDO_2`	"ido" = #. A new analysis may not begin until the previous analysis is terminated with "ido" equal to 5 or 6.
`IMSLS_COV_SINGULAR_1`	The variance-covariance matrix for population number # is singular. The computations cannot continue.
`IMSLS_COV_SINGULAR_2`	The pooled variance-covariance matrix is singular. The computations cannot continue.
`IMSLS_COV_SINGULAR_4`	A variance-covariance matrix is singular. The index of the first zero element is equal to #.

Chapter 10: Survival and Reliability Analysis

Routines

Survival Analysis		
Computes Kaplan-Meier estimates of survival probabilties	kaplan_meier_estimates	708
Analyzes survival and reliability data using Cox's proportional hazards model	prop_hazards_gen_lin	713
Analyzes survival data using the generalized linear model	survival_glm	727
Estimates using various parametric modes	survival_estimates	750
Reliability Analysis		
Estimates a reliability hazard function using a nonparametric approach	nonparam_hazard_rate	756
Actuarial Tables		
Produces population and cohort life tables	life_tables	764

Usage Notes

The functions described in this chapter have primary application in the areas of reliability and life testing, but they may find application in any situation in which analysis of binomial events over time is of interest. Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), Gross and Clark (1975), Lawless (1982), and Chiang (1968) and Tanner and Wong (1984) are references for discussing the models and methods desribed in this chapter.

Function `imsls_f_kaplan_meier_estimates` produces Kaplan-Meier (product-limit) estimates of the survival distribution in a single population, and these can be printed using the `IMSLS_PRINT` optional argument.

Function `imsls_f_prop_hazards_gen_lin` computes the parameter estimates in a proportional hazards model.

Function `imsls_f_survival_glm` fits any of several generalized linear models for survival data, and `imsls_f_survival_estimates` computes estimates of survival probabilities based upon the same models.

Function imsls_f_nonparam_hazard_rate performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.
Function imsls_f_life_tables computes and (optionally) prints an actuarial table based either upon a cohort followed over time or a cross-section of a population.

kaplan_meier_estimates

Computes Kaplan-Meier estimates of survival probabilities in stratified samples.

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_kaplan_meier_estimates`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `ncol` (Input)
Number of columns in `x`.

float `x[]` (Input)
Two-dimensional data array of size `n_observations*ncol`.

Return Value

Pointer to an array of length `n_observations*2`. The first column contains the estimated survival probabilities, and the second column contains Greenwood's estimate of the standard deviation of these probabilities. If the *i*-th observation contains censor codes out of range or if a variable is missing, then the corresponding elements of the return value are set to missing (NaN, not a number). Similarly, if an element in the return value is not defined, then it is set to missing.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_kaplan_meier_estimates` (*int* `n_observations`, *int* `ncol`,
float `x[]`,
`IMSLS_RETURN_USER`, *float* `table[]`,
`IMSLS_PRINT`,
`IMSLS_X_RESPONSE_COL`, *int* `irt`,
`IMSLS_CENSOR_CODES_COL`, *int* `icen`,
`IMSLS_FREQ_RESPONSE_COL_COL`, *int* `ifrq`,
`IMSLS_STRATUM_NUMBER_COL`, *int* `igrp`,
`IMSLS_SORTED`,
`IMSLS_N_MISSING`, *int* `*nrmiss`,
0)

Optional Arguments

IMSLS_RETURN_USER, *float* table[] (Output)
: User supplied storage of an array of length n_observations*2 containing the estimated survival probabilities and their associated standard deviations. See Return Value section.

IMSLS_PRINT, (Input)
: Print Kaplan-Meier estimates of survival probabilities in stratified samples.

IMSLS_X_RESPONSE_COL, *int* irt (Input)
: Column index for the response times in the data array, x. The interpretation of these times as either right-censored or exact failure times depends on IMSLS_CENSOR_CODES_COL.
Default: irt = 0.

IMSLS_CENSOR_CODES_COL, *int* icen (Input)
: Column index for the optional censoring codes in the data array, x. If x[i, icen]= 0, the failure time x[i, irt] is treated as an exact time of failure. Otherwise it is treated as a right-censored time.
Default: It is assumed that there is no censor code column in x. All observations are assumed to be exact failure times.

IMSLS_FREQ_RESPONSE_COL_COL, *int* ifrq (Input)
: Column index for the number of responses associated with each row in the data array, x.
Default: It is assumed that there is no frequency response column in x. Each observation in the data array is assumed to be for a single failure.

IMSLS_STRATUM_NUMBER_COL, *int* igrp (Input)
: Column index for the stratum number for each observation in the data array, x. Column igrp of x contains a unique value for each stratum in the data. Kaplan-Meier estimates are computed within each stratum.
Default: It is assumed that there is no stratum number column in x. The data is assumed to come from one stratum.

IMSLS_SORTED, (Input)
: If this option is used, column irt of x is assumed to be sorted in ascending order within each stratum. Otherwise, a detached sort is conducted prior to analysis. If sorting is performed, all censored individuals are assumed to follow tied failures.
Default: Column irt of x is not sorted.

IMSLS_N_MISSING, *int* *nrmiss (Output)
: Number of rows of data in x containing missing values.

Description

Function imsls_f_kaplan_meier_estimates computes Kaplan-Meier (or product-limit) estimates of survival probabilities for a sample of failure times that can be right censored or exact times. A survival probability $S(t)$ is defined as
$1 - F(t)$, where $F(t)$ is the cumulative distribution function of the failure times (t).

Greenwood's estimate of the standard errors of the survival probability estimates are also computed. (See Kalbfleisch and Prentice, 1980, pages 13 and 14.)

Let (t_i, δ_i), for i = 1,…, n denote the failure censoring times and the censoring codes for the n observations in a single sample. Here, t_i = x$_{i-1,\, irt}$ is a failure time if δ_i is 0, where δ_i = x$_{i-1,\, icen}$. Also, t_i is a right censoring time if δ_i is 1. Rows in `x` containing values other than 0 or 1 for δ_i are ignored. Let the number of observations in the sample that have not failed by time $s_{(t)}$ be denoted by $n_{(t)}$, where $s_{(t)}$ is an ordered (from smallest to largest) listing of the distinct failure times (censoring times are omitted). Then the Kaplan-Meier estimate of the survival probabilities is a step function, which in the interval from $s_{(t)}$ to $s_{(i+1)}$ (including the lower endpoint) is given by

$$\hat{S}(t) = \prod_{j=1}^{i} \left(\frac{n_{(j)} - d_{(j)}}{n_{(j)}} \right)$$

where $d_{(j)}$ denotes the number of failures occurring at time $s_{(j)}$, and $n_{(\varphi)}$ is the number of observation that have not failed prior to$s_{(j)}$.

Note that one row of X may correspond to more than one failed (or censored) observation when the frequency option is in effect (`ifrq` is specified). The Kaplan-Meier estimate of the survival probability prior to time $s_{(1)}$ is 1.0, while the Kaplan-Meier estimate of the survival probability after the last failure time is not defined.

Greenwood's estimate of the variance of

$$\hat{S}(t)$$

in the interval from $s_{(i)}$ to $s_{(i+1)}$ is given as

$$\text{est. var}(\hat{S}(t)) = \hat{S}^2(t) \sum_{j=1}^{i} \frac{d_{(j)}}{n_{(j)}(n_{(j)} - d_{(j)})}$$

Function `imsls_f_kaplan_meier_estimates` computes the single sample estimates of the survival probabilities for all samples of data included in `x` during a single call. This is accomplished through the `igrp` column of `x`, which if present, must contain a distinct code for each sample of observations. If `igrp` is not specified, there is no grouping column, and all observations are assumed to come from the same sample.

When failures and right-censored observations are tied and the data are to be sorted by `imsls_f_kaplan_meier_estimates` (`IMSLS_SORTED` optional argument is not used), `imsls_f_kaplan_meier_estimates` assumes that the time of censoring for the tied-censored observations is immediately after the tied failure (within the same sample). When the `IMSLS_SORTED` optional argument is used, the data are assumed to be sorted from smallest to largest according to column `irt` of `x` within each stratum. Furthermore, a small increment of time is assumed (theoretically) to elapse between the failed and censored observations that are tied (in the same sample). Thus, when the

IMSLS_SORTED optional argument is used, the user must sort all of the data in x from smallest to largest according to column irt (and column igrp, if present). By appropriate sorting of the observations, the user can handle censored and failed observations that are tied in any manner desired.

The IMSLS_PRINT option prints life tables. One table for each stratum is printed. In addition to the survival probabilities at each failure point, the following is also printed: the number of individuals remaining at risk, Greenwood's estimate of the standard errors for the survival probabilities, and the Kaplan-Meier log-likelihood. The Kaplan-Meier log-likelihood is computed as:

$$\ell = \sum_j d_{(j)} \ln d_{(j)} + (n_{(j)} - d_{(j)}) \ln(n_{(j)} - d_{(j)}) - n_{(j)} \ln n_{(j)}$$

where the sum is with respect to the distinct failure times $s_{(j)}$, $d_{(j)}$.

Example

The following example is taken from Kalbfleisch and Prentice (1980, page 1). The first column in *x* contains the death/censoring times for rats suffering from vaginal cancer. The second column contains information as to which of two forms of treatment were provided, while the third column contains the censoring code. Finally, the fourth column contains the frequency of each observation. The product-limit estimates of the survival probabilities are computed for both groups with one call to imsls_f_kaplan_meier_estimates.

Function imsls_f_kaplan_meier_estimates could have been called with the IMSLS_SORTED optional argument if the censored observations had been sorted with respect to the failure time variable. IMSLS_PRINT option is used to print the life tables.

```
#include "imsls.h"

void main ()
{
  int icen = 2, ifrq = 3, igrp = 1, ncol = 4, n_observations = 33;
  float x[] = {
    143, 5, 0, 1,
    164, 5, 0, 1,
    188, 5, 0, 2,
    190, 5, 0, 1,
    192, 5, 0, 1,
    206, 5, 0, 1,
    209, 5, 0, 1,
    213, 5, 0, 1,
    216, 5, 0, 1,
    220, 5, 0, 1,
    227, 5, 0, 1,
    230, 5, 0, 1,
    234, 5, 0, 1,
    246, 5, 0, 1,
    265, 5, 0, 1,
    304, 5, 0, 1,
    216, 5, 1, 1,
    244, 5, 1, 1,
    142, 7, 0, 1,
    156, 7, 0, 1,
    163, 7, 0, 1,
    198, 7, 0, 1,
    205, 7, 0, 1,
    232, 7, 0, 2,
    233, 7, 0, 4,
    239, 7, 0, 1,
    240, 7, 0, 1,
    261, 7, 0, 1,
    280, 7, 0, 2,
    296, 7, 0, 2,
    323, 7, 0, 1,
    204, 7, 1, 1,
    344, 7, 1, 1
  };

  imsls_f_kaplan_meier_estimates (n_observations, ncol, x,
                              IMSLS_PRINT,
                              IMSLS_FREQ_RESPONSE_COL_COL, ifrq,
                              IMSLS_CENSOR_CODES_COL, icen,
                              IMSLS_STRATUM_NUMBER_COL, igrp,
                              0);
}
```

Output

```
                  Kaplan Meier Survival Probabilities
                     For Group Value = 5

      Number      Number                Survival     Estimated
     at risk     Failing        Time  Probability    Std. Error
          19           1         143      0.94737      0.051228

          18           1         164      0.89474      0.070406

          17           2         188      0.78947      0.093529

          15           1         190      0.73684       0.10102

          14           1         192      0.68421       0.10664

          13           1         206      0.63158       0.11066

          12           1         209      0.57895       0.11327

          11           1         213      0.52632       0.11455

          10           1         216      0.47368       0.11455

           8           1         220      0.41447       0.11452

           7           1         227      0.35526       0.11243
           6          1         230      0.29605       0.10816

           5          1         234      0.23684       0.10145

           3          1         246      0.15789      0.093431

           2          1         265     0.078947      0.072792

           1          1         304            0  ............

Total number in group    =      19
Total number failing     =      17
Product Limit Likelihood = -49.1692

                  Kaplan Meier Survival Probabilities
                     For Group Value = 7

      Number      Number                  Survival     Estimated
     at risk     Failing        Time  Probability    Std. Error
          21          1         142      0.95238      0.046471

          20          1         156      0.90476      0.064056

          19          1         163      0.85714       0.07636

          18          1         198      0.80952      0.085689

          16          1         205      0.75893      0.094092

          15          2         232      0.65774       0.10529

          13          4         233      0.45536       0.11137

           9          1         239      0.40476       0.10989

           8          1         240      0.35417       0.10717

           7          1         261      0.30357       0.10311

           6          2         280      0.20238      0.090214

           4          2         296      0.10119      0.067783

           2          1         323     0.050595      0.049281

Total number in group    =      21
Total number failing     =      19
Product Limit Likelihood = -50.4277
```

prop_hazards_gen_lin

Analyzes survival and reliability data using Cox’s proportional hazards model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_prop_hazards_gen_lin` (*int* `n_observations`,
int `n_columns`, *float* `x[]`, *int* `nef`, *int* `n_var_effects[]`,
int `indices_effects[]`, *int* `max_class`, *int* `*ncoef`, ..., 0)

The type *double* function is `imsls_d_prop_hazards_gen_lin`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `n_columns` (Input)
Number of columns in `x`.

float `x[]` (Input)
Array of length `n_observations` * `n_columns` containing the data. When optional argument `itie` = 1, the observations in `x` must be grouped by stratum and sorted from largest to smallest failure time within each stratum, with the strata separated.

int `nef` (Input)
Number of effects in the model. In addition to effects involving classification variables, simple covariates and the product of simple covariates are also considered effects.

int `n_var_effects[]` (Input)
Array of length `nef` containing the number of variables associated with each effect in the model.

int `indices_effects[]` (Input)
Index array of length `n_var_effects[0]` + … + `n_var_effects[nef-1]` containing the column indices of `x` associated with each effect. The first `n_var_effects[0]` elements of `indices_effects` contain the column indices of `x` for the variables in the first effect. The next `n_var_effects[1]` elements in `indices_effects` contain the column indices for the second effect, etc.

int `max_class` (Input)
An upper bound on the total number of different values found among the classification variables in `x`. For example, if the model consisted of two class variables, one with the values {1, 2, 3, 4} and a second with the values {0, 1}, then then the total number of different classification values is 4+2=6, and max_class >= 6.

int `*ncoef` (Output)
Number of estimated coefficients in the model.

Return Value

Pointer to an array of length `ncoef*4`, `coef`, containing the parameter estimates and associated statistics.

Column	Statistic
1	Coefficient estimate $\hat{\beta}$
2	Estimated standard deviation of the estimated coefficient.
3	Asymptotic normal score for testing that the coefficient is zero against the two-sided alternative.
4	*p*-value associated with the normal score in column 3.

Synopsis with Optional Arguments

```
#include <imsls.h>

float * imsls_f_prop_hazards_gen_lin (int n_observations,
        int n_columns, float x[], int nef, int n_var_effects[],
        int indices_effects[], int max_class, int *ncoef,
        IMSLS_RETURN_USER, float cov[],
        IMSLS_PRINT_LEVEL, int iprint,
        IMSLS_MAX_ITERATIONS, int max_iterations,
        IMSLS_CONVERGENCE_EPS, float eps,
        IMSLS_RATIO, float ratio,
        IMSLS_X_RESPONSE_COL, int irt,
        IMSLS_CENSOR_CODES_COL, int icen,
        IMSLS_STRATIFICATION_COL, int istrat,
        IMSLS_CONSTANT_COL, int ifix,
        IMSLS_FREQ_RESPONSE_COL, int ifrq,
        IMSLS_TIES_OPTION, int itie,
        IMSLS_MAXIMUM_LIKELIHOOD, float algl,
        IMSLS_N_MISSING, int *nrmiss,
        IMSLS_STATISTICS, float **case,
        IMSLS_STATISTICS_USER, float case[],
        IMSLS_X_MEAN, float **xmean,
        IMSLS_X_MEAN_USER, float xmean[],
        IMSLS_VARIANCE_COVARIANCE_MATRIX, float **cov,
        IMSLS_VARIANCE_COVARIANCE_MATRIX_USER, float cov[],
        IMSLS_INITIAL_EST_INPUT, float in_coef[],
        IMSLS_UPDATE, float **gr,
        IMSLS_UPDATE_USER, float gr[],
        IMSLS_DUMP, int n_class_var, int index_class_var[],
        IMSLS_STRATUM_NUMBER, int **igrp,
        IMSLS_STRATUM_NUMBER_USER, int igrp[],
        IMSLS_CLASS_VARIABLES, int **n_class_values,
             float **class_values,
        IMSLS_CLASS_VARIABLES_USER, int n_class_values[],
             float class_values[],
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `coef[]` (Output)
If specified, coef is an array of length `ncoef`*4 containing the parameter estimates and associated statistics. See Return Value.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option. Default: `iprint` = 0.

Iprint	**Action**
0	No printing is performed.
1	Printing is performed, but observational statistics are not printed.
2	All output statistics are printed.

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations. `max_iterations` = 30 will usually be sufficient. Use `max_iterations` = 0 to compute the Hessian and gradient, stored in `cov` and `gr`, at the initial estimates. When `max_iterations` = 0, `IMSLS_INITIAL_EST_INPUT` must be used.
Default: `max_iterations` = 30.

`IMSLS_CONVERGENCE_EPS`, *float* `eps` (Input)
Convergence criterion. Convergence is assumed when the relative change in `algl` from one iteration to the next is less than `eps`. If `eps` is zero, `eps` = 0.0001 is assumed.
Default: `eps` = 0.0001.

`IMSLS_RATIO`, *float* `ratio` (Input)
Ratio at which a stratum is split into two strata.
Default: `ratio` = 1000.0.
Let

$$r_k = \exp(z_k \hat{\beta} + w_k)$$

be the observation proportionality constant, where z_k is the design row vector for the k-th observation and w_k is the optional fixed parameter specified by $x_{k,\,\mathtt{ifix}}$. Let $r_{\min}$ be the minimum value r_k in a stratum, where, for failed observations, the minimum is over all times less than or equal to the time of occurrence of the k-th observation. Let $r_{\max}$ be the maximum value of r_k for the remaining observations in the group. Then, if $r_{\min} >$ `ratio` $r_{\max}$, the observations in the group are divided into two groups at k. `ratio` = 1000 is usually a good value. Set `ratio` = −1.0 if no division into strata is to be made.

`IMSLS_X_RESPONSE_COL`, *int* `irt` (Input)
Column index in `x` containing the response variable. For point observations, $x_{i,\,\mathtt{irt}}$ contains the time of the i-th event. For right-censored observations, $x_{i,\,\mathtt{irt}}$ contains the right-censoring time. Note that because `imsls_f_prop_hazards_gen_lin` only uses the order of the events,

negative "times" are allowed.
Default: irt = 0.

IMSLS_CENSOR_CODES_COL, *int* icen (Input)
Column index in x containing the censoring code for each observation.
Default: A censoring code of 0 is assumed for all observations.

$x_{i,icen}$	Censoring
0	Exact censoring time $x_{i,irt}$.
1	Right censored. The exact censoring time is greater than $x_{i,irt}$.

IMSLS_STRATIFICATION_COL, *int* istrat (Input)
Column number in x containing the stratification variable. Column istrat in x contains a unique number for each stratum. The risk set for an observation is determined by its stratum.
Default: All observations are considered to be in one stratum.

IMSLS_CONSTANT_COL, *int* ifix (Input)
Column index in x containing a constant, w_i, to be added to the linear response. The linear response is taken to be $w_i + z_i\hat{\beta}$
where w_i is the observation constant, z_i is the observation design row vector, and $\hat{\beta}$ is the vector of estimated parameters. The "fixed" constant allows one to test hypotheses about parameters via the log-likelihoods.
Default: w_i is assumed to be 0 for all observations.

IMSLS_FREQ_RESPONSE_COL, *int* ifrq (Input)
Column index in x containing the number of responses for each observation.
Default: A response frequency of 1 for each observation is assumed.

IMSLS_TIES_OPTION, *int* itie (Input)
Method for handling ties. Default: itie = 0.

Itie	Method
0	Breslow's approximate method.
1	Failures are assumed to occur in the same order as the observations input in x. The observations in x must be sorted from largest to smallest failure time within each stratum, and grouped by stratum. All observations are treated as if their failure/censoring times were distinct when computing the log-likelihood.

IMSLS_MAXIMUM_LIKELIHOOD, *float* *algl (Output)
The maximized log-likelihood.

IMSLS_N_MISSING, *int* *nrmiss (Output)
Number of rows of data in X that contain missing values in one or more columns irt, ifrq, ifix, icen, istrat, index_class_var, or indices_effects of x.

IMSLS_STATISTICS, *float* **case (Output)
Address of a pointer to an array of length n_observations * 5 containing the case statistics for each observation.

Column	Statistic
1	Estimated survival probability at the observation time.
2	Estimated observation influence or leverage.
3	A residual estimate.
4	Estimated cumulative baseline hazard rate.
5	Observation proportionality constant.

IMSLS_STATISTICS_USER, *float* case[] (Output)
Storage for case is provided by the user. See IMSLS_STATISTICS.

IMSLS_X_MEAN, *float* **xmean (Output)
Address of a pointer to an array of length ncoef containing the means of the design variables.

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

IMSLS_VARIANCE_COVARIANCE_MATRIX, *float* **cov (Output)
Address of a pointer to an array of length ncoef*ncoef containing the estimated asymptotic variance-covariance matrix of the parameters. For max_iterations = 0, the return value is the inverse of the Hessian of the negative of the log-likelihood, computed at the estimates input in in_coef.

IMSLS_VARIANCE_COVARIANCE_MATRIX_USER, *float* cov[] (Output)
Storage for cov is provided by the user. See IMSLS_VARIANCE_COVARIANCE_MATRIX.

IMSLS_INITIAL_EST_INPUT, *float* *in_coef (Input)
An array of length ncoef containing the initial estimates on input to prop_hazards_gen_lin.
Default: all initial estimates are taken to be 0.

IMSLS_UPDATE, *float* **gr (Output)
Address of a pointer to an array of length ncoef containing the last parameter updates (excluding step halvings). For max_iterations = 0, gr contains the inverse of the Hessian times the gradient vector computed at the estimates input in in_coef.

IMSLS_UPDATE_USER, *float* gr[] (Output)
Storage for gr is provided by the user. See IMSLS_UPDATE.

IMSLS_DUMP, *int* n_class_var, *int* index_class_var[] (Input)
Variable n_class_var is the number of classification variables. Dummy variables are generated for classification variables using the dummy_method = IMSLS_LEAVE_OUT_LAST of the IMSLS_DUMMY option of imsls_f_regressors_for_glm function (see Chapter 2, Regression).

Argument index_class_var is an index array of length n_class_var containing the column numbers of x that are the classification variables. (if n_class_var is is equal to zero, index_class_var is not used).
Default: n_class_var = 0.

IMSLS_STRATUM_NUMBER, *int* **igrp (Output)
Address of a pointer to an array of length n_observations giving the stratum number used for each observation. If ratio is not −1.0, additional "strata" (other than those specified by column istrat of x) may be generated. igrp also contains a record of the generated strata. See the "Description" section for more detail.

IMSLS_STRATUM_NUMBER_USER, *int* igrp[] (Output)
Storage for igrp is provided by the user. See IMSLS_STRATUM_NUMBER.

IMSLS_CLASS_VARIABLES, *int* **n_class_values, *float* **class_values (Output)
n_class_values is an address of a pointer to an array of length n_class_var containing the number of values taken by each classification variable. n_class_values[*i*] is the number of distinct values for the *i*-th classification variable. class_values is an address of a pointer to an array of length n_class_values[0] + n_class_values[1] + … + n_class_values[n_class_var-1] containing the distinct values of the classification variables. The first n_class_values[0] elements of class_values contain the values for the first classification variable, the next n_class_values[1] elements contain the values for the second classification variable, etc.

IMSLS_CLASS_VARIABLES_USER, *int* n_class_values[], *float* class_values[] (Output)
Storage for n_class_values and class_values is provided by the user. The length of class_values will not be known in advance, use max_class as the maximum length of class_values. See IMSLS_CLASS_VARIABLES.

Description

Function imsls_f_prop_hazards_gen_lin computes parameter estimates and other statistics in Proportional Hazards Generalized Linear Models. These models were first proposed by Cox (1972). Two methods for handling ties are allowed in imsls_f_prop_hazards_gen_lin. Time-dependent covariates are not allowed. The user is referred to Cox and Oakes (1984), Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), or Lawless (1982), among other texts, for a thorough discussion of the Cox proportional hazards model.

Let $\lambda(t, z_i)$ represent the hazard rate at time t for observation number i with covariables contained as elements of row vector z_i. The basic assumption in the proportional hazards model (the proportionality assumption) is that the hazard rate can be written as a product of a time varying function $\lambda_0(t)$, which depends only on time, and a function $f(z_i)$, which depends only on the covariable values. The function $f(z_i)$ used in imsls_f_prop_hazards_gen_lin is given as $f(z_i) = \exp(w_i + \beta z_i)$ where w_i is a fixed constant assigned to the observation, and β is a vector of coefficients to be estimated. With this function one obtains a hazard rate $\lambda(t, z_i) = \lambda_0(t) \exp(w_i + \beta z_i)$. The form of $\lambda_0(t)$ is not important in proportional hazards models.

The constants w_i may be known theoretically. For example, the hazard rate may be proportional to a known length or area, and the w_i can then be determined from this known length or area. Alternatively, the w_i may be used to fix a subset of the coefficients β (say, β_1) at specified values. When w_i is used in this way, constants $w_i = \beta_1 z_{1i}$ are used, while the remaining coefficients in β are free to vary in the optimization algorithm. If user-specified constants are not desired, the user should set `ifix` to 0 so that $w_i = 0$ will be used.

With this definition of $\lambda(t, z_i)$, the usual partial (or marginal, see Kalbfleisch and Prentice (1980)) likelihood becomes

$$L = \prod_{i=1}^{n_d} \frac{\exp(w_i + \beta z_i)}{\sum_{j \in R(t_i)} \exp(w_j + \beta z_j)}$$

where $R(t_i)$ denotes the set of indices of observations that have not yet failed at time t_i (the risk set), t_i denotes the time of failure for the i-th observation, n_d is the total number of observations that fail. Right-censored observations (i.e., observations that are known to have survived to time t_i, but for which no time of failure is known) are incorporated into the likelihood through the risk set $R(t_i)$. Such observations never appear in the numerator of the likelihood. When `itie` = 0, all observations that are censored at time t_i are not included in $R(t_i)$, while all observations that fail at time t_i are included in $R(t_i)$.

If it can be assumed that the dependence of the hazard rate upon the covariate values remains the same from stratum to stratum, while the time-dependent term, $\lambda_0(t)$, may be different in different strata, then `imsls_f_prop_hazards_gen_lin` allows the incorporation of strata into the likelihood as follows. Let k index the m = `istrat` strata. Then, the likelihood is given by

$$L_s = \prod_{k=1}^{m} \left[\prod_{i=1}^{n_k} \frac{\exp(w_{ki} + \beta z_{ki})}{\sum_{j \in R(t_{ki})} \exp(w_{kj} + \beta z_{kj})} \right]$$

In `imsls_f_prop_hazards_gen_lin`, the log of the likelihood is maximized with respect to the coefficients β. A quasi-Newton algorithm approximating the Hessian via the matrix of sums of squares and cross products of the first partial derivatives is used in the initial iterations (the "Q-N" method in the output). When the change in the log-likelihood from one iteration to the next is less than 100`*eps`, Newton-Raphson iteration is used (the "N-R" method). If, during any iteration, the initial step does not lead to an increase in the log-likelihood, then step halving is employed to find a step that will increase the log-likelihood.

Once the maximum likelihood estimates have been computed, `imsls_f_prop_hazards_gen_lin` computes estimates of a probability associated with each failure. Within stratum k, an estimate of the probability that the i-th observation fails at time t_i given the risk set $R(t_{ki})$ is given by

$$p_{ki} = \frac{\exp(w_{ki} + z_{ki}\beta)}{\sum_{j\in R(t_{ki})} \exp(w_{kj} + z_{kj}\beta)}$$

A diagnostic "influence" or "leverage" statistic is computed for each noncensored observation as:

$$l_{ki} = -g'_{ki} H_s^{-1} g'_{ki}$$

where H_s is the matrix of second partial derivatives of the log-likelihood, and

$$g'_{ki}$$

is computed as:

$$g'_{ki} = z_{ki} - \frac{z_{ki}\exp(w_{ki} + z_{ki}\beta)}{\sum_{j\in R(t_{ki})} \exp(w_{kj} + z_{kj}\beta)}$$

Influence statistics are not computed for censored observations.

A "residual" is computed for each of the input observations according to methods given in Cox and Oakes (1984, page 108). Residuals are computed as

$$r_{ki} = \exp(w_{ki} + z_{ki}\hat{\beta}) \quad \sum_{j\in R(t_{ki})} \frac{d_{kj}}{\sum_{l\in R(t_{kj})} \exp(w_{kl} + z_{kl}\hat{\beta})}$$

where d_{kj} is the number of tied failures in group k at time t_{kj}. Assuming that the proportional hazards assumption holds, the residuals should approximate a random sample (with censoring) from the unit exponential distribution. By subtracting the expected values, centered residuals can be obtained. (The j-th expected order statistic from the unit exponential with censoring is given as

$$e_j = \sum_{l\le j} \tfrac{1}{h-l+1}$$

where h is the sample size, and censored observations are not included in the summation.)

An estimate of the cumulative baseline hazard within group k is given as

$$\hat{H}_{k0}(t_{ik}) = \sum_{t_{kj}\le t_{ki}} \frac{d_{kj}}{\sum_{l\in R(t_{kj})} \exp(w_{kl} + z_{kl}\hat{\beta})}$$

The observation proportionality constant is computed as

$$\exp(w_{ki} + z_{ki}\hat{\beta})$$

Programming Notes

1. The covariate vectors z_{ki} are computed from each row of the input matrix `x` via function `imsls_f_regressors_for_glm` (see Chapter 2, Regression). Thus, class variables are easily incorporated into the z_{ki}. The reader is referred to the document for `imsls_f_regressors_for_glm` in the regression chapter for a more detailed discussion.
   Note that `imsls_f_prop_hazards_gen_lin` calls `imsls_f_regressors_for_glm` with `dummy_method` = `IMSLS_LEAVE_OUT_LAST` of the `IMSLS_DUMMY` option.

2. The average of each of the explanatory variables is subtracted from the variable prior to computing the product $z_{ki}\beta$. Subtraction of the mean values has no effect on the computed log-likelihood or the estimates since the constant term occurs in both the numerator and denominator of the likelihood. Subtracting the mean values does help to avoid invalid exponentiation in the algorithm and may also speed convergence.

3. Function `imsls_f_prop_hazards_gen_lin` allows for two methods of handling ties. In the first method (`itie` = 1), the user is allowed to break ties in any manner desired. When this method is used, it is
   assumed that the user has sorted the rows in `X` from largest to smallest with respect to the failure/censoring times $x_{i,\,\mathtt{irt}}$ within each stratum (and across strata), with tied observations (failures or censored) broken in the manner desired. The same effect can be obtained with `itie` = 0 by adding (or subtracting) a small amount from each of the tied observations failure/censoring times $t_i = x_{i,\,\mathtt{irt}}$ so as to break the ties in the desired manner.

The second method for handling ties (`itie` = 0) uses an approximation for the tied likelihood proposed by Breslow (1974). The likelihood in Breslow's method is as specified above, with the risk set at time t_i including all observations that fail at time t_i, while all observations that are censored at time t_i are not included. (Tied censored observations are assumed to be censored immediately prior to the time t_i).

4. `IMSLS_INITIAL_EST_INPUT` option is used, then it is assumed that the user has provided initial estimates for the model coefficients β in `in_coef`. When initial estimates are provided by the user, care should be taken to ensure that the estimates correspond to the generated covariate vector z_{ki}. If `IMSLS_INITIAL_EST_INPUT` option is not used, then initial estimates of zero are used for all of the coefficients. This corresponds to no effect from any of the covariate values.

5. If a linear combination of covariates is monotonically increasing or decreasing with increasing failure times, then one or more of the estimated coefficients is infinite and extended maximum likelihood estimates must be computed. Such estimates may be written as $\hat{\beta} = \hat{\beta}_f + \rho\hat{\gamma}$ where $\rho = \infty$ at the supremum of the likelihood so that $\hat{\beta}_f$ is the finite part of the solution. In

`imsls_f_prop_hazards_gen_lin`, it is assumed that extended maximum likelihood estimates must be computed if, within any group *k*, for any time *t*,

$$\min_{t_{ki}<t} \exp(w_{ki} + z_{ki}\hat{\beta}) > \rho \max_{t_{ki}<t} \exp(w_{ki} + z_{ki}\hat{\beta})$$

where ρ = `ratio` is specified by the user. Thus, for example, if ρ = 10000, then `imsls_f_prop_hazards_gen_lin` does not compute extended maximum likelihood estimates until the estimated proportionality constant

$$\exp(w_{ki} + z_{ki}\hat{\beta})$$

is 10000 times larger for all observations prior to *t* than for all observations after *t*. When this occurs, `imsls_f_prop_hazards_gen_lin` computes estimates for $\hat{\beta}_f$ by splitting the failures in stratum *k* into two strata at *t* (see Bryson and Johnson 1981). Censored observations in stratum *k* are placed into a stratum based upon the associated value for

$$\exp(w_{ki} + z_{ki}\hat{\beta})$$

The results of the splitting are returned in `igrp`.
The estimates $\hat{\beta}_f$ based upon the stratified likelihood represent the finite part of the extended maximum likelihood solution. Function `imsls_f_prop_hazards_gen_lin` does not compute $\hat{\gamma}$ explicitly, but an estimate for $\hat{\gamma}$ may be obtained in some circumstances by setting `ratio` = −1 and optimizing the log-likelihood without forming additional strata. The solution $\hat{\beta}$ obtained will be such that $\hat{\beta} = \hat{\beta}_f + \rho\hat{\gamma}$ for some finite value of ρ > 0. At this solution, the Newton-Raphson algorithm will not have "converged" because the Newton-Raphson step sizes returned in `gr` will be large, at least for some variables. Convergence will be declared, however, because the relative change in the log-likelihood during the final iterations will be small.

Example

The following data are taken from Lawless (1982, page 287) and involve the survival of lung cancer patients based upon their initial tumor types and treatment type. In the first example, the likelihood is maximized with no strata present in the data. This corresponds to Example 7.2.3 in Lawless (1982, page 367). The input data is printed in the output. The model is given as:

$$\ln(\lambda) = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \alpha_i + \gamma_j$$

where α_i and γ_j correspond to dummy variables generated from column indices 5 and 6 of `x`, respectively, x_1 corresponds to column index 2, x_2 corresponds to column index 3, and x_3 corresponds to column index 4 of `x`.

```
#include "imsls.h"

#define NOBS 40
```

```
#define NCOL 7
#define NCLVAR 2
#define NEF 5

void main ()
{
  int icen = 1, iprint = 2, maxcl = 6, ncoef;
  int indef[NEF] = { 2, 3, 4, 5, 6 };
  int nvef[NEF] = { 1, 1, 1, 1, 1 };
  int indcl[NCLVAR] = { 5, 6 };
  float *coef, ratio = 10000.0;
  float x[NOBS * NCOL] = {
    411, 0, 7, 64, 5, 1, 0,
    126, 0, 6, 63, 9, 1, 0,
    118, 0, 7, 65, 11, 1, 0,
    92, 0, 4, 69, 10, 1, 0,
    8, 0, 4, 63, 58, 1, 0,
    25, 1, 7, 48, 9, 1, 0,
    11, 0, 7, 48, 11, 1, 0,
    54, 0, 8, 63, 4, 2, 0,
    153, 0, 6, 63, 14, 2, 0,
    16, 0, 3, 53, 4, 2, 0,
    56, 0, 8, 43, 12, 2, 0,
    21, 0, 4, 55, 2, 2, 0,
    287, 0, 6, 66, 25, 2, 0,
    10, 0, 4, 67, 23, 2, 0,
    8, 0, 2, 61, 19, 3, 0,
    12, 0, 5, 63, 4, 3, 0,
    177, 0, 5, 66, 16, 4, 0,
    12, 0, 4, 68, 12, 4, 0,
    200, 0, 8, 41, 12, 4, 0,
    250, 0, 7, 53, 8, 4, 0,
    100, 0, 6, 37, 13, 4, 0,
    999, 0, 9, 54, 12, 1, 1,
    231, 1, 5, 52, 8, 1, 1,
    991, 0, 7, 50, 7, 1, 1,
    1, 0, 2, 65, 21, 1, 1,
    201, 0, 8, 52, 28, 1, 1,
    44, 0, 6, 70, 13, 1, 1,
    15, 0, 5, 40, 13, 1, 1,
    103, 1, 7, 36, 22, 2, 1,
    2, 0, 4, 44, 36, 2, 1,
    20, 0, 3, 54, 9, 2, 1,
    51, 0, 3, 59, 87, 2, 1,
    18, 0, 4, 69, 5, 3, 1,
    90, 0, 6, 50, 22, 3, 1,
    84, 0, 8, 62, 4, 3, 1,
    164, 0, 7, 68, 15, 4, 1,
    19, 0, 3, 39, 4, 4, 1,
    43, 0, 6, 49, 11, 4, 1,
    340, 0, 8, 64, 10, 4, 1,
    231, 0, 7, 67, 18, 4, 1
  };

  coef = imsls_f_prop_hazards_gen_lin (NOBS, NCOL, x, NEF,
                                nvef, indef, maxcl, &ncoef,
                                IMSLS_PRINT_LEVEL, iprint,
                                IMSLS_CENSOR_CODES_COL, icen,
                                IMSLS_RATIO, ratio,
                                IMSLS_DUMMY, NCLVAR, &indcl[0], 0);
}
```

Output

```
                    Initial Estimates
      1        2        3        4        5        6        7
 0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000

Method  Iteration  Step size  Maximum scaled     Log
                               coef. update      likelihood
  Q-N        0                                       -102.4
  Q-N        1      1.0000           0.5034          -91.04
  Q-N        2      1.0000           0.5782          -88.07
  N-R        3      1.0000           0.1131          -87.92
  N-R        4      1.0000          0.06958          -87.89
  N-R        5      1.0000        0.0008145          -87.89

Log-likelihood               -87.88778

                 Coefficient Statistics
    Coefficient      Standard    Asymptotic    Asymptotic
                        error   z-statistic       p-value
1        -0.585         0.137        -4.272         0.000
2        -0.013         0.021        -0.634         0.526
3         0.001         0.012         0.064         0.949
4        -0.367         0.485        -0.757         0.449
5        -0.008         0.507        -0.015         0.988
6         1.113         0.633         1.758         0.079
7         0.380         0.406         0.936         0.349

                   Asymptotic Coefficient Covariance
             1             2             3             4             5
1      0.01873      0.000253     0.0003345      0.005745       0.00975
2                  0.0004235    -4.12e-005     -0.001663    -0.0007954
3                                0.0001397     0.0008111     -0.001831
4                                                  0.235       0.09799
5                                                               0.2568

             6             7
1     0.004264      0.002082
2    -0.003079     -0.002898
3    0.0005995      0.001684
4       0.1184       0.03735
5       0.1253      -0.01944
6       0.4008       0.06289
7                     0.1647

                              Case Analysis
        Survival     Influence      Residual    Cumulative         Prop.
     Probability                                    hazard      constant
 1          0.00          0.04          2.05          6.10          0.34
 2          0.30          0.11          0.74          1.21          0.61
 3          0.34          0.12          0.36          1.07          0.33
 4          0.43          0.16          1.53          0.84          1.83
 5          0.96          0.56          0.09          0.05          2.05
 6          0.74   ............         0.13          0.31          0.42
 7          0.92          0.37          0.03          0.08          0.42
 8          0.59          0.26          0.14          0.53          0.27
 9          0.26          0.12          1.20          1.36          0.88
10          0.85          0.15          0.97          0.17          5.76
11          0.55          0.31          0.21          0.60          0.36
12          0.74          0.21          0.96          0.31          3.12
13          0.03          0.06          3.02          3.53          0.86
14          0.94          0.09          0.17          0.06          2.71
15          0.96          0.16          1.31          0.05         28.89
16          0.89          0.23          0.59          0.12          4.82
17          0.18          0.09          2.62          1.71          1.54
18          0.89          0.19          0.33          0.12          2.68
19          0.14          0.23          0.72          1.96          0.37
20          0.05          0.09          1.66          2.95          0.56
21          0.39          0.22          1.17          0.94          1.25
22          0.00          0.00          1.73         21.11          0.08
23          0.08   ............         2.19          2.52          0.87
24          0.00          0.00          2.46          8.89          0.28
25          0.99          0.31          0.05          0.01          4.28
26          0.11          0.17          0.34          2.23          0.15
27          0.66          0.25          0.16          0.41          0.38
28          0.87          0.22          0.15          0.14          1.02
29          0.39   ............         0.45          0.94          0.48
30          0.98          0.25          0.06          0.02          2.53
31          0.77          0.26          1.03          0.26          3.90
32          0.63          0.35          1.80          0.46          3.88
33          0.82          0.26          1.06          0.19          5.47
34          0.47          0.26          1.65          0.75          2.21
35          0.51          0.32          0.39          0.67          0.58
36          0.22          0.18          0.49          1.53          0.32
37          0.80          0.26          1.08          0.23          4.77
38          0.70          0.16          0.26          0.36          0.73
39          0.01          0.23          0.87          4.66          0.19
40          0.08          0.20          0.81          2.52          0.32

                          Last Coefficient Update
           1             2             3             4             5             6
-1.296e-008   2.269e-009  -5.894e-009  -4.782e-007  -1.787e-007   1.509e-007

           7
 4.327e-008

                              Covariate Means
           1             2             3             4             5             6
        5.65         56.58         15.65          0.35          0.28          0.13

           7
        0.53
```

```
Distinct Values For Each Class Variable
Variable 1:           1           2           3           4

Variable 2:           0           1

                     Stratum Numbers For Each Observation
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
20
 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1

21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
40
 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1

Number of Missing Values          0
```

survival_glm

Analyzes censored survival data using a generalized linear model.

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_survival_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.

Model	PDF of the Response Variable
0	Exponential
1	Linear hazard
2	Log-normal
3	Normal
4	Log-logistic

Model	PDF of the Response Variable
5	Logistic
6	Log least extreme value
7	Least extreme value
8	Log extreme value
9	Extreme value
10	Weibull

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 in the model.

Synopsis with Optional Arguments

```
#include <imsls.h>

int imsls_f_survival_glm (int n_observations, int n_class,
       int n_continuous, int model, float x[],
       IMSLS_X_COL_CENSORING, int icen, int ilt, int irt,
       IMSLS_X_COL_DIM, int x_col_dim,
       IMSLS_X_COL_FREQUENCIES, int ifrq,
       IMSLS_X_COL_FIXED_PARAMETER, int ifix,
       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,
       IMSLS_INFINITY_CHECK, int lp_max
       IMSLS_NO_INFINITY_CHECK
       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_SURVIVAL_INFO, *Imsls_f_survival* **survival_info
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_CENSORING, *int* icen, *int* ilt, *int* irt (Input)
Parameter icen is the column in x containing the censoring code for each observation.

x [*i*] [icen]	**Censoring type**
0	Exact failure at x [*i*] [irt]
1	Right Censored. The response is greater than x [*i*] [irt].
2	Left Censored. The response is less than or equal to x [*i*] [irt].
3	Interval Censored. The response is greater than x [*i*] [irt], but less than or equal to x [*i*] [ilt].

Parameter ilt is the column number of x containing the upper endpoint of the failure interval for interval- and left-censored observations. If there are no left-censored or interval-censored observations, ilt should be set to −1.

Parameter irt is the column number of x containing the lower endpoint of the failure interval for interval- and right-censored observations. If there are no left-censored or interval-censored observations, irt should be set to −1.

Exact failure times are specified in column `iy` of `x`. By default, `iy` is column `n_class` + `n_continuous` of `x`. The default can be changed if keyword `IMSLS_X_COL_VARIABLES` is specified.

Note that it is allowable to set `iy` = `irt`, since a row with an `iy` value will never have an `irt` value, and vice versa. This use is illustrated in Example 2.

`IMSLS_FREQUENCIES`, *int* `ifrq` (Input)
: Column number of `x` containing the frequency of response for each observation.

`IMSLS_FIXED_PARAMETER`, *int* `ifix` (Input)
: Column number 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_VARIABLES` *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 from 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` corresponds to the column of `x` which contains the dependent 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_INFINITY_CHECK`, *int* `lp_max` (Input)

Remove a right- or left-censored observation from the log-likelihood whenever the probability of the observation exceeds 0.995. At convergence, use linear programming to check that all removed observations actually have infinite linear response

$$z_i\hat{\beta}$$

`obs_status` [*i*] is set to 2 if the linear response is infinite (See optional argument `IMSLS_OBS_STATUS`). If not all removed observations have infinite linear response, re-compute the estimates based upon the observations with finite

$$z_i\hat{\beta}$$

Parameter `lp_max` is the maximum number of observations that can be handled in the linear programming. Setting `lp_max` = `n_observations` is always sufficient.
Default: No infinity checking; `lp_max` = 0

`IMSLS_NO_INFINITY_CHECK`

Iterates without checking for infinite estimates. This option is the default.

`IMSLS_EFFECTS`, *int* `n_effects`, *int* `n_var_effects[]`,
int `indices_effects[]` (Input)

Use this keyword to specify the effects in the model.

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 `survival_glm`). See optional argument `IMSLS_COEF_STAT` for a description of the “nuisance” parameter, which is the first element of array `estimates`.

`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. Internal workspace usage can be significantly reduced with an appropriate choice of `max_class`.
Default: `max_class` = `n_observations` * `n_class`

`IMSLS_CLASS_INFO`, *int* `**n_class_values`, *float* `**class_values` (Output)

Argument `n_class_values` is 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

$$\sum_{i=0}^{\text{n\_class-1}} \texttt{n\_class\_values}[i]$$

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:

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.

When present in the model, the first coefficient in `coef_statistics` is the estimate of the "nuisance" parameter, and the remaining coefficients are estimates of the parameters associated with the "linear" model, beginning with the intercept, if present. Nuisance parameters are as follows:

model	
0	No nuisance parameter
1	Coefficient of the quadratic term in time, θ
2-9	Scale parameter, σ
10	Shape parameter, θ

`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` by `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.

`IMSLS_COV_USER`, *float* `cov[]` (Ouput)
Storage for array `cov` is provided by the user. See `IMSLS_COV`.

`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` – *m* if `IMSLS_NO_INTERCEPT` is specified, and of length `n_coefficients` – *m* – 1 otherwise. Here, *m* is equal to 0 if `model` = 0, and equal to 1 otherwise.

`IMSLS_MEANS_USER`, *float* `means[]` (Output)
Storage for array means is provided by the user. See `IMSLS_MEANS`.

`IMSLS_CASE_ANALYSIS`, *float* `**case_statistics` (Output)
Address of a pointer to the internally allocated array of size `n_observations` by 5 containing the case analysis below:

Column	Statistic
0	Estimated predicted value.
1	Estimated influence or leverage.
2	Estimated residual.
3	Estimated cumulative hazard.
4	Non-censored observations: Estimated density at the observation failure time and covariate values. Censored observations: The corresponding estimated probability.

If `max_iterations` = 0, `case_statistics` is an array of length `n_observations` containing the estimated probability (for censored observations) or the estimated density (for non-censored observations)

IMSLS_CASE_ANALYSIS_USER, *float* case_statistics[] (Output)

Storage for array case_statistics 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). Parameter last_step is computed as the inverse of the matrix of second partial derivatives times the vector of first partial derivatives of the log-likelihood. When max_iterations = 0, the derivatives are computed at the initial 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_ITERATIONS, *int* *n, *float* **iterations (Output)

Address of a pointer to the internally allocated array of size, n by 5 containing information about each iteration of the analysis, where n is equal to the number of iterations.

Column	**Statistic**
0	Method of iteration
	Q-N Step = 0
	N-R Step = 1
1	Iteration number
2	Step size
3	Maximum scaled coefficient update
4	Log-likelihood

IMSLS_ITERATIONS_USER, *int* *n, *float* iterations[] (Output)

Storage for array iterations is provided by the user. See IMSLS_ITERATIONS.

IMSLS_SURVIVAL_INFO, *Imsls_f_survival* **survival_info (Output)
Address of the pointer to an internally allocated structure of type *Imsls_f_survival* containing information about the survival analysis. This structure is required input for function imsls_f_survival_estimates.

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 vectors or columns of x: iy, icen, ilt, irt, ifrq, ifix, iclass, icontinuous, or indices_effects.

Comments

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 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_survival_glm computes the maximum likelihood estimates of parameters and associated statistics in generalized linear models commonly found in survival (reliability) analysis. Although the terminology used will be from the survival area, the methods discussed have applications in many areas of data analysis, including reliability analysis and event history analysis. These methods can be used anywhere a random variable from one of the discussed distributions is parameterized via one of the models available in imsls_f_survival_glm. Thus, while it is not advisable to do so, standard multiple linear regression can be performed by routine imsls_f_survival_glm. Estimates for any of 10 standard models can be computed. Exact, left-censored, right-censored, or interval-censored observations are allowed (note that left censoring is the same as interval censoring with the left endpoint equal to the left endpoint of the support of the distribution).

Let $\eta = x^T\beta$ be the linear parameterization, where x is a design vector obtained by imsls_f_survival_glm via function imsls_f_regressors_for_glm from a row of x, and β is a vector of parameters associated with the linear model. Let T denote the random response variable and $S(t)$ denote the probability that $T > t$. All models considered also allow a fixed parameter w_i for observation i (input in column `ifix` of `x`). Use of this parameter is discussed below. There also may be nuisance parameters $\theta > 0$, or $\sigma > 0$ to be estimated (along with β) in the various models. Let Φ denote the cumulative normal distribution. The survival models available in `imsls_f_survival_glm` are:

Model	**Name**	***S* (*t*)**
0	Exponential	$\exp\left[-t\exp(w_i + \eta)\right]$
1	Linear hazard	$\exp\left[-\left(t + \frac{\theta t^2}{2}\right)\exp\left(w_i + \eta\right)\right]$
2	Log-normal	$1 - \Phi\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)$
3	Normal	$1 - \Phi\left(\frac{t - \eta - w_i}{\sigma}\right)$
4	Log-logistic	$\{1 + \exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}^{-1}$
5	Logistic	$\{1 + \exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}^{-1}$
6	Log least extreme value	$\exp\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}$
7	Least extreme value	$\exp\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}$
8	Log extreme value	$1 - \exp\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}$
9	Extreme value	$1 - \exp\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}$
10	Weibull	$\exp\{-\left[\frac{t}{\exp\left(w_i + \eta\right)}\right]^{\theta}\}$

Note that the log-least-extreme-value model is a reparameterization of the Weibull model. Moreover, models 0, 1, 2, 4, 6, 8, and 10 require that $T > 0$, while all of the remaining models allow any value for T, $-\infty < T < \infty$.

Each row vector in the data matrix can represent a single observation; or, through the use of vector 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 constant parameter W_i is input in x and may be used for a number of purposes. For example, if the parameter in an exponential model is known to depend upon the size of the area tested, volume of a radioactive mass, or population density, etc., then a multiplicative factor of the exponential parameter $\lambda = \exp(x\beta)$ may be known apriori. This factor can be input in W_i (W_i is the log of the factor).

An alternate use of W_i is as follows: It may be that $\lambda = \exp(x_1\beta_1 + x_2\beta_2)$, where β_2 is known. Letting $W_i = x_2\beta_2$, estimates for β_1 can be obtained via `imsls_f_survival_glm` with the known fixed values for β_2. Standard methods can then be used to test hypothesis about β_1 via computed log-likelihoods.

Computational Details

The computations proceed as follows:

1. The input parameters are checked for consistency and validity.
   - Estimates of the means of the "independent" or design variables are computed. Means are computed as

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

2. If initial estimates are not provided by the user (see optional argument `IMSLS_INITIAL_EST_INPUT`), the initial estimates are calculated as follows:
   - Models 2-10

A. Kaplan-Meier estimates of the survival probability,

$$\hat{S}(t)$$

at the upper limit of each failure interval are obtained. (Because upper limits are used, interval- and left-censored data are assumed to be exact failures at the upper endpoint of the failure interval.) The Kaplan-Meier estimate is computed under the assumption that all failure distributions are identical (i.e., all β's but the intercept, if present, are assumed to be zero).

B. If there is an intercept in the model, a simple linear regression is performed predicting

$$S^{-1}\left(\hat{S}(t)\right) - w_i = \alpha + \phi t'$$

where t' is computed at the upper endpoint of each failure interval, $t' = t$ in models 3, 5, 7, and 9, and $t' = \ln(t)$ in models 2, 4, 6, 8, and 10, and w_i is the fixed constant, if present.

If there is no intercept in the model, then α is fixed at zero, and the model

$$S^{-1}\left(\hat{S}\left(t\right)\right)-\hat{\phi}t'-w_i=x^T\beta$$

is fit instead. In this model, the coefficients β are used in place of the location estimate α above. Here

$$\hat{\phi}$$

is estimated from the simple linear regression with $\alpha = 0$.

C. If the intercept is in the model, then in log-location-scale models (models 1-8),

$$\hat{\sigma}=\hat{\phi}$$

and the initial estimate of the intercept is assumed to be $\hat{\alpha}$.

In the Weibull model

$$\hat{\theta}=1/\hat{\phi}$$

and the intercept is assumed to be $\hat{\alpha}$.

Initial estimates of all parameters β, other than the intercept, are assumed to be zero.

If there is no intercept in the model, the scale parameter is estimated as above, and the estimates

$$\hat{\beta}$$

from Step 2 are used as initial estimates for the β's.

- Models 0 and 1

  For the exponential models (`model` = 0 or 1), the "average total time on" test statistic is used to obtain an estimate for the intercept. Specifically, let T_t denote the total number of failures divided by the total time on test. The initial estimates for the intercept is then $\ln(T_t)$. Initial estimates for the remaining parameters β are assumed to be zero, and if `model` = 1, the initial estimate for the linear hazard parameter θ is assumed to be a small positive number. When the intercept is not in the model, the initial estimate for the parameter θ is assumed to be a small positive number, and initial estimates of the parameters β are computed via multiple linear regression as in Part A.

3. A quasi-Newton algorithm is used in the initial iterations based on a Hessian estimate

$$\hat{H}_{\kappa_j \kappa_l} = \sum_i l'_{i\alpha_j i\alpha_l}$$

where $l'_{i\alpha j}$ is the partial derivative of the i-th term in the log-likelihood with respect to the parameter α_j, and a_j denotes one of the parameter to be estimated.

When the relative change in the log-likelihood from one iteration to the next is 0.1 or less, exact second partial derivatives are used for the Hessian so the Newton-Rapheson iteration is used.

If the initial step size results in an increase in the log-likelihood, the full step is used. If the log-likelihood decreases for the initial step size, the step size is halved, and a check for an increase in the log-likelihood performed. Step-halving is performed (as a simple line search) until an increase in the log-likelihood is detected, or until the step size becomes very small (the initial step size is 1.0).

4. 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 very small step size with no increase in the log-likelihood.

5. If requested (see optional argument `IMSLS_INFINITY_CHECK`), then the methods of Clarkson and Jennrich (1988) are used to check for the existence of infinite estimates in

$$\eta_i = x_i^T \beta$$

As an example of a situation in which infinite estimates can occur, suppose that observation j is right-censored with $t_j > 15$ in a normal distribution model in which the mean is

$$\mu_j = x_j^T \beta = \eta_j$$

where x_j is the observation design vector. If the design vector x_j for parameter β_m is such that $x_{jm} = 1$ and $x_{im} = 0$ for all $i \neq j$, then the optimal estimate of β_m occurs at

$$\hat{\beta}_m = \infty$$

leading to an infinite estimate of both β_m and η_j. In `imsls_f_survival_glm`, such estimates can be "computed".

In all models fit by `imsls_f_survival_glm`, infinite estimates can only occur when the optimal estimated probability associated with the left- or right-censored observation is 1. If infinity checking is on, left- or right-censored observations that have estimated probability greater than 0.995 at some point during the iterations are excluded from the log-likelihood, and the iterations proceed with a log-likelihood based on the remaining observations. This allows convergence of the algorithm when the maximum relative change in the estimated coefficients is small and also allows for a more precise determination of observations with infinite

$$\eta_i = x_i^T \beta$$

At convergence, linear programming is used to ensure that the eliminated observations have infinite η_i. If some (or all) of the removed observations should not have been removed (because their estimated η_i's must be finite), then the iterations are restarted with a log-likelihood based upon the finite η_i observations. See Clarkson and Jennrich (1988) for more details.

When infinity checking is turned off (see optional argument `IMSLS_NO_INFINITY_CHECK`), no observations are eliminated during the iterations. In this case, the infinite estimates occur, some (or all) of the coefficient estimates

$$\hat{\beta}$$

will become large, and it is likely that the Hessian will become (numerically) singular prior to convergence.

6. The case statistics are computed as follows: Let I_i (θ_i)denote the log-likelihood
of the *i*-th observation evaluated at θ_i, let I'_i denote the vector of derivatives of I_i with respect to all parameters, $I'_{\eta,i}$ denote the derivative of I_i with respect to $\eta = x^T\beta$, *H* denote the Hessian, and *E* denote expectation. Then the columns of `case_statistics` are:

   A. Predicted values are computed as E (T/x) according to standard formulas. If model is 4 or 8, and if $s \geq 1$, then the expected values cannot be computed because they are infinite.

   B. Following Cook and Weisberg (1982), the influence (or leverage) of the *i*-th observation is assumed to be

   $$\left(I_i'\right)^T H^{-1} I_i'$$

   This quantity is a one-step approximation of the change in the estimates when the *i*-th observation is deleted (ignoring the nuisance parameters).

   C. The “residual” is computed as $I'_{\eta,i}$.

   D. The cumulative hazard is computed at the observation covariate values and, for interval observations, the upper endpoint of the failure interval. The cumulative hazard also can be used as a “residual” estimate. If the model is correct, the cumulative hazards should follow a standard exponential distribution. See Cox and Oakes (1984).

Programming Notes

Indicator (dummy) variables are created for the classification variables using function `imsls_f_regressors_for_glm` (Chapter 2, “Regression”) using keyword `IMSLS_LEAVE_OUT_LAST` as the argument to the `IMSLS_DUMMY` optional argument.

Examples

Example 1

This example is taken from Lawless (1982, p. 287) and involves the mortality of patients suffering from lung cancer. An exponential distribution is fit for the model

$$\eta = \mu + \alpha_i + \gamma_k + \beta_6 x_3 + \beta_7 x_4 + \beta_8 x_5$$

where α_i is associated with a classification variable with four levels, and γ_k is associated with a classification variable with two levels. Note that because the computations are performed in single precision, there will be some small variation in the estimated coefficients across different machine environments.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,    0.0,    6.0,   63.0,    9.0,  126.0,    0.0,
        1.0,    0.0,    7.0,   65.0,   11.0,  118.0,    0.0,
        1.0,    0.0,    4.0,   69.0,   10.0,   92.0,    0.0,
        1.0,    0.0,    4.0,   63.0,   58.0,    8.0,    0.0,
        1.0,    0.0,    7.0,   48.0,    9.0,   25.0,    1.0,
        1.0,    0.0,    7.0,   48.0,   11.0,   11.0,    0.0,
        2.0,    0.0,    8.0,   63.0,    4.0,   54.0,    0.0,
        2.0,    0.0,    6.0,   63.0,   14.0,  153.0,    0.0,
        2.0,    0.0,    3.0,   53.0,    4.0,   16.0,    0.0,
        2.0,    0.0,    8.0,   43.0,   12.0,   56.0,    0.0,
        2.0,    0.0,    4.0,   55.0,    2.0,   21.0,    0.0,
        2.0,    0.0,    6.0,   66.0,   25.0,  287.0,    0.0,
        2.0,    0.0,    4.0,   67.0,   23.0,   10.0,    0.0,
        3.0,    0.0,    2.0,   61.0,   19.0,    8.0,    0.0,
        3.0,    0.0,    5.0,   63.0,    4.0,   12.0,    0.0,
        4.0,    0.0,    5.0,   66.0,   16.0,  177.0,    0.0,
        4.0,    0.0,    4.0,   68.0,   12.0,   12.0,    0.0,
        4.0,    0.0,    8.0,   41.0,   12.0,  200.0,    0.0,
        4.0,    0.0,    7.0,   53.0,    8.0,  250.0,    0.0,
        4.0,    0.0,    6.0,   37.0,   13.0,  100.0,    0.0,
        1.0,    1.0,    9.0,   54.0,   12.0,  999.0,    0.0,
        1.0,    1.0,    5.0,   52.0,    8.0,  231.0,    1.0,
        1.0,    1.0,    7.0,   50.0,    7.0,  991.0,    0.0,
        1.0,    1.0,    2.0,   65.0,   21.0,    1.0,    0.0,
        1.0,    1.0,    8.0,   52.0,   28.0,  201.0,    0.0,
        1.0,    1.0,    6.0,   70.0,   13.0,   44.0,    0.0,
        1.0,    1.0,    5.0,   40.0,   13.0,   15.0,    0.0,
        2.0,    1.0,    7.0,   36.0,   22.0,  103.0,    1.0,
        2.0,    1.0,    4.0,   44.0,   36.0,    2.0,    0.0,
        2.0,    1.0,    3.0,   54.0,    9.0,   20.0,    0.0,
        2.0,    1.0,    3.0,   59.0,   87.0,   51.0,    0.0,
        3.0,    1.0,    4.0,   69.0,    5.0,   18.0,    0.0,
        3.0,    1.0,    6.0,   50.0,   22.0,   90.0,    0.0,
        3.0,    1.0,    8.0,   62.0,    4.0,   84.0,    0.0,
        4.0,    1.0,    7.0,   68.0,   15.0,  164.0,    0.0,
        4.0,    1.0,    3.0,   39.0,    4.0,   19.0,    0.0,
        4.0,    1.0,    6.0,   49.0,   11.0,   43.0,    0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   n_coef;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    float *coef_stat;
    char *fmt = "%12.4f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_COEF_STAT, &coef_stat,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);
}
```

Output

```
                Coefficient Statistics
 coefficient         s.e.           z           p
     -1.1027       1.3140     -0.8392      0.4016
     -0.3626       0.4446     -0.8156      0.4149
      0.1271       0.4863      0.2613      0.7939
      0.8690       0.5861      1.4825      0.1385
      0.2697       0.3882      0.6948      0.4873
     -0.5400       0.1081     -4.9946      0.0000
     -0.0090       0.0197     -0.4594      0.6460
     -0.0034       0.0117     -0.2912      0.7710
```

Example 2

This example is the same as Example 1, but more optional arguments are demonstrated.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,     0.0,     6.0,    63.0,     9.0,  126.0,     0.0,
        1.0,     0.0,     7.0,    65.0,    11.0,  118.0,     0.0,
        1.0,     0.0,     4.0,    69.0,    10.0,   92.0,     0.0,
        1.0,     0.0,     4.0,    63.0,    58.0,    8.0,     0.0,
        1.0,     0.0,     7.0,    48.0,     9.0,   25.0,     1.0,
        1.0,     0.0,     7.0,    48.0,    11.0,   11.0,     0.0,
        2.0,     0.0,     8.0,    63.0,     4.0,   54.0,     0.0,
        2.0,     0.0,     6.0,    63.0,    14.0,  153.0,     0.0,
        2.0,     0.0,     3.0,    53.0,     4.0,   16.0,     0.0,
        2.0,     0.0,     8.0,    43.0,    12.0,   56.0,     0.0,
        2.0,     0.0,     4.0,    55.0,     2.0,   21.0,     0.0,
        2.0,     0.0,     6.0,    66.0,    25.0,  287.0,     0.0,
        2.0,     0.0,     4.0,    67.0,    23.0,   10.0,     0.0,
        3.0,     0.0,     2.0,    61.0,    19.0,    8.0,     0.0,
        3.0,     0.0,     5.0,    63.0,     4.0,   12.0,     0.0,
        4.0,     0.0,     5.0,    66.0,    16.0,  177.0,     0.0,
        4.0,     0.0,     4.0,    68.0,    12.0,   12.0,     0.0,
        4.0,     0.0,     8.0,    41.0,    12.0,  200.0,     0.0,
        4.0,     0.0,     7.0,    53.0,     8.0,  250.0,     0.0,
        4.0,     0.0,     6.0,    37.0,    13.0,  100.0,     0.0,
        1.0,     1.0,     9.0,    54.0,    12.0,  999.0,     0.0,
        1.0,     1.0,     5.0,    52.0,     8.0,  231.0,     1.0,
        1.0,     1.0,     7.0,    50.0,     7.0,  991.0,     0.0,
        1.0,     1.0,     2.0,    65.0,    21.0,    1.0,     0.0,
        1.0,     1.0,     8.0,    52.0,    28.0,  201.0,     0.0,
        1.0,     1.0,     6.0,    70.0,    13.0,   44.0,     0.0,
        1.0,     1.0,     5.0,    40.0,    13.0,   15.0,     0.0,
        2.0,     1.0,     7.0,    36.0,    22.0,  103.0,     1.0,
        2.0,     1.0,     4.0,    44.0,    36.0,    2.0,     0.0,
        2.0,     1.0,     3.0,    54.0,     9.0,   20.0,     0.0,
        2.0,     1.0,     3.0,    59.0,    87.0,   51.0,     0.0,
        3.0,     1.0,     4.0,    69.0,     5.0,   18.0,     0.0,
        3.0,     1.0,     6.0,    50.0,    22.0,   90.0,     0.0,
        3.0,     1.0,     8.0,    62.0,     4.0,   84.0,     0.0,
        4.0,     1.0,     7.0,    68.0,    15.0,  164.0,     0.0,
        4.0,     1.0,     3.0,    39.0,     4.0,   19.0,     0.0,
        4.0,     1.0,     6.0,    49.0,    11.0,   43.0,     0.0,
        4.0,     1.0,     8.0,    64.0,    10.0,  340.0,     0.0,
        4.0,     1.0,     7.0,    67.0,    18.0,  231.0,     0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   n_coef;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    int   n, *ncv, nrmiss, *obs;
    float *iterations, *cv, criterion;
    float *coef_stat, *casex;
    char *fmt = "%12.4f";
    char *fmt2 = "%4d%4d%6.4f%8.4f%8.1f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};
    static char *clabels2[] = {"", "Method", "Iteration", "Step Size",
        "Coef Update", "Log-Likelihood"};
    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_COEF_STAT, &coef_stat,
        IMSLS_ITERATIONS, &n, &iterations,
        IMSLS_CASE_ANALYSIS, &casex,
        IMSLS_CLASS_INFO, &ncv, &cv,
        IMSLS_OBS_STATUS, &obs,
        IMSLS_CRITERION, &criterion,
        IMSLS_N_ROWS_MISSING, &nrmiss,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);

    imsls_f_write_matrix("Iteration Information", n, 5, iterations,
        IMSLS_WRITE_FORMAT, fmt2,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels2, 0);

    printf("\nLog-Likelihood = %12.5f\n", criterion);

    imsls_f_write_matrix("Case Analysis", 1, n_observations, casex,
        IMSLS_WRITE_FORMAT, fmt,
        0);

    imsls_f_write_matrix(
        "Distinct Values for Classification Variable 1",
        1, ncv[0], &cv[0], IMSLS_NO_COL_LABELS, 0);

    imsls_f_write_matrix(
        "Distinct Values for Classification Variable 2",
        1, ncv[1], &cv[ncv[0]], IMSLS_NO_COL_LABELS, 0);

    imsls_i_write_matrix("Observation Status", 1, n_observations,
        obs, 0);

    printf("\nNumber of Missing Values = %2d\n", nrmiss);
}
```

Output

```
              Coefficient Statistics
 coefficient          s.e.             z             p
     -1.1027        1.3140       -0.8392        0.4016
     -0.3626        0.4446       -0.8156        0.4149
      0.1271        0.4863        0.2613        0.7939
      0.8690        0.5861        1.4825        0.1385
      0.2697        0.3882        0.6948        0.4873
     -0.5400        0.1081       -4.9946        0.0000
     -0.0090        0.0197       -0.4594        0.6460
     -0.0034        0.0117       -0.2912        0.7710

                  Iteration Information
Method  Iteration  Step Size  Coef Update  Log-Likelihood
     0          0     ......     ........          -224.0
     0          1     1.0000       0.9839          -213.4
     1          2     1.0000       3.6033          -207.3
     1          3     1.0000      10.1236          -204.3
     1          4     1.0000       0.1430          -204.1
     1          5     1.0000       0.0117          -204.1

Log-Likelihood =   -204.13916

                            Case Analysis
          1             2             3             4             5
   262.6884        0.0450       -0.5646        1.5646        0.0008

          6             7             8             9            10
   153.7777        0.0042        0.1806        0.8194        0.0029

         11            12            13            14            15
   270.5347        0.0482        0.5638        0.4362        0.0024

         16            17            18            19            20
    55.3168        0.0844       -0.6631        1.6631        0.0034

         21            22            23            24            25
    61.6845        0.3765        0.8703        0.1297        0.0142

         26            27            28            29            30
   230.4414        0.0025       -0.1085        0.1085        0.8972

         31            32            33            34            35
   232.0135        0.1960        0.9526        0.0474        0.0041

         36            37            38            39            40
   272.8432        0.1677        0.8021        0.1979        0.0030

 Distinct Values for Classification Variable 1
         1           2           3           4

Distinct Values for Classification Variable 2
                    0          1

                              Observation Status
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

Number of Missing Values =  0
```

Example 3

In this example, the same data and model as Example 1 are used, but `max_iterations` is set to zero iterations with model coefficients restricted such that $\mu = -1.25$, $\beta_6 = -0.6$, and the remaining six coefficients are equal to zero. A chi-squared statistic, with 8 degrees of freedom for testing the coefficients is specified as above (versus the alternative that it is not as specified), can be computed, based on the output, as

$$\chi^2 = g^T \hat{\Sigma}^{-1} g$$

where

$$\hat{\Sigma}$$

is output in `cov`. The resulting test statistic, $\chi^2 = 6.107$, based upon no iterations is comparable to likelihood ratio test that can be computed from the log-likelihood output in this example (–206.6835) and the log-likelihood output in Example 2 (–204.1392).

$$\chi^2_{LR} = 2\left(206.6835 - 204.1392\right) = 5.0886$$

Neither statistic is significant at the $\alpha = 0.05$ level.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,    0.0,    6.0,   63.0,    9.0,  126.0,    0.0,
        1.0,    0.0,    7.0,   65.0,   11.0,  118.0,    0.0,
        1.0,    0.0,    4.0,   69.0,   10.0,   92.0,    0.0,
        1.0,    0.0,    4.0,   63.0,   58.0,    8.0,    0.0,
        1.0,    0.0,    7.0,   48.0,    9.0,   25.0,    1.0,
        1.0,    0.0,    7.0,   48.0,   11.0,   11.0,    0.0,
        2.0,    0.0,    8.0,   63.0,    4.0,   54.0,    0.0,
        2.0,    0.0,    6.0,   63.0,   14.0,  153.0,    0.0,
        2.0,    0.0,    3.0,   53.0,    4.0,   16.0,    0.0,
        2.0,    0.0,    8.0,   43.0,   12.0,   56.0,    0.0,
        2.0,    0.0,    4.0,   55.0,    2.0,   21.0,    0.0,
        2.0,    0.0,    6.0,   66.0,   25.0,  287.0,    0.0,
        2.0,    0.0,    4.0,   67.0,   23.0,   10.0,    0.0,
        3.0,    0.0,    2.0,   61.0,   19.0,    8.0,    0.0,
        3.0,    0.0,    5.0,   63.0,    4.0,   12.0,    0.0,
        4.0,    0.0,    5.0,   66.0,   16.0,  177.0,    0.0,
        4.0,    0.0,    4.0,   68.0,   12.0,   12.0,    0.0,
        4.0,    0.0,    8.0,   41.0,   12.0,  200.0,    0.0,
        4.0,    0.0,    7.0,   53.0,    8.0,  250.0,    0.0,
        4.0,    0.0,    6.0,   37.0,   13.0,  100.0,    0.0,
        1.0,    1.0,    9.0,   54.0,   12.0,  999.0,    0.0,
        1.0,    1.0,    5.0,   52.0,    8.0,  231.0,    1.0,
        1.0,    1.0,    7.0,   50.0,    7.0,  991.0,    0.0,
        1.0,    1.0,    2.0,   65.0,   21.0,    1.0,    0.0,
        1.0,    1.0,    8.0,   52.0,   28.0,  201.0,    0.0,
        1.0,    1.0,    6.0,   70.0,   13.0,   44.0,    0.0,
        1.0,    1.0,    5.0,   40.0,   13.0,   15.0,    0.0,
        2.0,    1.0,    7.0,   36.0,   22.0,  103.0,    1.0,
        2.0,    1.0,    4.0,   44.0,   36.0,    2.0,    0.0,
        2.0,    1.0,    3.0,   54.0,    9.0,   20.0,    0.0,
        2.0,    1.0,    3.0,   59.0,   87.0,   51.0,    0.0,
        3.0,    1.0,    4.0,   69.0,    5.0,   18.0,    0.0,
        3.0,    1.0,    6.0,   50.0,   22.0,   90.0,    0.0,
        3.0,    1.0,    8.0,   62.0,    4.0,   84.0,    0.0,
        4.0,    1.0,    7.0,   68.0,   15.0,  164.0,    0.0,
        4.0,    1.0,    3.0,   39.0,    4.0,   19.0,    0.0,
        4.0,    1.0,    6.0,   49.0,   11.0,   43.0,    0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    int   n_coef_input = 8;
    static float estimates[8] = {-1.25, 0.0, 0.0, 0.0,
        0.0, -0.6, 0.0, 0.0};

    int   n_coef;
    float *coef_stat, *means, *cov;
    float criterion, *last_step;

    char *fmt = "%12.4f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_INITIAL_EST_INPUT, n_coef_input, estimates,
        IMSLS_MAX_ITERATIONS, 0,
        IMSLS_COEF_STAT, &coef_stat,
        IMSLS_MEANS, &means,
        IMSLS_COV, &cov,
        IMSLS_CRITERION, &criterion,
        IMSLS_LAST_STEP, &last_step,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);

    imsls_f_write_matrix("Covariate Means", 1, n_coef-1, means, 0);

    imsls_f_write_matrix("Hessian", n_coef, n_coef, cov,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_PRINT_UPPER,
        0);

    printf("\nLog-Likelihood = %12.5f\n", criterion);

    imsls_f_write_matrix("Newton-Raphson Step", 1, n_coef, last_step,
        IMSLS_WRITE_FORMAT, fmt, 0);

}
```

Output

```
                 Coefficient Statistics
 coefficient          s.e.           z           p
     -1.2500        1.3833     -0.9036      0.3664
      0.0000        0.4288      0.0000      1.0000
      0.0000        0.5299      0.0000      1.0000
      0.0000        0.7748      0.0000      1.0000
      0.0000        0.4051      0.0000      1.0000
     -0.6000        0.1118     -5.3652      0.0000
      0.0000        0.0215      0.0000      1.0000
      0.0000        0.0109      0.0000      1.0000

                            Covariate Means
        1           2           3           4           5           6
     0.35        0.28        0.12        0.53        5.65       56.58

        7
    15.65


                            Hessian
            1            2            3            4            5
1      1.9136      -0.0906      -0.1641      -0.1681       0.0778
2                   0.1839       0.0996       0.1191       0.0358
3                                0.2808       0.1264      -0.0226
4                                             0.6003       0.0460
5                                                          0.1641

            6            7            8
1     -0.0818      -0.0235      -0.0012
2     -0.0005      -0.0008       0.0006
3      0.0104       0.0005      -0.0021
4      0.0193      -0.0016       0.0007
5      0.0060      -0.0040       0.0017
6      0.0125       0.0000       0.0003
7                   0.0005      -0.0001
8                                0.0001

Log-Likelihood =    -206.68349

                       Newton-Raphson Step
          1            2            3            4            5
     0.1706      -0.3365       0.1333       1.2967       0.2985

          6            7            8
```

```
0.0625      -0.0112      -0.0026
```

Warning Errors

`IMSLS_CONVERGENCE_ASSUMED_1`	Too many step halvings. Convergence is assumed.
`IMSLS_CONVERGENCE_ASSUMED_2`	Too many step iterations. Convergence is assumed.
`IMSLS_NO_PREDICTED_1`	“estimates[0]” > 1.0. The expected value for the log logistic distribution (“model” = 4) does not exist. Predicted values will not be calculated.
`IMSLS_NO_PREDICTED_2`	“estimates[0]” > 1.0. The expected value for the log extreme value distribution(“model” = 8) does not exist. Predicted values will not be calculated.
`IMSLS_NEG_EIGENVALUE`	The Hessian has at least one negative eigenvalue. An upper bound on the absolute value of the minimum eigenvalue is # corresponding to variable index #.
`IMSLS_INVALID_FAILURE_TIME_4`	“x[#][“ilt”= #]” = # and “x[#][“irt”= #]” = #. The censoring interval has length 0.0. The censoring code for this observation is being set to 0.0.

Fatal Error

`IMSLS_MAX_CLASS_TOO_SMALL`	The number of distinct values of the classification variables exceeds “max_class” = #.
`IMSLS_TOO_FEW_COEF`	`IMSLS_INITIAL_EST_INPUT` is specified, and “n_coef_input” = #. The model specified requires # coefficients.
`IMSLS_TOO_FEW_VALID_OBS`	“n_observations” = # and “n_rows_missing” = #. “n_observations”–”n_rows_missing” must be greater than or equal to 2 in order to estimate the coefficients.
`IMSLS_SVGLM_1`	For the exponential model (“model” = 0) with “n_effects” = # and no intercept, “n_coef” has been determined to equal 0. With no coefficients in the model, processing cannot continue.
`IMSLS_INCREASE_LP_MAX`	Too many observations are to be deleted from the model. Either use a different model or increase the workspace.

`IMSLS_INVALID_DATA_8`	“n_class_values[#]” = #. The number of distinct values for each classification variable must be greater than one.

survival_estimates

Estimates survival probabilities and hazard rates for the various parametric models.

Synopsis

#include `<imsls.h>`

int `*imsls_f_survival_estimates` (*Imsls_f_survival* `*survival_info`,
int `n_observations`, *float* `xpt[]`, *float* `time`, *int* `npt`, *float* `delta`,
…, 0)

The type *double* function is `imsls_d_survival_estimates`.

Required Arguments

Imsls_f_survival `*survival_info` (Input)
Pointer to structure of type *Imsls_f_survival* containing the estimated survival coefficients and other related information. See `imsls_f_survival_glm`.

int `n_observations` (Input)
Number of observations for which estimates are to be calculated.

float `xpt[]` (Input)
Array `xpt` is an array of size `n_observations` by `x_col_dim` containing the groups of covariates for which estimates are desired, where `x_col_dim` is described in the documentation for `imsls_f_survival_glm`. The covariates must be specified exactly as in the call to `imsls_f_survival_glm` which produced `survival_info`.

float `time` (Input)
Beginning of the time grid for which estimates are desired. Survival probabilities and hazard rates are computed for each covariate vector over the grid of time points `time` + *i*`*delta` for *i* = 0, 1, …, `npt` − 1.

int `npt` (Input)
Number of points on the time grid for which survival probabilities are desired.

float `delta` (Input)
Increment between time points on the time grid.

Return Value

An array of size `npt` by (2 ∗ `n_observations` + 1) containing the estimated survival probabilities for the covariate groups specified in `xpt`. Column 0 contains the survival time. Columns 1 and 2 contain the estimated survival probabilities and hazard rates, respectively, for the covariates in the first row of `xpt`. In general, the survival and hazard for row i of `xpt` is contained in columns $2i - 1$ and $2i$, respectively, for $i = 1, 2, \ldots,$ `npt`.

Synopsis with Optional Arguments

#include <imsls.h>

int *imsls_f_survival_estimates (*Imsls_f_survival* survival_info,
int n_observations, *float* xpt[], *float* time, *int* npt, *float* delta,
IMSLS_XBETA, *float* **xbeta,
IMSLS_XBETA_USER, *float* xbeta[],
IMSLS_RETURN_USER, *float* sprob[],
0)

Optional Arguments

IMSLS_XBETA, *float* **xbeta (Output)
Address of a pointer to an array of length n_observations containing the estimated linear response

$$w + x\hat{\beta}$$

for each row of xpt.

IMSLS_XBETA_USER, *float* xbeta[] (Output)
Storage for array xbeta is provided by the user. See IMSLS_XBETA.

IMSLS_RETURN_USER, *float* sprob[] (Output)
User supplied array of size npt by (2 ∗ n_observations + 1) containing the estimated survival probabilities for the covariate groups specified in xpt. Column 0 contains the survival time. Columns 1 and 2 contain the estimated survival probabilities and hazard rates, respectively, for the covariates in the first row of xpt. In general, the survival and hazard for row i of xpt is contained in columns $2i - 1$ and $2i$, respectively, for $i = 1, 2, \ldots,$ npt.

Description

Function imsls_f_survival_estimates computes estimates of survival probabilities and hazard rates for the parametric survival/reliability models fit by function imsls_f_survival_glm.

Let $\eta = x^T\beta$ be the linear parameterization, where x is the design vector corresponding to a row of xpt (imsls_f_survival_estimates generates the design vector using function imsls_f_regressors_for_glm), and β is a vector of parameters associated with the linear model. Let T denote the random response variable and $S(t)$ denote the probability that $T > t$. All models considered also allow a fixed parameter w (input in column ifix of xpt). Use of the parameter is discussed in function imsls_f_survival_glm. There also may be nuisance parameters $\theta > 0$ or $\sigma > 0$. Let Φ denote the cumulative normal distribution. The survival models available in imsls_f_survival_estimates are:

Model	**Name**	***S* (*t*)**
0	Exponential	$\exp\left[-t\exp(w_i+\eta)\right]$
1	Linear hazard	$\exp\left[-\left(t+\frac{\theta t^2}{2}\right)\exp(w_i+\eta)\right]$
2	Log-normal	$1-\Phi\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)$
3	Normal	$1-\Phi\left(\frac{t-\eta-w_i}{\sigma}\right)$
4	Log-logistic	$\{1+\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}^{-1}$
5	Logistic	$\{1+\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}^{-1}$
6	Log least extreme value	$\exp\{-\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}$
7	Least extreme value	$\exp\{-\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}$
8	Log extreme value	$1-\exp\{-\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}$
9	Extreme value	$1-\exp\{-\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}$
10	Weibull	$\exp\{-\left[\frac{t}{\exp(w_i+\eta)}\right]^{\theta}\}$

Let $\lambda(t)$ denote the hazard rate at time t. Then $\lambda(t)$ and $S(t)$ are related at

$$S(t)=\exp(\int_{-\infty}^{t}\lambda(s)\,ds)$$

Models 0, 1, 2, 4, 6, 8, and 10 require that $T > 0$ (in which case assume $\lambda(s) = 0$ for $s < 0$), while the remaining models allow arbitrary values for T, $-\infty < T < \infty$. The computations proceed in function `imsls_f_survival_estimates` as follows:

1. The input arguments are checked for consistency and validity.
2. For each row of `xpt`, the explanatory variables are generated from the classification and variables and the covariates using function `imsls_f_regressors_for_glm` (See Chapter 2, "Regression") with

`dummy_method` = `IMSLS_LEAVE_OUT_LAST`. Given the explanatory variables x, η is computed as $\eta = x^T\beta$, where β is input in `survival_info`.

3. For each point requested in the time grid, the survival probabilities and hazard rates are computed.

Example

This example is a continuation of the first example given for function imsls_f_survival_glm. Prior to calling `survival_estimates`, `imsls_f_survival_glm` is invoked to compute the parameter estimates (contained in the structure `survival_info`). The example is taken from Lawless (1982, p. 287) and involves the mortality of patients suffering from lung cancer.

```
#include <imsls.h>
#include <stdlib.h>
main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,     5.0,  411.0,     0.0,
        1.0,    0.0,    6.0,   63.0,     9.0,  126.0,     0.0,
        1.0,    0.0,    7.0,   65.0,    11.0,  118.0,     0.0,
        1.0,    0.0,    4.0,   69.0,    10.0,   92.0,     0.0,
        1.0,    0.0,    4.0,   63.0,    58.0,    8.0,     0.0,
        1.0,    0.0,    7.0,   48.0,     9.0,   25.0,     1.0,
        1.0,    0.0,    7.0,   48.0,    11.0,   11.0,     0.0,
        2.0,    0.0,    8.0,   63.0,     4.0,   54.0,     0.0,
        2.0,    0.0,    6.0,   63.0,    14.0,  153.0,     0.0,
        2.0,    0.0,    3.0,   53.0,     4.0,   16.0,     0.0,
        2.0,    0.0,    8.0,   43.0,    12.0,   56.0,     0.0,
        2.0,    0.0,    4.0,   55.0,     2.0,   21.0,     0.0,
        2.0,    0.0,    6.0,   66.0,    25.0,  287.0,     0.0,
        2.0,    0.0,    4.0,   67.0,    23.0,   10.0,     0.0,
        3.0,    0.0,    2.0,   61.0,    19.0,    8.0,     0.0,
        3.0,    0.0,    5.0,   63.0,     4.0,   12.0,     0.0,
        4.0,    0.0,    5.0,   66.0,    16.0,  177.0,     0.0,
        4.0,    0.0,    4.0,   68.0,    12.0,   12.0,     0.0,
        4.0,    0.0,    8.0,   41.0,    12.0,  200.0,     0.0,
        4.0,    0.0,    7.0,   53.0,     8.0,  250.0,     0.0,
        4.0,    0.0,    6.0,   37.0,    13.0,  100.0,     0.0,
        1.0,    1.0,    9.0,   54.0,    12.0,  999.0,     0.0,
        1.0,    1.0,    5.0,   52.0,     8.0,  231.0,     1.0,
        1.0,    1.0,    7.0,   50.0,     7.0,  991.0,     0.0,
        1.0,    1.0,    2.0,   65.0,    21.0,    1.0,     0.0,
        1.0,    1.0,    8.0,   52.0,    28.0,  201.0,     0.0,
        1.0,    1.0,    6.0,   70.0,    13.0,   44.0,     0.0,
        1.0,    1.0,    5.0,   40.0,    13.0,   15.0,     0.0,
        2.0,    1.0,    7.0,   36.0,    22.0,  103.0,     1.0,
        2.0,    1.0,    4.0,   44.0,    36.0,    2.0,     0.0,
        2.0,    1.0,    3.0,   54.0,     9.0,   20.0,     0.0,
        2.0,    1.0,    3.0,   59.0,    87.0,   51.0,     0.0,
        3.0,    1.0,    4.0,   69.0,     5.0,   18.0,     0.0,
        3.0,    1.0,    6.0,   50.0,    22.0,   90.0,     0.0,
        3.0,    1.0,    8.0,   62.0,     4.0,   84.0,     0.0,
        4.0,    1.0,    7.0,   68.0,    15.0,  164.0,     0.0,
        4.0,    1.0,    3.0,   39.0,     4.0,   19.0,     0.0,
        4.0,    1.0,    6.0,   49.0,    11.0,   43.0,     0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};

    int   n_observations = 40;
    int   n_estimates = 2;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    float time = 10.0;
    int   npt = 10;
    float delta = 20.0;

    int   n_coef;
    float *sprob;
    Imsls_f_survival *survival_info;
    char *fmt = "%12.2f%10.4f%10.6f%10.4f%10.6f";
    char *clabels[] = {"", "Time", "S1", "H1", "S2", "H2"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous,
        model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_SURVIVAL_INFO, &survival_info,
        0);

    sprob = imsls_f_survival_estimates(survival_info, n_estimates,
        &x[0][0], time, npt, delta, 0);

    imsls_f_write_matrix("Survival and Hazard Estimates",
        npt, 2*n_estimates+1, sprob,
        IMSLS_WRITE_FORMAT, fmt, IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels, 0);

    free (survival_info);
    free (sprob);
}
```

Output

```
               Survival and Hazard Estimates

        Time        S1         H1         S2         H2
       10.00    0.9626   0.003807     0.9370   0.006503
       30.00    0.8921   0.003807     0.8228   0.006503
       50.00    0.8267   0.003807     0.7224   0.006503
       70.00    0.7661   0.003807     0.6343   0.006503
       90.00    0.7099   0.003807     0.5570   0.006503
      110.00    0.6579   0.003807     0.4890   0.006503
      130.00    0.6096   0.003807     0.4294   0.006503
      150.00    0.5649   0.003807     0.3770   0.006503
      170.00    0.5235   0.003807     0.3310   0.006503
      190.00    0.4852   0.003807     0.2907   0.006503
```

Note that the hazard rate is constant over time for the exponential model.

Warning Errors

`IMSLS_CONVERGENCE_ASSUMED_1`	Too many step halvings. Convergence is assumed.
`IMSLS_CONVERGENCE_ASSUMED_2`	Too many step iterations. Convergence is assumed.
`IMSLS_NO_PREDICTED_1`	"estimates[0]" > 1.0. The expected value for the log logistic distribution ("model" = 4) does not exist. Predicted values will not be calculated.
`IMSLS_NO_PREDICTED_2`	"estimates[0]" > 1.0. The expected value for the log extreme value distribution ("model" = 8) does not exist. Predicted values will not be calculated.
`IMSLS_NEG_EIGENVALUE`	The Hessian has at least one negative eigenvalue. An upper bound on the absolute value of the minimum eigenvalue is # corresponding to variable index #.
`IMSLS_INVALID_FAILURE_TIME_4`	"x[#]["ilt"= #]" = # and "x[#]["irt"= #]" = #. The censoring interval has length 0.0. The censoring code for this observation is being set to 0.0.

Fatal Error

`IMSLS_MAX_CLASS_TOO_SMALL`	The number of distinct values of the classification variables exceeds "max_class" = #.
`IMSLS_TOO_FEW_COEF`	`IMSLS_INITIAL_EST_INPUT` is specified, and "n_coef_input" = #. The model specified requires # coefficients.
`IMSLS_TOO_FEW_VALID_OBS`	"n_observations" = %(i1) and "n_rows_missing" = #. "n_observations"–"n_rows_missing" must be greater than or equal to 2 in order to estimate the coefficients.
`IMSLS_SVGLM_1`	For the exponential model ("model" = 0) with "n_effects" = # and no intercept, "n_coef" has been determined to equal 0. With no coefficients in the model, processing cannot continue.

`IMSLS_INCREASE_LP_MAX`	Too many observations are to be deleted from the model. Either use a different model or increase the workspace.
`IMSLS_INVALID_DATA_8`	"n_class_values[#]" = #. The number of distinct values for each classification variable must be greater than one.

nonparam_hazard_rate

Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.

Synopsis

#include `<imsls.h>`

float `*imsls_f_nonparam_hazard_rate` (*int* `n_observations`,
float `t[]`, *int* `n_hazard`, *float* `hazard_min`,
float `hazard_increment`, ..., 0)

The type *double* function is `imsls_d_nonparam_hazard_rate`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `t[]` (Input)
An array of `n_observations` containing the failure times. If optional argument `IMSLS_CENSOR_CODES` is used, the values of `t` may be treated as exact failure times, as right-censored times, or a combination of exact and right censored times. By default, all times in `t` are assumed to be exact failure times.

int `n_hazard` (Input)
Number of grid points at which to compute the hazard. The function computes the hazard rates over the range given by:
`hazard_min` + *j* * `hazard_increment`, for *j* = 0, …, `n_hazard` – 1.

float `hazard_min` (Input)
First grid value.

float `hazard_increment` (Input)
Increment between grid values.

Return Value

Pointer to an array of length `n_hazard` containing the estimated hazard rates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_nonparam_hazard_rate` (*int* `n_observations`,
float `t[]`, *int* `n_hazard`, *float* `hazard_min`,

float hazard_increment
IMSLS_RETURN_USER, *float* haz[],
IMSLS_PRINT_LEVEL, *int* iprint,
IMSLS_CENSOR_CODES, *int* censor_codes[],
IMSLS_WEIGHT, *int* iwto,
IMSLS_SORT_OPTION, *int* isort,
IMSLS_K_GRID, *int* n_k, *float* k_min, *float* k_increment,
IMSLS_BETA_GRID, *int* n_beta_grid, *float* beta_start,
float beta_increment,
IMSLS_N_MISSING, *int* *nmiss,
IMSLS_ALPHA, *float* *alpha,
IMSLS_BETA, *float* *beta,
IMSLS_CRITERION, *float* *vml,
IMSLS_K, *int* *k,
IMSLS_SORTED_EVENT_TIMES, *float* **event_times,
IMSLS_SORTED_EVENT_TIMES_USER, *float* event_times[],
IMSLS_SORTED_CENSOR_CODES, *int* **isorted_censor,
IMSLS_SORTED_CENSOR_CODES_USER, *int* isorted_censor[],
0)

Optional Arguments

IMSLS_RETURN_USER, *float* haz[] (Output)
If specified, haz is a user supplied array of length n_hazard containing the estimated hazard rates.

IMSLS_PRINT_LEVEL, *int* iprint (Input)
Printing option. Default: iprint = 0.

iprint	**Action**
0	No printing is performed.
1	The grid estimates and the optimized estimates are printed for each value of *k*.

IMSLS_CENSOR_CODES, *int* censor_codes[] (Input)
censor_codes is an array of length n_observations containing the censoring codes for each time in t. If censor_codes[i]=0 the failure time t[i] is treated as an exact time of failure. Otherwise it is treated as a right-censored time; that is, the exact time of failure is greater than t[i].
Default: All failure times are treated as exact times of failure with no censoring.

IMSLS_WEIGHT_OPTION, *int* iwto (Input)
Weight option . If iwto = 1, then
$\texttt{weight} = \ln\left(1+1/\left(\texttt{n\_observations}-i\right)\right)$ is used for the *i*-th smallest observation.
Otherwise, $\texttt{weight} = 1/\left(\texttt{n\_observations}-i\right)$ is used.
Default: iwto = 0.

`IMSLS_SORT_OPTION`, *int* `isort` (Input)
: Sorting option . If `isort` = 1, then the event times are not automatically sorted by the function. Otherwise, sorting is performed with exact failure times following tied right-censored times.
Default: `isort` = 0.

`IMSLS_K_GRID`, *int* `n_k`, *float* `k_min`, *float* `k_increment` (Input)
: Finds the optimal value of k over the range given by: `kmin` + (j – 1) * `k_increment`, for j = 1, …, `n_k`. Where `n_k` is the number of values of k to be considered. `k_min` is the minimum value for parameter k. `k_increment` is the increment between successive values of parameter k. Parameter k is the number of nearest neighbors to be used in computing the k-th nearest neighbor distance.
Default: `k_min` is the smallest possible value of k, `k_increment` =2, and `n_k` will be at most 10 points.

`IMSLS_BETA_GRID`, *int* `n_beta_grid`, *float* `beta_start`, *float* `beta_increment` (Input)
: For `n_beta_grid` > 0, a user-defined grid is used. This grid is defined as `beta_start` + (j – 1)*`beta_increment`, for j = 1, …, `n_beta_grid`. `beta_start` is the first value to be used in the user-defined grid and `beta_increment` is the increment between successive grid values of `beta`.
Default: The values in the initial beta search are given as follows: Let $\beta^* = -8, -4, -2, -1, -0.5, 0.5, 1$, and 2, and

$$\beta = e^{-\beta^*}$$

For each value of β, `vml` is computed at the optimizing β. The maximizing β is used to initiate the iterations. If the initial β^* is determined from the search to be less than −6, then it is presumed that β is infinite, and an analytic estimate of α based upon infinite β is used. Infinite β corresponds to a flat hazard rate.

`IMSLS_N_MISSING`, *int* `*nmiss` (Output)
: Number of missing (NaN, not a number) failure times in `t`.

`IMSLS_ALPHA`, *float* `*alpha` (Output)
: Optimal estimate for the parameter α.

`IMSLS_BETA`, *float* `*beta` (Output)
: Optimal estimate for the parameter β.

`IMSLS_CRITERION`, *float* `*vml` (Output)
: Optimum value of the criterion function.

`IMSLS_K`, *int* `*k` (Output)
: Optimal estimate for the parameter k.

`IMSLS_SORTED_EVENT_TIMES`, *float* `**event_times` (Output)
: Address of a pointer to an array of length `n_observations` containing the times of occurrence of the events, sorted from smallest to largest.

`IMSLS_SORTED_EVENT_TIMES_USER`, *float* `event_times[]` (Output)
Storage for `event_times` is provided by the user. See `IMSLS_SORTED_EVENT_TIMES`.

`IMSLS_SORTED_CENSOR_CODES`, *int* `**isorted_censor` (Output)
Address of a pointer to an array of length `n_observations` containing the sorted censor codes. Censor codes are sorted corresponding to the events `event_times[i]`, with censored observations preceding tied failures.

`IMSLS_SORTED_CENSOR_CODES_USER`, *int* `isorted_censor[]` (Output)
Storage for `isorted_censor` is provided by the user. See `IMSLS_SORTED_CENSOR_CODE`.

Description

Function `imsls_f_nonparam_hazard_rate` is an implementation of the methods discussed by Tanner and Wong (1984) for estimating the hazard rate in survival or reliability data with right censoring. It uses the biweight kernel,

$$K(x) = \begin{cases} \frac{15}{16}(1-x^2)^2 & \text{for} \left|x\right| < 1 \\ 0 & \text{elsewhere} \end{cases}$$

and a modified likelihood to obtain data-based estimates of the smoothing parameters α, β, and *k* needed in the estimation of the hazard rate. For kernel *K*(*x*), define the “smoothed” kernel
$K_s(x - x(j)$ as follows:

$$K_S(x - x_{(j)}) = \frac{1}{\alpha d_{jk}} K\left(\frac{x - x(j)}{\beta d_{jk}}\right)$$

where d_{jk} is the distance to the *k*-th nearest failure from *x*(*j*), and *x*(*j*) is the *j*-th ordered observation (from smallest to largest). For given α and β, the hazard at point *x* is then

$$h(x) = \sum_{i=1}^{N} \{(1-\delta_i) w_i K_s(x - x_{(i)})\}$$

where $N =$ `n_observations`, δ_i is the *i*-th observation’s censor code (1 = censored, 0 = failed), and w_i is the *i*-th ordered observation’s weight, which may be chosen as either $1/(N - i + 1)$, or
$\ln(1 + 1/(N - i + 1))$. Let

$$H(x) = \int_0^x h(s)\, ds$$

The likelihood is given by

$$L = \prod\nolimits_{i=1}^{N} \{h(x_i)^{(1-\delta_i)} \exp(-H(x_{(i)}))\},$$

where Π denotes product. Since the likelihood leads to degenerate estimates, Tanner and Wong (1984) suggest the use of a modified likelihood. The modification consists of deleting observation x_i in the calculation of $h(x_i)$ and $H(x_i)$ when the likelihood term for x_i is computed using the usual optimization techniques. α and β for given k can then be estimated.

Estimates for α and β are computed as follows: for given β, a closed form solution is available for α. The problem is thus reduced to the estimation of β.
A grid search for β is first performed. Experience indicates that if the initial estimate of β from this grid search is greater than, say, e^6, then the modified likelihood is degenerate because the hazard rate does not change with time. In this situation, β should be taken to be infinite, and an estimate of α corresponding to infinite β should be directly computed. When the estimate of β from the grid search is less than e^6, a secant algorithm is used to optimize the modified likelihood. The secant algorithm iteration stops when the change in β from one iteration to the next is less than 10^{-5}. Alternatively, the iterations may cease when the value of β becomes greater than e^6, at which point an infinite β with a degenerate likelihood is assumed.

To find the optimum value of the likelihood with respect to k, a user-specified grid of k-values is used. For each grid value, the modified likelihood is optimized with respect to α and β. That grid point, which leads to the smallest likelihood, is taken to be the optimal k.

Programming Notes

1. If sorting of the data is performed by `imsls_f_nonparam_hazard_rate`, then the sorted array will be such that all censored observations at a given time precede all failures at that time. To specify an arbitrary pattern of censored/failed observations at a given time point, the `isort` = 1 option must be used. In this case, it is assumed that the times have already been sorted from smallest to largest.

2. The smallest value of k must be greater than the largest number of tied failures since d_{jk} must be positive for all j. (Censored observations are not counted.) Similarly, the largest value of k must be less than the total number of failures. If the grid specified for k includes values outside the allowable range, then a warning error is issued; but k is still optimized over the allowable grid values.

3. The secant algorithm iterates on the transformed parameter $\beta^* = \exp(-\beta)$. This assures a positive β, and it also seems to lead to a more desirable grid search. All results returned to the user are in the original parameterization, however.

4. Since local minimums have been observed in the modified likelihood, it is recommended that more than one grid of initial values for α and β be used.

5. Function `imsls_f_nonparam_hazard_rate` assumes that the hazard grid points are new data points.

Example

The following example is taken from Tanner and Wong (1984). The data are from Stablein, Carter, and Novak (1981) and involve the survival times of individuals with nonresectable gastric carcinoma. Only individuals treated with both radiation and chemotherapy are used. For each value of k from 18 to 22 with increment of 2, the default grid search for β is performed. Using the optimal value of β in the grid, the optimal parameter estimates of α and β are computed for each value of *k*. The final solution is the parameter estimates for the value of *k* which optimizes the modified likelihood (`vml`). Because the `iprint` = 1 is in effect, `imsls_f_nonparam_hazard_rate` prints all of the results in the output.

```
#include "imsls.h"

void main ()
{
  int n_observations = 45, iprint = 1, kmin = 18;
  int increment_k = 2, n_k = 3, isort = 1, nmiss, *isorted_censor;
  float *event_times, *haz;
  int n_hazard=100;
  float hazard_min = 0.0, hazard_inc = 10;

  float t[] = { 17.0, 42.0, 44.0, 48.0, 60.0, 72.0, 74.0, 95.0,
                       103.0, 108.0, 122.0, 144.0, 167.0, 170.0, 183.0,
                       185.0, 193.0, 195.0, 197.0, 208.0, 234.0, 235.0,
                       254.0, 307.0, 315.0, 401.0, 445.0, 464.0, 484.0,
                       528.0, 542.0, 567.0, 577.0, 580.0, 795.0, 855.0,
                       882.0, 892.0,1031.0,1033.0,1306.0,1335.0,1366.0,
                       1452.0, 1472.0};
  float censor_codes[] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 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, 1.0};

  haz = imsls_f_nonparam_hazard_rate I  (n_observations, t,
                                         n_hazard, hazard_min, hazard_inc,
                                         IMSLS_K_GRID, n_k, kmin,
                                               increment_k,
                                         IMSLS_PRINT_LEVEL, iprint,
                                         IMSLS_N_MISSING, &nmiss,
                                         IMSLS_SORT_OPTION, isort,
                                         IMSLS_CENSOR_CODES, censor_codes,
                                         IMSLS_SORTED_EVENT_TIMES,
                                               &event_times,
                                         IMSLS_SORTED_CENSOR_CODES,
                                               &isorted_censor,
                                         0);

  printf ("\nnmiss = %d\n", nmiss);
  imsls_f_write_matrix ("Sorted Event Times", 1, n_observations,
                       event_times, IMSLS_WRITE_FORMAT, "%7.1f", 0);
  imsls_i_write_matrix ("Sorted Censors", 1, n_observations,
                       isorted_censor, 0);
  imsls_f_write_matrix ("Hazard Rates", 1, n_hazard, haz, 0);

}
```

Output

```
                  *** Grid search for k =    18 ***
          alpha                   beta                   vml
        4.57832                2980.96              -266.805
        4.54312                54.5982               -266.62
        4.33646                20.0855              -265.541
        4.01933                12.1825              -264.001
        3.54274                7.38906               -262.54
        2.99058                4.48169              -262.512
        2.35154                2.71828              -262.634
        1.58417                1.64872              -262.158
       0.966332                      1              -262.868

                 *** Optimal parameter estimates ***
          alpha                   beta                   vml
        1.69515                1.76926              -262.119

                  *** Grid search for k =    20 ***
          alpha                   beta                   vml
        4.05393                2980.96              -266.526
        4.03284                54.5982              -266.401
        3.90505                20.0855              -265.648
        3.68782                12.1825              -264.402
        3.30434                7.38906              -262.666
        2.82272                4.48169               -262.08
        2.25276                2.71828              -262.445
        1.55578                1.64872              -261.772
       0.955586                      1              -262.618

                 *** Optimal parameter estimates ***
          alpha                   beta                   vml
        1.54053                1.63155              -261.771

    *** Grid search for k =    22 ***
          alpha                   beta                   vml
        3.65641                2980.96              -267.595
        3.64159                54.5982              -267.499
        3.55056                20.0855              -266.904
        3.38875                12.1825              -265.859
        3.07147                7.38906              -264.066
        2.64504                4.48169              -263.039
         2.1374                2.71828              -263.335
        1.51261                1.64872               -262.64
       0.936368                      1              -262.683

                 *** Optimal parameter estimates ***
          alpha                   beta                   vml
        1.34217                1.45001              -262.561

             *** The final solution     (k =    20) ***
          alpha                   beta                   vml
        1.54053                1.63155              -261.771

nmiss = 0
```

```
                         Sorted Event Times
       1          2          3          4          5          6          7          8
    17.0       42.0       44.0       48.0       60.0       72.0       74.0       95.0

       9         10         11         12         13         14         15         16
   103.0      108.0      122.0      144.0      167.0      170.0      183.0      185.0

      17         18         19         20         21         22         23         24
   193.0      195.0      197.0      208.0      234.0      235.0      254.0      307.0

      25         26         27         28         29         30         31         32
   315.0      401.0      445.0      464.0      484.0      528.0      542.0      567.0

      33         34         35         36         37         38         39         40
   577.0      580.0      795.0      855.0      882.0      892.0     1031.0     1033.0

      41         42         43         44         45
  1306.0     1335.0     1366.0     1452.0     1472.0

                               Sorted Censors
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

20 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1

39  40  41  42  43  44  45
1    1   1   1   1   1   1

                            Hazard Rates
          1            2            3            4            5            6
   0.000962     0.001111     0.001276     0.001451     0.001634     0.001819

          7            8            9           10           11           12
   0.002004     0.002185     0.002359     0.002523     0.002675     0.002813

         13           14           15           16           17           18
   0.002935     0.003040     0.003126     0.003193     0.003240     0.003266

         19           20           21           22           23           24
   0.003273     0.003260     0.003229     0.003179     0.003114     0.003034

         25           26           27           28           29           30
   0.002941     0.002838     0.002727     0.002612     0.002495     0.002381

         31           32           33           34           35           36
   0.002273     0.002175     0.002084     0.001998     0.001917     0.001841

         37           38           39           40           41           42
   0.001771     0.001709     0.001655     0.001608     0.001569     0.001537

         43           44           45           46           47           48
   0.001510     0.001484     0.001459     0.001435     0.001411     0.001388
```

```
       49          50          51          52          53          54
 0.001365    0.001343    0.001323    0.001304    0.001285    0.001266

       55          56          57          58          59          60
 0.001247    0.001228    0.001208    0.001188    0.001167    0.001146

       61          62          63          64          65          66
 0.001125    0.001103    0.001081    0.001060    0.001040    0.001020

       67          68          69          70          71          72
 0.000999    0.000979    0.000958    0.000936    0.000913    0.000891

       73          74          75          76          77          78
 0.000868    0.000845    0.000821    0.000798    0.000775    0.000752

       79          80          81          82          83          84
 0.000730    0.000708    0.000685    0.000662    0.000640    0.000617

       85          86          87          88          89          90
 0.000595    0.000573    0.000552    0.000530    0.000510    0.000490

       91          92          93          94          95          96
 0.000471    0.000452    0.000434    0.000416    0.000399    0.000383

       97          98          99         100
 0.000366    0.000351    0.000336    0.000321
```

Fatal Errors

`IMSLS_ALL_OBSERVATIONS_MISSING`

All observations are missing (NaN, not a number) values.

life_tables

Produces population and cohort life tables.

Synopsis

#include `<imsls.h>`

float `*imsls_f_life_tables` (*int* `n_classes`, *float* `age[]`, *float* `a[]`,
int `n_cohort[]`, ..., 0)

The type *double* function is `imsls_d_life_tables`.

Required Arguments

int `n_classes` (Input)
Number of age classes.

float `age[]` (Input)
Array of length `n_classes` + 1 containing the lowest age in each age interval, and in `age[n_classes]`, the endpoint of the last age interval. Negative `age[0]` indicates that the age intervals are all of length `|age[0]|`

and that the initial age interval is from 0.0 to `|age[0]|`. In this case, all other elements of `age` need not be specified. `age[n_classes]` need not be specified when getting a cohort table.

float `a[]` (Input)
Array of length `n_classes` containing the fraction of those dying within each interval who die before the interval midpoint. A common choice for all `a[i]` is 0.5. This choice may also be specified by setting `a[0]` to any negative value. In this case, the remaining values of `a` need not be specified.

int `n_cohort[]` (Input)
Array of length `n_classes` containing the cohort sizes during each interval. If the `IMSL_POPULATION_LIFE_TABLE` option is used, then `n_cohort[i]` contains the size of the population at the midpoint of interval `i`. Otherwise, `n_cohort[i]` contains the size of the cohort at the beginning of interval `i`. When requesting a population table, the population sizes in `n_cohort` may need to be adjusted to correspond to the number of deaths in `n_deaths`. See the Description section for more information.

Return Value

Pointer to an array of length `n_classes` by 12 containing the life table. The function returns a cohort table by default. If the `IMSL_POPULATION_LIFE_TABLE` option is used, a population table is returned. Entries in the *i*th row are for the age interval defined by `age[i]`. Column definitions are described in the following table.

Column	Description
0	Lowest age in the age interval.
1	Fraction of those dying within the interval who die before the interval midpoint.
2	Number surviving to the beginning of the interval.
3	Number of deaths in the interval.
4	Death rate in the interval. For cohort table, this column is set to NaN (not a number).
5	Proportion dying in the interval.
6	Standard error of the proportion dying in the interval.
7	Proportion of survivors at the beginning of the interval.
8	Standard error of the proportion of survivors at the beginning of the interval.
9	Expected lifetime at the beginning of the interval.
10	Standard error of the expected life at the beginning of the interval.
11	Total number of time units lived by all of the population in the interval.

Synopsis with Optional Arguments

#include `<imsls.h>`

float * imsls_f_life_tables (*int* n_classes, *float* age[],
float a[], *int* n_cohort[],
IMSLS_RETURN_USER, *float* table[],
IMSLS_PRINT_LEVEL, *int* iprint,
IMSLS_POPULATION_SIZE, *int* initial_pop,
IMSLS_POPULATION_LIFE_TABLE, *int* *n_deaths,
0)

Optional Arguments

IMSLS_RETURN_USER, *float* table[] (Output)
If specified, table is an user-specified array of length n_classes*12 containing the life table.

IMSLS_PRINT_LEVEL, *int* iprint (Input)
Printing option.
Default: iprint = 0.

Iprint	Action
0	No printing is performed.
1	The life table is printed.

IMSLS_POPULATION_SIZE, *int* initial_pop (Input)
The population size at the beginning of the first age interval in requesting population table. A default value of 10,000 is used to allow easy entry of n_cohorts and n_deaths when numbers are available as percentages.
Default: initial_pop = 10000.

IMSLS_POPULATION_LIFE_TABLE, *int* *n_deaths (Input)
Compute a population table. n_deaths is an array of length n_classes containing the number of deaths in each age interval.

Description

Function imsls_f_life_tables computes population (current) or cohort life tables based upon the observed population sizes at the middle (for population table) or the beginning (for cohort table) of some userspecified age intervals. The number of deaths in each of these intervals must also be observed.

The probability of dying prior to the middle of the interval, given that death occurs somewhere in the interval, may also be specified. Often, however, this probability is taken to be 0.5. For a discussion of the probability models underlying the life table here, see the references.

Let t_i, for $i = 0, 1, \ldots, t_n$ denote the time grid defining the n age intervals, and note that the length of the age intervals may vary. Following Gross and Clark (1975, page 24), let d_i denote the number of individuals dying in age interval i, where age interval i ends at time t_i. For population table, the death rate at the middle of the interval is given by $r_i = d_i/(M_i h_i)$, where M_i is the number of individuals alive at the middle of the interval, and $h_i = t_i - t_{i-1}$, $t_0 = 0$. The number of individuals alive at the beginning of the interval may be estimated by $P_i = M_i + (1 - a_i)d_i$ where a_i is the probability that an individual dying in the interval dies prior to the interval midpoint. For cohort table, P_i is input directly while the death rate in the interval, r_i, is not needed.

The probability that an individual dies during the age interval from t_{i-1} to t_i is given by $q_i = d_i/P_i$. It is assumed that all individuals alive at the beginning of the last interval die during the last interval. Thus, $q_n = 1.0$. The asymptotic variance of q_i can be estimated by

$$\sigma_i^2 = q_i(1-q_i)/P_i$$

For population table, the number of individuals alive in the middle of the time interval (input in `n_cohort[i]`) must be adjusted to correspond to the number of deaths observed in the interval. Function `imsls_f_life_tables` assumes that the number of deaths observed in interval h_i occur over a time period equal to h_i. If d_i is measured over a period u_i, where $u_i \neq d_i$, then `n_cohort[i]` must be adjusted to correspond to d_i by multiplication by u_i/h_i, i.e., the value M_i input into `imsls_f_life_tables` as `n_cohort[i]` is computed as

$$M_i^* = M_i u_i / h_i$$

Let S_i denote the number of survivors at time t_i from a hypothetical (for population table) or observed (for cohort table) population. Then, S_0 = `initial_pop` for population table, and S_0 = `n_cohort[0]` for cohort table, and S_i is given by $S_i = S_{i-1} - \delta_{i-1}$ where $\delta_i = S_i q_i$ is the number of individuals who die in the i-th interval. The proportion of survivors in the interval is given by $V_i = S_i/S_0$ while the asymptotic variance of V_i can be estimated as follows.

$$\text{var}(V_i) = V_i^2 \sum_{j=1}^{i-1} \frac{\sigma_j^2}{(1-q_j)^2}$$

The expected lifetime at the beginning of the interval is calculated as the total lifetime remaining for all survivors alive at the beginning of the interval divided by the number of survivors at the beginning of the interval. If e_i denotes this average expected lifetime, then the variance of e_i can be estimated as (see Chiang 1968)

$$\text{var}(e_i) = \frac{\sum_{j=i}^{n-1} P_j^2 \sigma_j^2 [e_{j+1} + h_{j+1}(1-a_j)]^2}{P_j^2}$$

where $\text{var}(e_n) = 0.0$.

Finally, the total number of time units lived by all survivors in the time interval can be estimated as:

$$U_i = h_i[S_i - \delta_i(1-a_i)]$$

Example

This example is taken from Chiang (1968). The cohort life table has thirteen equally spaced intervals, so `age[0]` is set to −5.0. Similarly, the probabilities of death prior to the middle of the interval are all taken to be 0.5, so `a[0]` is set to −1.0. Since `IMSLS_PRINT_LEVEL` option is used, `imsls_f_life_tables` prints the life table.

```
#include "imsls.h"

#define N_CLASSES 13

void main ()
{
  int iprint = 1;
  int n_cohort[] =
    { 270, 268, 264, 261, 254, 251, 248, 232, 166, 130, 76, 34, 13 };
  float age[N_CLASSES + 1], a[N_CLASSES];
  float *result;

  age[0] = -5.0;
  a[0] = -1.0;
  result = imsls_f_life_tables (N_CLASSES, age, a, n_cohort,
                               IMSLS_PRINT_LEVEL, iprint, 0);
}
```

Output

```
                              Life Table
Age Class           Age      PDHALF        Alive       Deaths  Death Rate
        1             0         0.5          270            2  ..........
        2             5         0.5          268            4  ..........
        3            10         0.5          264            3  ..........
        4            15         0.5          261            7  ..........
        5            20         0.5          254            3  ..........
        6            25         0.5          251            3  ..........
        7            30         0.5          248           16  ..........
        8            35         0.5          232           66  ..........
        9            40         0.5          166           36  ..........
       10            45         0.5          130           54  ..........
       11            50         0.5           76           42  ..........
       12            55         0.5           34           21  ..........
       13            60         0.5           13           13  ..........

Age Class          P(D)    Std(P(D))         P(S)    Std(P(S))     Lifetime
        1      0.007407     0.005218            1            0        43.19
        2       0.01493     0.007407       0.9926     0.005218        38.49
        3       0.01136     0.006523       0.9778     0.008971        34.03
        4       0.02682         0.01       0.9667      0.01092         29.4
        5       0.01181     0.006779       0.9407      0.01437        25.14
        6       0.01195     0.006859       0.9296      0.01557        20.41
        7       0.06452       0.0156       0.9185      0.01665        15.63
        8        0.2845      0.02962       0.8593      0.02116        11.53
        9        0.2169      0.03199       0.6148      0.02962        10.12
       10        0.4154      0.04322       0.4815      0.03041        7.231
       11      0.5526     0.05704      0.2815     0.02737       5.592
       12      0.6176     0.08334      0.1259     0.02019       4.412
       13           1           0     0.04815     0.01303         2.5

Age Class   Std(Life)  Time Units
        1      0.6993        1345
        2      0.6707        1330
        3       0.623        1313
        4       0.594        1288
        5      0.5403        1263
        6      0.5237        1248
        7      0.5149        1200
        8      0.4982         995
        9      0.4602         740
       10      0.4328         515
       11      0.4361         275
       12      0.4167       117.5
       13           0        32.5
```

Chapter 11: Probability Distribution Functions and Inverses

Routines

Discrete Random Variables: Distribution Functions and Probability Functions

Distribution Functions		
Binomial distribution function	binomial_cdf	774
Binomial probability function	binomial_pdf	775
Hypergeometric distribution function	hypergeometric_cdf	777
Hypergeometric probability function	hypergeometric_pdf	778
Poisson distribution function	poisson_cdf	779
Poisson probability function	poisson_pdf	781

Continuous Random Variables

Distribution Functions and Their Inverses		
Beta distribution function	beta_cdf	783
Inverse beta distribution function	beta_inverse_cdf	785
Bivariate normal distribution function	bivariate_normal_cdf	786
Chi-squared distribution function	chi_squared_cdf	788
Inverse chi-squared distribution function	chi_squared_inverse_cdf	789
Noncentral chi-squared distribution function	non_central_chi_sq	791
Inverse of the noncentral chi-squared distribution function	non_central_chi_sq_inv	793
F distribution function	F_cdf	794
Inverse *F* distribution function	F_inverse_cdf	796
Gamma distribution function	gamma_cdf	798
Inverse gamma distribution function	gamma_inverse_cdf	799
Normal (Gaussian) distribution function	normal_cdf	801
Inverse normal distribution function	normal_inverse_cdf	802
Student's *t* distribution function	t_cdf	804
Inverse Student's *t* distribution function	t_inverse_cdf	805
Noncentral Students's *t* distribution function	non_central_t_cdf	807

Inverse of the noncentral Student's *t* distribution function non_central_t_inv_cdf 809

Usage Notes

Definitions and discussions of the terms basic to this chapter can be found in Johnson and Kotz (1969, 1970a, 1970b). These are also good references for the specific distributions.

In order to keep the calling sequences simple, whenever possible, the subprograms described in this chapter are written for standard forms of statistical distributions. Hence, the number of parameters for any given distribution may be fewer than the number often associated with the distribution. For example, while a gamma distribution is often characterized by two parameters (or even a third, "location"), there is only one parameter that is necessary, the "shape".
The "scale" parameter can be used to scale the variable to the standard gamma distribution. Also, the functions relating to the normal distribution, `imsls_f_normal_cdf` and `imsls_f_normal_inverse_cdf` , are for a normal distribution with mean equal to zero and variance equal to one. For other means and variances, it is very easy for the user to standardize the variables by subtracting the mean and dividing by the square root of the variance.

The *distribution function* for the (real, single-valued) random variable X is the function F defined for all real x by

$$F(x) = \text{Prob}(X \le x)$$

where Prob(·) denotes the probability of an event. The distribution function is often called the *cumulative distribution function* (CDF).

For distributions with finite ranges, such as the beta distribution, the CDF is 0 for values less than the left endpoint and 1 for values greater than the right endpoint. The subprograms described in this chapter return the correct values for the distribution functions when values outside of the range of the random variable are input, but warning error conditions are set in these cases.

Discrete Random Variables

For discrete distributions, the function giving the probability that the random variable takes on specific values is called the *probability function*, defined by

$$p(x) = \text{Prob}(X = x)$$

The "`PR`" routines described in this chapter evaluate probability functions.

The CDF for a discrete random variable is

$$F(x) = \sum_{A} p(k)$$

where A is the set such that $k \le x$. The "DF" routines in this chapter evaluate cumulative distribution functions. Since the distribution function is a step function, its inverse does not exist uniquely.

Continuous Distributions

For continuous distributions, a probability function, as defined above, would not be useful because the probability of any given point is 0. For such distributions, the useful analog is the *probability density function* (PDF). The integral of the PDF is the probability over the interval, if the continuous random variable X has PDF f, then

$$\text{Prob}(a < X \le b) = \int_a^b f(x)dx$$

The relationship between the CDF and the PDF is

$$F(x) = \int_{-\infty}^{x} f(t)dt \quad .$$

The "`_cdf`" functions described in this chapter evaluate cumulative distribution functions.

For (absolutely) continuous distributions, the value of F(x) uniquely determines x within the support of the distribution. The "`_inverse_cdf`" functions described in this chapter compute the inverses of the distribution functions, that is, given F(x) (called "P" for "probability"), a routine such as `imsls_f_beta_inverse_cdf` computes `x`. The inverses are defined only over the open interval (0,1).

Additional Comments

Whenever a probability close to 1.0 results from a call to a distribution function or is to be input to an inverse function, it is often impossible to achieve good accuracy because of the nature of the representation of numeric values. In this case, it may be better to work with the complementary distribution function (one minus the distribution function). If the distribution is symmetric about some point (as the normal distribution, for example) or is reflective about some point (as the beta distribution, for example), the complementary distribution function has a simple relationship with the distribution function. For example, to evaluate the standard normal distribution at 4.0, using `imsls_f_normal_inverse_cdf` directly, the result to six places is 0.999968. Only two of those digits are really useful, however. A more useful result may be 1.000000 minus this value, which can be obtained to six significant figures as 3.16713E-05 by evaluating `imsls_f_normal_inverse_cdf` at –4.0. For the normal distribution, the two values are related by $\Phi(x) = 1 - \Phi(-x)$, where $\Phi(\cdot)$ is the normal distribution function. Another example is the beta distribution with parameters 2 and 10. This distribution is skewed to the right, so evaluating `imsls_f_beta_cdf` at 0.7, 0.999953 is obtained. A more precise result is obtained by evaluating `imsls_f_beta_cdf` with parameters 10 and 2 at 0.3. This yields 4.72392E-5. (In both of these examples, it is wise not to trust the last digit.)

Many of the algorithms used by routines in this chapter are discussed by Abramowitz and Stegun (1964). The algorithms make use of various expansions and recursive relationships and often use different methods in different regions.

Cumulative distribution functions are defined for all real arguments, however, if the input to one of the distribution functions in this chapter is outside the range of the random variable, an error of Type 1 is issued, and the output is set to zero or one, as appropriate. A Type 1 error is of lowest severity, a “note”, and, by default, no printing or stopping of the program occurs. The other common errors that occur in the routines of this chapter are Type 2, “alert”, for a function value being set to zero due to underflow, Type 3, “warning”, for considerable loss of accuracy in the result returned, and Type 5, “terminal”, for incorrect and/or inconsistent input, complete loss of accuracy in the result returned, or inability to represent the result (because of overflow). When a Type 5 error occurs, the result is set to NaN (not a number, also used as a missing value code).

binomial_cdf

Evaluates the binomial distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_binomial_cdf` (*int* `k`, *int* `n`, *float* `p`)

The type *double* function is `imsls_d_binomial_cdf`.

Required Arguments

int `k` (Input)
Argument for which the binomial distribution function is to be evaluated.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial.

Return Value

The probability that *k* or fewer successes occur in *n* independent Bernoulli trials, each of which has a probability *p* of success.

Description

The `imsls_f_binomial_cdf` function evaluates the distribution function of a binomial random variable with parameters *n* and *p*. It does this by summing probabilities of the random variable taking on the specific values in its range. These probabilities are computed by the recursive relationship:

$$Pr(X = j) = \frac{(n+1-j)p}{j(1-p)} Pr(X = j-1)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0 if *k* is not greater than $n \times p$; otherwise, they are computed backward from *n*. The smallest positive machine number, ε, is used as the starting value for summing the probabilities, which are rescaled by $(1 - p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is used.

For the special case of $p = 0$, `imsls_f_binomial_cdf` is set to 1; for the case $p = 1$, `imsls_f_binomial_cdf` is set to 1 if $k = n$ and is set to 0 otherwise.

Example

Suppose X is a binomial random variable with $n = 5$ and $p = 0.95$. In this example, the function finds the probability that X is less than or equal to 3.

```
#include <imsls.h>

void main()
{
    int         k = 3;
    int         n = 5;
    float       p = 0.95;
    float       pr;

    pr = imsls_f_binomial_cdf(k,n,p);
    printf("Pr(x <= 3) = %6.4f\n", pr);
}
```

Output

```
Pr(x <= 3) = 0.0226
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since “k” = # is less than zero, the distribution function is set to zero.
`IMSLS_GREATER_THAN_N`	The input argument, *k*, is greater than the number of Bernoulli trials, *n*.

binomial_pdf

Evaluates the binomial probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_binomial_pdf` (*int* `k`, *int* `n`, *float* `p`,..., 0)

The type *double* function is `imsls_d_binomial_pdf`.

Required Arguments

int `k` (Input)
Argument for which the binomial probability function is to be evaluated.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial.

Return Value

The probability that a binomial random variable takes on a value equal to `k`.

Description

The function `imsls_f_binomial_pdf` evaluates the probability that a binomial random variable with parameters *n* and *p* takes on the value *k*. It does this by computing probabilities of the random variable taking on the values in its range less than (or the values greater than) *k*. These probabilities are computed by the recursive relationship

$$\Pr(X = j) = \frac{(n+1-j)p}{j(1-p)} \Pr(X = j-1)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0, if *k* is not greater than *n* times *p*, and are computed backward from *n*, otherwise. The smallest positive machine number, ε, is used as the starting value for computing the probabilities, which are rescaled by $(1-p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done.

For the special case of $p = 0$, `imsls_f_binomial_pdf` is set to 0 if *k* is greater than 0 and to 1 otherwise; and for the case $p = 1$, `imsls_f_binomial_pdf` is set to 0 if *k* is less than *n* and to 1 otherwise.

Example 1

Suppose *X* is a binomial random variable with $n = 5$ and $p = 0.95$. In this example, we find the probability that *X* is equal to 3.

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

void main()
{
  int k, n;
  float p, prob;

  k = 3;
  n = 5;
  p = 0.95;
  prob = imsls_f_binomial_pdf(k, n, p);

  printf("The probability that X is equal to 3 is %f\n", prob);
 }
```

Output

```
The probability that X is equal to 3 is 0.021434
```

hypergeometric_cdf

Evaluates the hypergeometric distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_hypergeometric_cdf` (*int* `k`, *int* `n`, *int* `m`, *int* `l`)

The type *double* function is `imsls_d_hypergeometric_cdf`.

Required Arguments

int `k` (Input)
Argument for which the hypergeometric distribution function is to be evaluated.

int `n` (Input)
Sample size. Argument `n` must be greater than or equal to `k`.

int `m` (Input)
Number of defectives in the lot.

int `l` (Input)
Lot size. Argument `l` must be greater than or equal to `n` and `m`.

Return Value

The probability that k or fewer defectives occur in a sample of size n drawn from a lot of size l that contains m defectives.

Description

Function `imsls_f_hypergeometric_cdf` evaluates the distribution function of a hypergeometric random variable with parameters n, l, and m. The hypergeometric random variable x can be thought of as the number of items of a given type in a random sample of size n that is drawn without replacement from a population of size l containing m items of this type. The probability function is

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

where $i = max\ (0, n - l + m)$.

If k is greater than or equal to i and less than or equal to min (n, m), `imsls_f_hypergeometric_cdf` sums the terms in this expression for j going from i up to k; otherwise, 0 or 1 is returned, as appropriate. To avoid rounding in the accumulation, `imsls_f_hypergeometric_cdf` performs the summation differently, depending on whether or not k is greater than the mode of the distribution, which is the greatest integer less than or equal to $(m + 1)\ (n + 1)/(l + 2)$.

Example

Suppose X is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. In this example, evaluate the distribution function at 7.

```
#include <imsls.h>

void main()
{
    int         k = 7;
    int         l = 1000;
    int         m = 70;
    int         n = 100;
    float       p;

    p = imsls_f_hypergeometric_cdf(k,n,m,l);
    printf("\nPr (x <= 7) = %6.4f", p);
}
```

Output

```
Pr (x <= 7) = 0.599
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since "k" = # is less than zero, the distribution function is set to zero.
`IMSLS_K_GREATER_THAN_N`	The input argument, *k*, is greater than the sample size.

Fatal Errors

`IMSLS_LOT_SIZE_TOO_SMALL`	Lot size must be greater than or equal to *n* and *m*.

hypergeometric_pdf

Evaluates the hypergeometric probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_hypergeometric_pdf` (*int* `k`, *int* `n`, *int* `m`, *int* `l`)

The type *double* function is `imsls_d_hypergeometric_pdf`.

Required Arguments

int `k` (Input)
Argument for which the hypergeometric probability function is to be evaluated.

int `n` (Input)
Sample size. `n` must be greater than zero and greater than or equal to `k`.

int `m` (Input)
Number of defectives in the lot.

int `l` (Input)
Lot size. `l` must be greater than or equal to `n` and `m`.

Return Value

The probability that a hypergeometric random variable takes a value equal to `k`. This value is the probability that exactly `k` defectives occur in a sample of size `n` drawn from a lot of size `l` that contains `m` defectives.

Description

The function `imsls_f_hypergeometic_pdf` evaluates the probability function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type. The probability function is

$$\Pr(X = k) = \frac{\binom{m}{k}\binom{l-m}{n-k}}{\binom{l}{n}} \quad \text{for } k = i, i+1, i+2, \ldots \min(n, m)$$

where $i = \max(0, n - l + m)$. `imsls_f_hypergeometic_pdf` evaluates the expression using log gamma functions.

Example

Suppose *X* is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. In this example, we evaluate the probability function at 7.

```
include "imsls.h"
void main()
{
  int k=7, n=100, l=1000, m=70;
  float pr;
  pr = imsls_f_hypergeometic_pdf(k, n, m, l);
  printf("  The probability that X is equal to 7 is %6.4f\n", pr);
}
```

Output

```
  The probability that X is equal to 7 is 0.1628
```

poisson_cdf

Evaluates the Poisson distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_poisson_cdf` (*int* `k`, *float* `theta`)

The type *double* function is `imsls_d_poisson_cdf`.

Required Arguments

int `k` (Input)
Argument for which the Poisson distribution function is to be evaluated.

float `theta` (Input)
Mean of the Poisson distribution. Argument `theta` must be positive.

Return Value

The probability that a Poisson random variable takes a value less than or equal to *k*.

Description

Function `imsls_f_poisson_cdf` evaluates the distribution function of a Poisson random variable with parameter `theta`. The mean of the Poisson random variable, `theta`, must be positive. The probability function (with θ = `theta`) is as follows:

$$f(x) = e^{-\theta}\theta^{x} / x!, \qquad \text{for } x = 0, 1, 2, \ldots$$

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. Function `imsls_f_poisson_cdf` uses the recursive relationship

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

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

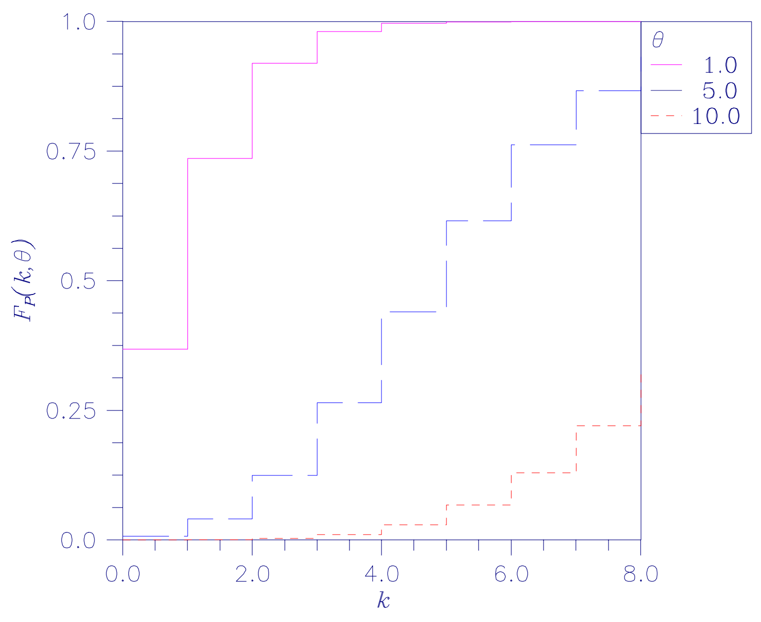


Figure 11-1 Plot of $F_p(k, \theta)$

Example

Suppose X is a Poisson random variable with θ = 10. In this example, we evaluate the probability that X is less than or equal to 7.

```
#include <imsls.h>

void main()
{
    int          k = 7;
    float        theta = 10.0;
    float        p;

    p = imsls_f_poisson_cdf(k, theta);
    printf("Pr(x <= 7) = %6.4f\n", p);
}
```

Output

```
Pr(x <= 7) = 0.2202
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since "k" = # is less than zero, the distribution function is set to zero.

poisson_pdf

Evaluates the Poisson probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_poisson_pdf` (*int* `k`, *float* `theta`)

The type *double* function is `imsls_d_poisson_pdf`.

Required Arguments

int `k` (Input)
Argument for which the Poisson distribution function is to be evaluated.

float `theta` (Input)
Mean of the Poisson distribution. `theta` must be positive.

Return Value

Function value, the probability that a Poisson random variable takes a value equal to `k`.

Description

Function `imsls_f_poisson_pdf` evaluates the probability function of a Poisson random variable with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with θ = `theta`) is

$$f(x) = e^{-\theta}\, \theta^{k}/k!, \quad \text{for } k = 0, 1, 2, \ldots$$

`imsls_f_poisson_pdf` evaluates this function directly, taking logarithms and using the log gamma function.

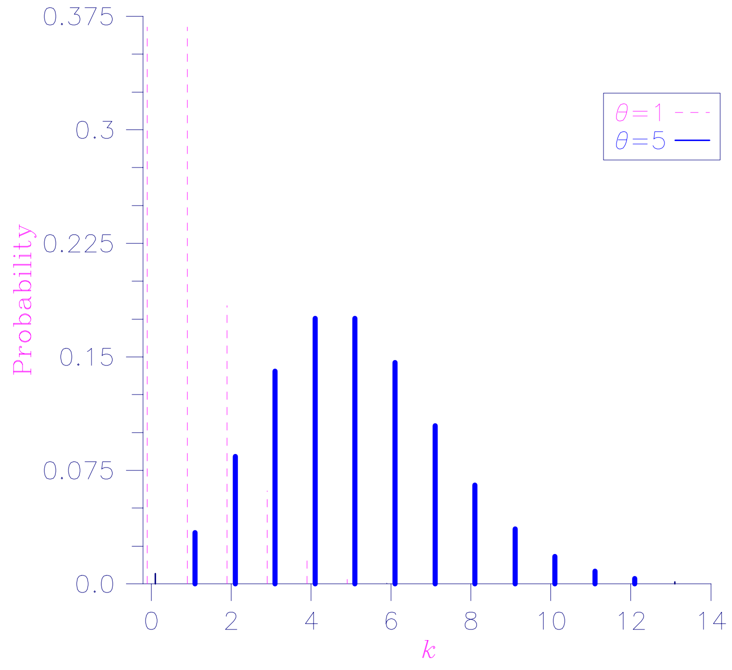


Figure 11-2   Poisson Probability Function

Example

Suppose X is a Poisson random variable with θ = 10. In this example, we evaluate the probability function at 7.

```
#include "imsls.h"

void main () {
  int k = 7;
  float theta = 10.0;

  printf ("The probability that X is equal to 7 is %g.\n",
         imsls_f_poisson_pdf (k, theta));
}
```

Output

```
The probability that X is equal to 7 is 0.0900792.
```

beta_cdf

Evaluates the beta probability distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta_cdf` (*float* `x`, *float* `pin`, *float* `qin`)

The type *double* function is `imsls_d_beta_cdf`.

Required Arguments

float `x` (Input)
Argument for which the beta probability distribution function is to be evaluated.

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

The probability that a beta random variable takes on a value less than or equal to x.

Description

Function `imsls_f_beta_cdf` evaluates the distribution function of a beta random variable with parameters `pin` and `qin`. This function is sometimes called the incomplete beta ratio and, with p = `pin` and q = `qin`, is denoted by $I_x(p, q)$. It is given by

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

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function by $I_x(p, q)$ is the probability that the random variable takes a value less than or equal to x.

The integral in the expression above is called the incomplete beta function and is denoted by $\beta_x(p, q)$. The constant in the expression is the reciprocal of the beta function (the incomplete function evaluated at 1) and is denoted by $\beta(p, q)$.

Function `imsls_f_beta_cdf` uses the method of Bosten and Battiste (1974).

Example

Suppose X is a beta random variable with parameters 12 and 12 (X has a symmetric distribution). This example finds the probability that X is less than 0.6 and the probability that X is between 0.5 and 0.6. (Since X is a symmetric beta random variable, the probability that it is less than 0.5 is 0.5.)

```
#include <imsls.h>

main()
{
        float           p, pin, qin, x;

        pin = 12.0;
        qin = 12.0;
        x = 0.6;
        p = imsls_f_beta_cdf(x, pin, qin);
        printf("The probability that X is less than 0.6 is %6.4f\n",
                p);
        x = 0.5;
        p -= imsls_f_beta_cdf(x, pin, qin);
        printf("The probability that X is between 0.5 and");
        printf(" 0.6 is %6.4f\n", p);
}
```

Output

```
The probability that X is less than 0.6 is 0.8364
The probability that X is between 0.5 and 0.6 is 0.3364
```

beta_inverse_cdf

Evaluates the inverse of the beta distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta_inverse_cdf` (*float* `p`, *float* `pin`, *float* `qin`)

The type *double* function is `imsls_d_beta_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the beta distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

Function `imsls_f_beta_inverse_cdf` returns the inverse distribution function of a beta random variable with parameters `pin` and `qin`.

Description

With P = `p`, p = `pin`, and q = `qin`, the beta_inverse_cdf returns x such that

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

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is P.

Example

Suppose *X* is a beta random variable with parameters 12 and 12 (*X* has a symmetric distribution). In this example, we find the value *x* such that the probability that *X* is less than or equal to *x* is 0.9.

```
#include <imsls.h>

main()
{
        float           p, pin, qin, x;


        pin = 12.0;
        qin = 12.0;
        p = 0.9;
        x = imsls_f_beta_inverse_cdf(p, pin, qin);
        printf(" X is less than %6.4f with probability 0.9.\n",
                x);
}
```

Output

```
X is less than 0.6299 with probability 0.9.
```

bivariate_normal_cdf

Evaluates the bivariate normal distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_bivariate_normal_cdf` (*float* `x`, *float* `y`, *float* `rho`)

The type *double* function is `imsls_d_bivariate_normal_cdf`.

Required Arguments

float `x` (Input)
The *x*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `y` (Input)
The *y*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `rho` (Input)
Correlation coefficient.

Return Value

The probability that a bivariate normal random variable with correlation `rho` takes a value less than or equal to *x* and less than or equal to *y*.

Description

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

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

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

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

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

Function imsls_f_bivariate_normal_cdf uses the method of Owen (1962, 1965). Computation of Owen's T-function is based on code by M. Patefield and D. Tandy (2000). For $|\rho| = 1$, the distribution function is computed based on the univariate statistic, $Z = \min(x, y)$, and on the normal distribution function imsls_f_normal_cdf.

Example

Suppose (X, Y) is a bivariate normal random variable with mean (0, 0) and variance-covariance matrix as follows:

$$\begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}$$

In this example, we find the probability that X is less than –2.0 and Y is less than 0.0.

```
#include <imsls.h>

main()
{
        float           p, rho, x, y;

        x = -2.0;
        y = 0.0;
        rho = 0.9;
        p = imsls_f_bivariate_normal_cdf(x, y, rho);
        printf(" The probability that X is less than -2.0\n"
               " and Y is less than 0.0 is %6.4f\n", p);

}
```

Output

```
The probability that X is less than -2.0
and Y is less than 0.0 is 0.0228
```

chi_squared_cdf

Evaluates the chi-squared distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_chi_squared_cdf` (*float* `chi_squared`, *float* `df`)

The type *double* function is `imsls_d_chi_squared_cdf`.

Required Arguments

float `chi_squared` (Input)
Argument for which the chi-squared distribution function is to be evaluated.

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. Argument `df` must be greater than or equal to 0.5.

Return Value

The probability that a chi-squared random variable takes a value less than or equal to `chi_squared`.

Description

Function `imsls_f_chi_squared_cdf` evaluates the distribution function, F, of a chi-squared random variable x = `chi_squared` with ν = `df`. Then,

$$F(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

For $\nu > 65$, `imsls_f_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) to the normal distribution, and function `imsls_f_normal_cdf` is used to evaluate the normal distribution function.

For $\nu \le 65$, `imsls_f_chi_squared_cdf` uses series expansions to evaluate the distribution function. If $x <$ `max` $(\nu / 2, 26)$, `imsls_f_chi_squared_cdf` uses the series 6.5.29 in Abramowitz and Stegun (1964); otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.

Example

Suppose X is a chi-squared random variable with two degrees of freedom. In this example, we find the probability that X is less than 0.15 and the probability that X is greater than 3.0.

```
#include <imsls.h>

void main()
{
    float       chi_squared = 0.15;
    float       df = 2.0;
    float       p;

    p    = imsls_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared\n",
        "with 2 df is less than 0.15 is", p);

    chi_squared = 3.0;
    p    = 1.0 - imsls_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared\n",
        "with 2 df is greater than 3.0 is", p);
}
```

Output

```
The probability that chi-squared
 with 2 df is less than 0.15 is 0.0723
The probability that chi-squared
 with 2 df is greater than 3.0 is 0.2231
```

Informational Errors

`IMSLS_ARG_LESS_THAN_ZERO`	Since “chi_squared” = # is less than zero, the distribution function is zero at “chi_squared.”

Alert Errors

`IMSLS_NORMAL_UNDERFLOW`	Using the normal distribution for large degrees of freedom, underflow would have occurred.

chi_squared_inverse_cdf

Evaluates the inverse of the chi-squared distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_chi_squared_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsls_d_chi_squared_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the chi-squared distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. Argument `df` must be greater than or equal to 0.5.

Return Value

The inverse at the chi-squared distribution function evaluated at `p`. The probability that a chi-squared random variable takes a value less than or equal to `imsls_f_chi_squared_inverse_cdf` is `p`.

Description

Function `imsls_f_chi_squared_inverse_cdf` evaluates the inverse distribution function of a chi-squared random variable with ν = `df` and with probability *p*. That is, it determines x = `imsls_f_chi_squared_inverse_cdf` (`p`, `df`), such that

$$p = \frac{1}{2^{\nu/2}\Gamma\left(\nu/2\right)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is p.

For $\nu < 40$, `imsls_f_chi_squared_inverse_cdf` uses bisection (if $\nu \le 2$ or $p > 0.98$) or regula falsi to find the point at which the chi-squared distribution function is equal to p. The distribution function is evaluated using IMSL function `imsls_f_chi_squared_cdf`.

For $40 \le \nu < 100$, a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.18) to the normal distribution is used. IMSL function `imsls_f_normal_cdf` is used to evaluate the inverse of the normal distribution function. For $\nu \ge 100$, the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) is used.

Example

In this example, we find the 99-th percentage point of a chi-squared random variable with 2 degrees of freedom and of one with 64 degrees of freedom.

```
#include <imsls.h>

void main ()
{
    float       df, x;
    float       p = 0.99;

    df = 2.0;
    x  = imsls_f_chi_squared_inverse_cdf(p, df);
    printf("For p = .99 with  2 df, x = %7.3f.\n", x);

    df = 64.0;
    x  = imsls_f_chi_squared_inverse_cdf(p,df);
    printf("For p = .99 with 64 df, x = %7.3f.\n", x);
}
```

Output

```
For p = .99 with  2 df, x =   9.210.
For p = .99 with 64 df, x =  93.217.
```

Warning Errors

`IMSLS_UNABLE_TO_BRACKET_VALUE`	The bounds that enclose "p" could not be found. An approximation for `imsls_f_chi_squared_inverse_cdf` is returned.
`IMSLS_CHI_2_INV_CDF_CONVERGENCE`	The value of the inverse chi-squared could not be found within a specified number of iterations. An approximation for `imsls_f_chi_squared_inverse_cdf` is returned.

non_central_chi_sq

Evaluates the noncentral chi-squared distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_chi_sq` *(float* `chi_squared`*, float* `df` *, float* `delta`*)*

The type *double* function is `imsls_d_non_central_chi_sq`.

Required Arguments

float `chi_squared` (Input)
Argument for which the noncentral chi-squared distribution function is to be evaluated.

float `df` (Input)
Number of degrees of freedom of the noncentral chi-squared distribution. Argument `df` must be greater than or equal to 0.5

float `delta` (Input)
The noncentrality parameter. `delta` must be nonnegative, and `delta + df` must be less than or equal to 200,000.

Return Value

The probability that a noncentral chi-squared random variable takes a value less than or equal to `chi_squared`.

Description

Function `imsls_f_non_central_chi_sq` evaluates the distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, that is, with v = `df`, λ = `alam`, and x = `chi_squared`,

$$non\_central\_chi\_sq(x) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^x \frac{t^{(v+2i)/2-1}e^{-t/2}}{2^{(v+2i)/2}\Gamma\left(\frac{v+2i}{2}\right)}dt$$

where Γ(·) is the gamma function. This is a series of central chi-squared distribution functions with Poisson weights. The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

The noncentral chi-squared random variable can be defined by the distribution function above, or alternatively and equivalently, as the sum of squares of independent normal random variables. If Y_i have independent normal distributions with means μ_i and variances equal to one and

$$X = \sum_{i=1}^{n} Y_i^2$$

then X has a noncentral chi-squared distribution with n degrees of freedom and noncentrality parameter equal to

$$\sum_{i=1}^{n} \mu_i^2$$

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the chi-squared distribution.

Function imsls_f_non_central_chi_sq determines the point at which the Poisson weight is greatest, and then sums forward and backward from that point, terminating when the additional terms are sufficiently small or when a maximum of 1000 terms have been accumulated. The recurrence relation 26.4.8 of Abramowitz and Stegun (1964) is used to speed the evaluation of the central chi-squared distribution functions.

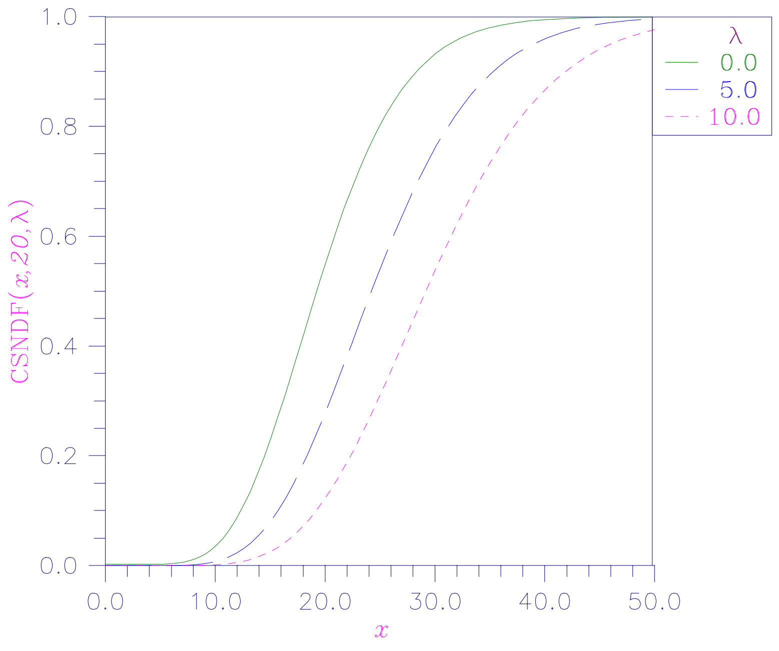


Figure 11-3 Noncentral Chi-squared Distribution Function

Example

In this example, `imsls_f_non_central_chi_sq` is used to compute the probability that a random variable that follows the noncentral chi-squared distribution with noncentrality parameter of 1 and with 2 degrees of freedom is less than or equal to 8.642.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float chsq = 8.642;
        float df = 2.0;
        float alam = 1.0;
        float p;
        p = imsls_f_non_central_chi_sq(chsq, df, alam);
        printf("The probability that a noncentral chi-squared random\n"
        "variable with %2.0f df and noncentrality parameter %3.1f is less\n"
        "than %5.3f is %5.3f.\n", df, alam, chsq, p);
}
```

Output

```
The probability that a noncentral chi-squared random
variable with 2 df and noncentrality parameter 1.0 is less
than 8.642 is 0.950
```

1.

non_central_chi_sq_inv

Evaluates the inverse of the noncentral chi-squared function.

Synopsis

#include <`imsls.h`>

float `imsls_f_non_central_chi_sq_inv` *(float* `p`, *float* `df`, *float* `delta`*)*

The type *double* function is `imsls_d_non_central_chi_sq_inv`.

Required Arguments

float p (Input)
Probability for which the inverse of the noncentral chi-squared distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Number of degrees of freedom of the noncentral chi-squared distribution. Argument df must be greater than or equal to 0.5

float `delta` (Input)
The noncentrality parameter. `delta` must be nonnegative, and `delta + df` must be less than or equal to 200,000.

Return Value

The probability that a noncentral chi-squared random variable takes a value less than or equal to `imsls_f_non_central_chi_sq_inv` is *p*.

Description

Function `imsls_f_non_central_chi_sq_inv` evaluates the inverse distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, ν = `df`, and λ = `delta`, it determines c_0 (= `imsls_f_non_central_chi_sq_inv (p, df, delta)`), such that

$$P = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^{c_0} \frac{x^{(\nu+2i)/2-1} e^{-x/2}}{2^{(\nu+2i)/2}\,\Gamma(\frac{\nu+2i}{2})} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to c_0 is P.

Function `imsls_f_non_central_chi_sq_inv` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using routine `imsls_f_non_central_chi_sq`. See `imsls_f_non_central_chi_sq` for an alternative definition of the noncentral chi-squared random variable in terms of normal random variables.

Example

In this example, we find the 95-th percentage point for a noncentral chi-squared random variable with 2 degrees of freedom and noncentrality parameter 1.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float p = .95;
        int df = 2;
        float delta = 1.0;
        float chi_squared;
        chi_squared = imsls_f_non_central_chi_sq_inv(p, df, delta);
       printf("The 0.05 noncentral chi-squared critical value is %6.4f.\n",
               chi_squared);
}
```

Output

```
The 0.05 noncentral chi-squared critical value is  8.6422.
```

F_cdf

Evaluates the *F* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_F_cdf` (*float* `f`, *float* `df_numerator`, *float* `df_denominator`)

The type *double* function is `imsls_d_F_cdf`.

Required Arguments

float `f` (Input)
: Point at which the *F* distribution function is to be evaluated.

float `df_numerator` (Input)
: The numerator degrees of freedom. Argument `df_numerator` must be positive.

float `df_denominator` (Input)
: The denominator degrees of freedom. Argument `df_denominator` must be positive.

Return Value

The probability that an *F* random variable takes a value less than or equal to the input point, `f`.

Description

Function `imsls_f_F_cdf` evaluates the distribution function of a Snedecor's *F* random variable with `df_numerator` and `df_denominator`. The function is evaluated by making a transformation to a beta random variable, then evaluating the incomplete beta function. If X is an F variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1 X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. Function `imsls_f_F_cdf` also uses a relationship between *F* random variables that can be expressed as

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

where F_F is the distribution function for an *F* random variable.

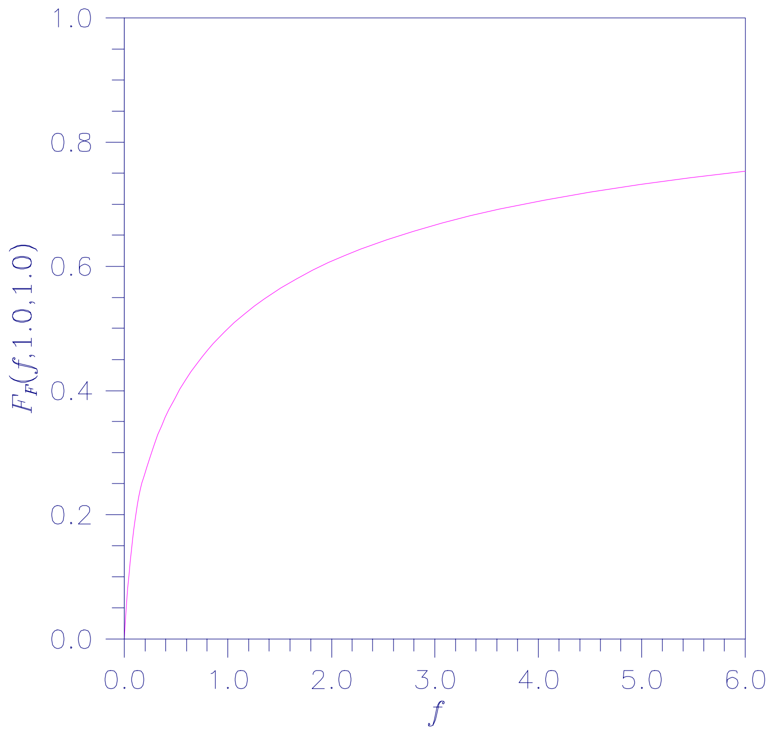


Figure 11-4 Plot of F_F*(f, 1.0, 1.0)*

Example

This example finds the probability that an *F* random variable with one numerator and one denominator degree of freedom is greater than 648.

```
#include <imsls.h>

main()
{
    float       p;
    float       F = 648.0;
    float       df_numerator = 1.0;
    float       df_denominator = 1.0;

    p = 1.0 - imsls_f_F_cdf(F,df_numerator, df_denominator);
    printf("%s %s %6.4f.\n", "The probability that an F(1,1) variate",
        "is greater than 648 is", p);
}
```

Output

```
The probability that an F(1,1) variate is greater than 648 is 0.0250.
```

F_inverse_cdf

Evaluates the inverse of the *F* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_F_inverse_cdf` (*float* `p`, *float* `df_numerator`,
float `df_denominator`)

The type *double* function is `imsls_d_F_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the *F* distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df_numerator` (Input)
Numerator degrees of freedom. Argument `df_numerator` must be positive.

float `df_denominator` (Input)
Denominator degrees of freedom. Argument `df_denominator` must be positive.

Return Value

The value of the inverse of the *F* distribution function evaluated at `p`. The probability that an *F* random variable takes a value less than or equal to `imsls_f_F_inverse_cdf` is `p`.

Description

Function `imsls_f_F_inverse_cdf` evaluates the inverse distribution function of a Snedecor's *F* random variable with ν_1 = `df_numerator` numerator degrees of freedom and ν_2 = `df_denominator` denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable, then evaluating the inverse of an incomplete beta function. If X is an F variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1 X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. If $p \leq 0.5$, `imsls_f_F_ inverse_cdf` uses this relationship directly; otherwise, it also uses a relationship between *F* random variables that can be expressed as follows:

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

Example

This example finds the 99-th percentage point for an *F* random variable with 7 and 1 degrees of freedom.

```
#include <imsls.h>

main()
{
    float        df_denominator = 1.0;
    float        df_numerator = 7.0;
    float        f;
    float        p = 0.99;
    f = imsls_f_F_inverse_cdf(p, df_numerator, df_denominator);

    printf("The F(7,1) 0.01 critical value is %6.3f\n", f);
}
```

Output

```
The F(7,1) 0.01 critical value is 5928.370
```

Fatal Errors

`IMSLS_F_INVERSE_OVERFLOW`	Function `imsls_f_F_inverse_cdf` overflows. This is because `df_numerator` or `df_denominator` and *p* are too large. The return value is set to machine infinity.

gamma_cdf

Evaluates the gamma distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_cdf` (*float* `x`, *float* `a`)

The type *double* function is `imsls_d_gamma_cdf`.

Required Arguments

float `x` (Input)
Argument for which the gamma distribution function is to be evaluated.

float `a` (Input)
Shape parameter of the gamma distribution. This parameter must be positive.

Return Value

The probability that a gamma random variable takes a value less than or equal to `x`.

Description

Function `imsls_f_gamma_cdf` evaluates the distribution function, *F*, of a gamma random variable with shape parameter *a*,

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

where Γ(·) is the gamma function. (The gamma function is the integral from 0 to ∞ of the same integrand as above.) The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The gamma distribution is often defined as a two-parameter distribution with a scale parameter *b* (which must be positive) or as a three-parameter distribution in which the third parameter *c* is a location parameter. In the most general case, the probability density function over (*c*, ∞) is as follows:

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}$$

If T is a random variable with parameters a, b, and c, the probability that $T \le t_0$ can be obtained from `imsls_f_gamma_cdf` by setting $x = (t_0 - c)/b$.

If x is less than a or less than or equal to 1.0, `imsls_f_gamma_cdf` uses a series expansion; otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun 1964.)

Example

Let *X* be a gamma random variable with a shape parameter of four. (In this case, it has an *Erlang distribution* since the shape parameter is an integer.) This example finds the probability that *X* is less than 0.5 and the probability that *X* is between 0.5 and 1.0.

```
#include <imsls.h>

main()
{
    float       p, x;
    float       a = 4.0;

    x = 0.5;
    p = imsls_f_gamma_cdf(x,a);
    printf("The probability that X is less than 0.5 is %6.4f\n", p);

    x = 1.0;
    p = imsls_f_gamma_cdf(x,a) - p;
    printf("The probability that X is between 0.5 and 1.0 is %6.4f\n",
        p);
}
```

Output

```
The probability that X is less than 0.5 is 0.0018
The probability that X is between 0.5 and 1.0 is 0.0172
```

Informational Errors

`IMSLS_ARG_LESS_THAN_ZERO`	Since "x" = # is less than zero, the distribution function is zero at "x."

Fatal Errors

`IMSLS_X_AND_A_TOO_LARGE`	Since "x" = # and "a" = # are so large, the algorithm would overflow.

gamma_inverse_cdf

Evaluates the inverse of the gamma distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_inverse_cdf` (*float* `p`, *float* `a`)

The type *double* function is `imsls_d_gamma_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the gamma distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

float `a` (Input)
The shape parameter of the gamma distribution. This parameter must be positive.

Return Value

The probability that a gamma random variable takes a value less than or equal to the returned value is `p`.

Description

Function `imsls_f_gamma_inverse_cdf` evaluates the inverse distribution function of a gamma random variable with shape parameter *a*, that is, it determines *x* (=`imsls_f_gamma_inverse_cdf` (`p`, `a`)), such that

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

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to *x* is *P*. See the documentation for function `imsls_f_gamma_cdf` for further discussion of the gamma distribution.

Function `imsls_f_gamma_inverse_cdf` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using function `imsls_f_gamma_cdf`.

Example

In this example, we find the 95-th percentage point for a gamma random variable with shape parameter of 4.

```
include "imsls.h"
void main()
{
  float p = .95, a = 4.0, x;
  x = imsls_f_gamma_inverse_cdf(p,a);
  printf("The 0.05 gamma(4) critical value is %6.4f\n", x);
}
```

Output

```
The 0.05 gamma(4) critical value is 7.7537
```

normal_cdf

Evaluates the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_cdf` (*float* `x`)

The type *double* function is `imsls_d_normal_cdf`.

Required Arguments

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

Return Value

The probability that a normal random variable takes a value less than or equal to x.

Description

Function `imsls_f_normal_cdf` evaluates the distribution function, Φ, of a standard normal (Gaussian) random variable as follows:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

The standard normal distribution (for which `imsls_f_normal_cdf` is the distribution function) has mean of 0 and variance of 1. The probability that a normal random variable with mean μ and variance σ^2 is less than y is given by `imsls_f_normal_cdf` evaluated at $(y - \mu)/\sigma$.

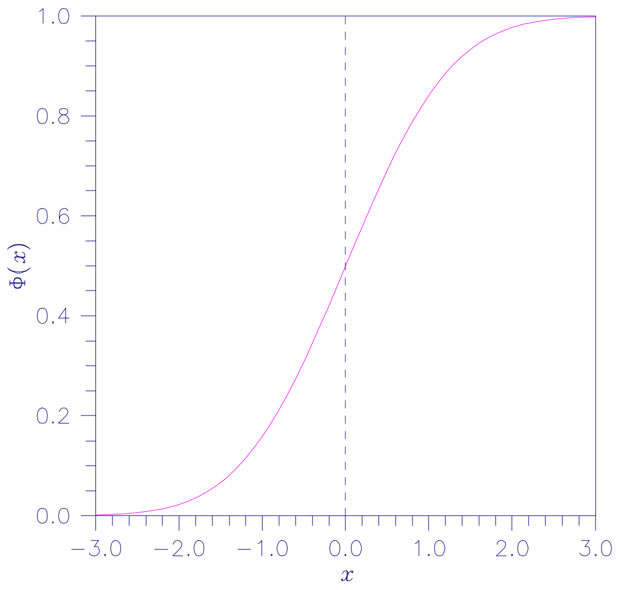


Figure 11-5 Plot of Φ(x)

Example

Suppose *X* is a normal random variable with mean 100 and variance 225. This example finds the probability that *X* is less than 90 and the probability that *X* is between 105 and 110.

```
#include <imsls.h>

main()
{
    float      p, x1, x2;

    x1  = (90.0-100.0)/15.0;
    p   = imsls_f_normal_cdf(x1);
    printf("The probability that X is less than 90 is %6.4f\n", p);

    x1 = (105.0-100.0)/15.0;
    x2 = (110.0-100.0)/15.0;
    p  = imsls_f_normal_cdf(x2) - imsls_f_normal_cdf(x1);
    printf("The probability that X is between 105 and 110 is %6.4f\n",
        p);
}
```

Output

```
The probability that X is less than 90 is 0.2525
The probability that X is between 105 and 110 is 0.1169
```

normal_inverse_cdf

Evaluates the inverse of the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_inverse_cdf` (*float* `p`)

The type *double* function is `imsls_d_normal_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the normal distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

Return Value

The inverse of the normal distribution function evaluated at `p`. The probability that a standard normal random variable takes a value less than or equal to `imsls_f_normal_inverse_cdf` is `p`.

Description

Function `imsls_f_normal_inverse_cdf` evaluates the inverse of the distribution function, Φ, of a standard normal (Gaussian) random variable, `imsls_f_normal_inverse_cdf`(p) = $\Phi^{-1}(x)$, where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x. The standard normal distribution has a mean of 0 and a variance of 1.

Function `imsls_f_normal_inverse_cdf` (p) is evaluated by use of minimax rational-function approximations for the inverse of the error function. General descriptions of these approximations are given in Hart et al. (1968) and Strecok (1968). The rational functions used in `imsls_f_normal_inverse_cdf` are described by Kinnucan and Kuki (1968).

Example

This example computes the point such that the probability is 0.9 that a standard normal random variable is less than or equal to this point.

```
#include <imsls.h>

main()
{
    float       x;
    float       p = 0.9;

    x = imsls_f_normal_inverse_cdf(p);
    printf("The 90th percentile of a standard normal is %6.4f.\n", x);
}
```

Output

```
The 90th percentile of a standard normal is 1.2816.
```

t_cdf

Evaluates the Student's *t* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_t_cdf` (*float* `t`, *float* `df`)

The type *double* function is `imsls_d_t_cdf`.

Required Arguments

float `t` (Input)
Argument for which the Student's *t* distribution function is to be evaluated.

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The probability that a Student's *t* random variable takes a value less than or equal to the input *t*.

Description

Function `imsls_f_t_cdf` evaluates the distribution function of a Student's *t* random variable with ν = `df` degrees of freedom. If the square of *t* is greater than or equal to ν, the relationship of a *t* to an *F* random variable (and subsequently, to a beta random variable) is exploited, and percentage points from a beta distribution are used. Otherwise, the method described by Hill (1970) is used. If ν is not an integer, is greater than 19, or is greater than 200, a Cornish- Fisher expansion is used to evaluate the distribution function. If ν is less than 20 and $|t|$ is less than 2.0, a trigonometric series is used (see Abramowitz and Stegun 1964, Equations 26.7.3 and 26.7.4 with some rearrangement). For the remaining cases, a series given by Hill (1970) that converges well for large values of *t* is used.

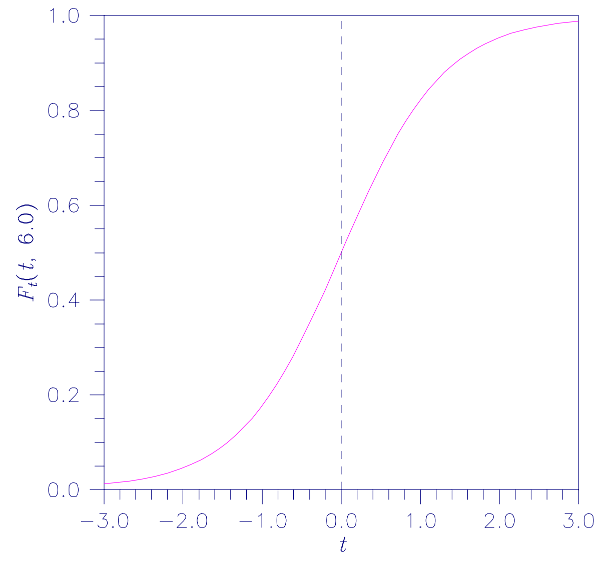


Figure 11-6 Plot of F_t *(t, 6.0)*

Example

This example finds the probability that a *t* random variable with 6 degrees of freedom is greater in absolute value than 2.447. The fact that *t* is symmetric about 0 is used.

```
#include <imsls.h>

main ()
{
    float       p;
    float       t = 2.447;
    float       df = 6.0;

    p  = 2.0*imsls_f_t_cdf(-t,df);
    printf("Pr(|t(6)| > 2.447) = %6.4f\n", p);
}
```

Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

t_inverse_cdf

Evaluates the inverse of the Student's *t* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_t_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsls_d_t_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the Student's *t* distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The inverse of the Student's *t* distribution function evaluated at `p`. The probability that a Student's *t* random variable takes a value less than or equal to `imsls_f_t_inverse_cdf` is `p`.

Description

Function `imsls_f_t_inverse_cdf` evaluates the inverse distribution function of a Student's *t* random variable with ν = `df` degrees of freedom. If ν equals 1 or 2, the inverse can be obtained in closed form. If ν is between 1 and 2, the relationship of a *t* to a beta random variable is exploited and the inverse of the beta distribution is used to evaluate the inverse; otherwise, the algorithm of Hill (1970) is used. For small values of ν greater than 2, Hill's algorithm inverts an integrated expansion in $1/(1 + t^2/\nu)$ of the *t* density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

Example

This example finds the 0.05 critical value for a two-sided *t* test with 6 degrees of freedom.

```
#include <imsls.h>

void main()
{
    float       df = 6.0;
    float       p = 0.975;
    float       t;

    t  = imsls_f_t_inverse_cdf(p,df);

    printf("The two-sided t(6) 0.05 critical value is %6.3f\n", t);
}
```

Output

```
The two-sided t(6) 0.05 critical value is  2.447
```

Informational Errors

`IMSLS_OVERFLOW`	Function `imsls_f_t_inverse_cdf` is set to machine infinity since overflow would occur upon modifying the inverse value for the *F* distribution with the result obtained from the inverse beta distribution.

non_central_t_cdf

Evaluates the noncentral Student's *t* distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_t_cdf` (*float* `t`, *int* `df` , *float* `delta`)

The type *double* function is `imsls_d_non_central_t_cdf`.

Required Arguments

float `t` *(Input)*

Argument for which the noncentral Student's *t* distribution function is to be evaluated.

int `df` *(Input)*

Number of degrees of freedom of the noncentral Student's *t* distribution. Argument df must be greater than or equal to 0.0

float `delta` *(Input)* The noncentrality parameter.

Return Value

The probability that a noncentral Student's *t* random variable takes a value less than or equal to `t`.

Description

Function `imsls_f_non_central_t_cdf` evaluates the distribution function *F* of a noncentral *t* random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with v = `df`, δ = `delta` , and t_0 = `t`,

$$F(t_0) = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(v/2)(v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2)\left(\tfrac{\delta^i}{i!}\right)\left(\tfrac{2x^2}{v+x^2}\right)^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point t_0 is the probability that the random variable takes a value less than or equal to t_0.

The noncentral *t* random variable can be defined by the distribution function above, or alternatively and equivalently, as the ratio of a normal random variable and an independent chi-squared random variable. If w has a normal distribution with mean δ and variance equal to one, u has an independent chi-squared distribution with v degrees of freedom, and

$$x = w/\sqrt{u/v}$$

then x has a noncentral *t* distribution with degrees of freedom and noncentrality parameter δ.

The distribution function of the noncentral *t* can also be expressed as a double integral involving a normal density function (see, for example, Owen 1962, page 108). The function `TNDF` uses the method of Owen (1962, 1965), which uses repeated integration by parts on that alternate expression for the distribution function.

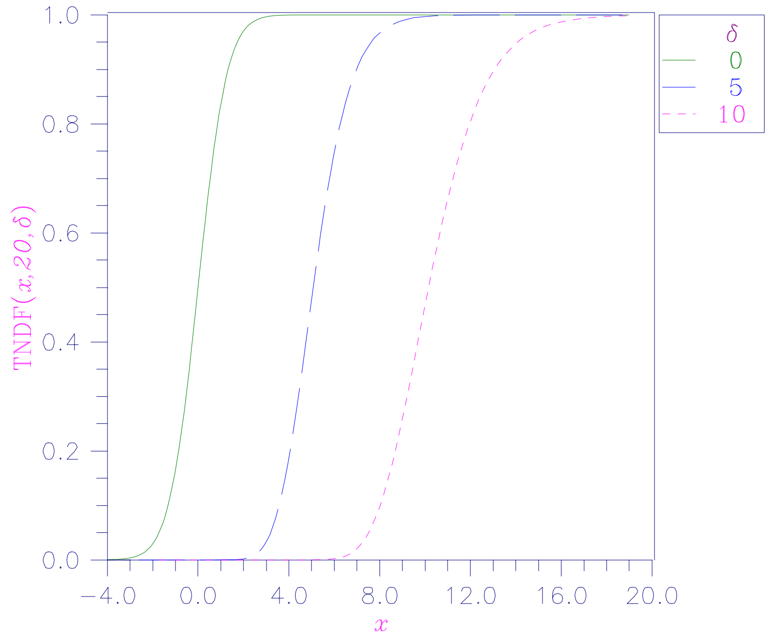


Figure 11-7 Noncentral Student's t *Distribution Function*

Example

Suppose `t` is a noncentral *t* random variable with 6 degrees of freedom and noncentrality parameter 6. In this example, we find the probability that `t` is less than 12.0. (This can be checked using the table on page 111 of Owen 1962, with $\eta = 0.866$, which yields $\lambda = 1.664$.)

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float t = 12.0;
        int df = 6;
        float delta = 6.0;
        float p;
        p = imsls_f_non_central_t_cdf(t, df, delta);
        printf("The probability that t is less than 12 is %6.4f.\n", p);
}
```

Output

```
The probability that T is less than 12.0 is 0.9501
```

non_central_t_inv_cdf

Evaluates the inverse of the noncentral Student's *t* distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_t_inv_cdf` (*float* `p`, *int* `df` , *float* `delta`)

The type *double* function is `imsls_d_non_central_t_inv_cdf`.

Required Arguments

float `p` (Input)
A Probability for which the inverse of the noncentral Student's *t* distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

int `df` (Input)
Number of degrees of freedom of the noncentral Student's *t* distribution. Argument `df` must be greater than or equal to 0.0

float `delta` (Input)
The noncentrality parameter.

Return Value

The probability that a noncentral Student's *t* random variable takes a value less than or equal to `t` is `p`.

Description

Function `imsls_f_non_central_t_inv_cdf` evaluates the inverse distribution function of a noncentral *t* random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, ν = `df`, and δ = `delta`, it determines t_0 (= `imsls_f_non_central_t_inv_cdf (p, df, delta`)), such that

$$P = \int_{-\infty}^{t_0} \frac{\nu^{\nu/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(\nu/2)(\nu+x^2)^{(\nu+1)/2}} \sum_{i=0}^{\infty} \Gamma((\nu+i+1)/2)(\tfrac{\delta^i}{i!})(\tfrac{2x^2}{\nu+x^2})^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to t_0 is P. See `imsls_f_non_central_t_cdf` (page) for an alternative definition in terms of normal and chi-squared random variables. The function `imsls_f_non_central_t_inv_cdf` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using routine `imsls_f_non_central_t_cdf`.

Example

In this example, we find the 95-th percentage point for a noncentral *t* random variable with 6 degrees of freedom and noncentrality parameter 6.

```
#include <imsls.h>
#include <stdio.h>
void main()
{
        float p = .95;
        int df = 6;
        float delta = 6.0;
        float t;
        t = imsls_f_non_central_t_inv_cdf(p, df, delta);
        printf("The 0.05 noncentral t critical value is %6.4f.\n", t);
}
```

Output

```
The 0.05 noncentral t critical value is 11.995.
```

Chapter 12: Random Number Generation

Routines

Univariate Discrete Distributions

Generates pseudorandom binomial numbers	random_binomial	816
Generates pseudorandom geometric numbers	random_geometric	818
Generates pseudorandom hypergeometric numbers	random_hypergeometric	819
Generates pseudorandom logarithmic numbers	random_logarithmic	822
Generates pseudorandom negative binomial numbers	random_neg_binomial	823
Generates pseudorandom Poisson numbers	random_poisson	825
Generates pseudorandom discrete uniform numbers	random_uniform_discrete	826
Generates pseudorandom numbers from a general discrete distribution	random_general_discrete	828
Sets up a table to generate pseudorandom numbers from a general discrete distribution	discrete_table_setup	832

Univariate Continuous Distributions

Generates pseudorandom beta numbers	random_beta	837
Generates pseudorandom Cauchy numbers	random_cauchy	838
Generates pseudorandom chi_squared numbers	random_chi_squared	840
Generates pseudorandom exponential numbers	random_exponential	841
Generates pseudorandom mixed exponential numbers	random_exponential_mix	843
Generates pseudorandom gamma numbers	random_gamma	845
Generates peudorandom lognormal numbers	random_lognormal	846
Generates pseudorandom normal numbers	random_normal	848

Generates pseudorandom numbers from a stable distribution	random_stable	850
Generates pseudorandom Student's t	random_student_t	852
Generates pseudorandom triangular numbers	random_triangular	853
Generates pseudorandom uniform numbers	random_uniform	854
Generates pseudorandom Von Mises numbers	random_von_mises	856
Generates pseudorandom Weibull numbers	random_weibull	857
Generates pseudorandom numbers from a general continuous distribution	random_general_continuous	859
Sets up table to generate pseudorandom numbers from a general continuous distribution	continuous_table_setup	862

Multivariate Continuous Distributions

Generates multivariate normal vectors	random_normal_multivariate	864
Generates a pseudorandom orthogonal matrix or a correlation matrix	random_orthogonal_matrix	866
Generates pseudorandom numbers from a multivariate distribution determined from a given sample	random_mvar_from_data	868
Generates pseudorandom numbers from a multinomial distribution	random_multinomial	871
Generates pseudorandom points on a unit circle or K-dimensional sphere	random_sphere	873
Generates a pseudorandom two-way table	random_table_twoway	875

Order Statistics

Generates pseudorandom order statistics from a standard normal distribution	random_order_normal	876
Generates pseudorandom order statistics from a uniform (0, 1) distribution	random_order_uniform	878

Stochastic Processes

Generates pseudorandom ARMA process numbers	random_arma	880
Generates pseudorandom numbers from a nonhomogeneous Poisson process	random_npp	884

Samples and Permutations

Generates a pseudorandom permutation	random_permutation	887
Generates a simple pseudorandom sample of indices	random_sample_indices	889
Generates a simple pseudorandom sample from a finite population	random_sample	890

Utility Functions

Selects the uniform (0, 1) generator	random_option	894

Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator	random_option_get	895
Retrieves the current value of the seed	random_seed_get	896
Retrieves a seed for the congruential generators	random_substream_seed_get	897
Initializes a random seed	random_seed_set	899
Sets the current table used in the shuffled generator	random_table_set	900
Retrieves the current table used in the shuffled generator	random_table_get	900
Sets the current able used in the GFSR generator	random_GFSR_table_set	901
Retrieves the current table used in the GFSR generator	random_GFSR_table_get	902
Initializes the 32-bit Mersenne Twister generator using an array.	random_MT32_init	905
Retrieves the current table used in the 32-bit Mersenne Twister generator.	random_MT32_table_get	905
Sets the current table used in the 32-bit Mersenne Twister generator.	random_MT32_table_set	907
Initializes the 64-bit Mersenne Twister generator using an array.	random_MT64_init	908
Retrieves the current table used in the 64-bit Mersenne Twister generator	random_MT64_table_get	908
Sets the current table used in the 64-bit Mersenne Twister generator.	random_MT64_table_set	910

Low-discrepancy sequence

Generates a shuffled Faure sequence	faure_next_point	911

Usage Notes

Overview of Random Number Generation

This chapter describes functions for the generation of random numbers that are useful for applications in Monte Carlo or simulation studies. Before using any of the random number generators, the generator must be initialized by selecting a *seed* or starting value. The user can do this by calling the function `imsls_random_seed_set`. If the user does not select a seed, one is generated using the system clock. A seed needs to be selected only once in a program, unless two or more separate streams of random numbers are maintained. Other utility functions in this chapter can be used to select the form of the basic generator to restart simulations and to maintain separate simulation streams.

In the following discussions, the phrases “random numbers,” “random deviates,” “deviates,” and “variates” are used interchangeably. The phrase “pseudorandom” is sometimes used to emphasize that the numbers generated are really not “random” since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they simulate the realizations of independent and identically distributed random variables.

Basic Uniform Generators

The random number generators in this chapter use either a multiplicative congruential method or a generalized feedback shift register. The selection of the type of generator is made by calling the routine `imsls_random_option`. If no selection is made explicitly, a multiplicative generator (with multiplier 16807) is used. Whatever distribution is being simulated, uniform (0, 1) numbers are first generated and then transformed if necessary. These routines are *portable* in the sense that, given the same seed and for a given type of generator, they produce the same sequence in all computer/compiler environments. There are many other issues that must be considered in developing programs for the methods described below (see Gentle 1981 and 1990).

The Multiplicative Congruential Generators

The form of the multiplicative congruential generators is

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0,1). If the multiplier, c, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator will have a maximal period of $2^{31} - 2$. There are several other considerations, however. See Knuth (1981) for a good general discussion. The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by the function `imsls_random_option`. The choice of 16807 will result in the fastest execution time, but other evidence suggests that the performance of 950706376 is best among these three choices (Fishman and Moore 1982). If no selection is made explicitly, the functions use the multiplier 16807, which has been in use for some time (Lewis et al. 1969).

The generation of uniform (0,1) numbers is done by the function `imsls_f_random_uniform`. This function is portable in the sense that, given the same seed, it produces the same sequence in all computer/compiler environments.

Shuffled Generators

The user also can select a shuffled version of these generators using `imsls_random_option`. The shuffled generators use a scheme due to Learmonth and Lewis (1973). In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number; and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table. This scheme is similar to that of Bays and Durham (1976), and their analysis is applicable to this scheme as well.

The Generalized Feedback Shift Register Generator

The GFSR generator uses the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$. This generator, which is different from earlier GFSR generators, was proposed by Fushimi (1990), who discusses the theory behind the generator and reports on several empirical tests of it. Background discussions on this type of generator can be found in Kennedy and Gentle (1980), pages 150–162.

Setting the Seed

The seed of the generator can be set in imsls_random_seed_set and can be retrieved by imsls_random_seed_get. Prior to invoking any generator in this section, the user can call imsls_random_seed_set to initialize the seed, which is an integer variable with a value between 1 and 2147483647. If it is not initialized by imsls_random_seed_set, a random seed is obtained from the system clock. Once it is initialized, the seed need not be set again.

If the user wants to restart a simulation, imsls_random_seed_get can be used to obtain the final seed value of one run to be used as the starting value in a subsequent run. Also, if two simultaneous random number streams are desired in one run, imsls_random_seed_set and imsls_random_seed_get can be used before and after the invocations of the generators in each stream.

If a shuffled generator or the GFSR generator is used, in addition to resetting the seed, the user must also reset some values in a table. For the shuffled generators, this is done using the routines imsls_f_random_table_get and imsls_f_random_table_set; and for the GFSR generator; the table is retrieved and set by the routines imsls_random_GFSR_table_get and imsls_random_GFSR_table_set. The tables for the shuffled generators are separate for single and double precision; so, if precisions are mixed in a program, it is necessary to manage each precision separately for the shuffled generators.

Timing Considerations

The generation of the uniform (0,1) numbers is done by the routine imsls_f_random_uniform. The particular generator selected in imsls_random_option, that is, the value of the multiplier and whether shuffling is done or whether the GFSR generator is used, affects the speed of imsls_f_random_uniform. The smaller multiplier (16807, selected by iopt = 1) is faster than the other multipliers. The multiplicative congruential generators that do not shuffle are faster than the ones that do. The GFSR generator is roughly as fast as the fastest multiplicative congruential generator, but the initialization for it (required only on the first invocation) takes longer than the generation of thousands of uniform random numbers. Precise statements of relative speeds depend on the computing system.

Distributions Other than the Uniform

The nonuniform generators use a variety of transformation procedures. All of the transformations used are exact (mathematically). The most straightforward transformation is the *inverse CDF technique*, but it is often less efficient than others involving *acceptance/rejection* and *mixtures*. See Kennedy and Gentle (1980) for discussion of these and other techniques.

Many of the nonuniform generators in this chapter use different algorithms depending on the values of the parameters of the distributions. This is particularly true of the generators for discrete distributions. Schmeiser (1983) gives an overview of techniques for generating deviates from discrete distributions.

Although, as noted above, the uniform generators yield the same sequences on different computers, because of rounding, the nonuniform generators that use acceptance/rejection may occasionally produce different sequences on different computer/compiler environments.

Although the generators for nonuniform distributions use fast algorithms, if a very large number of deviates from a fixed distribution are to be generated, it might be worthwhile to consider a table-sampling method, as implemented in the routines `imsls_f_random_general_discrete`, `imsls_f_discrete_table_setup`, `imsls_f_random_general_continuous`, and `imsls_f_continuous_table_setup`. After an initialization stage, which may take some time, the actual generation may proceed very fast.

Tests

Extensive empirical tests of some of the uniform random number generators available in `imsls_f_random_uniform` are reported by Fishman and Moore (1982 and 1986). Results of tests on the generator using the multiplier 16807 with and without shuffling are reported by Learmonth and Lewis (1973b). If the user wishes to perform additional tests, the routines in Chapter 7, "Tests of Goodness of Fit and Randomness," may be of use. Often in Monte Carlo applications, it is appropriate to construct an ad hoc test that is sensitive to departures that are important in the given application. For example, in using Monte Carlo methods to evaluate a one-dimensional integral, autocorrelations of order one may not be harmful, but they may be disastrous in evaluating a two-dimensional integral. Although generally the routines in this chapter for generating random deviates from nonuniform distributions use exact methods, and, hence, their quality depends almost solely on the quality of the underlying uniform generator, it is often advisable to employ an ad hoc test of goodness of fit for the transformations that are to be applied to the deviates from the nonuniform generator.

Additional Notes on Usage

The generators for continuous distributions are available in both single and double-precision versions. This is merely for the convenience of the user; the double-precision versions should not be considered more "accurate," except possibly for the multivariate distributions.

random_binomial

Generates pseudorandom numbers from a binomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_binomial` (*int* `n_random`, *int* `n`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_binomial`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial. Parameter `p` must be greater than 0.0 and less than 1.0.

Return Value

An integer array of length `n_random` containing the random binomial deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_binomial` (*int* `n_random`, *int* `n`, *float* `p`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random binomial deviates.

Description

Function `imsls_f_random_binomial` generates pseudorandom numbers from a binomial distribution with parameters *n* and *p*. Parameters *n* and *p* must be positive, and *p* must less than 1. The probability function (with *n* = `n` and *p* = `p`) is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for *x* = 0, 1, 2, …, *n*.

The algorithm used depends on the values of *n* and *p*. If *np* < 10 or *p* is less than machine epsilon (see `imsls_f_machine`, Chapter 15, "Utilities"), the inverse CDF technique is used; otherwise, the BTPE algorithm of Kachitvichyanukul and Schmeiser (see Kachitvichyanukul 1982) is used. This is an acceptance/rejection method using a composition of four regions. (TPE=Triangle, Parallelogram, Exponential, left and right.)

Example

In this example, `imsls_f_random_binomial` generates five pseudorandom binomial deviates from a binomial distribution with parameters 20 and 0.5.

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

```
#include <imsls.h>

void main()
{
    int   n_random = 5;
    int   n = 20;
    float p = 0.5;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_binomial(n_random, n, p, 0);
    imsls_i_write_matrix("Binomial (20, 0.5) random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Binomial (20, 0.5) random deviates:
       14    9   12   10   12
```

random_geometric

Generates pseudorandom numbers from a geometric distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_geometric` (*int* `n_random`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_geometric`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `p` (Input)
Probability of succes on each trial. Parameter `p` must be positive and less than 1.0.

Return Value

An integer array of length `n_random` containing the random geometric deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_geometric` (*int* `n_random`, *float* `p`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
: User-supplied integer array of length `n_random` containing the random geometric deviates.

Description

Function `imsls_f_random_geometric` generates pseudorandom numbers from a geometric distribution with parameter P, where P is the probability of getting a success on any trial. A geometric deviate can be interpreted as the number of trials until the first success (including the trial in which the first success is obtained). The probability function is

$$f(x) = P(1 - P)^{x-1}$$

for $x = 1, 2, \ldots$ and $0 < P < 1$.

The geometric distribution as defined above has mean $1/P$.

The i-th geometric deviate is generated as the smallest integer not less than $(\log(U_i))/(\log(1 - P))$, where the U_i are independent uniform(0, 1) random numbers (see Knuth 1981).

The geometric distribution is often defined on 0, 1, 2, ..., with mean $(1 - P)/P$. Such deviates can be obtained by subtracting 1 from each element of `ir` (the returned vector of random deviates).

Example

In this example, `imsls_f_random_geometric` generates five pseudorandom geometric deviates from a geometric distribution with parameter an equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float p = 0.3;
    int *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_geometric(n_random, p, 0);
    imsls_i_write_matrix("Geometric(0.3) random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Geometric(0.3) random deviates:
       1    4    1    2    1
```

random_hypergeometric

Generates pseudorandom numbers from a hypergeometric distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_hypergeometric` (*int* `n_random`, *int* `n`, *int* `m`,
int `l`, ..., 0)

The type *double* function is `imsls_d_random_hypergeometric`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `n` (Input)
Number of items in the sample. Parameter `n` must be positive.

int `m` (Input)
Number of special items in the population, or lot. Parameter `m` must be positive.

int `l` (Input)
Number of items in the lot. Parameter `l` must be greater than both `n` and `m`.

Return Value

An integer array of length `n_random` containing the random hypergeometric deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_hypergeometric` (*int* `n_random`, *int* `n`, *int* `m`,
int `l`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random hypergeometric deviates.

Description

Function `imsls_f_random_hypergeometric` generates pseudorandom numbers from a hypergeometric distribution with parameters *N*, *M*, and *L*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *N* that is drawn without replacement from a population of size *L* containing *M* items of this type. The probability function is

$$f(x) = \frac{\binom{M}{x}\binom{L-M}{N-x}}{\binom{L}{N}}$$

for $x = \max(0, N - L + M), 1, 2, \ldots, \min(N, M)$

If the hypergeometric probability function with parameters N, M, and L evaluated at $N - L + M$ (or at 0 if this is negative) is greater than the machine epsilon (see imsls_f_machine, Chapter 15, "Utilities"), and less than 1.0 minus the machine epsilon, then imsls_f_random_hypergeometric uses the inverse CDF technique. The routine recursively computes the hypergeometric probabilities, starting at $x = \max(0, N - L + M)$ and using the ratio

$$\frac{f(X = x+1)}{f(X = x)}$$

(see Fishman 1978, p. 475).

If the hypergeometric probability function is too small or too close to 1.0, the imsls_f_random_hypergeometric generates integer deviates uniformly in the interval $[1, L - i]$ for $i = 0, 1, ...$, and at the i-th step, if the generated deviate is less than or equal to the number of special items remaining in the lot, the occurence of one special item is tallied and the number of remaining special items is decreased by one. This process continues until the sample size of the number of special items in the lot is reached, whichever comes first. This method can be much slower than the inverse CDF technique. The timing depends on N. If N is more than half of L (which in practical examples is rarely the case), the user may wish to modify the problem, replacing N by $L - N$, and to consider the generated deviates to be the number of special items *not* included in the sample.

Example

In this example, imsls_f_random_hypergeometric generates five pseudorandom hypergeometric deviates from a hypergeometric distribution to simulate taking random samples of size 4 from a lot containing 20 items, of which 12 are defective. The resulting hypergeometric deviates represent the numbers of defectives in each of the five samples of size 4.

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

void main()
{
    int n_random = 5;
    int n = 4;
    int m = 12;
    int l = 20;
    int *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_hypergeometric(n_random, n, m, l, 0);
    imsls_i_write_matrix("Hypergeometric random deviates: ",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Hypergeometric random deviates:
        4   2   3   3   3
```

Fatal Errors

`IMSLS_LOT_SIZE_TOO_SMALL`	The lot size must be greater than the sample size and the number of defectives in the lot. Lot size = #. Sample size = #. Number of defectives in the lot = #.

random_logarithmic

Generates pseudorandom numbers from a logarithmic distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_logarithmic` (*int* `n_random`, *float* `a`, ..., 0)

The type *double* function is `imsls_d_random_logarithmic`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Parameter of the logarithmic distribution. Parameter `a` must be positive and less than 1.0.

Return Value

An integer array of length `n_random` containing the random logarithmic deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_logarithmic` (*int* `n_random`, *float* `a`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random logarithmic deviates.

Description

Function `imsls_f_random_logarithmic` generates pseudorandom numbers from a logarithmic distribution with parameter `a`. The probability function is

$$f(x) = -\frac{a^x}{x \ln(1-a)}$$

for $x = 1, 2, 3, \ldots$, and $0 < a < 1$

The methods used are described by Kemp (1981) and depend on the value of *a*. If *a* is less than 0.95, Kemp's algorithm LS, which is a "chop-down" variant of an inverse CDF technique, is used. Otherwise, Kemp's algorithm LK, which gives special treatment to the highly probable values of 1 and 2 is used.

Example

In this example, `imsls_f_random_logarithmic` generates five pseudorandom logarithmic deviates from a logarithmic distribution with parameter a equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float a = 0.3;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_logarithmic(n_random, a, 0);
    imsls_i_write_matrix("logarithmic random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
logarithmic random deviates:
      2   1   1   1   2
```

random_neg_binomial

Generates pseudorandom numbers from a negative binomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_neg_binomial` (*int* `n_random`, *float* `rk`, *float* `p`, ..., 0)

The type double function is `imsls_d_random_neg_binomial`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `rk` (Input)
Negative binomial parameter. Parameter `rk` must be positive. If `rk` is an integer, the generated deviates can be thought of as the number of failures in a sequence of Bernoulli trials before `rk` successes occur.

float `p` (Input)
Probability of failure on each trial. Parameter `p` must be greater than machine epsilon (see `imsls_f_machine`, Chapter 15, "Utilities") and less than 1.0.

Return Value

An integer array of length `n_random` containing the random negative binomial deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_neg_binomial` (*int* `n_random`, *float* `rk`, *float* `p`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random negative binomial deviates.

Description

Function `imsls_f_random_neg_binomial` generates pseudorandom numbers from a negative binomial distribution with parameters `rk` and `p`. Parameters `rk` and `p` must be positive and `p` must be less than 1. The probability function (with r = `rk` and p = `p`) is

$$f(x) = \binom{r+x-1}{x}(1-p)^r p^x$$

for $x = 0, 1, 2, \ldots$

If r is an integer, the distribution is often called the Pascal distribution and can be thought of as modeling the length of a sequence of Bernoulli trials until r successes are obtained, where p is the probability of getting a failure on any trial. In this form, the random variable takes values r, $r + 1$, $r + 2$, … and can be obtained from the negative binomial random variable defined above by adding r to the negative binomial variable. This latter form is also equivalent to the sum of r geometric random variables defined as taking values 1, 2, 3, ...

If $rp/(1 - p)$ is less than 100 and $(1 - p)^r$ is greater than the machine epsilon, `imsls_f_random_neg_binomial` uses the inverse CDF technique; otherwise, for each negative binomial deviate, `imsls_f_random_neg_binomial` generates a gamma $(r, p/(1 - p))$ deviate Y and then generates a Poisson deviate with parameter Y.

Example

In this example, `imsls_f_random_neg_binomial` generates five pseudorandom negative binomial deviates from a negative binomial (Pascal) distribution with parameters r equal to 4 and p equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float rk = 4.0;
    float p = 0.3;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_neg_binomial(n_random, rk, p, 0);
    imsls_i_write_matrix(
        "Negative Binomial (4.0, 0.3) random deviates: ",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Negative Binomial (4.0, 0.3) random deviates:
               5   1   3   2   3
```

random_poisson

Generates pseudorandom numbers from a Poisson distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_random_poisson` (*int* `n_random`, *float* `theta`, ..., 0)

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `theta` (Input)
Mean of the Poisson distribution. Argument `theta` must be positive.

Return Value

An array of length `n_random` containing the random Poisson deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_poisson` (*int* `n_random`, *float* `theta`,
`IMSLS_RETURN_USER`, *int* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `r[]` (Output)
User-supplied array of length `n_random` containing the random Poisson deviates.

Description

Function `imsls_random_poisson` generates pseudorandom numbers from a Poisson distribution with positive mean `theta`. The probability function (with θ = `theta`) is

$$f(x) = \left(e^{-\theta}\theta^x\right)/x! \qquad \text{for } x = 0, 1, 2, \ldots$$

If `theta` is less than 15, `imsls_random_poisson` uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) (see also Schmeiser 1983) is used. The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

Example

In this example, `imsls_random_poisson` is used to generate five pseudorandom deviates from a Poisson distribution with mean equal to 0.5.

```
#include <imsls.h>

#define N_RANDOM  5

void main()
{
    int         *r;
    int         seed = 123457;
    float       theta = 0.5;

    imsls_random_seed_set (seed);
    r = imsls_random_poisson (N_RANDOM, theta, 0);
    imsls_i_write_matrix ("Poisson(0.5) random deviates", 1, N_RANDOM, r,
0);
}
```

Output

```
Poisson(0.5) random deviates
      1   2   3   4   5
      2   0   1   0   1
```

random_uniform_discrete

Generates pseudorandom numbers from a discrete uniform distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_uniform_discrete` (*int* `n_random`, *int* `k`, ..., 0)

The type *double* function is `imsls_d_random_uniform_discrete`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `k` (Input)
Parameter of the discrete uniform distribution. The integers 1, 2, ..., `k` occur with equal probability. Parameter `k` must be positive.

Return Value

An integer array of length `n_random` containing the random discrete uniform deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_uniform_discrete` (*int* `n_random`, *int* `k`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random discrete uniform deviates.

Description

Function `imsls_f_random_uniform_discrete` generates pseudorandom numbers from a uniform discrete distribution over the integers 1, 2, ...`k`. A random integer is generated by multiplying `k` by a uniform (0, 1) random number, adding 1.0, and truncating the result to an integer. This, of course, is equivalent to sampling with replacement from a finite population of size `k`

Example

In this example, `imsls_f_random_uniform_discrete` generates five pseudorandom discrete uniform deviates from a discrete uniform distribution over the integers 1 to 6.

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

void main()
{
    int n_random = 5;
    int k = 6;
    int *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_uniform_discrete(n_random, k, 0);
    imsls_i_write_matrix("Discrete uniform (1, 6) random deviates:" ,
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
Discrete uniform (1, 6) random deviates:
              6    2    5    4    6
```

random_general_discrete

Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_general_discrete` (*int* `n_random`, *int* `imin`, *int* `nmass`, *float* `probs[]`,..., 0)

The type *double* function is `imsls_d_random_general_discrete`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `imin` (Input)
Smallest value the random deviate can assume.
This is the value corresponding to the probability in `probs`[0].

int `nmass` (Input)
Number of mass points in the discrete distribution.

float `probs[]` (Input)
Array of length `nmass` containing probabilities associated with the individual mass points. The elements of `probs` must be nonnegative and must sum to 1.0.

If the optional argument `IMSLS_TABLE` is used, then `probs` is a vector of length at least `nmass` + 1 containing in the first `nmass` positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities.
IMSL routine `imsls_f_discrete_table_setup` can be used to initialize `probs` properly. If no elements of `probs` are used as indexes, `probs` [`nmass`] is 0.0 on input. The value in `probs`[0] is the probability of `imin`. The value in `probs` [`nmass-1`] must be exactly 1.0 (since this is the `CDF` at the upper range of the distribution.)

Return Value

An integer array of length `n_random` containing the random discrete deviates. To release this space, use `free`.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_f_random_general_discrete (int n_random, int imin, int
        nmass, float probs[],
        IMSLS_GET_INDEX_VECTORS, int **iwk, float **wk,
        IMSLS_GET_INDEX_VECTORS_USER, int iwk[], float wk[],
        IMSLS_SET_INDEX_VECTORS, int iwk[], float wk[],
        IMSLS_RETURN_USER, int ir[],
        IMSLS_TABLE,
         0)
```

Optional Arguments

`IMSLS_GET_INDEX_VECTORS`, *int* `**iwk`, *float* `**wk` (Output)
: Retrieve indexing vectors that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` with the same values in `probs`.

`IMSLS_GET_INDEX_VECTORS_USER`, *int* `iwk[]`, *float* `wk[]` (Output)
: User-supplied arrays of length `nmass` used for retrieve indexing vectors that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` with the same values in `probs`.

`IMSLS_SET_INDEX_VECTORS`, *int* `*iwk`, *float* `*wk` (Input)
: Arrays of length `nmass` that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` the same values in `probs`. These arrays are obtained by using one of the options `IMSLS_GET_INDEX_VECTORS` or `IMSLS_GET_INDEX_VECTORS_USER` in the first call to `imsls_f_random_general_discrete`.

`IMSLS_TABLE` (Input)
: Generate pseudorandom numbers from a general discrete distribution using a table lookup method. If this option is used, then `probs` is a vector of length at least `nmass` + 1 containing in the first `nmass` positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities.

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
: User-supplied array of length `n_random` containing the random discrete deviates.

Description

Routine `imsls_f_random_general_discrete` generates pseudorandom numbers from a discrete distribution with probability function given in the vector `probs`; that is

$$\Pr(X = i) = p_j$$

for $i = i_{\min}, i_{\min} + 1, \ldots, i_{\min} + n_m - 1$ where $j = i - i_{\min} + 1$, p_j = `probs`[*j-1*], $i_{\min}$ = `imin`, and n_m = `nmass`.

The algorithm is the *alias* method, due to Walker (1974), with modifications suggested by Kronmal and Peterson (1979). The method involves a setup phase, in which the vectors `iwk` and `wk` are filled. After the vectors are filled, the generation phase is very fast. To increase efficiency, the first call to `imsls_f_random_general_discrete` can retrieve the arrays `iwk` and `wk` using the optional arguments `IMSLS_GET_INDEX_VECTORS` or `IMSLS_GET_INDEX_VECTORS_USER` , then subsequent calls can be made using the optional argument `IMSLS_SET_INDEX_VECTORS`.

If the optional argument `IMSLS_TABLE` is used, `imsls_f_random_general_discrete` generates pseudorandom deviates from a discrete distribution, using the table `probs`, which contains the cumulative probabilities of the distribution and, possibly, indexes to speed the search of the table. The routine `imsls_f_discrete_table_setup` can be used to set up the table `probs`. `imsls_f_random_general_discrete` uses the inverse CDF method to generate the variates.

Example 1

In this example, `imsls_f_random_general_discrete` is used to generate five pseudorandom variates from the discrete distribution:

$$\Pr(X = 1) = .05$$

$$\Pr(X = 2) = .45$$

$$\Pr(X = 3) = .31$$

$$\Pr(X = 4) = .04$$

$$\Pr(X = 5) = .15$$

When `imsls_f_random_general_discrete` is called the first time, `IMSLS_GET_INDEX_VECTORS` is used to initialize the index vectors `iwk` and `wk`. In the next call, `IMSLS_GET_INDEX_VECTORS` is used, so the setup phase is bypassed.

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

void main()
{
  int nr = 5, nmass = 5, iopt = 0, imin = 1, *iwk, *ir;

  float probs[] = {.05, .45, .31, .04, .15};
  float *wk;

  imsls_random_seed_set(123457);


  ir = imsls_f_random_general_discrete(nr, imin, nmass, probs,
                                       IMSLS_GET_INDEX_VECTORS, &iwk, &wk,
                                0);

  imsls_i_write_matrix("Random deviates", 1, 5, ir,
                  IMSLS_NO_COL_LABELS,
                  0);
  free(ir);

  ir = imsls_f_random_general_discrete(nr, imin, nmass, probs,
                                IMSLS_SET_INDEX_VECTORS, iwk, wk,
                                0);

  imsls_i_write_matrix("Random deviates", 1, 5, ir,
                  IMSLS_NO_COL_LABELS,
                  0);

}
```

Output

```
  Random deviates
 3   2   2   3   5

  Random deviates
 1   3   4   5   3
```

Example 2

In this example, imsls_f_discrete_table_setup is used to set up a table and then imsls_f_random_general_discrete is used to generate five pseudorandom variates from the binomial distribution with parameters 20 and 0.5.

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

float prf(int ix);
void main()
{
  int nndx = 12, imin = 0, nmass = 21, nr = 5;
  float del = 0.00001, *cumpr;
  int *ir = NULL;


  cumpr = imsls_f_discrete_table_setup (prf,  del, nndx,  &imin, &nmass, 0);

  imsls_random_seed_set(123457);

  ir = imsls_f_random_general_discrete(nr, imin, nmass, cumpr,
                                IMSLS_TABLE, 0);

  imsls_i_write_matrix("Binomial (20, 0.5) random deviates", 1, 5, ir,
                  IMSLS_NO_COL_LABELS,
                  0);

}
```

```
float prf(int ix)
{
  int n = 20;
  float  p = .5;
  return imsls_f_binomial_probability (ix, n, p);
}
```

Output

```
Binomial (20, 0.5) random deviates
       14    9   12   10   12
```

discrete_table_setup

Sets up table to generate pseudorandom numbers from a general discrete distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_discrete_table_setup` (*float* `prf()`, *float* `del`,
  int `nndx`, *int* `*imin`, *int* `*nmass`, ..., 0)

The type *double* function is `imsls_d_discrete_table_setup`.

Required Arguments

float `prf(`*int* `ix)` (Input)
: User-supplied function to compute the probability associated with each mass point of the distribution The argument to the function is the point at which the probability function is to be evaluated. `ix` can range from `imin` to the value at which the cumulative probability is greater than or equal to 1.0 – `del`.

float `del` (Input)
: Maximum absolute error allowed in computing the cumulative probability. Probabilities smaller than `del` are ignored; hence, `del` should be a small positive number. If `del` is too small, however, the return value, `cumpr` [`nmass-1`] must be exactly 1.0 since that value is compared to 1.0 – `del`.

int `nndx` (Input)
: The number of elements of `cumpr` available to be used as indexes. `nndx` must be greater than or equal to 1. In general, the larger `nndx` is, to within sixty or seventy percent of `nmass`, the more efficient the generation of random numbers using `imsls_f_random_general_discrete` will be.

int `*imin` (Input/Output)
: Pointer to a scalar containing the smallest value the random deviate can assume.   (Input/Output)
`imin` is not used if optional argument `IMSLS_INDEX_ONLY` is used. By default, `prf` is evaluated at `imin`. If this value is less than `del`, `imin` is incremented by 1 and again `prf` is evaluated at `imin`. This process is

continued until `prf(imin)` ≥ `del`. `imin` is output as this value and the return value `cumpr` [0] is output as `prf(imin)`.

int `*nmass` (Input/Output)
Pointer to a scalar containing the number of mass points in the distribution. Input, if `IMSLS_INDEX_ONLY` is used; otherwise, output.
By default, `nmass` is the smallest integer such that
`prf(imin` + `nmass` – 1) > 1.0 – `del`. `nmass` does include the points $\texttt{imin}_{\text{in}}$ + *j* for which `prf`($\texttt{imin}_{\text{in}}$ + *j*) < `del`, for *j* = 0, 1, …,
$\texttt{imin}_{\text{out}}$ – $\texttt{imin}_{\text{in}}$, where $\texttt{imin}_{\text{in}}$ denotes the input value of `imin` and $\texttt{imin}_{\text{out}}$ denotes its output value.

Return Value

Array, `cumpr`, of length `nmass` + `nndx` containing in the first `nmass` positions, the cumulative probabilities and in some of the remaining positions, indexes to speed access to the probabilities. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_discrete_table_setup` (*float* `prf()`, *float* `del`, *int* `nndx`, *int* `*imin`, *int* `*nmass`,
`IMSLS_INDEX_ONLY`,
`IMSLS_RETURN_USER`, *float* `cumpr[]`, *int* `lcumpr`,
`IMSLS_FCN_W_DATA`, *float* `prf()`, *void* `*data`,
0)

Optional Arguments

`IMSLS_INDEX_ONLY` (Intput)
Fill only the index portion of the result, `cumpr`, using the values in the first `nmass` positions. `prf` is not used and may be a dummy function; also, `imin` is not used. The optional argument `IMSLS_RETURN_USER` is required if `IMSLS_INDEX_ONLY` is used.

`IMSLS_RETURN_USER`, *float* `cumpr[]`, *int* `lcumpr` (Input/Output)
`cumpr` is a user-allocated array of length `nmass` + `nndx` containing in the first `nmass` positions, the cumulative probabilities and in some of the remaining positions, indexes to speed access to the probabilities. `lcumpr` is the actual length of `cumpr` as specified in the calling function. Since, by default, the logical length of `cumpr` is determined in `imsls_f_discrete_table_setup`, `lcumpr` is used for error checking. If the option `IMSLS_INDEX_ONLY` is used, then only the index portion of `cumpr` are filled.

`IMSLS_FCN_W_DATA`, *float* `prf(`*int* `ix)`, *void* `*data`, (Input)
User-supplied function to compute the probability associated with each mass point of the distribution, 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

Routine imsls_f_discrete_table_setup sets up a table that routine imsls_f_random_general_discrete uses to generate pseudorandom deviates from a discrete distribution. The distribution can be specified either by its probability function prf or by a vector of values of the cumulative probability function. Note that prf is *not* the cumulative probability distribution function. If the cumulative probabilities are already available in cumpr, the only reason to call imsls_f_discrete_table_setup is to form an index vector in the upper portion of cumpr so as to speed up the generation of random deviates by the routine imsls_f_random_general_discrete.

Example 1

In this example, imsls_f_discrete_table_setup is used to set up a table to generate pseudorandom variates from the discrete distribution:

$$\Pr(X = 1) = .05$$

$$\Pr(X = 2) = .45$$

$$\Pr(X = 3) = .31$$

$$\Pr(X = 4) = .04$$

$$\Pr(X = 5) = .15$$

In this simple example, we input the cumulative probabilities directly in cumpr and request 3 indexes to be computed (nndx = 4). Since the number of mass points is so small, the indexes would not have much effect on the speed of the generation of the random variates.

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

float prf(int ix);
void main()
{
  int i, lcumpr = 9, ir[5];
  int nndx = 4, imin = 1, nmass = 5, nr = 5;

  float cumpr[9], del = 0.00001, *p_cumpr = NULL;
  i = 0;
  cumpr[i++] = .05;
  cumpr[i++] = .5;
  cumpr[i++] = .81;
  cumpr[i++] = .85;
  cumpr[i++] = 1.0;

 imsls_f_discrete_table_setup (prf,  del,
                              nndx,   &imin, &nmass,
                              IMSLS_INDEX_ONLY,
                              IMSLS_RETURN_USER, cumpr, lcumpr,
                              0);
  imsls_f_write_matrix("Cumulative probabilities and indexes",
                     1, lcumpr, cumpr, 0);

}

float prf(int ix)
{
  return 0.;

}
```

Output

1.

```
          Cumulative probabilities and indexes
    1           2           3           4           5           6
 0.05        0.50        0.81        0.85        1.00        3.00

    7           8           9
 1.00        2.00        5.00
```

Example 2

This example, `imsls_f_random_general_discrete` is used to set up a table to generate binomial variates with parameters 20 and 0.5. The routine `imsls_f_binomial_probabililty` (Chapter 11, Probability Distribution Functions and Inverses) is used to compute the probabilities.

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

float prf(int ix);
void main()
{
  int lcumpr = 33;
  int nndx = 12, imin = 0, nmass = 21, nr = 5;
  float del = 0.00001, *cumpr;
  int *ir = NULL;


  cumpr = imsls_f_discrete_table_setup (prf,  del, nndx,  &imin, &nmass, 0);

  printf("The smallest point with positive probability using \n");
  printf("the given del is %d and all points after \n", imin);
  printf("point number %d (counting from the input value\n", nmass);
  printf("of IMIN) have zero probability.\n");
  imsls_f_write_matrix("Cumulative probabilities and indexes",
                     nmass+nndx, 1, cumpr,
                     IMSLS_WRITE_FORMAT, "%11.7f", 0);

}
```

```
float prf(int ix)
{
  int n = 20;
  float  p = .5;
  return imsls_f_binomial_probability(ix, n, p);
}
```

Output

2.

```
The smallest point with positive probability using
the given del is 1 and all points after
point number 19 (counting from the input value
of IMIN) have zero probability.

Cumulative probabilities and indexes
            1    0.0000191
            2    0.0002003
            3    0.0012875
            4    0.0059080
            5    0.0206938
            6    0.0576583
            7    0.1315873
            8    0.2517219
            9    0.4119013
           10    0.5880987
           11    0.7482781
           12    0.8684127
           13    0.9423417
           14    0.9793062
           15    0.9940920
           16    0.9987125
           17    0.9997997
           18    0.9999809
           19    1.0000000
           20   11.0000000
           21    1.0000000
           22    7.0000000
           23    8.0000000
           24    9.0000000
           25    9.0000000
           26   10.0000000
           27   11.0000000
           28   11.0000000
           29   12.0000000
           30   13.0000000
           31   19.0000000
```

random_beta

Generates pseudorandom numbers from a beta distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_beta` (*int* `n_random`, *float* `pin`, *float* `qin`, ..., 0)

The type *double* function is `imsls_d_random_beta`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

If no optional arguments are used, `imsls_f_random_beta` returns an array of length `n_random` containing the random standard beta deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_beta` (*int* `n_random`, *float* `pin`, *float* `qin`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
Array of length `n_random` containing the random standard beta deviates.

Description

Function `imsls_f_random_beta` generates pseudorandom numbers from a beta distribution with parameters `pin` and `qin`, both of which must be positive. With p = `pin` and q = `qin`, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1} (1-x)^{q-1} \qquad \text{for } 0 \le x \le 1$$

where $\Gamma(\cdot)$ is the gamma function.

The algorithm used depends on the values of p and q. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all of the methods use acceptance/rejection. If p and q are both less than 1, the method of Jöhnk (1964) is used. If either *p* or *q* is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both *p* and *q* are greater than 1, algorithm BB (Cheng 1978), which requires very little setup time, is used if `n_random` is less than 4; and algorithm B4PE of Schmeiser and Babu (1980) is used if `n_random` is greater than or equal to 4. Note that for *p* and *q* both greater than 1, calling `imsls_f_random_beta` in a loop getting less than four variates on each call will not yield the same set of deviates as calling `imsls_f_random_beta` once and getting all the deviates at once because two different algorithms are used.

The values returned in `r` are less than 1.0 and greater than ε, where ε is the smallest positive number such that 1.0 – ε is less than 1.0.

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

Example

In this example, `imsls_f_random_beta` generates five pseudorandom beta (3, 2) variates.

```
#include <imsls.h>

main()
{

    int         n_random = 5;
    int         seed = 123457;
    float       pin = 3.0;
    float       qin = 2.0;
    float       *r;

    imsls_random_seed_set (seed);
    r = imsls_f_random_beta (n_random, pin, qin, 0);
    imsls_f_write_matrix("Beta (3,2) random deviates", 1, n_random,
                         r, 0);
}
```

Output

```
                Beta (3,2) random deviates
         1           2           3           4           5
    0.2814      0.9483      0.3984      0.3103      0.8296
```

random_cauchy

Generates pseudorandom numbers from a Cauchy distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_cauchy` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_cauchy`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random Cauchy deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_cauchy` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random Cauchy deviates.

Description

Function `imsls_f_random_cauchy` generates pseudorandom numbers from a Cauchy distribution. The probability density function is

$$f(x) = \frac{S}{\pi[S^2 + (x-T)^2]}$$

where T is the median and $T - S$ is the first quartile. This function first generates standard Cauchy random numbers ($T = 0$ and $S = 1$) using the technique described below, and then scales the values using T and S.

Use of the inverse CDF technique would yield a Cauchy deviate from a uniform (0, 1) deviate, u, as tan [π (u − 0.5)]. Rather than evaluating a tangent directly, however, `random_cauchy` generates two uniform (−1, 1) deviates, x_1 and x_2. These values can be thought of as sine and cosine values. If

$$x_1^2 + x_2^2$$

is less than or equal to 1, then x_1/x_2 is delivered as the unscaled Cauchy deviate; otherwise, x_1 and x_2 are rejected and two new uniform (−1, 1) deviates are generated. This method is also equivalent to taking the ration of two independent normal deviates.

Example

In this example, `imsls_f_random_cauchy` generates five pseudorandom Cauchy numbers. The generator used is a simple multiplicative congruential with a multiplier of 16807.

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

```
void main()
{
    int n_random = 5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_cauchy(n_random, 0);
    printf("Cauchy random deviates: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
        r[0], r[1], r[2], r[3], r[4]);

}
```

Output

```
Cauchy random deviates:   3.5765  0.9353 15.5797  2.0815 -0.1333
```

random_chi_squared

Generates pseudorandom numbers from a chi-squared distribution.

Synopsis

#include <imsls.h>

float *imsls_f_random_chi_squared (*int* n_random, *float* df, ..., 0)

The type *double* function is imsls_d_random_chi_squared.

Required Arguments

int n_random (Input)
Number of random numbers to generate.

float df (Input)
Degrees of freedom. Parameter df must be positive.

Return Value

An array of length n_random containing the random chi-squared deviates.

Synopsis with Optional Arguments

#include <imsls.h>

float *imsls_f_random_chi_squared (*int* n_random, *float* df,
IMSLS_RETURN_USER, *float* r[],
0)

Optional Arguments

IMSLS_RETURN_USER, *float* r[] (Output)
User-supplied array of length n_random containing the random chi-squared deviates.

Description

Function `imsls_f_random_chi_squared` generates pseudorandom numbers from a chi-squared distribution with `df` degrees of freedom. If `df` is an even integer less than 17, the chi-squared deviate r is generated as

$$r = -2\ln\left(\prod_{i=1}^{n} u_i\right)$$

where n = `df`/2 and the u_i are independent random deviates from a uniform (0, 1) distribution. If `df` is an odd integer less than 17, the chi-squared deviate is generated in the same way, except the square of a normal deviate is added to the expression above. If `df` is is greater than 16 or is not an integer, and if it is not too large to cause overflow in the gamma random number generator, the chi-squared deviate is generated as a special case of a gamma deviate, using function `imsls_f_random_gamma`. If overflow would occur in `imsls_f_random_gamma`, the chi-squared deviate is generated in the manner described above, using the logarithm of the product of uniforms, but scaling the quantities to prevent underflow and overflow.

Example

In this example, `imsls_f_random_chi_squared` generates five pseudorandom chi-squared deviates with five degrees of freedom.

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

void main()
{
    int   n_random = 5;
    float df = 5.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_chi_squared(n_random, df, 0);
    imsls_f_write_matrix("Chi-Squared random deviates: ",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
             Chi-Squared random deviates:
     12.09        0.48        1.80       14.87        1.75
```

random_exponential

Generates pseudorandom numbers from a standard exponential distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_exponential` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_exponential`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random standard exponential deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_exponential` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random standard exponential deviates.

Description

Function `imsls_f_random_exponential` generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. Function `imsls_f_random_exponential` uses an antithetic inverse CDF technique; that is, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at $1.0 - U$ to yield the exponential deviate.

Deviates from the exponential distribution with mean θ can be generated by using `imsls_f_random_exponential` and then multiplying each entry in `r` by θ.

Example

In this example, `imsls_f_random_exponential` generates five pseudorandom deviates from a standard exponential distribution.

```
#include <imsls.h>

#define N_RANDOM    5

main()

{
        int             seed = 123457;
        int             n_random = N_RANDOM;
        float           *r;

        imsls_random_seed_set(seed);
        r = imsls_f_random_exponential(n_random, 0);
        printf("%s: %8.4f%8.4f%8.4f%8.4f\n",
               "Exponential random deviates",
              r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Exponential random deviates:   0.0344  1.3443  0.2662  0.5633  0.1686
```

random_exponential_mix

Generates pseudorandom numbers from a mixture of two exponential distributions.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_exponential_mix` (*int* `n_random`, *float* `theta1`,
    float `theta2`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_exponential_mix`.

Required Arguments

int `n_random` (Input)
    Number of random numbers to generate.

float `theta1` (Input)
    Mean of the exponential distribution which has the larger mean.

float `theta2` (Input)
    Mean of the exponential distribution which has the smaller mean. Parameter `theta2` must be positive and less than or equal to `theta1`.

float `p` (Input)
    Mixing parameter. Parameter `p` must be non-negative and less than or equal to `theta1`/(`theta1` − `theta2`).

Return Value

An array of length `n_random` containing the random deviates of a mixture of two exponential distributions.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_exponential_mix` (*int* `n_random`, *float* `theta1`,
    float `theta2`, *float* `p`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
    User-supplied array of length `n_random` containing the random deviates.

Description

Function imsls_f_random_exponential_mix generates pseudorandom numbers from a mixture of two exponential distributions. The probability density function is

$$f(x) = \frac{p}{\theta_1} e^{-x/\theta_1} + \frac{1-p}{\theta_2} e^{-x/\theta_2}$$

for $x > 0$, where p = p, θ_1 = theta1, and θ_2 = theta2.

In the case of a convex mixture, that is, the case $0 < p < 1$, the mixing parameter p is interpretable as a probability; and imsls_f_random_exponential_mix with probability p generates an exponential deviate with mean θ_1, and with probability $1 - p$ generates an exponential with mean θ_2. When p is greater than 1, but less than $\theta_1/(\theta_1 - \theta_2)$, then either an exponential deviate with mean θ_1 or the sum of two exponentials with means θ_1 and θ_2 is generated. The probabilities are $q = p - (p - 1)(\theta_1/\theta_2)$ and $1 - q$, respectively, for the single exponential and the sum of the two exponentials.

Example

In this example, imsls_f_random_exponential_mix is used to generate five pseudorandom deviates from a mixture of exponentials with means 2 and 1, respecctively, and with mixing parameter 0.5.

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

void main()
{
    int   n_random = 5;
    float theta1 = 2.0;
    float theta2 = 1.0;
    float p = 0.5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_exponential_mix(n_random, theta1, theta2, p, 0);
    imsls_f_write_matrix("Mixed exponential random deviates: ",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
              Mixed exponential random deviates:
     0.070       1.302       0.630       1.976       0.372
```

random_gamma

Generates pseudorandom numbers from a standard gamma distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_gamma` (*int* `n_random`, *float* `a`, ..., 0)

The type *double* function is `imsls_d_random_gamma`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Shape parameter of the gamma distribution. This parameter must be positive.

Return Value

An array of length `n_random` containing the random standard gamma deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_gamma` (*int* `n_random`, *float* `a`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_USER_RETURN`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random standard gamma deviates.

Description

Function `imsls_f_random_gamma` generates pseudorandom numbers from a gamma distribution with shape parameter a and unit scale parameter. The probability density function is

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \qquad \text{for } x \ge 0$$

Various computational algorithms are used depending on the value of the shape parameter a. For the special case of $a = 0.5$, squared and halved normal deviates are used; for the special case of $a = 1.0$, exponential deviates are generated. Otherwise, if a is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used. If a is greater than 1.0, a ten-region rejection procedure developed by Schmeiser and Lal (1980) is used.

Deviates from the two-parameter gamma distribution with shape parameter *a* and scale parameter *b* can be generated by using imsls_f_random_gamma and then multiplying each entry in *r* by *b*. The following statements (in single precision) would yield random deviates from a gamma (*a*, *b*) distribution.

```
float *r;
r = imsls_f_random_gamma(n_random, a, 0);
for (i=0; i<n_random; i++) *(r+i) *= b;
```

The Erlang distribution is a standard gamma distribution with the shape parameter having a value equal to a positive integer; hence, imsls_f_random_gamma generates pseudorandom deviates from an Erlang distribution with no modifications required.

Function imsls_random_seed_set can be used to initialize the seed of the random number generator; function imsls_random_option can be used to select the form of the generator.

Example

In this example, `imsls_f_random_gamma` generates five pseudorandom deviates from a gamma (Erlang) distribution with shape parameter equal to 3.0.

```
#include <imsls.h>

void main()
{
    int         seed = 123457;
    int         n_random = 5;
    float       a = 3.0;
    float       *r;

    imsls_random_seed_set(seed);
    r = imsls_f_random_gamma(n_random, a, 0);
    imsls_f_write_matrix("Gamma(3) random deviates", 1, n_random, r, 0);
}
```

Output

```
          Gamma(3) random deviates
     1          2          3          4          5
 6.843      3.445      1.853      3.999      0.779
```

random_lognormal

Generates pseudorandom numbers from a lognormal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_lognormal` (*int* `n_random`, *float* `mean`, *float* `std`, ..., 0)

The type *double* function is `imsls_d_random_lognormal`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `mean` (Input)
Mean of the underlying normal distribution.

float `std` (Input)
Standard deviation of the underlying normal distribution.

Return Value

An array of length `n_random` containing the random deviates of a lognormal distribution. The log of each element of the vector has a normal distribution with mean `mean` and standard deviation `std`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_lognormal` (*int* `n_random`, *float* `mean`, *float* `std`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random lognormal deviates.

Description

Function `imsls_f_random_lognormal` generates pseudorandom numbers from a lognormal distribution with parameters `mean` and `std`. The scale parameter in the underlying normal distribution, `std`, must be positive. The method is to generate normal deviates with mean `mean` and standard deviation `std` and then to exponentiate the normal deviates.

With μ = `mean` and σ = `std`, the probability density function for the lognormal distribution is

$$f(x) = \frac{1}{\sigma x \sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma^2}(\ln x - \mu)^2\right]$$

for $x > 0$. The mean and variance of the lognormal distribution are $\exp(\mu + \sigma^2/2)$ and $\exp(2\mu + 2\sigma^2) - \exp(2\mu + \sigma^2)$, respectively.

Example

In this example, `imsls_f_random_lognormal` is used to generate five pseudorandom lognormal deviates with a mean of 0 and standard deviation of 1.

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

```
void main()
{
    int   n_random = 5;
    float mean = 0.0;
    float std = 1.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_lognormal(n_random, mean, std, 0);
    imsls_f_write_matrix("lognormal random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                 lognormal random deviates:
     7.780       2.954       1.086       3.588       0.293
```

random_normal

Generates pseudorandom numbers from a normal, N (μ, σ^2), distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_normal` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_normal`.

Required Arguments

int `n_random` (Input)
  Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random normal deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_normal` (*int* `n_random`,
  `IMSLS_MEAN`, *float* `mean`,
  `IMSLS_VARIANCE`, *float* `variance`,
  `IMSLS_ACCEPT_REJECT_METHOD`,
  `IMSLS_RETURN_USER`, *float* `r[]`,
  0)

Optional Arguments

`IMSLS_MEAN`, *float* `mean` (Input)
  Parameter `mean` contains the mean, μ, of the N(μ, σ^2) from which random normal deviates are to be generated.
  Default: `mean` = 0.0

IMSLS_VARIANCE, *float* variance (Input)

Parameter variance contains the variance of the N (μ, σ^2) from which random normal deviates are to be generated.
Default: variance = 1.0

IMSLS_ACCEPT_REJECT_METHOD

By default, random numbers are generated using an inverse CDF technique. When optional argument IMSLS_ACCEPT_REJECT_METHOD is specified, an acceptance/ rejection method is used instead. See the "Description" section for details about each method.

IMSLS_RETURN_USER, *float* r[] (Output)

User-supplied array of length n_random containing the generated random standard normal deviates.

Description

By default, function imsls_f_random_normal generates pseudorandom numbers from a normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0, 1) random deviate is generated. The inverse of the normal distribution function is then evaluated at that point, using the function imsls_f_normal_inverse_cdf (Chapter 11, Probablility Distribution Functions and Inverses).

If optional argument IMSLS_ACCEPT_REJECT_METHOD is specified, function imsls_f_random_normal generates pseudorandom numbers using an acceptance/rejection technique due to Kinderman and Ramage (1976). In this method, the normal density is represented as a mixture of densities over which a variety of acceptance/rejection method due to Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia et al. (1964) are applied. This method is faster than the inverse CDF technique.

Remarks

Function imsls_random_seed_set can be used to initialize the seed of the random number generator; function imsls_random_option can be used to select the form of the generator.

Example

In this example, imsls_f_random_normal generates five pseudorandom deviates from a standard normal distribution.

```
#include <imsls.h>
#define N_RANDOM  5

void main()
{
    int         seed = 123457;
    int         n_random = N_RANDOM;
    float       *r;

    imsls_random_seed_set (seed);
    r = imsls_f_random_normal(n_random, 0);
    printf("%s:\n%8.4f%8.4f%8.4f%8.4f%8.4f\n",
           "Standard normal random deviates",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Standard normal random deviates:
  1.8279 -0.6412  0.7266  0.1747  1.0145
```

random_stable

Generates pseudorandom numbers from a stable distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_stable` (*int* `n_random`, *float* `alpha`,
    float `bprime`, ..., 0)

The type *double* function is `imsls_d_random_stable`.

Required Arguments

int `n_random` (Input)
    Number of random numbers to generate.

float `alpha` (Input)
    Characteristic exponent of the stable distribution. This parameter must be positive and less than or equal to 2.

float `bprime` (Input)
    Skewness parameter of the stable distribution. When `bprime` = 0, the distribution is symmetric. Unless `alpha` = 1, `bprime` is not the usual skewness parameter of the stable distribution. `bprime` must be greater than or equal to – 1 and less than or equal to 1.

Return Value

An integer array of length `n_random` containing the random deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_binomial` (*int* `n_random`, *float* `alpha`,
    float `bprime`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
    User-supplied array of length `n_random` containing the random deviates.

Description

Function imsls_f_random_stable generates pseudorandom numbers from a stable distribution with parameters `alpha` and `bprime`. `alpha` is the usual characteristic exponent parameter α and `bprime` is related to the usual skewness parameter β of the stable distribution. With the restrictions $0 < \alpha \le 2$ and $-1 \le \beta \le 1$, the characteristic function of the distribution is

$$\varphi(t) = \exp[-|\,t\,|^{\alpha} \exp(-\pi i\beta(1 - |1 - \alpha|)\text{sign}(t)/2)] \qquad \text{for } \alpha \ne 1$$

and

$$\varphi(t) = \exp[-|\,t\,|(1 + 2i\beta \ln|\,t\,|)\text{sign}(t)/\pi)] \qquad \text{for } \alpha = 1$$

When $\beta = 0$, the distribution is symmetric. In this case, if $\alpha = 2$, the distribution is normal with mean 0 and variance 2; and if $\alpha = 1$, the distribution is Cauchy.

The parameterization using `bprime` and the algorithm used here are due to Chambers, Mallows, and Stuck (1976). The relationship between `bprime` = β' and the standard β is

$$\beta' = -\tan(\pi(1 - \alpha)/2)\tan(-\pi\beta(1 - |1 - \alpha|)/2) \quad \text{for } \alpha \ne 1$$

and

$$\beta' = \beta \qquad \text{for } \alpha = 1$$

The algorithm involves formation of the ratio of a uniform and an exponential random variate.

Example

In this example, `imsls_f_random_stable` is used to generate five pseudorandom symmetric stable variates with characteristic exponent 1.5. The tails of this distribution are heavier than those of a normal distribution, but not so heavy as those of a Cauchy distribution. The variance of this distribution does not exist, however. (This is the case for any stable distribution with characteristic exponent less than 2.)

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

void main()
{
  int  nr = 5;
  float alpha = 1.5, bprime = 0.0, *r;

  imsls_random_seed_set(123457);

  r = imsls_f_random_stable(nr, alpha, bprime, 0);
  imsls_f_write_matrix("Stable random deviates", 5, 1, r,
                       IMSLS_NO_ROW_LABELS, 0);

}
```

Output

```
Stable random deviates
          4.409
          1.056
          2.546
          5.672
          2.166
```

random_student_t

Generates pseudorandom numbers from a Student's *t* distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_student_t` (*int* `n_random`, *float* `df`, ..., 0)

The type *double* function is `imsls_d_random_student_t`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `df` (Input)
Degrees of freedom. Parameter `df` must be positive.

Return Value

An array of length `n_random` containing the random deviates of a Student's *t* distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_student_t` (*int* `n_random`, *float* `df`,
`IMSLS_RETURN_USER`, *float* `r[]`,
`IMSLS_MEAN`, *float* `mean`,
`IMSLS_VARIANCE`, *float* `variance`,
0)

Optional Arguments

`IMSLS_MEAN`, *float* `mean` (Input)
Mean of the Student's *t* distribution.
Default: `mean` = 0.0

`IMSLS_VARIANCE`, *float* `variance` (Input)
Variance of the Student's *t* distribution.
Default: `variance` = 1.0

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random Student's *t* deviates.

Description

Function `imsls_f_random_student_t` generates pseudorandom numbers from a Student's *t* distribution with `df` degrees of freedom, using a method suggested by Kinderman et al. (1977). The method ("TMX" in the reference) involves a representation of the *t* density as the sum of a triangular density over (−2, 2) and the difference of this and the *t* density. The mixing probabilities depend on the degrees of freedom of the *t* distribution. If the triangular density is chosen, the variate is generated as the sum of two uniforms; otherwise, an acceptance/rejection method is used to generate the difference density.

random_triangular

Generates pseudorandom numbers from a triangular distribution on the interval (0, 1).

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_triangular` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_triangular`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random deviates of a triangular distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_triangular` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random triangular deviates.

Description

Function `imsls_f_random_triangular` generates pseudorandom numbers from a triangular distribution over the unit interval. The probability density function is $f(x) = 4x$, for $0 \le x \le 0.5$, and $f(x) = 4(1 - x)$, for $0.5 < x \le 1$. An inverse CDF technique is used.

Example

In this example, `imsls_f_random_triangular` is used to generate five pseudorandom deviates from a triangular distribution.

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

void main()
{
    int   n_random = 5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_triangular(n_random, 0);
    imsls_f_write_matrix("Triangular random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
            Triangular random deviates:
    0.8700      0.3610      0.6581      0.5360      0.7215
```

random_uniform

Generates pseudorandom numbers from a uniform (0, 1) distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_uniform` (*int* `n_random`, …, 0)

The type *double* function is `imsls_d_random_uniform`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random uniform (0, 1) deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_uniform` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random uniform (0, 1) deviates.

Description

Function `imsls_f_random_uniform` generates pseudorandom numbers from a uniform (0, 1) distribution using a multiplicative congruential method. The form of the generator is as follows:

$$x_i \equiv c x_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0, 1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by the function `imsls_random_option`. The choice of 16807 will result in the fastest execution time. If no selection is made explicitly, the functions use the multiplier 16807.

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

The user can select a shuffled version of these generators. In this scheme, a table is filled with the first 128 uniform (0, 1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number, and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table.

The values returned by `imsls_f_random_uniform` are positive and less than 1.0. However, some values returned may be smaller than the smallest relative spacing; hence, it may be the case that some value, for example `r [i]`, is such that 1.0 – `r [i]` = 1.0.

Deviates from the distribution with uniform density over the interval (a, b) can be obtained by scaling the output from `imsls_f_random_uniform`. The following statements (in single precision) would yield random deviates from a uniform (a, b) distribution.

```
float *r;
r = imsls_f_random_uniform (n_random, 0);
for (i=0; i<n_random; i++) r[i] = r[i]*(b-a) + a;
```

Example

In this example, `imsls_f_random_uniform` generates five pseudorandom uniform numbers. Since function `imsls_random_option` is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

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

#define N_RANDOM  5
```

```
void main()
{
    float     *r;

    imsls_random_seed_set(123457);

    r = imsls_f_random_uniform(N_RANDOM, 0);

    printf("Uniform random deviates: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
            r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Uniform random deviates:   0.9662  0.2607  0.7663  0.5693  0.8448
```

random_von_mises

Generates pseudorandom numbers from a von mises distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_von_mises` (*int* `n_random`, *float* `c`, …, 0)

The type *double* function is `imsls_d_random_von_mises`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `c` (Input)
Parameter of the von Mises distribution. This parameter must be greater than one-half of machine epsilon (On many machines, the lower bound for `c` is 10^{-3}).

Return Value

An array of length `n_random` containing the random deviates of a von Mises distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_von_mises` (*int* `n_random`, *float* `c`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random von mises deviates.

Description

Function `imsls_f_random_von_mises` generates pseudorandom numbers from a von Mises distribution with parameter `c`, which must be positive. With c = `c`, the probability density function is

$$f(x) = \frac{1}{2\pi I_0(c)} \exp\left[c \cos(x)\right]$$

for $-\pi < x < \pi$, where $I_0(c)$ is the modified Bessel function of the first kind of order 0. The probability density is equal to 0 outside the interval $(-\pi, \pi)$.

The algorithm is an acceptance/rejection method using a wrapped Cauchy distribution as the majorizing distribution. It is due to Nest and Fisher (1979).

Example

In this example, `imsls_f_random_von_mises` is used to generate five pseudorandom von Mises variates with $c = 1$.

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

void main()
{
    int   n_random = 5;
    float c = 1.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_von_mises(n_random, c, 0);
    imsls_f_write_matrix("Von Mises random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
              Von Mises random deviates:
     0.247      -2.433      -1.022      -2.172      -0.503
```

random_weibull

Generates pseudorandom numbers from a Weibull distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_weibull` (*int* `n_random`, *float* `a`, …, 0)

The type *double* function is `imsls_d_random_weibull`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Shape parameter of the Weibull distribution. This parameter must be positive.

Return Value

An array of length `n_random` containing the random deviates of a Weibull distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_weibull` (*int* `n_random`, *float* `a`,
`IMSLS_B`, *float* `b`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_B`, *float* `b` (Input)
Scale parameter of the two parameter Weibull distribution.
Default: `b` = 1.0

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random Weibull deviates.

Description

Function `imsls_f_random_weibull` generates pseudorandom numbers from a Weibull distribution with shape parameter *a* and scale parameter *b*. The probability density function is

$$f(x) = abx^{a-1}\exp\left(-bx^{a}\right)$$

for $x \ge 0$, $a > 0$, and $b > 0$. Function `imsls_f_random_weibull` uses an antithetic inverse CDF technique to generate a Weibull variate; that is, a uniform random deviate *U* is generated and the inverse of the Weibull cumulative distribution function is evaluated at $1.0 - U$ to yield the Weibull deviate.

Note that the Rayleigh distribution with probability density function

$$r(x) = \frac{1}{\alpha^2} x e^{-\left(x^2/\left(2\alpha^2\right)\right)}$$

for $x \ge 0$ is the same as a Weibull distribution with shape parameter *a* equal to 2 and scale parameter *b* equal to

$$\sqrt{2}\alpha$$

Example

In this example, `imsls_f_random_weibull` is used to generate five pseudorandom deviates from a two-parameter Weibull distribution with shape parameter equal to 2.0 and scale parameter equal to 6.0—a Rayleigh distribution with the following parameter:

$$\alpha = 3\sqrt{2}$$

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

void main()
{
    int   n_random = 5;
    float a = 3.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_weibull(n_random, a, 0);
    imsls_f_write_matrix("Weibull random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
           Weibull random deviates:
     0.325       1.104       0.643       0.826       0.552
```

Warning Errors

`IMSLS_SMALL_A`	The shape parameter is so small that a relatively large proportion of the values of deviates from the Weibull cannot be represented.

random_general_continuous

Generates pseudorandom numbers from a general continuous distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_general_continuous` (*int* `n_random`, *int* `ndata`, *float* `table[]`,..., 0)

The type *double* function is `imsls_d_random_general_continuous`.

Required Arguments

int `n_random` (Input)
: Number of random numbers to generate.

int `ndata` (Input)
: Number of points at which the CDF is evaluated for interpolation. `ndata` must be greater than or equal to 4.

float `*table` (Input/Ouput)
: `ndata` by 5 table to be used for interpolation of the cumulative distribution function.
The first column of `table` contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing beginning with 0.0 and ending at 1.0) and the remaining columns contain values used in interpolation. This table is set up using routine `imsls_f_continous_table_setup`.

Return Value

An array of length `n_random` containing the random discrete deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_general_continuous` (*int* `n_random`, *int* `ndata`, *float* `table[]`,
`IMSLS_TABLE_COL_DIM`, *int* `table_col_dim`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_TABLE_COL_DIM`, *int* `table_col_dim` (Intput)
: Column dimension of the matrix `table`.
Default: `table_col_dim` = 5

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
: User-supplied array of length `n_random` containing the random continuous deviates.

Description

Routine `imsls_f_random_general_continuous` generates pseudorandom numbers from a continuous distribution using the inverse CDF technique, by interpolation of points of the distribution function given in `table`, which is set up by routine `imsls_f_continuous_table_setup`. A strictly monotone increasing distribution function is assumed. The interpolation is by an algorithm attributable to Akima (1970), using piecewise cubics. The use of this technique for generation of random numbers is due to Guerra, Tapia, and Thompson (1976), who give a description of the algorithm and accuracy comparisons between this method and linear interpolation. The relative errors using the Akima interpolation are generally considered very good.

Example 1

In this example, `imsls_f_continuous_table_setup` is used to set up a table for generation of beta pseudorandom deviates. The CDF for this distribution is computed by the routine `imsls_f_beta_cdf` (Chapter 11, Probability Distribution Functions and Inverses). The table contains 100 points at which the CDF is evaluated and that are used for interpolation.

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

float cdf(float);
void main()
{
  int i, iopt=0, ndata= 100;
  float table[100][5], x = 0.0, *r;

  for (i=0;i<ndata;i++) {
    table[i][0] = x;
    x += .01;
  }

  imsls_f_continuous_table_setup(cdf, iopt, ndata, (float*)table);

  imsls_random_seed_set(123457);
  r = imsls_f_random_general_continuous (5,  ndata, table, 0);
  imsls_f_write_matrix("Beta  (3, 2) random deviates", 5, 1, r, 0);

}

float cdf(float x)
{
  return imsls_f_beta_cdf(x, 3., 2.);
}
```

Output

```
*** WARNING  Error  from imsls_f_continuous_table_setup.  The values of the
***          CDF in the second column of table did not begin at 0.0 and end
***          at 1.0, but they have been adjusted. Prior to adjustment,
***          table[0][1] = 0.000000e+00 and table[ndata-1][1]= 9.994079e-01.

Beta  (3, 2) random deviates
       1      0.9208
       2      0.4641
       3      0.7668
       4      0.6536
       5      0.8171
```

continuous_table_setup

Sets up table to generate pseudorandom numbers from a general continuous distribution.

Synopsis

#include `<imsls.h>`

void `imsls_f_continuous_table_setup` (*float* `cdf()`, *int* `iopt`, *int* `ndata`, *float* `*table`, ..., 0)

The type *double* function is `imsls_d_continuous_table_setup`.

Required Arguments

float `cdf(`*float* `x)` (Input)
User-supplied function to compute the cumulative distribution function. The argument to the function is the point at which the distribution function is to be evaluated

int `iopt` (Input)
Indicator of the extent to which `table` is initialized prior to calling `imsls_f_continuous_table_setup`.

iopt	**Action**
0	`imsls_f_continuous_table_setup` fills the last four columns of `table`. The user inputs the points at which the CDF is to be evaluated in the first column of `table`. These must be in ascending order.
1	`imsls_f_continuous_table_setup` fills the last three columns of `table`. The user supplied function `cdf` is not used and may be a dummy function; instead, the cumulative distribution function is specified in the first two columns of `table`. The abscissas (in the first column) must be in ascending order and the function must be strictly monotonically increasing.

int `ndata` (Input)
Number of points at which the CDF is evaluated for interpolation. `ndata` must be greater than or equal to 4.

float `*table` (Input/Ouput)
`ndata` by 5 table to be used for interpolation of the cumulative distribution function.
The first column of `table` contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing), and the remaining columns contain values used in interpolation. The first row of `table` corresponds to the left limit of the support of the distribution and the
last row corresponds to the right limit of the support; that is,
`table`[0][1] = 0.0 and `table`[`ndata`-1][1] = 1.0.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_continuous_table_setup` (*float* `cdf()`, *int* `iopt`,
    int `ndata`, *float* `table[]`,
    `IMSLS_TABLE_COL_DIM`,
    `IMSLS_FCN_W_DATA`, *float* `cdf()`, *void* `*data`,
    0)

Optional Arguments

`IMSLS_TABLE_COL_DIM`, *int* `table_col_dim` (Intput)
Column dimension of the array `table`.
Default: `table_col_dim` = 5

`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

Routine `imsls_f_continuous_table_setup` sets up a table that routine `imsls_f_random_general_continuous` can use to generate pseudorandom deviates from a continuous distribution. The distribution is specified by its cumulative distribution function, which can be supplied either in tabular form in `table` or by a function `cdf`. See the documentation for the routine `imsls_f_random_general_continuous` for a description of the method.

Example 1

In this example, `imsls_f_continuous_table_setup` is used to set up a table to generate pseudorandom variates from a beta distribution. This example is continued in the documentation for routine `imsls_f_random_general_continuous` to generate the random variates.

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

float cdf(float);
void main()
{
  int i, iopt=0, ndata= 100;
  float table[100][5], x = 0.0;

  for (i=0;i<ndata;i++) {
    table[i][0] = x;
    x += .01;
  }

  imsls_f_continuous_table_setup(cdf, iopt, ndata, table);
  printf("The first few values from the table:\n");
  for (i=0;i<10;i++) printf("%4.2f\t%8.4f\n", table[i][0], table[i][1]);

}

float cdf(float x)
{
  return imsls_f_beta_cdf(x, 3., 2.);
}
```

Output

```
*** WARNING  Error  from imsls_f_continuous_table_setup.  The values of the
***          CDF in the second column of table did not begin at 0.0 and end
***          at 1.0, but they have been adjusted. Prior to adjustment,
***          table[0][1] = 0.000000e+00 and table[ndata-1][1]= 9.994079e-01.

The first few values from the table:
0.00     0.0000
0.01     0.0000
0.02     0.0000
0.03     0.0001
0.04     0.0002
0.05     0.0005
0.06     0.0008
0.07     0.0013
0.08     0.0019
0.09     0.0027
```

random_normal_multivariate

Generates pseudorandom numbers from a multivariate normal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_normal_multivariate` (*int* `n_vectors`, *int* `length`,
float `*covariances`, …, 0)

The type *double* function is `imsls_d_random_normal_multivariate`.

Required Arguments

int `n_vectors` (Input)
Number of random multivariate normal vectors to generate.

int `length` (Input)
Length of the multivariate normal vectors.

float `*covariances` (Input)
Array of size `length` × `length` containing the variance-covariance matrix.

Return Value

An array of length `n_vectors` × `length` containing the random multivariate normal vectors stored consecutively.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_normal_multivariate` (*int* `n_vectors`, *int* `length`,
  float `*covariances`,
  `IMSLS_RETURN_USER`, *float* `r[]`,
  0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
  User-supplied array of length `n_vectors` × `length` containing the random multivariate normal vectors stored consecutively.

Description

Function `imsls_f_random_normal_multivariate` generates pseudorandom numbers from a multivariate normal distribution with mean vector consisting of all zeros and variance-covariance matrix `imsls_f_covariances`. First, the Cholesky factor of the variance-covariance matrix is computed. Then, independent random normal deviates with mean 0 and variance 1 are generated, and the matrix containing these deviates is postmultiplied by the Cholesky factor. Because the Cholesky factorization is performed in each invocation, it is best to generate as many random vectors as needed at once.

Deviates from a multivariate normal distribution with means other than 0 can be generated by using `imsls_f_random_normal_multivariate` and then by adding the vectors of means to each row of the result.

Example

In this example, `imsls_f_random_normal_multivariate` generates five pseudorandom normal vectors of length 2 with variance-covariance matrix equal to the following:

$$\begin{bmatrix} 0.500 & 0.375 \\ 0.375 & 0.500 \end{bmatrix}$$

```
#include <imsls.h>

void main()
{
    int n_vectors = 5;
    int length = 2;
    float covariances[] = {.5, .375, .375, .5};
    float *random;

    imsls_random_seed_set (123457);
    random = imsls_f_random_normal_multivariate (n_vectors, length,
        covariances, 0);

    imsls_f_write_matrix ("multivariate normal random deviates",
        n_vectors, length, random, 0);
}
```

Output

```
multivariate normal random deviates
                  1            2
     1        1.451        1.246
     2        0.766       -0.043
     3        0.058       -0.669
     4        0.903        0.463
     5       -0.867       -0.933
```

random_orthogonal_matrix

Generates a pseudorandom orthogonal matrix or a correlation matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_orthogonal_matrix` (*int* `n`, ..., 0)

The type *double* function is `imsls_d_random_orthogonal_matrix`.

Required Arguments

int `n` (Input)
The order of the matrix to be generated.

Return Value

`n` by `n` random orthogonal matrix. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_orthogonal_matrix` (*int* `n`,
`IMSLS_EIGENVALUES`, *float* `*eignevalues[]`,
`IMSLS_A_MATRIX`, *float* `*a`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_EIGENVALUES`, *float* `*eigenvalues` (Input)
A vector of length `n` containing the eigenvalues of the correlation matrix to be generated. The elements of `eigenvalues` must be positive, they must sum to `n`, and they cannot all be equal.

`IMSLS_A_MATRIX`, *float* `*a` (Input)
`n` by `n` random orthogonal matrix. A random correlation matrix is generated using the orthogonal matrix input in `a`. The option `IMSLS_EIGENVALUES` must also be supplied if `IMSLS_A_MATRIX` is used.

`IMSLS_A_COL_DIM`, *int* `a_col_dim` (Input)
Column dimension of the matrix `a`.
Default: `a_col_dim` = `n`

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n` × `n` containing the random correlation matrix.

Description

Routine `imsls_f_random_orthogonal_matrix` generates a pseudorandom orthogonal matrix from the invariant Haar measure. For each column, a random vector from a uniform distribution on a hypersphere is selected and then is projected onto the orthogonal complement of the columns already formed. The method is described by Heiberger (1978). (See also Tanner and Thisted 1982.)

If the optional argument `IMSLS_EIGENVALUES` is used, a correlation matrix is formed by applying a sequence of planar rotations to the matrix $A^T DA$, where D = diag(`eigenvalues`[0], …, `eigenvalues` [`n-1`]), so as to yield ones along the diagonal. The planar rotations are applied in such an order that in the two by two matrix that determines the rotation, one diagonal element is less than 1.0 and one is greater than 1.0. This method is discussed by Bendel and Mickey (1978) and by Lin and Bendel (1985).

The distribution of the correlation matrices produced by this method is not known. Bendel and Mickey (1978) and Johnson and Welch (1980) discuss the distribution.

For larger matrices, rounding can become severe; and the double precision results may differ significantly from single precision results.

Example

In this example, `imsls_f_random_orthogonal_matrix` is used to generate a 4 by 4 pseudorandom correlation matrix with eigenvalues in the ratio 1:2:3:4.

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

void main()
{
  int   i, n = 4;
  float *a, *cor;
  float ev[] = {1., 2., 3., 4.};

  for (i=0;i<4;i++) ev[i] = 4.*ev[i]/10.;

  imsls_random_seed_set(123457);

  a = imsls_f_random_orthogonal_matrix(n, 0);
  imsls_f_write_matrix("Random orthogonal matrix",
                  4, 4, (float*)a, 0);
  cor = imsls_f_random_orthogonal_matrix(n,
                              IMSLS_EIGENVALUES, ev,
                              IMSLS_A_MATRIX, a,
                              0);
  imsls_f_write_matrix("Random correlation matrix",
                    4, 4, (float*)cor, 0);

}
```

Output

```
            Random orthogonal matrix
            1           2           3           4
1     -0.8804     -0.2417      0.4065     -0.0351
2      0.3088     -0.3002      0.5520      0.7141
3     -0.3500      0.5256     -0.3874      0.6717
4     -0.0841     -0.7584     -0.6165      0.1941

            Random correlation matrix
            1           2           3           4
1       1.000      -0.236      -0.326      -0.110
2      -0.236       1.000       0.191      -0.017
3      -0.326       0.191       1.000      -0.435
4      -0.110      -0.017      -0.435       1.000
```

random_mvar_from_data

Generates pseudorandom numbers from a multivariate distribution determined from a given sample.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_mvar_from_data` (*int* `n_random`, *int* `ndim`, *int* `nsamp`, *float* `x[]`, *int* `nn`, ..., 0)

The type *double* function is `imsls_d_random_mvar_from_data`.

Required Arguments

int `n_random` (Input)
Number of random multivariate vectors to generate.

int `ndim` (Input)
The length of the multivariate vectors, that is, the number of dimensions.

int `nsamp` (Input)
Number of given data points from the distribution to be simulated.

float `x[]` (Input)
Array of size `nsamp` × `ndim` matrix containing the given sample.

int `nn` (Input)

Number of nearest neighbors of the randomly selected point in `x` that are used to form the output point in the result.

Return Value

`n_random` × `ndim` matrix containing the random multivariate vectors in its rows. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float * imsls_f_random_mvar_from_data (int n_random, int ndim,
         int nsamp, float x[], int nn,
         IMSLS_X_COL_DIM, int x_col_dim,
         IMSLS_RETURN_USER, float r[],
          0)
```

Optional Arguments

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

Column dimension of the matrix `x`.
Default: `x_col_dim` = `ndim`

`IMSLS_RETURN_USER`, *float* `r[]` (Output)

User-supplied array of length `n_random` × `ndim` containing the random correlation matrix.

Description

Given a sample of size n (= `nsamp`) of observations of a k-variate random variable, `imsls_f_random_mvar_from_data` generates a pseudorandom sample with approximately the same moments as the given sample. The sample obtained is essentially the same as if sampling from a Gaussian kernel estimate of the sample density. (See Thompson 1989.) Routine `imsls_f_random_mvar_from_data` uses methods described by Taylor and Thompson (1986).

Assume that the (vector-valued) observations x_i are in the rows of x. An observation, x_j, is chosen randomly; its nearest m (= `nn`) neighbors,

$$x_{j_1}, x_{j_2}, \ldots, x_{j_m}$$

are determined; and the mean

$$\bar{x}_j$$

of those nearest neighbors is calculated. Next, a random sample

$u_1, u_2, \ldots, u_m$ is generated from a uniform distribution with lower bound

$$\frac{1}{m} - \sqrt{\frac{3(m-1)}{m^2}}$$

and upper bound

$$\frac{1}{m}+\sqrt{\frac{3(m-1)}{m^2}}$$

The random variate delivered is

$$\sum_{l=1}^{m} u_l\left(x_{jl}-\bar{x}_j\right)+\bar{x}_j$$

The process is then repeated until `n_random` such simulated variates are generated and stored in the rows of the result.

Example

In this example, `imsls_f_random_mvar_from_data` is used to generate 5 pseudorandom vectors of length 4 using the initial and final systolic pressure and the initial and final diastolic pressure from Data Set A in Afifi and Azen (1979) as the fixed sample from the population to be modeled. (Values of these four variables are in the seventh, tenth, twenty-first, and twenty-fourth columns of data set number nine in routine `imsls_f_data_sets`, Chapter 15, "Utilities".)

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

void main()
{
  int i, nrrow, nrcol, nr = 5, k=4, nsamp = 113, nn = 5;
  float x[113][4], rdata[113][34], *r;

  imsls_random_seed_set(123457);


  imsls_f_data_sets(9,
                IMSLS_N_OBSERVATIONS, &nrrow,
                IMSLS_N_VARIABLES, &nrcol,
                IMSLS_RETURN_USER, rdata,
                0);
  for (i=0;i<nrrow;i++) x[i][0] = rdata[i][6];
  for (i=0;i<nrrow;i++) x[i][1] = rdata[i][9];
  for (i=0;i<nrrow;i++) x[i][2] = rdata[i][20];
  for (i=0;i<nrrow;i++) x[i][3] = rdata[i][23];

  r = imsls_f_random_mvar_from_data(nr, k, nsamp, x, nn, 0);
  imsls_f_write_matrix("Random variates", 5, 4, r, 0);
 }
```

Output

```
                 Random variates
            1           2           3           4
1       162.8        90.5       153.7       104.9
```

```
2       153.4        78.3       176.7        85.2
3        93.7        48.2       153.5        71.4
4       101.8        54.2       113.1        56.3
5        91.7        58.8        48.4        28.1
```

random_multinomial

Generates pseudorandom numbers from a multinomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_random_multinomial` (*int* `n_random`, *int* `n`, *int* `k`,
float `p[]`, …, 0)

Required Arguments

int `n_random` (Input)
Number of random multinomial vectors to generate.

int `n` (Input)
Multinomial parameter indicating the number of independent trials.

int `k` (Input)
The number of mutually exclusive outcomes on any trial. `k` is the length of the multinomial vectors. `k` must be greater than or equal to 2.

float `p[]` (Input)
Vector of length `k` containing the probabilities of the possible outcomes. The elements of `p` must be positive and must sum to 1.0.

Return Value

`n_random` by `k` matrix containing the random multinomial vectors in its rows. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_multinomial` (*int* `n_random`, *int* `n`, *int* `k`,
float `p[]`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` × `k` containing the random deviates.

Description

Routine `imsls_random_multinomial` generates pseudorandom numbers from a `K`-variate multinomial distribution with parameters `n` and `p`. `k` and `n` must be positive.

Each element of p must be positive and the elements must sum to 1. The probability function (with n = n, k = k, and p_i = p[i+1]) is

$$f\left(x_1, x_2, \ldots, x_k\right) = \frac{n!}{x_1!x_2!\ldots x_k!} p_1^{x_1} p_2^{x_2} \ldots p_k^{x_k}$$

for $x_i \geq 0$ and

$$\sum_{i=0}^{k-1} x_i = n$$

The deviate in each row of r is produced by generation of the binomial deviate x_0 with parameters n and p_i and then by successive generations of the conditional binomial deviates x_j given $x_0, x_1, \ldots, x_{j-2}$ with parameters $n - x_0 - x_1 - \ldots - x_{j-2}$ and $p_j / (1 - p_0 - p_1 - \ldots - p_{j-2})$.

Example

In this example, imsls_random_multinomial is used to generate five pseudorandom 3-dimensional multinomial variates with parameters n = 20 and p = [0.1, 0.3, 0.6].

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

void main()
{
  int nr = 5, n = 20, k = 3, *ir;
  float p[3] = {.1, .3, .6};

  imsls_random_seed_set(123457);

  ir = imsls_random_multinomial(nr, n, k, p, 0);

  imsls_i_write_matrix("Multinomial random_deviates", 5, 3, ir,
                       IMSLS_NO_ROW_LABELS,
                       IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Multinomial random_deviates
         5    4   11
         3    6   11
         3    3   14
         5    5   10
         4    5   11
```

random_sphere

Generates pseudorandom points on a unit circle or `K`-dimensional sphere

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_sphere` (*int* `n_random`, *int* `k`,..., 0)

The type *double* function is `imsls_d_random_sphere`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `k` (Input)
Dimension of the circle (`k` = 2) or of the sphere.

Return Value

`n_random` by `k` matrix containing the random Cartesian coordinates on the unit circle or sphere. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_sphere` (*int* `n_random`, *int* `k`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of size `n_random` by `k` containing the random Cartesian coordinates on the unit circle or sphere.

Description

Routine `imsls_f_random_sphere` generates pseudorandom coordinates of points that lie on a unit circle or a unit sphere in `K`-dimensional space. For points on a circle (`k` = 2), pairs of uniform (– 1, 1) points are generated and accepted only if they fall within the unit circle (the sum of their squares is less than 1), in which case they are scaled so as to lie on the circle.

For spheres in three or four dimensions, the algorithms of Marsaglia (1972) are used. For three dimensions, two independent uniform (– 1, 1) deviates U_1 and U_2 are generated and accepted only if the sum of their squares S_1 is less than 1. Then, the coordinates

$$Z_1 = 2U_1\sqrt{1-S_1},\ Z_2 = 2U_2\sqrt{1-S_1},\ \text{and}\ Z_3 = 1-2S_1$$

are formed. For four dimensions, U_1, U_2, and S_1 are produced as described above. Similarly, U_3, U_4, and S_2 are formed. The coordinates are then

$$Z_1 = U_1, Z_2 = U_2, Z_3 = U_3\sqrt{(1-S_1)/S_2}$$

and

$$Z_4 = U_4\sqrt{(1-S_1)/S_2}$$

For spheres in higher dimensions, `K` independent normal deviates are generated and scaled so as to lie on the unit sphere in the manner suggested by Muller (1959).

Example

In this example, `imsls_f_random_sphere` is used to generate two uniform random deviates from the surface of the unit sphere in three space.

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

void main()
{
  int n_random = 2;
  int k = 3;
  float *z;
  char *rlabel[] = {"First point",
                  "Second point"};

  imsls_random_seed_set(123457);

  z = imsls_f_random_sphere(n_random, k, 0);

  imsls_f_write_matrix("Coordinates", n_random, k, z,
                     IMSLS_ROW_LABELS, rlabel,
                     IMSLS_NO_COL_LABELS,
                     0);
 }
```

Output

```
                   Coordinates
First point       0.8893      0.2316       0.3944
Second point      0.1901      0.0396      -0.9810
```

random_table_twoway

Generates a pseudorandom two-way table.

Synopsis

#include `<imsls.h>`

int `*imsls_random_table_twoway` (*int* `nrow`, *int* `ncol`, *int* `nrtot[]`,
int `nctot[]`,..., 0)

Required Arguments

int `nrow` (Input)
Number of rows in the table.

int `ncol` (Input)
Number of columns in the table.

int `nrtot[]` (Input)
Array of length `nrow` containing the row totals.

int `nctot[]` (Input)
Array of length `ncol` containing the column totals. (Input)
The elements of `nrtot` and `nctot` must be nonnegative and must sum to the same quantity.

Return Value

`nrow` by `ncol` random matrix with the given row and column totals. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_table_twoway` (*int* `nrow`, *int* `ncol`, *int* `nrtot[]`,
int `nctot[]`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of size `nrow` by `ncol` containing the random matrix with the given row and column totals.

Description

Routine `imsls_random_table_twoway` generates pseudorandom entries for a two-way contingency table with fixed row and column totals. The method depends on the size of the table and the total number of entries in the table. If the total number of entries is less than twice the product of the number of rows and columns, the method described by Boyette (1979) and by Agresti, Wackerly, and Boyette (1979) is used. In this method, a work vector is filled with row indices so that the number of times each index appears equals the given row total. This vector is then randomly permuted and used to increment the entries in each row so that the given row total is attained.

For tables with larger numbers of entries, the method of Patefield (1981) is used. This method can be considerably faster in these cases. The method depends on the conditional probability distribution of individual elements, given the entries in the previous rows. The probabilities for the individual elements are computed starting from their conditional means.

Example

In this example, `imsls_random_table_twoway` is used to generate a two by three table with row totals 3 and 5, and column totals 2, 4, and 2.

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

void main()
{
  int *itable, nrow = 2, ncol = 3;
  int nrtot[2] = {3, 5};
  int nctot[3] = {2, 4, 2};
  char *title = "A random contingency table with fixed marginal totals";

  imsls_random_seed_set(123457);


  itable = imsls_random_table_twoway(nrow, ncol, nrtot, nctot, 0);

  imsls_i_write_matrix(title, nrow, ncol, itable,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    0);
  }
```

Output

```
A random contingency table with fixed marginal totals
                     0   2   1
                     2   2   1
```

random_order_normal

Generates pseudorandom order statistics from a standard normal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_order_normal` (*int* `ifirst`, *int* `ilast`, *int* `n`,..., 0)

The type *double* function is `imsls_d_random_order_normal`.

Required Arguments

int `ifirst` (Input)
First order statistic to generate.

int `ilast` (Input)
Last order statistic to generate.
`ilast` must be greater than or equal to `ifirst`. The full set of order statistics from `ifirst` to `ilast` is generated. If only one order statistic is desired, set `ilast` = `ifirst`.

int `n` (Input)
Size of the sample from which the order statistics arise.

Return Value

An array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.
The first element is the `ifirst` order statistic in a random sample of size `n` from the standard normal distribution. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_order_normal` (*int* `ifirst`, *int* `ilast`, *int* `n`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.

Description

Routine `imsls_f_random_order_normal` generates the `ifirst` through the `ilast` order statistics from a pseudorandom sample of size `N` from a normal (0, 1) distribution. Routine `imsls_f_random_order_normal` uses the routine `imsls_f_random_order_uniform` to generate order statistics from the uniform (0, 1) distribution and then obtains the normal order statistics using the inverse CDF transformation.

Each call to `imsls_f_random_order_normal` yields an independent event so order statistics from different calls may not have the same order relations with each other.

Example

In this example, `imsls_f_random_order_normal` is used to generate the fifteenth through the nineteenth order statistics from a sample of size twenty.

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

void main()
```

```
{
  float *r = NULL;

  imsls_random_seed_set(123457);

  r = imsls_f_random_order_normal(15, 19, 20, 0);

  printf("The 15th through the 19th order statistics from a \n");
  printf("random sample of size 20 from a normal distribution\n");
  imsls_f_write_matrix("", 5, 1, r, 0);
}
```

Output

```
The 15th through the 19th order statistics from a
random sample of size 20 from a normal distribution

1      0.4056
2      0.4681
3      0.4697
4      0.9067
5      0.9362
```

random_order_uniform

Generates pseudorandom order statistics from a uniform (0, 1) distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_order_uniform` (*int* `ifirst`, *int* `ilast`, *int* `n`,..., 0)

The type *double* function is `imsls_d_random_order_uniform`.

Required Arguments

int `ifirst` (Input)
First order statistic to generate.

int `ilast` (Input)
Last order statistic to generate.
`ilast` must be greater than or equal to `ifirst`. The full set of order statistics from `ifirst` to `ilast` is generated. If only one order statistic is desired, set `ilast` = `ifirst`.

int `n` (Input)
Size of the sample from which the order statistics arise.

Return Value

An array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.
The first element is the `ifirst` order statistic in a random sample of size `n` from the uniform (0, 1) distribution. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_order_uniform` (*int* `ifirst`, *int* `ilast`, *int* `n`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.

Description

Routine `imsls_f_random_order_uniform` generates the `ifirst` through the `ilast` order statistics from a pseudorandom sample of size `n` from a uniform (0, 1) distribution. Depending on the values of `ifirst` and `ilast`, different methods of generation are used to achieve greater efficiency. If `ifirst` = 1 and `ilast` = `n`, that is, if the full set of order statistics are desired, the spacings between successive order statistics are generated as ratios of exponential variates. If the full set is not desired, a beta variate is generated for one of the order statistics, and the others are generated as extreme order statistics from conditional uniform distributions. Extreme order statistics from a uniform distribution can be obtained by raising a uniform deviate to an appropriate power.

Each call to `imsls_f_random_order_uniform` yields an independent event. This means, for example, that if on one call the fourth order statistic is requested and on a second call the third order statistic is requested, the “fourth” may be smaller than the “third”. If both the third and fourth order statistics from a given sample are desired, they should be obtained from a single call to `imsls_f_random_order_uniform` (by specifying `ifirst` less than or equal to 3 and `ilast` greater than or equal to 4).

Example

In this example, `imsls_f_random_order_uniform` is used to generate the fifteenth through the nineteenth order statistics from a sample of size twenty.

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

void main()
{
  float *r = NULL;

  imsls_random_seed_set(123457);
  r = imsls_f_random_order_uniform(15, 19, 20, 0);

  printf("The 15th through the 19th order statistics from a \n");
  printf("random sample of size 20 from a uniform distribution\n");
  imsls_f_write_matrix("", 5, 1, r, 0);
}
```

Output

```
The 15th through the 19th order statistics from a
random sample of size 20 from a uniform distribution

1      0.6575
2      0.6802
3      0.6807
4      0.8177
5      0.8254
```

random_arma

Generates a time series from a specific ARMA model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_arma` (*int* `n_observations`, *int* `p`, *float* `ar[]`, *int* `q`, *float* `ma[]`, ..., 0)

The type double function is `imsls_d_random_arma`.

Required Arguments

int `n_observations` (Input)
Number of observations to be generated. Parameter `n_observations` must be greater than or equal to one.

int `p` (Input)
Number of autoregressive parameters. Paramater `p` must be greater than or equal to zero.

float `ar[]` (Input)
Array of length `p` containing the autoregressive parameters.

int `q` (Input)
Number of moving average parameters. Parameter `q` must be greater than or equal to zero.

float `ma[]` (Input)
Array of length `q` containing the moving average parameters.

Return Value

An array of length `n_observations` containing the generated time series.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_arma` (*int* `n_observations`, *int* `p`, *float* `ar[]`,
int q, *float* `ma[]`,
`IMSLS_ARMA_CONSTANT`, *float* `constant`,
`IMSLS_VAR_NOISE`, *float* `*a_variance`,
`IMSLS_INPUT_NOISE`, *float* `*a_input`,
`IMSLS_OUTPUT_NOISE`, *float* `**a_return`,
`IMSLS_OUTPUT_NOISE_USER`, *float* `a_return[]`,
`IMSLS_NONZERO_ARLAGS`, *int* `*ar_lags`,
`IMSLS_NONZERO_MALAGS`, *int* `*ma_lags`,
`IMSLS_INITIAL_W`, *float* `*w_initial`,
`IMSLS_ACCEPT_REJECT_METHOD`,
`IMSLS_RETURN_USER`, *float* `w[]`,
0)

Optional Arguments

`IMSLS_ARMA_CONSTANT`, *float* `constant` (Input)
Overall constant. See "Description".
Default: `constant` = 0

`IMSLS_VAR_NOISE`, *float* `a_variance` (Input)
If `IMSLS_VAR_NOISE` is specified (and `IMSLS_INPUT_NOISE` is *not* specified) the noise a_t will be generated from a normal distribution with mean 0 and variance `a_variance`.
Default: `a_variance` = 1.0

`IMSLS_INPUT_NOISE`, *float* `*a_input` (Input)
If `IMSLS_INPUT_NOISE` is specified, the user will provide an array of length `n_observations` + max (`ma_lags`[*i*]) containing the random noises. If this option is specified, then `IMSLS_VAR_NOISE` should not be specified (a warning message will be issued and the option `IMSLS_VAR_NOISE` will be ignored).

`IMSLS_OUTPUT_NOISE`, *float* `**a_return` (Output)
An address of a pointer to an internally allocated array of length `n_observations` + max (`ma_lags`[*i*]) containing the random noises.

`IMSLS_OUTPUT_NOISE_USER`, *float* `a_return[]` (Output)
Storage for array `a_return` is provided by user. See `IMSLS_OUTPUT_NOISE`.

`IMSLS_NONZERO_ARLAGS`, *int* `ar_lags[]` (Input)
An array of length `p` containing the order of the nonzero autoregressive parameters.
Default: `ar_lags` = [1, 2, ..., `p`]

`IMSLS_NONZERO_MALAGS`, *int* `ma_lags` (Input)
An array of length `q` containing the order of the nonzero moving average parameters.
Default: `ma_lags` = [1, 2, ..., `q`]

`IMSLS_INITIAL_W`, *float* `w_initial[]` (Input)
: Array of length max (`ar_lags`[*i*]) containing the initial values of the time series.
: Default: all the elements in `w_initial` = `constant`/(1 – `ar` [0] – `ar` [1] – … – `ar` [`p` – 1])

`IMSLS_ACCEPT_REJECT_METHOD` (Input)
: If `IMSLS_ACCEPT_REJECT_METHOD` is specified, the random noises will be generated from a normal distribution using an acceptance/rejection method. If `IMSLS_ACCEPT_REJECT_METHOD` is not specified, the random noises will be generated using an inverse normal CDF method. This argument will be ignored if `IMSLS_INPUT_NOISE` is specified.

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
: User-supplied array of length `n_random` containing the generated time series.

Description

Function `imsls_f_random_arma` simulates an ARMA(p, q) process, $\{W_t\}$, for $t = 1, 2, ..., n$ (with n = `n_observations`, p = `p`, and q = `q`). The model is

$$\phi(B)W_t = \theta_0 + \theta(B)A_t \qquad t \in Z$$

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^P$$
$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$$

Let μ be the mean of the time series $\{W_t\}$. The overall constant θ_0 (`constant`) is

$$\theta_0 = \begin{cases} \mu & p = 0 \\ \mu\left(1 - \sum_{i=1}^{p} \phi_i\right) & p > 0 \end{cases}$$

Time series whose innovations have a nonnormal distribution may be simulated by providing the appropriate innovations in `a_input` and start values in `w_initial`.

The time series is generated according to the followng model:

X[i] = `constant` + `ar[0]` · X[i – `ar_lags[0]`] + … +
`ar[p – 1]` · X[i – `ar_lags[p – 1]`] +
A[i] – `ma[0]` · A[i – `ma_lags[0]`] – … –
`ma[q – 1]` · A[i – `ma_lags[q – 1]`]

where the constant is related to the mean of the series,

$$\overline{W}$$

as follows:

$$\text{constant} = \overline{W} \cdot (1 - \text{ar}[0] - \ldots - \text{ar}[q-1])$$

and where

$$X[t] = W[t], \quad t = 0, 1, \ldots, \texttt{n\_observations} - 1$$

and

$$W[t] = \texttt{w\_initial}[t + p], \qquad t = -p, -p + 1, \ldots, -2, -1$$

and A is either `a_input` (if `IMSLS_INPUT_NOISE` is specified) or `a_return` (otherwise).

Examples

Example 1

In this example, `imsls_f_random_arma` is used to generate a time series of length five, using an ARMA model with three autoregressive parameters and two moving average parameters. The start values are 0.1000, 0.0500, and 0.0375.

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

void main()
{
    int   n_random = 5;
    int   np = 3;
    float phi[3] = {0.5, 0.25, 0.125};
    int   nq = 2;
    float theta[2] = {-0.5, -0.25};
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_arma(n_random, np, phi, nq, theta, 0);
    imsls_f_write_matrix("ARMA random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
              ARMA random deviates:
     0.863       0.809       1.904       0.110       2.266
```

Example 2

In this example, a time series of length 5 is generated using an ARMA model with 4 autoregressive parameters and 2 moving average parameters. The start values are 0.1, 0.05 and 0.0375.

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

void main()
{
    int   n_random = 5;
    int   np = 3;
    float phi[3] = {0.5, 0.25, 0.125};
    int   nq = 2;
    float theta[2] = {-0.5, -0.25};
    float wi[3] = {0.1, 0.05, 0.0375};
    float theta0 = 1.0;
    float avar   = 0.1;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_arma(n_random, np, phi, nq, theta,
        IMSLS_ACCEPT_REJECT_METHOD,
        IMSLS_INITIAL_W, wi,
        IMSLS_ARMA_CONSTANT, theta0,
        IMSLS_VAR_NOISE, avar,
        0);
    imsls_f_write_matrix("ARMA random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                ARMA random deviates:
     1.403       2.220       2.286       2.888       2.832
```

Warning Errors

`IMSLS_RNARM_NEG_VAR`	VAR(a) = “a_variance” = #, VAR(a) must be greater than 0. The absolute value of # is used for VAR(a).
`IMSLS_RNARM_IO_NOISE`	Both `IMSLS_INPUT_NOISE` and `IMSLS_OUTPUT_NOISE` are specified. `IMSLS_INPUT_NOISE` is used.

random_npp

Generates pseudorandom numbers from a nonhomogeneous Poisson process.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_npp` (*float* `tbegin`, *float* `tend`, *float* `ftheta()`, *float* `theta_min`, *float* `theta_max`, *int* `neub`, *int* `*ne`, ..., 0)

The type *double* function is `imsls_d_random_npp`.

Required Arguments

float `tbegin` (Input)
Lower endpoint of the time interval of the process.
`tbegin` must be nonnegative. Usually, `tbegin` = 0.

float `tend` (Input)
Upper endpoint of the time interval of the process.
`tend` must be greater than `tbegin`.

float `ftheta` (*float* `t`) (Input)
User-supplied function to provide the value of the rate of the process as a function of time. This function must be defined over the interval from `tbegin` to `tend` and must be nonnegative in that interval.

float `theta_min` (Input)
Minimum value of the rate function `ftheta()` in the interval (`tbegin`, `tend`).
If the actual minimum is unknown, set `theta_min` = 0.0.

float `theta_max` (Input)
Maximum value of the rate function `ftheta()` in the interval (`tbegin`, `tend`).
If the actual maximum is unknown, set `theta_max` to a known upper bound of the maximum. The efficiency of `imsls_f_random_npp` is less the greater `theta_max` exceeds the true maximum.

int `neub` (Input)
Upper bound on the number of events to be generated.
In order to be reasonably sure that the full process through time `tend` is generated, calculate `neub` as `neub` = `X` + 10.0 `*` `SQRT`(`X`), where
`X` = `theta_max` `*` (`tend` – `tbegin`).

int `*ne` (Output)
Number of events actually generated.
If `ne` is less that `neub`, the time `tend` is reached before `neub` events are realized.

Return Value

An array of length `neub` containing the the times to events in the first `ne` elements. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_npp` (*float* `tbegin`, *float* `tend`, *float* `ftheta()`, *float* `theta_min`, *float* `theta_max`, *int* `neub`, *int* `*ne`, `IMSLS_RETURN_USER`, *float* `r[]`,
`IMSLS_FCN_W_DATA`, *float* `ftheta()`, *void* `*data`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `neub` containing the the times to events in the first `ne` elements.

`IMSLS_FCN_W_DATA`, *float* `ftheta` (*float* `t`), *void* `*data`, (Input)
User-supplied function to provide the value of the rate of the process as a function of time, 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

Routine `imsls_f_random_npp` simulates a one-dimensional nonhomogeneous Poisson process with rate function `ftheta` in a fixed interval (`tbegin`, `tend`].

Let $\lambda(t)$ be the rate function and t_0 = `tbegin` and t_1 = `tend`. Routine `imsls_f_random_npp` uses a method of thinning a nonhomogeneous Poisson process $\{N*(t), t \geq t_0\}$ with rate function $\lambda*(t) \geq \lambda(t)$ in $(t_0, t_1]$, where the number of events, $N*$, in the interval $(t_0, t_1]$ has a Poisson distribution with parameter

$$\mu_0 = \int_{t_0}^{t_1} \lambda(t)\, dt$$

The function

$$\Lambda(t) = \int_0^{t'} \lambda(t)\, dt$$

is called the *integrated rate function*.) In `imsls_f_random_npp`, $\lambda*(t)$ is taken to be a constant $\lambda*$(= `theta_max`) so that at time t_i, the time of the next event t_{i+1} is obtained by generating and cumulating exponential random numbers

$$E^*_{1,i}, E^*_{2,i}, \ldots,$$

with parameter $\lambda*$, until for the first time

$$u_{j,i} \leq \left(t_i + E^*_{1,i} + \ldots + E^*_{j,i}\right) / \lambda^*$$

where the $u_{j,i}$ are independent uniform random numbers between 0 and 1. This process is continued until the specified number of events, `neub`, is realized or until the time, `tend`, is exceeded. This method is due to Lewis and Shedler (1979), who also review other methods. The most straightforward (and most efficient) method is by inverting the integrated rate function, but often this is not possible.

If `theta_max` is actually greater than the maximum of $\lambda(t)$ in $(t_0, t_1]$, the routine will work, but less efficiently. Also, if $\lambda(t)$ varies greatly within the interval, the efficiency is reduced. In that case, it may be desirable to divide the time interval into subintervals within which the rate function is less variable. This is possible because the process is without memory.

If no time horizon arises naturally, `tend` must be set large enough to allow for the required number of events to be realized. Care must be taken, however, that `ftheta` is defined over the entire interval.

After simulating a given number of events, the next event came be generated by setting `tbegin` to the time of the last event (the sum of the elements in `R`) and calling `imsls_f_random_npp` again. Cox and Lewis (1966) discuss modeling applications of nonhomogeneous Poisson processes.

Example

In this example, `imsls_f_random_npp` is used to generate the first five events in the time 0 to 20 (if that many events are realized) in a nonhomogeneous process with rate function

$$\lambda(t) = 0.6342\ \mathrm{e}0.001427^{t}$$

for $0 < t \le 20$.

Since this is a monotonically increasing function of t, the minimum is at $t = 0$ and is 0.6342, and the maximum is at $t = 20$ and is 0.6342 e0.02854 = 0.652561.

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

void main()
{
  int i, neub = 5, ne;
  float  *r, tmax= .652561, tmin = .6342, tbeg=0., tend=20.;

  imsls_random_seed_set(123457);

  r = imsls_f_random_npp(tbeg, tend, ftheta, tmin, tmax, neub, &ne, 0);

  printf("Inter-event times for the first %d events in the process:\n", ne);
  for (i=0; i<ne; i++) printf("\t%f\n", r[i]);

}
```

Output

```
Inter-event times for the first 5 events in the process:
       0.052660
       0.407979
       0.258399
       0.019767
       0.167641
```

random_permutation

Generates a pseudorandom permutation.

Synopsis

#include `<imsls.h>`

int `*imsls_random_permutation` (*int* `k`, ..., 0)

Required Arguments

int `k` (Input)
Number of integers to be permuted.

Return Value

An array of length `k` containing the random permutation of the integers from 1 to `k`. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_permutation` (*int* `k`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of length `k` containing the random permutation of the integers from 1 to `k`.

Description

Routine `imsls_random_permutation` generates a pseudorandom permutation of the integers from 1 to `k`. It begins by filling a vector of length `k` with the consecutive integers 1 to `k`. Then, with M initially equal to `k`, a random index J between 1 and M (inclusive) is generated. The element of the vector with the index M and the element with index J swap places in the vector. M is then decremented by 1 and the process repeated until $M = 1$.

Example

In this example, `imsls_random_permutation` is called to produce a pseudorandom permutation of the integers from 1 to 10.

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

void main()
{
  int *ir, k = 10;

  imsls_random_seed_set(123457);

  ir = imsls_random_permutation(k, 0);

  printf("Random permutation of the integers from 1 to 10\n");
  imsls_i_write_matrix("", 1, k, ir,
                       IMSLS_NO_COL_LABELS, 0);
 }
```

Output

```
Random permutation of the integers from 1 to 10

  5    9    2    8    1    6    4    7    3   10
```

random_sample_indices

Generates a simple pseudorandom sample of indices.

Synopsis

#include `<imsls.h>`

int `*imsls_random_sample_indices` (*int* `nsamp`, *int* `npop`, ..., 0)

Required Arguments

int `nsamp` (Input)
Sample size desired.

int `npop` (Input)
Number of items in the population.

Return Value

An array of length `nsamp` containing the indices of the sample. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_sample_indices` (*int* `nsamp`, *int* `npop`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of length `nsamp` containing the indices of the sample.

Description

Routine `imsls_random_sample_indices` generates the indices of a pseudorandom sample,without replacement, of size `nsamp` numbers from a population of size `npop`. If `nsamp` is greater than `npop`/2, the integers from 1 to `npop` are selected sequentially with a probability conditional on the number selected and the number remaining to be considered. If, when the i-th population index is considered, j items have been included in the sample, then the index i is included with probability (`nsamp` $- j$)/(`npop` $+ 1 - i$).

If `nsamp` is not greater than `npop`/2, a O(`nsamp`) algorithm due to Ahrens and Dieter (1985) is used. Of the methods discussed by Ahrens and Dieter, the one called SG* is used in `imsls_random_sample_indices`. It involves a preliminary selection of q indices using a geometric distribution for the distances between each index and the next one. If the preliminary sample size *q* is less than `nsamp`, a new preliminary sample is chosen, and this is continued until a preliminary sample greater in size than `nsamp` is chosen. This preliminary sample is then thinned using the same kind of sampling as described above for the case in which the sample size is greater than half of the population size. Routine `imsls_random_sample_indices` does not store the preliminary sample indices, but rather restores the state of the generator used in selecting the sample initially, and then passes through once again, making the final selection as the preliminary sample indices are being generated.

Example

In this example, `imsls_random_sample_indices` is used to generate the indices of a pseudorandom sample of size 5 from a population of size 100.

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

void main()
{
  int *ir, nsamp = 5, npop = 100;

  imsls_random_seed_set(123457);

  ir = imsls_random_sample_indices(nsamp, npop, 0);

  imsls_i_write_matrix("Random Sample", 1, nsamp, ir,
                      IMSLS_NO_COL_LABELS, 0);
 }
```

Output

```
     Random Sample
              2    22    53    61    79
```

random_sample

Generates a simple pseudorandom sample from a finite population.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_sample` (*int* `nrow`, *int* `nvar`, *float* `population[]`, *int* `nsamp`,..., 0)

The type *double* function is `imsls_d_random_sample`.

Required Arguments

int `nrow` (Input)
Number of rows of data in `population`.

int `nvar` (Input)

Number of variables in the population and in the sample.

float `population[]` (Input)

`nrow` by `nvar` matrix containing the population to be sampled. If either of the optional arguments `IMSLS_FIRST_CALL` or `IMSLS_ADDITIONAL_CALL` are specified, then `population` contains a different part of the population on each invocation, otherwise `population` contains the entire population.

int `nsamp` (Input)

The sample size desired.

Return Value

`nsamp` by `nvar` matrix containing the sample. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_sample` (*int* `nrow`, *int* `nvar`, *float* `population[]`, *int* `nsamp`,
`IMSLS_FIRST_CALL`, *int* `**index`, *int* `*npop`
`IMSLS_FIRST_CALL_USER`, *int* `index[]`, *int* `*npop`
`IMSLS_ADDITIONAL_CALL`, *int* `*index`, *int* `*npop`, *float* `*samp`,
`IMSLS_POPULATION_COL_DIM`, *int* `population_col_dim`,
`IMSLS_RETURN_USER`, *int* `samp[]`,
0)

Optional Arguments

`IMSLS_FIRST_CALL`, *int* `**index`, *int* `*npop` (Output)

This is the first invocation with this data; additional calls to `imsls_f_random_sample` may be made to add to the population. Additional calls should be made using the optional argument `IMSLS_ADDITIONAL_CALL` . Argument `index` is the address of a pointer to an internally allocated array of length `nsamp` containing the indices of the sample in the population. Argument `npop` returns the number of items in the population. If the population is input a few items at a time, the first call to `imsls_f_random_sample` should use `IMSLS_FIRST_CALL`, and subsequent calls should use `IMSLS_ADDITIONAL_CALL`. See example 2.

`IMSLS_FIRST_CALL_USER`, *int* `index[]`, *int* `*npop` (Output)

Storage for `index` is provided by the user. See `IMSLS_FIRST_CALL`.

`IMSLS_ADDITIONAL_CALL`, *int* `*index`, *int* `*npop`, *float* `*samp` (Input/Output)

This is an additional invocation of `imsls_f_random_sample`, and updating for the subpopulation in `population` is performed. Argument `index` is a pointer to an array of length `nsamp` containing the indices of the sample in the population, as returned using optional argument `IMSLS_FIRST_CALL`. Argument `npop`, also obtained using optional argument `IMSLS_FIRST_CALL`, returns the number of items in the population. It is not necessary to know the number of items in the population in advance. `npop` is used to cumulate the population size and should not be changed between calls to `imsls_f_random_sample`. Argument `samp` is a pointer to the array of size `nsamp` by `nvar` containing the sample. `samp` is the result of calling `imsls_f_random_sample` with optional argument `IMSLS_FIRST_CALL`. See example 2

`IMSLS_POPULATION_COL_DIM`, *int* `population_col_dim` (Input)
Column dimension of the matrix `population`.
Default: `x_col_dim` = `nvar`

`IMSLS_RETURN_USER`, *int* `samp[]` (Output)
User-supplied array of size `nrow` by `nvar` containing the sample. This option should not be used if `IMSLS_ADDITIONAL_CALL` is used.

Description

Routine `imsls_f_random_sample` generates a pseudorandom sample from a given population, without replacement, using an algorithm due to McLeod and Bellhouse (1983).

The first `nsamp` items in the population are included in the sample. Then, for each successive item from the population, a random item in the sample is replaced by that item from the population with probability equal to the sample size divided by the number of population items that have been encountered at that time.

Example 1

In this example, `imsls_f_random_sample` is used to generate a sample of size 5 from a population stored in the matrix `population`.

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

void main()
{
  int nrow = 176, nvar = 2, nsamp = 5;
  float *population;
  float *sample;

  population = imsls_f_data_sets(2, 0);

  imsls_random_seed_set(123457);

  sample = imsls_f_random_sample(nrow, nvar, population, nsamp, 0);

  imsls_f_write_matrix("The sample", nsamp, nvar, sample,
                       IMSLS_NO_ROW_LABELS,
                       IMSLS_NO_COL_LABELS,
                       0);
}
```

Output

```
      The sample
      1764          36
      1828          62
      1923           6
      1773          35
      1769         106
```

Example 2

Routine `imsls_f_random_sample` is now used to generate a sample of size 5 from the same population as in the example above except the data are input to `RNSRS` one observation at a time. This is the way `imsls_f_random_sample` may be used to sample from a file on disk or tape. Notice that the number of records need not be known in advance.

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

void main()
{
  int i, nrow = 176, nvar = 2, nsamp = 5;
  int *index, npop;
  float *population;
  float *sample;

  population = imsls_f_data_sets(2, 0);

  imsls_random_seed_set(123457);

  sample = imsls_f_random_sample(1, 2, population, nsamp,
                            IMSLS_FIRST_CALL, &index, &npop,
                            0);
  for (i = 1; i < 176; i++) {
    imsls_f_random_sample(1, 2, &population[2*i], nsamp,
                      IMSLS_ADDITIONAL_CALL, index, &npop, sample,
                      0);
  }
  printf("The population size is %d\n", npop);
  imsls_i_write_matrix("Indices of random sample", 5, 1, index, 0);


  imsls_f_write_matrix("The sample", nsamp, nvar, sample,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    0);
 }
```

Output

```
The population size is 176

Indices of random sample
         1    16
         2    80
         3   175
         4    25
         5    21
```

```
The sample
1764          36
1828          62
1923           6
1773          35
1769         106
```

random_option

Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator or a generalized feedback shift register (`GFSR`) method.

Synopsis

#include `<imsls.h>`

void `imsls_random_option` (*int* `generator_option`)

Required Arguments

int `generator_option` (Input)
Indicator of the generator. Argument `generator_option` is used to choose the multiplier and whether or not shuffling is done, or the `GFSR` method.

`generator_option`	Generator
1	The multiplier 16807 is used.
2	The multiplier 16807 is used with shuffling.
3	The multiplier 397204094 is used.
4	The multiplier 397204094 is used with shuffling.
5	The multiplier 950706376 is used.
6	The multiplier 950706376 is used with shuffling.
7	GFSR, with the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$ is used.
8	A 32-bit Mersenne Twister generator is used. The float and double random numbers are generated from 32-bit integers.
9	A 64-bit Mersenne Twister generator is used. The float and double random numbers are generated from 64-bit integers. This ensures that all bits of both float and doubles are random.

Description

The uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling. The value of the multiplier and whether or not to use shuffling are determined by `imsls_random_option`. The description of function `imsls_f_random_uniform` may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (see Lewis et al. 1969).

Both of the Mersenne Twister generators have a period of 2^{19937}-1 and a 623-dimensional equidistribution property. See Matsumoto et al. 1998 for details.

The IMSL Mersenne Twister generators are derived from code copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved. It is subject to the following notice:

The IMSL 32-bit Mersenne Twister generator is based on the Matsumoto and Nishimura code ‘mt19937ar’ and the 64-bit code is based on ‘mt19937-64’.

Example

See function `imsls_random_GFSR_table_get`.

random_option_get

Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.

Synopsis

#include `<imsls.h>`

int `imsls_random_option_get` ()

Return Value

Indicator of the generator.

Result	Generator
1	The multiplier 16807 is used.
2	The multiplier 16807 is used with shuffling.
3	The multiplier 397204094 is used.
4	The multiplier 397204094 is used with shuffling.
5	The multiplier 950706376 is used.
6	The multiplier 950706376 is used with shuffling.
7	GFSR, with the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$ is used

Description

The routine `imsls_random_option_get` retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator or the `GRSR` method. The uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling. The value of the multiplier and whether or not to use shuffling are determined by `imsls_random_option`.

random_seed_get

Retrieves the current value of the seed used in the random number generators.

Synopsis

#include `<imsls.h>`

int `imsls_random_seed_get` ( )

Return Value

The value of the seed.

Description

Function `imsls_random_seed_get` retrieves the current value of the “seed” used in the random number generators. A reason for doing this would be to restart a simulation, using function `imsls_random_seed_set` to reset the seed.

Example

This example illustrates the statements required to restart a simulation using `imsls_random_seed_get` and `imsls_random_seed_set`. The example shows that restarting the sequence of random numbers at the value of the seed last generated is the same as generating the random numbers all at once.

```
#include <imsls.h>

#define      N_RANDOM      5

main()
{
    int          seed = 123457;
    float       *r1, *r2, *r;
    imsls_random_seed_set(seed);
    r1 = imsls_f_random_uniform(N_RANDOM, 0);
    imsls_f_write_matrix ("First Group of Random Numbers", 1,
                           N_RANDOM, r1, 0);
    seed = imsls_random_seed_get();

    imsls_random_seed_set(seed);
    r2 = imsls_f_random_uniform(N_RANDOM, 0);
    imsls_f_write_matrix ("Second Group of Random Numbers", 1,
                           N_RANDOM, r2, 0);

    imsls_random_seed_set(123457);
    r = imsls_f_random_uniform(2*N_RANDOM, 0);
    imsls_f_write_matrix ("Both Groups of Random Numbers", 1,
                           2*N_RANDOM, r, 0);
}
```

Output

```
              First Group of Random Numbers
         1           2           3           4           5
    0.9662      0.2607      0.7663      0.5693      0.8448

             Second Group of Random Numbers
         1           2           3           4           5
    0.0443      0.9872      0.6014      0.8964      0.3809

                     Both Groups of Random Numbers
         1           2           3           4           5           6
    0.9662      0.2607      0.7663      0.5693      0.8448      0.0443

         7           8           9          10
    0.9872      0.6014      0.8964      0.3809
```

random_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.

Synopsis

#include `<imsls.h>`

int `imsls_random_substream_seed_get` (*int* `iseed1`)

Required Arguments

int `iseed1` (Input)
The seed that yields the first stream.

Return Value

The seed that yields a stream beginning 100,000 numbers beyond the stream that begins with `iseed1`.

Description

Given a seed, `iseed1`, `imsls_random_substream_seed_get` determines another seed, such that if one of the IMSL multiplicative congruential generators, using no shuffling, went through 100,000 generations starting with `iseed1`, the next number in that sequence would be the first number in the sequence that begins with the returned seed.

Note that `imsls_random_substream_seed_get` works only when a multiplicative congruential generator without shuffling is used. This means that either the routine `imsls_random_option` has not been called at all or that it has been last called with `generator_option` taking a value of 1, 3, or 5.

For many of the IMSL generators for nonuniform distributions that do not use the inverse CDF method, the distance between the sequences generated starting with `iseed1` and starting with the returned seed may be less than 100,000. This is because the nonuniform generators that use other techniques may require more than one uniform deviate for each output deviate.

The reason that one may want two seeds that generate sequences a known distance apart is for blocking Monte Carlo experiments or for running parallel streams

Example

In this example, `imsls_random_substream_seed_get` is used to determine seeds for 4 separate streams, each 200,000 numbers apart, for a multiplicative congruential generator without shuffling. (Since `imsls_random_option` is not invoked to select a generator, the multiplier is 16807.) Since the streams are 200,000 numbers apart, each seed requires two invocations of `imsls_random_substream_seed_get`. All of the streams are non-overlapping, since the period of the underlying generator is 2,147,483,646. The resulting seed are then verified by checking the seed after generating random sequences of length 200,000.

```
#include <imsls.h>

main()
{
  int i, is1, is2, is3, is4;
  float *r;

  is1 = 123457;
  is2 = imsls_random_substream_seed_get(is1);
  is2 = imsls_random_substream_seed_get(is2);
  is3 = imsls_random_substream_seed_get(is2);
  is3 = imsls_random_substream_seed_get(is3);
  is4 = imsls_random_substream_seed_get(is3);
  is4 = imsls_random_substream_seed_get(is4);
  printf("Seeds for four separate streams:\n");
  printf("%d\t%d\t%d\t%d\n\n", is1, is2, is3, is4);

  imsls_random_seed_set(is1);
  for (i=0;i<3;i++) {
    r = imsls_f_random_uniform(200000, 0);
    printf("seed after %d random numbers: %d\n", (i+1)*200000,
           imsls_random_seed_get());
    if (r) free(r);
  }
}
```

Output

```
Seeds for four separate streams:
123457 2016130173   85016329     979156171

seed after 200000 random numbers: 2016130173
seed after 400000 random numbers: 85016329
seed after 600000 random numbers: 979156171
```

random_seed_set

Initializes a random seed for use in the random number generators.

Synopsis

#include `<imsls.h>`

void `imsls_random_seed_set` (*int* `seed`)

Required Arguments

int `seed` (Input)
The seed of the random number generator. The argument `seed` must be in the range (0, 2147483646). If `seed` is 0, a value is computed using the system clock; hence, the results of programs using the random number generators will be different at various times.

Description

Function `imsls_random_seed_set` is used to initialize the seed used in the random number generators. The form of the generators is as follows:

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

The value of x_0 is the seed. If the seed is not initialized prior to invocation of any of the functions for random number generation by calling `imsls_random_seed_set`, the seed is initialized by the system clock. The seed can be reinitialized to a clock-dependent value by calling `imsls_random_seed_set` with `seed` set to 0.

The effect of `imsls_random_seed_set` is to set some global values used by the random number generators. A common use of `imsls_random_seed_set` is in conjunction with function `imsls_random_seed_get` to restart a simulation.

Example

See function `imsls_random_seed_get`.

random_table_set

Sets the current table used in the shuffled generator.

Synopsis

#include `<imsls.h>`

void `imsls_f_random_table_set` (*float* `table[]`)

The type *double* function is `imsls_d_random_table_set`.

Required Arguments

float `table[]` (Input)
Array of length 128 used in the shuffled generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive in except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_table_set`, on the next invocation of a routine to generate random numbers using a shuffled method, the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get`.

random_table_get

Retrieves the current table used in the shuffled generator.

Synopsis

#include `<imsls.h>`

void `imsls_f_random_table_get` (*float* `**table`, ..., 0)

The type *double* function is `imsls_d_random_table_get`.

Required Arguments

float `**table` (Output)
Address of a pointer to an array of length 128 containing the table used in the shuffled generators. Typically, *float* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_random_table_get` (*float* `**table`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
: User-supplied array of length 1565 containing the table used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_table_set`, on the next invocation of a routine to generate random numbers using a shuffled method, the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get`.

random_GFSR_table_set

Sets the current table used in the GFSR generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_GFSR_table_set` (*int* `table[]`)

Required Arguments

int `table []` (Input)
: Array of length 1565 used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_GFSR_table_set`, on the next invocation of a routine to generate random numbers using a GFSR method, the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get`.

random_GFSR_table_get

Retrieves the current table used in the GFSR generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_GFSR_table_get` (*int* `**table`, ..., 0)

Required Arguments

int `**table` (Output)
Address of a pointer to an array of length 1565 containing the table used in the GFSR generators. Typically, *int* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_random_GFSR_table_get` (*int* `**table`,
`IMSLS_RETURN_USER`, *int* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `r[]` (Output)
User-supplied array of length 1565 containing the table used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_GFSR_table_set`, on the next invocation of a routine to generate random numbers using a GFSR method, the appropriate array will be reinitialized.

Example

In this example, three separate simulation streams are used, each with a different form of the generator. Each stream is stopped and restarted. (Although this example is obviously an artificial one, there may be reasons for maintaining separate streams and stopping and restarting them because of the nature of the usage of the random numbers coming from the separate streams.)

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

void main()
{
  float *r, *table;
  int  nr, iseed1, iseed2, iseed7;
  int *itable;

  nr = 5;
  iseed1 = 123457;
  iseed2 = 123457;
  iseed7 = 123457;

  /* Begin first stream, iopt = 1 (by default) */
  imsls_random_seed_set (iseed1);
  r = imsls_f_random_uniform (nr, 0);
  iseed1 = imsls_random_seed_get ();
  imsls_f_write_matrix ("First stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed1);
  free(r);

  /* Begin second stream, iopt = 2 */
  imsls_random_option (2);
  imsls_random_seed_set (iseed2);
  r = imsls_f_random_uniform (nr, 0);
  iseed2 = imsls_random_seed_get ();
  imsls_f_random_table_get (&table, 0);
  imsls_f_write_matrix ("Second stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed2);
  free(r);

  /* Begin third stream, iopt = 7 */
  imsls_random_option (7);
  imsls_random_seed_set (iseed7);
  r = imsls_f_random_uniform (nr, 0);
  iseed7 = imsls_random_seed_get ();
  imsls_random_GFSR_table_get (&itable, 0);
  imsls_f_write_matrix ("Third stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed7);
  free(r);

  /* Reinitialize seed and resume first stream */
  imsls_random_option (1);
  imsls_random_seed_set (iseed1);
  r = imsls_f_random_uniform (nr, 0);
  iseed1 = imsls_random_seed_get ();
  imsls_f_write_matrix ("First stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed1);
  free(r);

  /*
   * Reinitialize seed and table for shuffling and
   * resume second stream
   */
  imsls_random_option (2);
  imsls_random_seed_set (iseed2);
  imsls_f_random_table_set (table);
  r = imsls_f_random_uniform (nr, 0);
  iseed2 = imsls_random_seed_get ();
  imsls_f_write_matrix ("Second stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed2);
  free(r);

  /*
   * Reinitialize seed and table for GFSR and
   * resume third stream.
   */
  imsls_random_option (7);
  imsls_random_seed_set (iseed7);
  imsls_random_GFSR_table_set (itable);
  r = imsls_f_random_uniform (nr, 0);
  iseed7 = imsls_random_seed_get ();
  imsls_f_write_matrix ("Third stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed7);
  free(r);

}
```

Output

```
                  First stream output
   0.9662      0.2607      0.7663      0.5693      0.8448
 Output seed 1814256879


                 Second stream output
   0.7095      0.1861      0.4794      0.6038      0.3790
 Output seed 1965912801


                  Third stream output
   0.3914      0.0263      0.7622      0.0281      0.8997
 Output seed 1932158269


                  First stream output
   0.0443      0.9872      0.6014      0.8964      0.3809
 Output seed 817878095


                 Second stream output
   0.2557      0.4788      0.2258      0.3455      0.5811
 Output seed 2108806573
```

```
                Third stream output
    0.7519      0.5084      0.9070      0.0910      0.6917
 Output seed 1485334679
```

random_MT32_init

Initializes the 32-bit Mersenne Twister generator using an array.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT32_table_init` (*int* `key_length`, *unsigned int* `key[]`)

Required Arguments

int `key_length` (Input)
Length of the array key.

unsigned int `key []` (Input)
Array of length `key_length` used to initialize the 32-bit Mersenne Twister generator.

Description

By default, the Mersenne Twister random number generator is initialized using the current seed value (see `imsls_random_seed_get`). The seed is limited to one integer for initialization. This function allows an arbitrary length array to be used for initialization.

This function completely replaces the use of the seed for initialization of the 32-bit Mersenne Twister generator.

Example

See function `imsls_random_MT32_table_get`.

random_MT32_table_get

Retrieves the current table used in the 32-bit Mersenne Twister generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT32_table_get` (*unsigned int* `**table`, ..., 0)

Required Arguments

unsigned int `**table` (Output)
Address of a pointer to an array of length 625 containing the table used in the 32-bit Mersenne Twister generator. Typically, *unsigned int* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include <imsls.h>

void imsls_random_MT32_table_get (*int* **table,
        IMSLS_RETURN_USER, *int* r[],
        0)

Optional Arguments

IMSLS_RETURN_USER, *int* r[] (Output)
User-supplied array of length 625 containing the table used in the 32-bit Mersenne Twister generator.

Description

The values in table contain the state of the 32-bit Mersenne Twister random number generator. The table can be used by imsls_random_MT32_table_set to set the generator back to this state.

Example

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore, the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
#include <imsls.h>

void main()
{
       const unsigned int init[] = {0x123, 0x234, 0x345, 0x456};
       float   *r;
       int     iseed = 123457;
       int     *itable;
       int     nr = 5;

       /* Initialize Mersenne Twister series with a seed */
       imsls_random_option (8);
       imsls_random_seed_set (iseed);
       r = imsls_f_random_uniform (nr, 0);
       imsls_f_write_matrix ("First stream output", 1, 5, r,
              IMSLS_NO_COL_LABELS,
              IMSLS_NO_ROW_LABELS,
              0);
       free(r);

       /* Reinitialize Mersenne Twister series with an array */
       imsls_random_option (8);
       imsls_random_MT32_init(4, init);
       /* Save the state of the series */
        imsls_random_MT32_table_get(&itable, 0);
       r = imsls_f_random_uniform (nr, 0);
       imsls_f_write_matrix ("Second stream output", 1, 5, r,
              IMSLS_NO_COL_LABELS,
              IMSLS_NO_ROW_LABELS,
              0);
       free(r);

       /* Restore the state of the series */
       imsls_random_MT32_table_set(itable);

       r = imsls_f_random_uniform (nr, 0);
       imsls_f_write_matrix ("Third stream output", 1, 5, r,
              IMSLS_NO_COL_LABELS,
              IMSLS_NO_ROW_LABELS,
              0);
       free(r);

       /* Reset the series - it will reinitialize from the seed */
       itable[0] = 1000;
       imsls_random_MT32_table_set(itable);

       r = imsls_f_random_uniform (nr, 0);
       imsls_f_write_matrix ("Fourth stream output", 1, 5, r,
              IMSLS_NO_COL_LABELS,
              IMSLS_NO_ROW_LABELS,
              0);
       free(r);
}
```

Output

```
                  First stream output
    0.4347       0.3522       0.0139       0.2091       0.4956

                 Second stream output
    0.2486       0.2226       0.1111       0.9563       0.9846

                  Third stream output
    0.2486       0.2226       0.1111       0.9563       0.9846

                 Fourth stream output
    0.4347       0.3522       0.0139       0.2091       0.4956
```

random_MT32_table_set

Sets the current table used in the 32-bit Mersenne Twister generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT32_table_set` (*unsigned int* `table[]`)

Required Arguments

unsigned int `table []` (Input)

Array of length 625 used in the 32-bit Mersenne Twister generator.

Description

The values in `table` are the state of the 32-bit Mersenne Twister random number generator obtained by a call to `imsls_random_MT32_table_set`. The values in the table can be used to restore the state of the generator.

Alternatively, if `table[0]` > 625 then the generator is set to its original, uninitialized, state.

Example

See function `imsls_random_MT32_table_get`.

random_MT64_init

Initializes the 64-bit Mersenne Twister generator using an array.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT64_table_init` (*int* `key_length`, *unsigned long long* `key[]`)

Required Arguments

int key_length (Input)

Length of the array key.

unsigned long long `key []` (Input)

Array of length `key_length` used to initialize the 64-bit Mersenne Twister generator.

Description

By default, the Mersenne Twister random number generator is initialized using the current seed value (see `imsls_random_seed_get`). The seed is limited to one integer for initialization. This function allows an arbitrary length array to be used for initialization.

This function completely replaces the use of the seed for initialization of the 64-bit Mersenne Twister generator.

Example

See function `imsls_random_MT64_table_get`.

random_MT64_table_get

Retrieves the current table used in the 64-bit Mersenne Twister generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT64_table_get` (*unsigned long long* `**table`, ..., 0)

Required Arguments

unsigned long long `**table` (Output)

Address of a pointer to an array of length 625 containing the table used in the 64-bit Mersenne Twister generator. Typically, *unsigned long long* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_random_MT64_table_get` (*unsigned long long* `**table`,
    `IMSLS_RETURN_USER`, *unsigned long long* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *unsigned long long* `r[]` (Output)
    User-supplied array of length 625 containing the table used in the 64-bit Mersenne Twister generator.

Description

The values in the `table` contain the state of the 64-bit Mersenne Twister random number generator. The table can be used by `imsls_random_MT64_table_set` to set the generator back to this state.

Example

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
#include <imsls.h>

void main()
{
       const unsigned long long init[] = {0x123, 0x234, 0x345, 0x456};
       float   *r;
       int     iseed = 123457;
       unsigned long long *itable;
       int     nr = 5;

       /* Initialize 64-bit Mersenne Twister series with a seed */
       imsls_random_option (9);
       imsls_random_seed_set (iseed);
       r = imsls_f_random_uniform (nr, 0);
       imsls_f_write_matrix ("First stream output", 1, 5, r,
              IMSLS_NO_COL_LABELS,
              IMSLS_NO_ROW_LABELS,
              0);
```

```
        free(r);

        /* Reinitialize Mersenne Twister series with an array */
        imsls_random_option (9);
        imsls_random_MT64_init(4, init);
        /* Save the state of the series */
        imsls_random_MT64_table_get(&itable, 0);

        r = imsls_f_random_uniform (nr, 0);
        imsls_f_write_matrix ("Second stream output", 1, 5, r,
               IMSLS_NO_COL_LABELS,
               IMSLS_NO_ROW_LABELS,
               0);
        free(r);

        /* Restore the state of the series */
        imsls_random_MT64_table_set(itable);

        r = imsls_f_random_uniform (nr, 0);
        imsls_f_write_matrix ("Third stream output", 1, 5, r,
               IMSLS_NO_COL_LABELS,
               IMSLS_NO_ROW_LABELS,
               0);
        free(r);

        /* Reset the series - it will reinitialize from the seed */
        itable[0] = 1000;
        imsls_random_MT64_table_set(itable);

        r = imsls_f_random_uniform (nr, 0);
        imsls_f_write_matrix ("Fourth stream output", 1, 5, r,
               IMSLS_NO_COL_LABELS,
               IMSLS_NO_ROW_LABELS,
               0);
        free(r);
}
```

Output

```
                 First stream output
    0.5799       0.9401       0.7102       0.1640       0.5457

                Second stream output
    0.4894       0.7397       0.5725       0.0863       0.7588

                 Third stream output
    0.4894       0.7397       0.5725       0.0863       0.7588

                Fourth stream output
    0.5799       0.9401       0.7102       0.1640       0.5457
```

random_MT64_table_set

Sets the current table used in the 64-bit Mersenne Twister generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_MT64_table_set` (*unsigned long long* `table[]`)

Required Arguments

unsigned long long `table []` (Input)

Array of length 625 used in the 64-bit Mersenne Twister generator.

Description

The values in `table` are the state of the 64-bit Mersenne Twister random number generator obtained by a call to `imsls_random_MT64_table_set`. The values in the table can be used to restore the state of the generator.

Alternatively, if `table`[0] > 625 then the generator is set to its original, uninitialized, state.

Example

See function `imsls_random_MT64_table_get`.

faure_next_point

Computes a shuffled Faure sequence.

Synopsis

#include `<imsls.h>`

*Imsls_faure** `imsls_faure_sequence_init` (*int* `ndim`, …, 0)

*float** `imsls_f_faure_next_point` (*Imsls_faure* `*state`, …, `0`)

void `imsls_faure_sequence_free` (*Imsls_faure* `*state`)

The type *double* function is `imsls_d_faure_next_point`. The functions `imsls_faure_sequence_init` and `imsls_faure_sequence_free` are precision independent.

Required Arguments for imsls_faure_sequence_init

int `ndim` (Input)
The dimension of the hyper-rectangle.

Return Value for imsls_faure_sequence_init

Returns a structure that contains information about the sequence. The structure should be freed using `imsls_faure_sequence_free` after it is no longer needed.

Required Arguments for imsls_faure_next_point

Imsls_faure `*state` (Input/Output)
Structure created by a call to `imsls_faure_sequence_init`.

Return Value for imsls_faure_next_point

Returns the next point in the shuffled Faure sequence. To release this space, use `imsls_faure_sequence_free`.

Required Arguments for imsls_faure_sequence_free

Imsls_faure *`state` (Input/Output)
Structure created by a call to `imsls_faure_sequence_init`.

Synopsis with Optional Arguments

#include `<imsls.h>`

Imsls_faure `*imsls_faure_sequence_init` (*int* `ndim`,
`IMSLS_BASE`, *int* `base`,
`IMSLS_SKIP`, *int* `skip`,
0)

*float** `imsls_f_faure_next_point` (*Imsls_faure* `*state`,
`IMSLS_RETURN_USER`, *float* `*user`,
`IMSLS_RETURN_SKIP`, *int* `*skip`,
0)

Optional Arguments

`IMSLS_BASE`, *int* `base` (Input)
The base of the Faure sequence.
Default: The smallest prime greater than or equal to `ndim`.

`IMSLS_SKIP`, *int* `*skip` (Input)
The number of points to be skipped at the beginning of the Faure sequence.
Default: $\lfloor \texttt{base}^{m/2-1} \rfloor$, where $m = \lfloor \log B / \log \texttt{base} \rfloor$ and B is the largest representable integer.

`IMSLS_RETURN_USER`, *float* *user (Output)
User-supplied array of length `ndim` containing the current point in the sequence.

`IMSLS_RETURN_SKIP`, *int* `*skip` (Output)
The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for `IMSLS_SKIP`, and using the same dimension for `ndim`.

Description

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set $x_1, \ldots, x_n \in [0,1]^d$, $d \geq 1$, is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = \left[0, t_1\right) \times \ldots \times \left[0, t_d\right),\ 0 \le t_j \le 1,\ 1 \le j \le d$$,

λ is the Lebesque measure, and $A(E;n)$ is the number of the x_j contained in E.

The sequence x_1, x_2, … of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d, such that

$$D_n^{(d)} \le c(d) \frac{(\log n)^d}{n}$$

for all n>1.

Generalized Faure sequences can be defined for any prime base $b \ge d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \ge d$, so the optional argument `IMSLS_BASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence x_1, x_2, …, is computed as follows:

Write the positive integer n in its b-ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers, $0 \le a_i(n) < b$.

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series, $c_{kd}^{(j)}$, is defined to be

$$c_{kd}^{(j)} = j^{\,d-k} c_{kd}$$

and c_{kd} is an element of the Pascal matrix,

$$c_{kd} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b*-ary Gray code. The function *G*(*n*) maps the positive integer *n* into the integer given by its *b*-ary expansion.

The sequence computed by this function is ***x***(*G*(*n*)), where ***x*** is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that `imsls_faure_sequence_init` is used to create a structure that holds the state of the sequence. Each call to `imsls_f_faure_next_point` returns the next point in the sequence and updates the *Imsls_faure* structure. The final call to `imsls_faure_sequence_free` frees data items, stored in the structure, that were allocated by `imsls_faure_sequence_init`.

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


void main()
{
       Imsl_faure    *state;
       float         *x;
       int            ndim = 3;
       int            k;

       state = imsl_faure_sequence_init(ndim, 0);

       for (k = 0;  k < 5;  k++) {
              x = imsl_f_faure_next_point(state, 0);
              printf("%10.3f %10.3f  %10.3f\n", x[0], x[1], x[2]);
             free(x);
       }

       imsl_faure_sequence_free(state);
}
```

Output

```
     0.334      0.493       0.064
     0.667      0.826       0.397
     0.778      0.270       0.175
     0.111      0.604       0.509
     0.445      0.937       0.842
```

Chapter 13: Neural Networks

Routines

Network

Multilayered feedforward neural network	mlff_network	934
Training	mlff_network_trainer	944
Forecasting	mlff_network_forecast	954

Preprocessing Filters

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.	scale_filter	960
Converts time series data to the format required for processing by a neural network.	time_series_filter	966
Converts time series data sorted within nominal classes.	time_series_class_filter	969
Converts nominal data into a series of binary encoded columns for input to a neural network.	unsupervised_nominal_filter	973
Converts ordinal data into proportions.	unsupervised_ordinal_filter	976

Usage Notes

Neural Networks – An Overview

Today, neural networks are used to solve a wide variety of problems, some of which have been solved by existing statistical methods, and some of which have not. These applications fall into one of the following three categories:

- *Forecasting: predicting one or more quantitative outcomes from both quantitative and categorical input data,*
- *Classification: classifying input data into one of two or more categories, or*
- *Statistical pattern recognition: uncovering patterns, typically spatial or temporal, among a set of variables.*

Forecasting, pattern recognition and classification problems are not new. They existed years before the discovery of neural network solutions in the 1980's. What is new is that neural networks provide a single framework for solving so many traditional problems and, in some cases, extend the range of problems that can be solved.

Traditionally, these problems were solved using a variety of widely known statistical methods:

- *linear regression and general least squares,*
- *logistic regression and discrimination,*
- *principal component analysis,*
- *discriminant analysis,*
- *k-nearest neighbor classification, and*
- *ARMA and NARMA time series forecasts.*

In many cases, simple neural network configurations yield the same solution as many traditional statistical applications. For example, a single-layer, feedforward neural network with linear activation for its output perceptron is equivalent to a general linear regression fit. Neural networks can provide more accurate and robust solutions for problems where traditional methods do not completely apply.

Mandic and Chambers (2001) identify the traditional methods for time series forecasting that are unsuitable when a time series:

- *is non-stationary,*
- *has large amounts of noise, such as a biomedical series, or*
- *is too short.*

ARIMA and other traditional time series approaches can produce poor forecasts when one or more of the above conditions exist. The forecasts of ARMA and non-linear ARMA (NARMA) depend heavily upon key assumptions about the model or underlying relationship between the output of the series and its patterns.

Neural networks, on the other hand, adapt to changes in a non-stationary series and can produce reliable forecasts even when the series contains a good deal of noise or when only a short series is available for training. Neural networks provide a single tool for solving many problems traditionally solved using a wide variety of statistical tools and for solving problems when traditional methods fail to provide an acceptable solution.

Although neural network solutions to forecasting, pattern recognition and classification problems can vary vastly, they are always the result of computations that proceed from the network inputs to the network outputs. The network inputs are referred to as *patterns*, and outputs are referred to as *classes*. Frequently the flow of these computations is in one direction, from the network input patterns to its outputs. Networks with forward-only flow are referred to as feedforward networks.

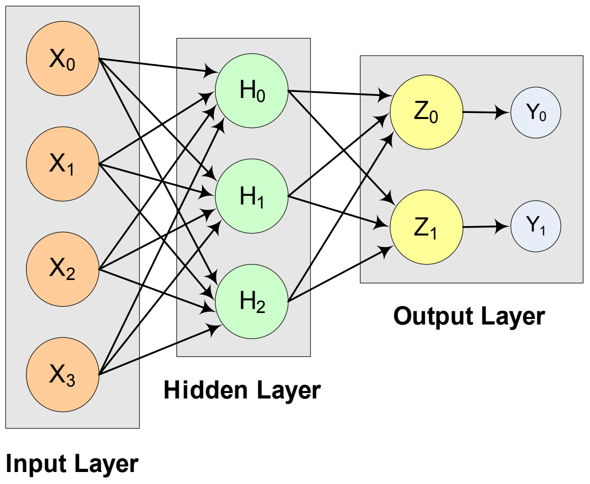


Figure 13-1: A 2-layer, Feedforward Network with 4 inputs and 2 outputs

Other networks, such as recurrent neural networks, allow data and information to flow in both directions, see Mandic and Chambers' (2001).

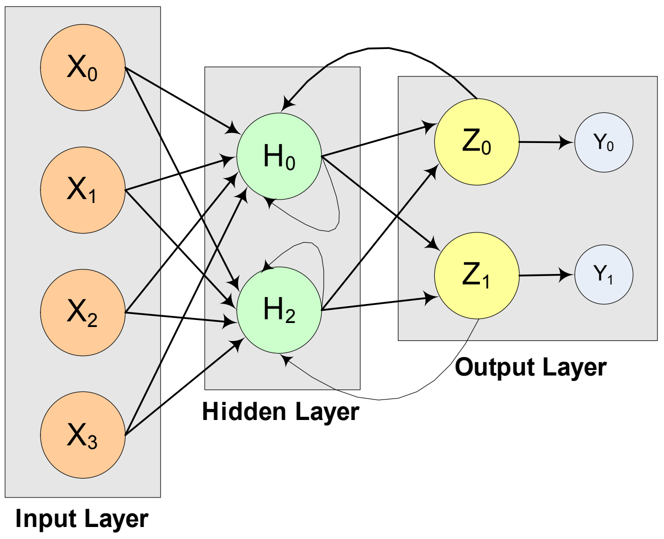


Figure 13-2: A recurrent neural network with 4 inputs and 2 outputs

A neural network is defined not only by its architecture and flow, or interconnections, but also by computations used to transmit information from one node or input to another node. These computations are determined by network weights. The process of fitting a network to existing data to determine these weights is referred to as *training* the network, and the data used in this process are referred to as *patterns*. Individual network inputs are referred to as *attributes* and outputs are referred to as *classes*. The table below lists terms used to describe neural networks that are synonymous to common statistical terminology.

Neural Network Terminology	Traditional Statistical Terminology	Description
Training	Model Fitting	Estimating unknown parameters or coefficients in the analysis
Patterns	Cases or Observations	A single observation of all input and output variables
Attributes	Independent Variables	Inputs to the network or model
Classes	Dependent Variables	Outputs from the network or model calculations

Table 1. Synonyms between Neural Network and Common Statistical Terminology

Neural Networks – History and Terminology

The Threshold Neuron

McCulloch and Pitts' (1943) wrote one of the first published works on neural networks. This paper describes the threshold neuron as a model for which the human brain stores and processes information.

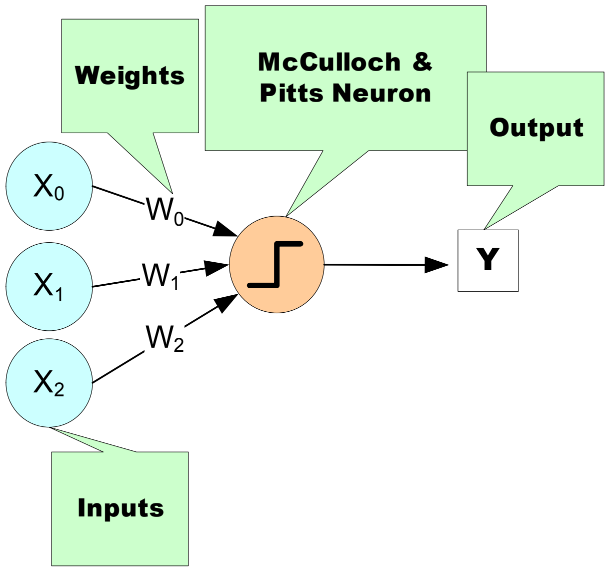


Figure 13-3: The McCulloch & Pitts Threshold Neuron

All inputs to a threshold neuron are combined into a single number, Z, using the following weighted sum:

$$Z = \sum_{i=1}^{m} w_i x_i - \mu ,$$

where w_i is the weight associated with the *i*th input (attribute) x_i. The term μ in this calculation is referred to as the *bias term*. In traditional statistical terminology it might be referred to as the *intercept*. The weights and bias terms in this calculation are estimated during network training.

In McCulloch and Pitts' (1943) description of the threshold neuron, the neuron does not respond to its inputs unless *Z* is greater than zero. If *Z* is greater than zero then the output from this neuron is set to 1. If *Z* is less than or equal to zero the output is zero:

$$Y = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \le 0 \end{cases},$$

where *Y* is the neuron's output.

Years following McCulloch and Pitts' (1943) article, interest in McCulloch and Pitts neural network was limited to theoretical discussions, such as Hebb (1949), which describe learning, memory and the brain's structure.

The Perceptron

The McCulloch and Pitts' neuron is also referred to as a threshold neuron since it abruptly changes its output from 0 to 1 when its potential, *Z*, crosses a threshold. Mathematically, this behavior can be viewed as a step function that maps the neuron's potential, *Z*, to the neuron's output, *Y*.

Rosenblatt (1958) extended the McCulloch and Pitts threshold neuron by replacing this step function with a continuous function that maps *Z* to *Y*. The Rosenblatt neuron is referred to as the perceptron, and the continuous function mapping *Z* to *Y* makes it easier to train a network of perceptrons than a network of threshold neurons.

Unlike the threshold neuron, the perceptron produces analog output rather than the threshold neuron's purely binary output. Carefully selecting the analog function, makes Rosenblatt's perceptron differentiable, whereas the threshold neuron is not. This simplifies the training algorithm.

Like the threshold neuron, Rosenblatt's perceptron starts by calculating a weighted sum of its inputs,

$$Z = \sum_{i=1}^{m} w_i x_i - \mu .$$

This is referred to as the perceptron's *potential*.

Rosenblatt's perceptron calculates its analog output from its potential. There are many choices for this calculation. The function used for this calculation is referred to as the activation function as shown in Figure 13-4 below.

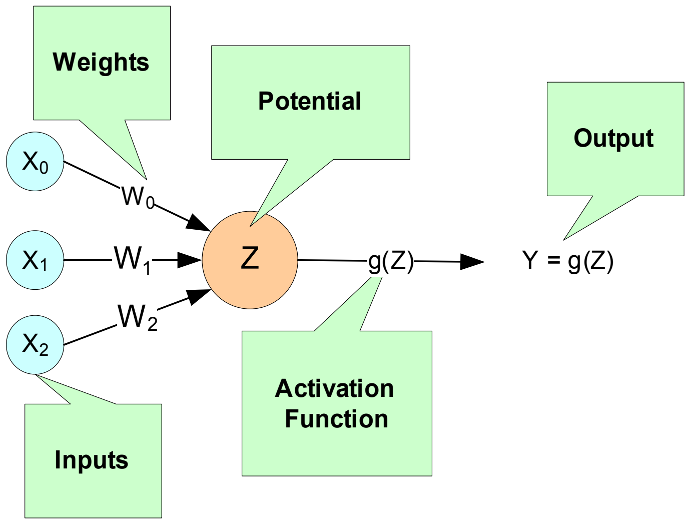


Figure 13-4: A Neural Net Perceptron

As shown in Figure 13-4, perceptrons consist of the following five components:

1. *Inputs – x_1, x_2, and x_3,*
2. *Input Weights – W_1, W_2, and W_3,*
3. *Potential –* $Z = \sum_{i=1}^{3} W_i x_i - \mu$ *, where μ is a bias correction,*
4. *Activation Function – g(Z), and*
5. *Output – Y = g(Z) .*

Like threshold neurons, perceptron inputs can be either the initial raw data inputs or the output from another perceptron. The primary purpose of network training is to estimate the weights associated with each perceptron's potential. The activation function maps this potential to the perceptron's output.

The Activation Function

Although theoretically any differentiable function can be used as an activation function, the identity and sigmoid functions are the two most commonly used.

The *identity activation* function, also referred to as a *linear activation* function, is a flow-through mapping of the perceptron's potential to its output:

$$g(Z) = Z.$$

Output perceptrons in a forecasting network often use the identity activation function.

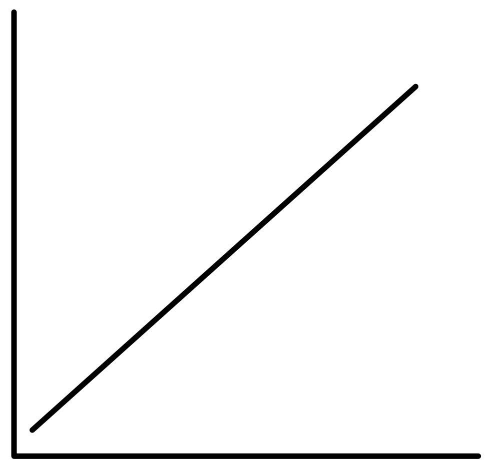


Figure 13-5: An Identity (Linear) Activation Function

If the identity activation function is used throughout the network, then it is easily shown that the network is equivalent to fitting a linear regression model of the form $Y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$, where $x_1, x_2, \cdots, x_k$ are the k network inputs, Y_i is the *i*th network output and $\beta_0, \beta_1, \cdots, \beta_k$ are the coefficients in the regression equation. As a result, it is uncommon to find a neural network with identity activation used in all its perceptrons.

Sigmoid activation functions are differentiable functions that map the perceptron's potential to a range of values, such as 0 to 1, i.e., $\Re^K \to \Re$ where K is the number of perceptron inputs.

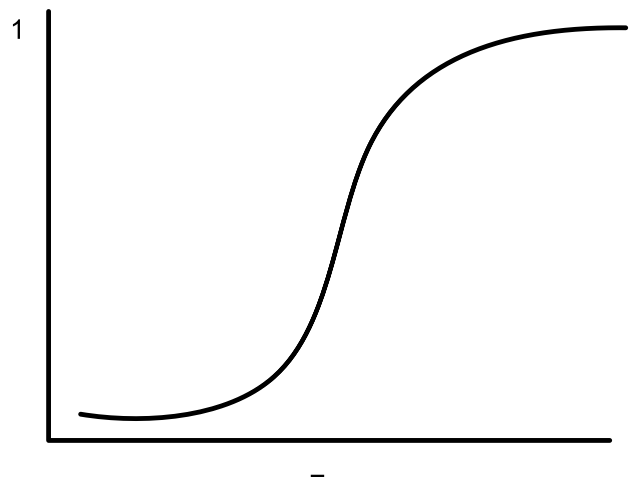


Figure 13-6: A Sigmoid Activation Function

In practice, the most common sigmoid activation function is the logistic function that maps the potential into the range 0 to 1:

$$g(Z) = \frac{1}{1+e^{-Z}} ,$$

Since $0 < g(Z) < 1$, the logistic function is very popular for use in networks that output probabilities.

Other popular sigmoid activation functions include:

- *the hyperbolic-tangent* $g(Z) = \tanh(Z) = \frac{e^{\alpha Z} - e^{-\alpha Z}}{e^{\alpha Z} + e^{-\alpha Z}}$,
- *the arc-tangent* $g(Z) = \frac{2}{\pi} \arctan\left(\frac{\pi Z}{2}\right)$*, and*
- *the squash activation function, see Elliott (1993),* $g(Z) = \frac{Z}{1+|Z|}$.

It is easy to show that the hyperbolic-tangent and logistic activation functions are linearly related. Consequently, forecasts produced using logistic activation should be close to those produced using hyperbolic-tangent activation. However, one function may be preferred over the other when training performance is a concern. Researchers report that the training time using the hyperbolic-tangent activation function is shorter than using the logistic activation function.

Network Applications

Forecasting using Neural Networks

There are numerous good statistical forecasting tools. Most require assumptions about the relationship between the variables being forecasted and the variables used to produce the forecast, as well as the distribution of forecast errors. Such statistical tools are referred to as *parametric methods*. ARIMA time series models, for example, assume that the time series is stationary, that the errors in the forecasts follow a particular ARIMA model, and that the probability distribution for the residual errors is Gaussian, see Box and Jenkins (1970). If these assumptions are invalid, then ARIMA time series forecasts can be substandard.

Neural networks, on the other hand, require few assumptions. Since neural networks can approximate highly non-linear functions, they can be applied without an extensive analysis of underlying assumptions.

Another advantage of neural networks over ARIMA modeling is the number of observations needed to produce a reliable forecast. ARIMA models generally require 50 or more equally spaced, sequential observations in time. In many cases, neural networks can also provide adequate forecasts with fewer observations by incorporating exogenous, or external, variables in the network's input.

For example, a company applying ARIMA time series analysis to forecast business expenses would normally require each of its departments, and each sub-group within each department, to prepare its own forecast. For large corporations this can require fitting hundreds or even thousands of ARIMA models. With a neural network approach, the department and sub-group information could be incorporated into the network as exogenous variables. Although this can significantly increase the network's training time, the result would be a single model for predicting expenses within all departments.

Linear least squares models are also popular statistical forecasting tools. These methods range from simple linear regression for predicting a single quantitative outcome to logistic regression for estimating probabilities associated with categorical outcomes. It is easy to show that simple linear least squares forecasts and logistic regression forecasts are equivalent to a feedforward network with a single layer. For this reason, single-layer feedforward networks are rarely used for forecasting. Instead multilayer networks are used.

Hutchinson (1994) and Masters (1995) describe using multilayer feedforward neural networks for forecasting. Multilayer feedforward networks are characterized by the forward-only flow of information in the network. The flow of information and computations in a feedforward network is always in one direction, mapping an M-dimensional vector of inputs to a C-dimensional vector of outputs, i.e., $\Re^M \rightarrow \Re^C$ where $C < M$.

There are many other types of networks without this feed forward requirement. Information and computations in a recurrent neural network, for example, flow in both directions. Output from one level of a recurrent neural network can be fed back, with some delay, as input into the same network (see Figure 13-2). Recurrent networks are very useful for time series prediction, see Mandic and Chambers (2001).

Pattern Recognition using Neural Networks

Neural networks are also extensively used in statistical pattern recognition. Pattern recognition applications that make wide use of neural networks include:

- *natural language processing: Manning and Schütze (1999)*
- *speech and text recognition: Lippmann (1989)*
- *face recognition: Lawrence, et al. (1997)*
- *playing backgammon, Tesauro (1990)*
- *classifying financial news, Calvo (2001).*

The interest in pattern recognition using neural networks has stimulated the development of important variations of feedforward networks. Two of the most popular are:

- *Self-Organizing Maps, also called Kohonen Networks, Kohonen (1995),*
- *and Radial Basis Function Networks, Bishop (1995).*

Useful mathematical descriptions of the neural network methods underlying these applications are given by Bishop (1995), Ripley (1996), Mandic and Chambers (2001), and Abe (2001). From a statistical viewpoint, Warner and Misra (1996) describes an excellent overview of neural networks.

Neural Networks for Classification

Classifying observations using prior concomitant information is possibly the most popular application of neural networks. Data classification problems abound in business and research. When decisions based upon data are needed, they can often be treated as a neural network data classification problem. Decisions to buy, sell, hold or remain with a stock are decisions involving four choices. Classifying loan applicants as good or bad credit risks, based upon their application, is a classification problem involving two choices. Neural networks are powerful tools for making decisions or choices based upon data.

These same tools are ideally suited for automatic selection or decision-making. Incoming email, for example, can be examined to separate spam from important email using a neural network trained for this task. A good overview of solving classification problems using multilayer feedforward neural networks is found in Abe (2001) and Bishop (1995).

There are two popular methods for solving data classification problems using multilayer feedforward neural networks, depending upon the number of choices (classes) in the classification problem. If the classification problem involves only two choices, then it can be solved using a neural network with a single logistic output. This output estimates the probability that the input data belong to one of the two choices.

For example, a multilayer feedforward network with a single logistic output can be used to determine whether a new customer is credit-worthy. The network's input would consist of information on the applicants credit application, such as age, income, etc. If the network output probability is above some threshold value (such as 0.5 or higher) then the applicant's credit application is approved.

This is referred to as binary classification using a multilayer feedforward neural network. If more than two classes are involved then a different approach is needed. A popular approach is to assign logistic output perceptrons to each class in the classification problem. The network assigns each input pattern to the class associated with the output perceptron that has the highest probability for that input pattern. However, this approach produces invalid probabilities since the sum of the individual class probabilities for each input is not equal to one, which is a requirement for any valid multivariate probability distribution.

To avoid this problem, the softmax activation function, see Bridle (1990), applied to the network outputs ensures that the outputs conform to the mathematical requirements of multivariate classification probabilities. If the classification problem has *C* categories, or classes, then each category is modeled by one of the network outputs. If Z_i is the weighted sum of products between its weights and inputs for the *i*th output, i.e.,

$$Z_i = \sum_j w_{ji} y_{ji}$$

then

$$softmax_i = \frac{e^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}} .$$

The softmax activation function ensures that all outputs conform to the requirements for multivariate probabilities. That is,

- *$0 < softmax_i < 1$, for all i = 1, 2, …, C and*
- $\sum_{i=1}^{C} softmax_i = 1$

A pattern is assigned to the *i*th classification when *softmax*$_i$ is the largest among all *C* classes.

However, multilayer feedforward neural networks are only one of several popular methods for solving classification problems. Others include:

- *Support Vector Machines (SVM Neural Networks), Abe (2001),*
- *Classification and Regression Trees (CART), Breiman, et al. (1984),*
- *Quinlan's classification algorithms C4.5 and C5.0, Quinlan (1993), and*
- *Quick, Unbiased and Efficient Statistical Trees (QUEST), Loh and Shih (1997).*

Support Vector Machines are simple modifications of traditional multilayer feedforward neural networks (MLFF) configured for pattern classification.

Multilayer Feedforward Neural Networks

A multilayer feedforward neural network is an interconnection of perceptrons in which data and calculations flow in a single direction, from the input data to the outputs. The number of layers in a neural network is the number of layers of perceptrons. The simplest neural network is one with a single input layer and an output layer of perceptrons. The network in Figure 13-7 illustrates this type of network. Technically, this is referred to as a one-layer feedforward network with two outputs because the output layer is the only layer with an activation calculation.

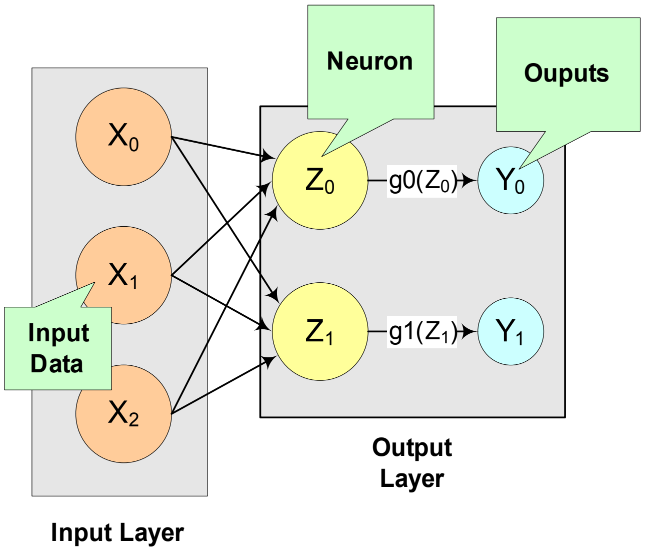


Figure 13- 7: A Single-Layer Feedforward Neural Net

In this single-layer feedforward neural network, the network's inputs are directly connected to the output layer perceptrons, Z_1 and Z_2.

The output perceptrons use activation functions, g_1 and g_2, to produce the outputs Y_1 and Y_2.

Since

$$Z_1 = \sum_{i=1}^{3} W_{1,i}x_i - \mu_1 \text{ and } Z_2 = \sum_{i=1}^{3} W_{2,i}x_i - \mu_2 ,$$

$$Y_1 = g_1(Z_1) = g_1(\sum_{i=1}^{3} W_{1,i}x_i - \mu_1) ,$$

and

$$Y_2 = g_2(Z_2) = g_2(\sum_{i=1}^{3} W_{2,i}x_i - \mu_2) .$$

When the activation functions g_1 and g_2 are identity activation functions, the single-layer neural net is equivalent to a linear regression model. Similarly, if g_1 and g_2 are logistic activation functions, then the single-layer neural net is equivalent to logistic regression. Because of this correspondence between single-layer neural networks and linear and logistic regression, single-layer neural networks are rarely used in place of linear and logistic regression.

The next most complicated neural network is one with two layers. This extra layer is referred to as a hidden layer. In general there is no restriction on the number of hidden layers. However, it has been shown mathematically that a two-layer neural network can accurately reproduce any differentiable function, provided the number of perceptrons in the hidden layer is unlimited.

However, increasing the number of perceptrons increases the number of weights that must be estimated in the network, which in turn increases the execution time for the network. Instead of increasing the number of perceptrons in the hidden layers to improve accuracy, it is sometimes better to add additional hidden layers, which typically reduce both the total number of network weights and the computational time. However, in practice, it is uncommon to see neural networks with more than two or three hidden layers.

Neural Network Error Calculations

Error Calculations for Forecasting

The error calculations used to train a neural network are very important. Researchers have investigated many error calculations in an effort to find a calculation with a short training time appropriate for the network's application. Typically error calculations are very different depending primarily on the network's application.

For forecasting, the most popular error function is the sum-of-squared errors, or one of its scaled versions. This is analogous to using the minimum least squares optimization criterion in linear regression. Like least squares, the sum-of-squared errors is calculated by looking at the squared difference between what the network predicts for each training pattern and the target value, or observed value, for that pattern. Formally, the equation is the same as one-half the traditional least squares error:

$$E = \tfrac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2 ,$$

where *N* is the total number of training cases, *C* is equal to the number of network outputs, t_{ij} is the observed output for the *i*th training case and the *j*th network output, and $\hat{t}_{ij}$ is the network's forecast for that case.

Common practice recommends fitting a different network for each forecast variable. That is, the recommended practice is to use *C*=1 when using a multilayer feedforward neural network for forecasting. For classification problems with more than two classes, it is common to associate one output with each classification category, i.e., *C*=number of classes.

Notice that in ordinary least squares, the sum-of-squared errors are not multiplied by one-half. Although this has no impact on the final solution, it significantly reduces the number of computations required during training.

Also note that as the number of training patterns increases, the sum-of-squared errors increases. As a result, it is often useful to use the root-mean-square (RMS) error instead of the unscaled sum-of-squared errors:

$$E^{RMS} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2}{\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \bar{t}\right)^2}$$

where $\bar{t}$ is the average output:

$$\bar{t} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C} t_{ij}}{N \cdot C} .$$

Unlike the unscaled sum-of-squared errors, E^{RMS} does not increase as N increases. The smaller values for E^{RMS}, indicate that the network predicts its training targets closer. The smallest value, $E^{RMS} = 0$, indicates that the network predicts every training target exactly. The largest value, $E^{RMS} = 1$, indicates that the network predicts the training targets only as well as setting each forecast equal to the mean of the training targets.

Notice that the root-mean-squared error is related to the sum-of-squared error by a simple scale factor:

$$E^{RMS} = \frac{2}{\bar{t}} \cdot E$$

Another popular error calculation for forecasting from a neural network is the Minkowski-R error. The sum-of-squared error, *E*, and the root-mean-squared error, E^{RMS}, are both theoretically motivated by assuming the noise in the target data is Gaussian. In many cases, this assumption is invalid. A generalization of the Gaussian distribution to other distributions gives the following error function, referred to as the Minkowski-R error:

$$E^{R} = \sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \hat{t}_{ij}\right|^{R} .$$

Notice that $E^R = 2E$ when *R*=2.

A good motivation for using E^R instead of *E* is to reduce the impact of outliers in the training data. The usual error measures, *E* and E^{RMS}, emphasize larger differences between the training data and network forecasts since they square those differences. If outliers are expected, then it is better to de-emphasize larger differences. This can be done by using the Minkowski-R error with *R*=1. When *R*=1, the Mindowski-R error simplifies to the sum of absolute differences:

$$L = E^1 = \sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \hat{t}_{ij}\right|.$$

L is also referred to as the Laplacian error. This name is derived from the fact that it can be theoretically justified by assuming the noise in the training data follows a Laplacian, rather than Gaussian, distribution.

Of course, similar to *E*, L generally increases when the number of training cases increases. Similar to E^{RMS}, a scaled version of the Laplacian error can be calculated using the following formula:

$$L^{RMS} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \hat{t}_{ij}\right|}{\sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \bar{t}\right|}.$$

Cross-Entropy Error for Binary Classification

As previously mentioned, multilayer feedforward neural networks can be used for both forecasting and classification applications. Training a forecasting network involves finding the network weights that minimize either the Gaussian or Laplacian distributions, *E* or L respectively, or equivalently their scaled versions, E^{RMS} or L^{RMS}. Although these error calculations can be adapted for use in classification by setting the target classification variable to zeros and ones, this is not recommended. Use of the sum-of-squared and Laplacian error calculations is based on the assumption that the target variable is continuous. In classification applications, the target variable is a discrete random variable with *C* possible values, where *C*=number of classes.

A multilayer feedforward neural network for classifying patterns into one of only two categories is referred to as a binary classification network. It has a single output: the estimated probability that the input pattern belongs to one of the two categories. The probability that it belongs to the other category is equal to one minus this probability, i.e., $P(C_2) = P(\text{not C}_1) = 1 - P(C_1)$.

Binary classification applications are very common. Any problem requiring yes/no classification is a binary classification application. For example, deciding to sell or buy a stock is a binary classification problem. Deciding to approve a loan application is also a binary classification problem. Deciding whether to approve a new drug or to provide one of two medical treatments are binary classification problems.

For binary classification problems, only a single output is used, *C*=1. This output represents the probability that the training case should be classified as "yes." A common choice for the activation function of the output of a binary classification network is the logistic activation function, which always results in an output in the range 0 to 1, regardless of the perceptron's potential.

One choice for training binary classification networks is to use sum-of-squared errors with the class value of *yes* patterns coded as a 1 and the *no* classes coded as a 0, *i.e.*:

$$t_i = \begin{cases} 1 & \text{if training pattern i = "yes"} \\ 0 & \text{if training pattern i = " no"}. \end{cases}$$

However, using either the sum-of-squared or Laplacian errors for training a network with these target values assumes that the noise in the training data are Gaussian. In binary classification, the zeros and ones are not Gaussian. They follow the Bernoulli distribution:

$$P(t_i = t) = p^t (1-p)^{1-t},$$

where p is equal to the probability that a randomly selected case belongs to the "yes" class.

Modeling the binary classes as Bernoulli observations leads to the use of the cross-entropy error function described by Hopfield (1987) and Bishop (1995):

$$E^C = -\sum_{i=1}^{N} \left\{ t_i \ln(\hat{t}_i) + (1-t_i)\ln(1-\hat{t}_i) \right\},$$

where N is the number of training patterns, t_i is the target value for the ith case (either 1 or 0), and $\hat{t}_i$ is the network output for the ith training pattern. This is equal to the neural network's estimate of the probability that the ith training pattern should be classified as "yes."

For situations in which the target variable is a probability in the range $0 < t_{ij} < 1$, the value of the cross-entropy at the network's optimum is equal to:

$$E^C_{\min} = -\sum_{i=1}^{N} \left\{ t_i \ln(t_i) + (1-t_i)\ln(1-t_i) \right\}$$

Subtracting $E^C_{\min}$ from E^C gives an error term bounded below by zero, i.e.,

$$E^{CE} \geq 0$$

where: $E^{CE} = E^C - E^C_{\min} = -\sum_{i=1}^{N} \left\{ t_i \ln\left[\frac{\hat{t}_i}{t_i}\right] + (1-t_i)\ln\left[\frac{1-\hat{t}_i}{1-t_i}\right] \right\}$.

This adjusted cross-entropy, E^{CE}, is normally reported when training a binary classification network where $0 < t_{ij} < 1$. Otherwise E^C**,** the unadjusted cross-entropy error, is used. For E^{CE} small values, i.e. values near zero, indicate that the training resulted in a network able to classify the training cases with a low error rate.

Cross-Entropy Error for Multiple Classes

Using a multilayer feedforward neural network for binary classification is relatively straightforward. A network for binary classification only has a single output that estimates the probability that an input pattern belongs to the "yes" class, i.e., $t_i = 1$. In classification problems with more than two mutually exclusive classes, the calculations and network configurations are not as simple.

One approach is to use multiple network outputs, one for each of the *C* classes. Using this approach, the *j*th output for the *i*th training pattern, t_{ij}, is the estimated probability that the *i*th pattern belongs to the jth class, denoted by $\hat{t}_{ij}$. An easy way to estimate these probabilities is to use logistic activation for each output. This ensures that each output satisfies the univariate probability requirements, i.e., $0 \le \hat{t}_{ij} \le 1$.

However, since the classification categories are mutually exclusive, each pattern can only be assigned to one of the *C* classes, which means that the sum of these individual probabilities should always equal 1. However, if each output is the estimated probability for that class, it is very unlikely that $\sum_{j=1}^{C} \hat{t}_{ij} = 1$. In fact, the sum of the individual probability estimates can easily exceed 1 if logistic activation is applied to every output.

Support Vector Machine (SVM) neural networks use this approach with one modification. An SVM network classifies a pattern as belonging to the *i*th category if the activation calculation for that category exceeds a threshold and the other calculations do not exceed this value. That is, the *i*th pattern is assigned to the jth category if and only if $\hat{t}_{ij} > \delta$ and $\hat{t}_{ik} \le \delta$ for all $k \ne j$, where δ is the threshold. If this does not occur, then the pattern is marked as *unclassified*.

Another approach to multi-class classification problems is to use the softmax activation function developed by Bridle (1990) on the network outputs. This approach produces outputs that conform to the requirements of a multinomial distribution. That is

$$\sum_{j=1}^{C} \hat{t}_{ij} = 1 \text{ for all } i = 1, 2, \cdots, N \text{ and } 0 \le \hat{t}_{ij} \le 1 \text{ for all } i = 1, 2, \cdots, N$$

and

$$j = 1, 2, \cdots, C$$

The softmax activation function estimates classification probabilities using the following softmax activation function:

$$\hat{t}_{ij} = \frac{e^{Z_{ij}}}{\sum_{j=1}^{C} e^{Z_{ij}}} \quad ,$$

where Z_{ij} is the potential for the *j*th output perceptron, or category, using the *i*th pattern.

For this activation function, it is clear that:

1. $0 \le \hat{t}_{ij} \le 1$ for all $i = 1, 2, \cdots, N$, $j = 1, 2, \cdots, C$ and

2. $\sum_{j=1}^{C} \hat{t}_{ij} = 1$ for all $i = 1, 2, \cdots, N$

Modeling the C network outputs as multinomial observations leads to the cross-entropy error function described by Hopfield (1987) and Bishop (1995):

$$E^C = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln(\hat{t}_{ij}),$$

where *N* is the number of training patterns, t_{ij} is the target value for the *j*th class of *i*th pattern (either 1 or 0), and $\hat{t}_{ij}$ is the network's *j*th output for the *i*th pattern. $\hat{t}_{ij}$ is equal to the neural network's estimate of the probability that the *i*th pattern should be classified into the *j*th category.

For situations in which the target variable is a probability in the range $0 < t_{ij} < 1$, the value of the cross-entropy at the networks optimum is equal to:

$$E^C_{\min} = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln(t_{ij})$$

Subtracting this from E^C gives an error term bounded below by zero, i.e., $E^{CE} \ge 0$ where:

$$E^{CE} = E^C - E^C_{\min} = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln\left[\frac{\hat{t}_{ij}}{t_{ij}}\right]$$

This adjusted cross-entropy is normally reported when training a binary classification network where $0 < t_{ij} < 1$. Otherwise E^C**,** the non-adjusted cross-entropy error, is used. That is, when 1-in-*C* encoding of the target variable is used,

$$t_{ij} = \begin{cases} 1 & \text{if the } i\text{th pattern belongs to the } j\text{th category} \\ 0 & \text{if the } i\text{th pattern does not belong to the } j\text{th category} \end{cases}$$

Small values, values near zero, indicate that the training resulted in a network with a low error rate and that patterns are being classified correctly most of the time.

Back-Propagation in Multilayer Feedforward Neural Networks

Sometimes a multilayer feedforward neural network is referred to incorrectly as a back-propagation network. The term back-propagation does not refer to the structure or architecture of a network. Back-propagation refers to the method used during network training. More specifically, back-propagation refers to a simple method for calculating the gradient of the network, that is the first derivative of the weights in the network.

The primary objective of network training is to estimate an appropriate set of network weights based upon a training dataset. Many ways have been researched for estimating these weights, but they all involve minimizing some error function. In forecasting the most commonly used error function is the sum-of-squared errors:

$$E = \tfrac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2 .$$

Training uses one of several possible optimization methods to minimize this error term. Some of the more common are: steepest descent, quasi-Newton, conjugant gradient and many various modifications of these optimization routines.

Back-propagation is a method for calculating the first derivative, or gradient, of the error function required by some optimization methods. It is certainly not the only method for estimating the gradient. However, it is the most efficient. In fact, some will argue that the development of this method by Werbos (1974), Parker (1985) and Rumelhart, Hinton and Williams (1986) contributed to the popularity of neural network methods by significantly reducing the network training time and making it possible to train networks consisting of a large number of inputs and perceptrons.

Simply stated, back-propagation is a method for calculating the first derivative of the error function with respect to each network weight. Bishop (1995) derives and describes these calculations for the two most common forecasting error functions – the sum-of-squared errors and Laplacian error functions. Abe (2001) gives the description for the classification error function - the cross-entropy error function. For all of these error functions, the basic formula for the first derivative of the network weight w_{ji} at the *i*th perceptron applied to the output from the *j*th
perceptron is:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j Z_i ,$$

where $Z_i = g(a_i)$ is the output from the *i*th perceptron after activation, and $\frac{\partial E}{\partial w_{ji}}$ is the derivative for a single output and a single training pattern. The overall estimate of the first derivative of w_{ji} is obtained by summing this calculation over all N training patterns and C network outputs.

The term back-propagation gets its name from the way the term δ_j in the back-propagation formula is calculated:

$$\delta_j = g'(a_j) \cdot \sum_k w_{kj} \delta_k ,$$

where the summation is over all perceptrons that use the activation from the *j*th perceptron, $g(a_j)$.

The derivative of the activation functions, $g'(a)$, varies among these functions. See the following table:

Activation Function	$g(a)$	$g'(a)$
Linear	$g(a) = a$	$g'(a) = 1$
Logistic	$g(a) = \frac{1}{1+e^{-a}}$	$g'(a) = g(a)(1-g(a))$
Hyperbolic-tangent	$g(a) = tanh(a)$	$g'(a) = sech^2(a) = 1 - tanh^2(a)$
Squash	$g(a) = \frac{a}{1+\lvert a \rvert}$	$g'(a) = \frac{1}{(1+\lvert a \rvert)^2} = (1 - \lvert g(a) \rvert)^2$

Table 2. Activation Functions and Their Derivatives

mlff_network

Creates a multilayered feedforward neural network.

Synopsis

#include `<imsls.h>`

Imsls_f_NN_Network *`ffnet imsls_f_mlff_network_init`
(*int* `n_inputs`, *int* `n_outputs`)

void `imsls_f_mlff_network` (*Imsls_f_NN_Network* *`ff_net`, ..., 0)

void `imsls_f_mlff_network_free` (*Imsls_f_NN_Network* *`ff_net`)

The type *double* functions are `imsls_d_mlff_network_init`, `imsls_d_mlff_network`, and `imsls_d_mlff_network_free`.

The function `imsl_f_mlff_network_init` is used to initialize the network, the function `imsl_f_mlff_network` is used to build up the network in preparation for training, and the function `imsl_f_mlff_network_free` is used to free the internally allocated structure `ff_net`. Descriptions of these functions are provided below.

Required Arguments for imsls_f_mlff_network_init

int `n_inputs` (Input/Output)
Number of input attributes in the network.

int `n_outputs` (Input)
Number of output attributes in the network.

Return Value for imsls_f_mlff_network_init

Pointer to structure of type *Imsls_f_NN_Network* containing the multilayered feed forward network.

Required Argument for imsls_f_mlff_network

Imsls_f_NN_Network `*ff_net` (Input/Output)
Pointer to structure of type *Imsls_f_NN_Network* containing the multilayered feed forward network.

Required Argument for imsls_f_mlff_network_free

Imsls_f_NN_Network `*ff_net` (Input)
Pointer to structure of type *Imsls_f_NN_Network* containing the multilayered feed forward network.

Synopsis with Optional Arguments

```
#include <imsls.h>

void imsls_f_mlff_network (Imsls_f_NN_Network *ff_net,
        IMSLS_CREATE_HIDDEN_LAYER, int n_perceptrons,
        IMSLS_ACTIVATION_FCN, int layer_id, int activation_fcn[],
        IMSLS_BIAS, int layer_id, float bias[],
        IMSLS_LINK_ALL,
        IMSLS_LINK_LAYER, int to,  int from,
        IMSLS_LINK_NODE, int to, int from,
        IMSLS_REMOVE_LINK, int to, int from,
        IMSLS_WEIGHTS, float weights[],
        IMSLS_N_LINKS, int *n_links,
        0)
```

Optional Arguments for imsls_f_mlff_network

`IMSLS_CREATE_HIDDEN_LAYER`, *int* `n_perceptrons` (Input)
Creates a hidden layer with `n_perceptrons`. To create one or more hidden layers `imsls_f_mlff_network` must be called multiple times with optional argument `IMSLS_CREATE_HIDDEN_LAYER`.
Default: No hidden layer is created.

`IMSLS_ACTIVATION_FCN`, *int* `layer_id`, *int* `activation_fcn[]` (Input)
Specifies the activation function for each perceptron in a hidden layer or the output layer, indicated by `layer_id`. `layer_id` must be between 1 and the number of layers. If a hidden layer has been created, `layer_id` set to 1 will indicate the first hidden layer. If there are zero hidden layers, `layer_id` set to 1 indicates the output layer. Argument `activation_fcn` is an array of length `n_perceptrons` in `layer_id`, where `n_perceptrons` is the number of perceptrons in `layer_id`. `activation_fcn` contains the activation function for the *i*th perceptron. Valid values for `activation_fcn` are:

`IMSLS_LINEAR`	Linear
`IMSLS_LOGISTIC`	Logistic
`IMSLS_TANH`	Hyperbolic-tangent
`IMSLS_SQUASH`	Squash

Default: Output Layer `activation_fcn[i]` = `IMSLS_LINEAR`. All hidden layers `activation_fcn[i]` = `IMSLS_LOGISTIC`.

`IMSLS_BIAS`, *int* `layer_id`, *float* `bias[]`, (Input)
Specifies the bias values for each perceptron in a hidden layer or the output layer, indicated by `layer_id`. `layer_id` must be between 1 and the number of layers. If a hidden layer has been created, `layer_id` set to 1 indicates the first hidden layer. If there are zero hidden layers, `layer_id` set to 1 indicates the output layer. Argument `bias` is an array of length `n_perceptrons` in `layer_id`, where `n_perceptrons` is the number of perceptrons in `layer_id`. `bias` contains the initial *bias* values for the *i*th perceptron.
Default: `bias`[i] = 0.0

`IMSLS_LINK_ALL`, (Input)
Connects all nodes in a layer to each node in the next layer, for all layers in the network. To create a valid network, use `IMSLS_LINK_ALL`, `IMSLS_LINK_LAYER`, or `IMSLS_LINK_NODE`.

`IMSLS_LINK_LAYER`, *int* `to`, *int* `from` (Input)
Creates a link between all nodes in layer `from` to all nodes in layer `to`. Layers are numbered starting at zero with the input layer, then the hidden layers in the order they are created, and finally the output layer. To create a valid network, use `IMSLS_LINK_ALL`, `IMSLS_LINK_LAYER`, or `IMSLS_LINK_NODE`.

or

`IMSLS_LINK_NODE`, *int* `to`, *int* `from` (Input)
Links node `from` to node `to`. Nodes are numbered starting at zero with the input nodes, then the hidden layer perceptrons, and finally the output perceptrons. To create a valid network, use `IMSLS_LINK_ALL`, `IMSLS_LINK_LAYER`, or `IMSLS_LINK_NODE`.

or

`IMSLS_REMOVE_LINK`, *int* `to`, *int* `from` (Input)
Removes the link between node `from` and node `to`. Nodes are numbered starting at zero with the input nodes, then the hidden layer perceptrons, and finally output perceptrons.

`IMSLS_WEIGHTS`, *float* `weights[]` (Input)
Array of length `n_links` containing the initial weight for the *i*th link in the

network. See keyword `IMSLS_N_LINKS`.
Default: `weights[]` = 1.0.

`IMSLS_N_LINKS`, *int* `*n_links` (Output)
Returns the number of links in the network.

Description

A multilayerd feedforward network contains an input layer, an output layer and zero or more hidden layers. The input and output layers are created by the function `imsls_f_mlff_network_init`, where `n_inputs` specifies the number of inputs in the input layer and `n_outputs` specifies the number of perceptrons in the output layer. The hidden layers are created by one or more calls to `imsls_f_mlff_network` with the keyword `IMSLS_CREATE_HIDDEN_LAYER`, where `n_perceptrons` specifies the number of perceptrons in the hidden layer.

The network also contains links or connections between nodes. Links are created by using one of the three optional arguments in the `imsls_f_mlff_network` function, `IMSLS_LINK_ALL`, `IMSLS_LINK_LAYER`, `IMSLS_LINK_NODE`. The most useful is the `IMSLS_LINK_ALL`, which connects every node in each layer to every node in the next layer. A feed forward network is a network in which links are only allowed from one layer to a following layer.

Each link has a *weight* and *gradient* value. Each perceptron node has a *bias* value. When the network is trained, the *weight* and *bias* values are used as initial guesses. After the network is trained using `imsls_f_mlff_network_trainer`, the *weight*, *gradient* and *bias* values are updated in the *Imsls_f_NN_Network* structure.

Each perceptron has an activation function *g*, and a *bias*μ. The value of the percepton is given by $g(Z)$, where *g* is the activation function and z is the potential calculated using

$$Z = \sum_{i=1}^{m} w_i x_i - \mu$$

where x_i are the values of nodes input to this perceptron with weights w_i.

All information for the network is stored in the structure called *Imsls_f_NN_Network*. (If the type is *double*, then the structure name is *Imsls_d_NN_Network*.) This structure describes the network that is trained by `imsls_f_mlff_network_trainer`.

The following code gives a detailed description of this structure:

```
typedef struct
{
  int                 n_layers;
  Imsls_NN_Layer      *layers;
  int                 n_links;
  int                 next_link;
  Imsls_f_NN_Link     *links;
  int                 n_nodes;
  Imsls_f_NN_Node     *nodes;
} Imsls_f_NN_Network;
```

Where *Imsls_NN_Layer* is:

```
typedef struct
{
  int          n_nodes;
  int          *nodes;

} Imsls_NN_Layer;
```

Imsls_NN_Link is:

```
typedef struct
{
  float        weight;
  int          to_node;
  int          from_node;
} Imsls_f_NN_Link;
```

And, *Imsls_NN_Node* is:

```
typedef struct
{
  int          layer_id;
  int          n_inLinks;
  int          n_outLinks;
  int          *inLinks;
  int          *outLinks;
  float        delta;
  float        bias;
  int          ActivationFcn;
} Imsls_f_NN_Node;
```

In particular, if `ff_net` is a pointer to the structure of type *Imsls_f_NN_Network* , then:

Structure member	**Description**
`ff_net->n_layers`	Number of layers in network. Layers are numbered starting at 0 for the input layer.
`ff_net->n_nodes`	Total number of nodes in network, including the input attributes.
`ff_net->n_links`	Total number of links or connections between input attributes and perceptrons and between perceptrons from layer to layer.
`ff_net->layers[0]`	Input layer with `n_inputs` attributes.

Structure member	Description
`ff_net->layers[ff_net->n_layers-1]`	Output layer with `n_outputs` perceptrons.
`ff_net->layers[0].n_nodes`	`n_inputs` (number of input attributes).
`ff_net->layers[ffnet->n_layers-1].n_nodes`	`n_outputs` (number of output perceptrons).
`ff_net->layers[1].n_nodes`	Number of output perceptrons in first hidden layer.
`ff_net->n_links[i].weight`	Initial weight for the *i*th link in network. After the training has completed the structure menber contains the weight used for forecasting.
`ff_net->n_nodes[i].bias`	Initial bias value for the *i*th node. After the training has completed the bias value is updated.

Table 3. Structure Members and Their Descriptions

Nodes are numbered starting at zero with the input nodes, then the hidden layer perceptrons and finally the output perceptrons.

Layers are numbered starting at zero with the input layer, then the hidden layers and finally the output layer. If there are zero hidden layers, the output layer is numbered one.

Use function `imsls_f_mlff_network_free` to free memory allocated by `imsls_f_mlff_network_init`.

Examples

Example 1

This code fragment creates a single-layer feedforward network. The network inputs are directly connected to the output perceptrons. The output perceptrons use the default linear activation function and default bias values of 0.0.

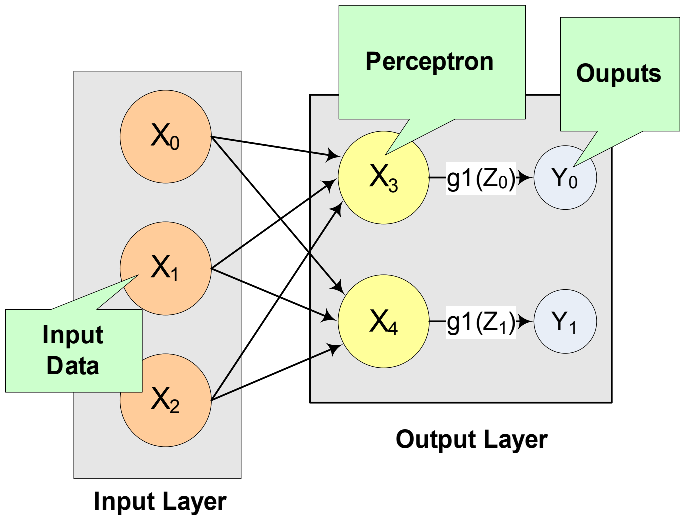


Figure 13- 8: A Single-Layer Feedforward Neural Net

```
#include "imsls.h"
void main()
{
    Imsls_f_NN_Network *ffnet;
    float *stats;
    int n_obs= 100, n_cat=2, n_cont=1;

    /* Data for categorical,continuous, and output omitted
       See imsls_f_mlff_network_trainer Example 1 for a complete
       source code example */
       …


    ffnet = imsls_f_mlff_network_init(3,2);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_ALL, 0);

    stats = imsls_f_mlff_network_trainer(ffnet, n_obs, n_cat, n_cont,
              categorical,continuous, output,0);

    imsls_f_mlff_network_free(ffnet);

}
```

Example 2

This code fragment creates a two-layer feedforward network with four inputs, one hidden layer with three perceptrons and two outputs.

Since the default activation function is linear for output and logistic for the hidden layers, to create a network that uses only linear activation you must specify the linear activation for each hidden layer in the network. This code fragment demonstrates how to change the activation function and bias values for hidden and output layer perceptrons as shown in Figure 13- 9 below.

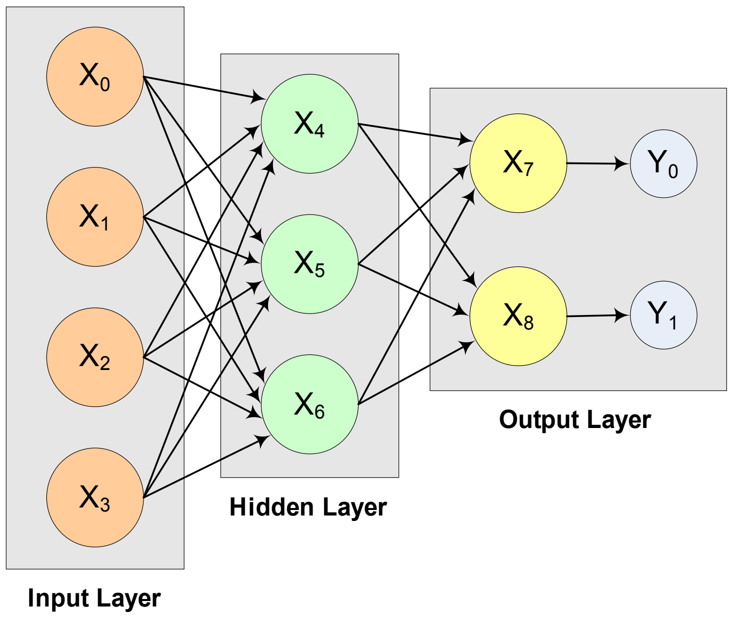


Figure 13- 9: A 2-layer, Feedforward Network with 4 Inputs and 2 Outputs

```
#include "imsls.h"
void main()
{
    Imsls_f_NN_Network *ffnet;
    float *stats;
    int n_obs= 100, n_cat=5, n_cont=1;
    int hidActFcn[3] ={IMSLS_LINEAR, IMSLS_LINEAR, IMSLS_LINEAR};
    int outbias[1] = {1.0};
    int hidbias[3] = {1.0, 1.0, 1.0};

    /* Data for categorical,continuous, and output Omitted
       See imsls_f_mlff_network_trainer Example 1 for a complete
       source code example */
       …


    ffnet = imsls_f_mlff_network_init(4,2);
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 3,
        IMSLS_ACTIVATION_FCN, 1, &hidActFcn,
        IMSLS_BIAS, 2, &outbias,
        IMSLS_LINK_ALL,  0);
    imsls_f_mlff_network(ffnet, IMSLS_BIAS, 1, &hidbias, 0);


    stats = imsls_f_mlff_trainer(ffnet, n_obs, n_cat, n_cont,
                                 categorical,continuous, output,
                                 0);
    imsls_f_mlff_network_free(ffnet);
}
```

Example 3

This example creates a three-layer feedforward network with six input nodes and they are not all connected to every node in the first hidden layer.

Note also that the four perceptrons in the first hidden layer are not connected to every node in the second hidden layer, and the perceptrons in the second hidden layer are not all connected to the two outputs.

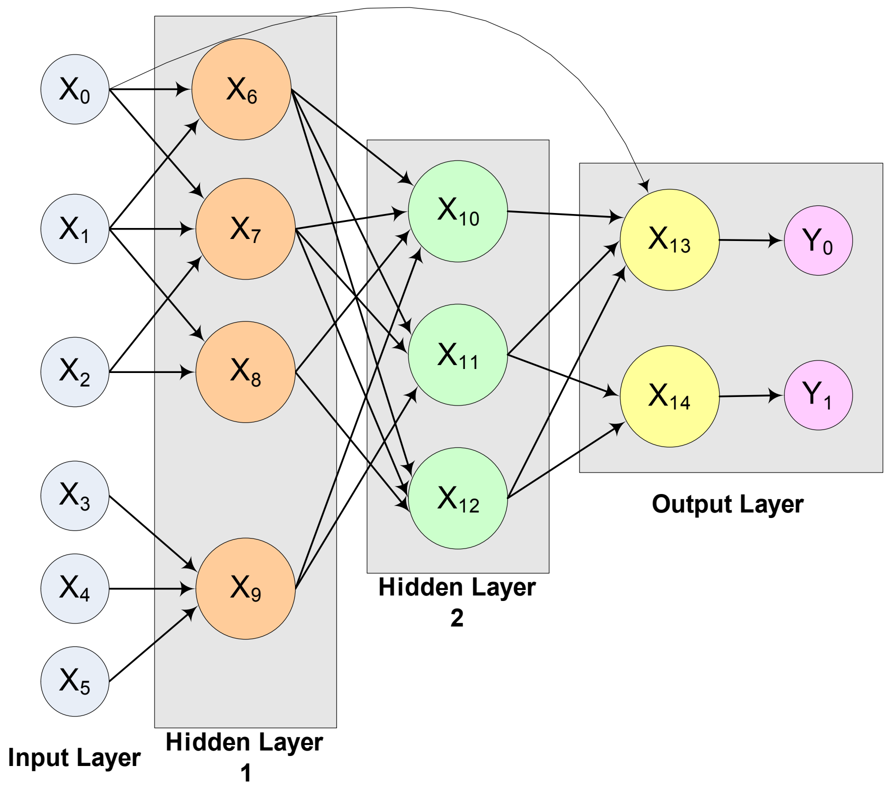


Figure 13- 10: This network uses a total of nine perceptrons to produce two forecasts from six input attributes.

Links among the input nodes and perceptrons can be created using one of several approaches. If all inputs are connected to every perceptron in the first hidden layer, and if all perceptrons are connected to every perceptron in the following layer, which is a standard architecture for feed forward networks, then a call to the `IMSLS_LINK_ALL` method can be used to create these links.

However, this example does not use that standard configuration. Some links are missing. The keyword `IMSLS_LINK_NODE` can be used is to construct individual links or an alternative approach is to first create all links and then remove those that are not needed. The code fragment below illustrates this approach.

```
#include "imsls.h"
void main()
{
    Imsls_f_NN_Network *ffnet;
    float *stats;
    int n_obs= 100, n_cat=4, n_cont=2;

    ffnet = imsls_f_mlff_network_init(6,2);
    /* Create 2 hidden layers and link all nodes 0 */
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 4, 0);
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 3,
        IMSLS_LINK_ALL,  0);
    /* Remove unwanted links from Input 0 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,8,0, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,9,0, 0);
    /* Remove unwanted links from Input 1 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,9,1, 0);
    /* Remove unwanted links from Input 2 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,6,2, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,9,2, 0);
    /* Remove unwanted links from Input 3*/
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,6,3, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,7,3, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,8,3, 0);
    /* Remove unwanted links from Input 4 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,6,4, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,7,4, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,8,4, 0);
    /* Remove unwanted links from Input 5 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,6,5, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,7,5, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,8,5, 0);
    /* Add link from Input 0 to Output Perceptron 0 */
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,13,0, 0);

    /* Remove unwanted links between hidden Layer 1 and hidden layer 2 */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,11,8, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,10,9, 0);

    /* Remove unwanted links between hidden Layer 2 and output layer */
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,14,10, 0);

    stats = imsls_f_network_trainer(ffnet, n_obs, n_cat, n_cont,
                                    categorical,continuous, output,
                                    0);

    imsls_f_mlff_network_free(ffnet);
}
```

Another approach is to use keywords `LINK_NODE` and `LINK_LAYER` to combine links between the two hidden layers, create individual links, and remove the links that are not needed. The following code fragment illustrates this approach:

```
#include "imsls.h"
void main()
{
    Imsls_f_NN_Network *ffnet;
    double *stats;
    int n_obs= 100, n_cat=4, n_cont=2;

    /* Data for categorical,continuous, and output Omitted
       See imsls_network_trainer Example 1 for complete
       source code example */
       …


    ffnet = imsls_f_mlff_network_init(6,2);
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 4, 0);
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 3, 0);

    /* Link input attributes to first hidden layer */
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,6,0, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,7,0, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,6,1, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,7,1, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,8,1, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,7,2, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,8,2, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,9,3, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,9,4, 0);
    imsls_f_mlff_network(ffnet, IMSLS_LINK_NODE,9,5, 0);

    /* Link hidden layer 1 to hidden layer 2 then remove unwanted links */
    imsls_f_mlff_network(ffnet, IMSLS_LINK_LAYER,2,1, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,11,8, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,10,9, 0);

    /* Link hidden layer 2 to output layer then remove unwanted links */
    imsls_f_mlff_network(ffnet, IMSLS_LINK_LAYER,3,2, 0);
    imsls_f_mlff_network(ffnet, IMSLS_REMOVE_LINK,14,10, 0);


    stats = imsls_f_mlff_network_trainer(ffnet, n_obs, n_cat, n_cont,
                                 categorical,continuous, output,
                                0);

    imsls_f_mlff_network_free(ffnet);
}
```

mlff_network_trainer

Trains a multilayered feedforward neural network.

Synopsis

#include `<imsls.h>`

float `*imsls_f_mlff_network_trainer` (*Imsls_f_NN_Network* `*ff_net`,
int `n_observations`, *int* `n_categorical`, *int* `n_continuous`,
int `categorical[]`, *float* `continuous[]`, *float* `output[]`, ..., 0)

The type *double* function is `imsls_d_mlff_network_trainer`.

Return Value

An array of length 5 containing the summary statistics from the network training, organized as follows:

`z[0]`	=	Error sum of squares at the optimum
`z[1]`	=	Total number of Stage I iterations
`z[2]`	=	Smallest error sum of squares after Stage I training
`z[3]`	=	Total number of Stage II iterations
`z[4]`	=	Smallest error sum of squares after Stage II training

If training is unsuccessful, `NULL` is returned.

Required Arguments

Imsls_f_NN_Network `*ff_net` (Input/Output)
Pointer to a structure of type *Imsls_f_NN_Network* containing the feedforward network. See `imsls_f_mlff_network`. On return, the weights and bias values are updated.

int `n_observations` (Input)
Number of network training patterns.

int `n_categorical` (Input)
Number of categorical attributes. `n_categorical` + `n_continuous` must equal `n_inputs`, where `n_inputs` is the number of input attributes in the network. `n_inputs = ff_net->layers[0].n_nodes`. For more details, see `imsls_f_mlff_network`.

int `n_continuous` (Input)
Number of continuous attributes. `n_categorical` + `n_continuous` must equal `n_inputs`, where `n_inputs` is the number of input attributes in the network. `n_inputs = ff_net->layers[0].n_nodes`. For more details, see `imsls_f_mlff_network`.

int `categorical[]` (Input)
Array of size `n_observations` by `n_categorical` containing the input training patterns. Each row of `categorical` contains a training pattern.

float `continuous[]` (Input)
Array of size `n_observations` by `n_continuous` containing the input training patterns. Each row of `continuous` contains a training pattern.

float `output[]` (Input)
Array of size `n_observations` by `n_outputs` containing the output training patterns, where `n_outputs` is the number of output perceptrons in the network.

`n_outputs` = `ff_net->layers[ff_net->n_layers-1].n_nodes`. For more details, see `imsls_f_mlff_network`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_mlff_network_trainer` (*Imsls_f_NN_Network* `*ff_net`,
  int `n_observations` *,* *int* `n_categorical`, *int* `n_continuous`,
  float `categorical[]` *,* *int* `continuous[]`, *float* `output[]`,
  `IMSLS_STAGE_I`, *int* `n_epochs`, *int* `epoch_size`,
  `IMSLS_NO_STAGE_II`,
  `IMSLS_MAX_STEP`, *float* `max_step`,
  `IMSLS_MAX_ITN`, *int* `max_itn`,
  `IMSLS_MAX_FCN`, *int* `max_fcn`,
  `IMSLS_REL_FCN_TOL`, *float* `rfcn_tol`,
  `IMSLS_GRAD_TOL`, *float* `grad_tol`,
  `IMSLS_TOLERANCE`, *float* `tolerance`,
  `IMSLS_PRINT`,
  `IMSLS_RESIDUAL`, *float* `*residuals`,
  `IMSLS_RESIDUAL_USER`, *float* `residuals[]`,
  `IMSLS_GRADIENT`, *float* `*gradients`,
  `IMSLS_GRADIENT_USER`, *float* `gradients[]`,
  `IMSLS_FORECASTS`, *float* `*forecasts`,
  `IMSLS_FORECASTS_USER`, *float* `forecasts[]`,
  `IMSLS_WEIGHTS`, *float* `*weights`,
  `IMSLS_WEIGHTS_USER`, *float* `weights[]`,
  `IMSLS_RETURN_USER`, `float z[]`,
  0)

Optional Arguments

`IMSLS_STAGE_I`, *int* `n_epochs`, *int* `epoch_size` (Input)
  Argument `n_epochs` is the number epochs used for Stage I training and argument `epoch_size` is the number of observations used during each epoch. If epoch training is not needed, set `epoch_size` = `n_observations` and `n_epochs`=1.
  Default: `n_epochs`=15, `epoch_size` = `n_observations`.

`IMSLS_NO_STAGE_II` (Input)
  Specifies no Stage II training is performed.
  Default: Stage II training is performed.

`IMSLS_MAX_STEP`, *float* `max_step` (Input)
  Maximum allowable step size in the optimizer.
  Default: `max_step` = 1000

`IMSLS_MAX_ITN`, *int* `max_itn` (Input)
  Maximum number of iterations in the optimizer, per epoch.
  Default: `max_itn`=1000

`IMSLS_MAX_FCN`, *int* `max_fcn` (Input)
Maximum number of function evaluations in the optimizer, per epoch.
Default: `max_fcn`=400

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

`IMSLS_GRAD_TOL`, *float* `grad_tol` (Input)
Scaled gradient tolerance in the optimizer.
Default: `grad_tol` = $\sqrt{\varepsilon}$, $\sqrt[3]{\varepsilon}$ in double where ε is the machine precision.

`IMSLS_TOLERANCE`, *float* `tolerance` (Input)
Absolute accuracy tolerance for the sum of squared errors in the optimizer.
Default: `tolerance` = 0.1

`IMSLS_PRINT` (Input)
Printing is performed.
Default: No printing is performed.

`IMSLS_RESIDUAL` *float* `**residuals` (Output)
The address of a pointer to an array with `n_observations` by `n_outputs` containing the residuals for each observation in the training data, where `n_outputs` is the number of output perceptrons in the network.
`n_outputs = ff_net->layers[ff_net->n_layers-1].n_nodes`.

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

`IMSLS_GRADIENT` *float* `**gradients` (Output)
The address of a pointer `gradients` to an array of size `n_links` + `n_nodes` – `n_inputs` to store the gradients for each weight found at the optimum training stage, where `n_links = ffnet->n_links`, `n_nodes = ff_net->n_nodes`, and `n_inputs = ff_net->layers[0].nodes`.

`IMSLS_GRADIENT_USER` *float* `gradients[]` (Output)
Storage for array `gradients` is provided by user. See `IMSLS_GRADIENT`.

`IMSLS_FORECASTS` *float* `**forecasts` (Output)
The address of a pointer `forecasts` to an array of size `n_observations` by `n_outputs`, where `n_outputs` is the number of output perceptrons in the network.
`n_outputs = ff_net->layers[ff_net->n_layers-1].n_nodes`. The values of the *i*th row are the forecasts for the outputs for the *i*th training pattern.

`IMSLS_FORECASTS_USER` *float* `forecasts[]` (Output)
Storage for array `forecasts` is provided by user. See `IMSLS_FORECASTS`.

`IMSLS_RETURN_USER`, *float* `z[]` (Output)
User-supplied array of length 5. Upon completion, `z` contains the return array of training statistics.

Description

Function `imsls_f_mlff_network_trainer` trains a multilayered feedforward neural network returning the forecasts for the training data, their residuals, the optimum weights and the gradients associated with those weights. Linkages among perceptrons allow for skipped layers, including linkages between inputs and perceptrons. The linkages and activation function for each perceptron, including output perceptrons, can be individually configured. For more details, see optional arguments `IMSLS_LINK_ALL`, `IMSLS_LINK_LAYER`, and `IMSLS_LINK_NODE` in `imsls_f_mlff_network`.

Training Data

Neural network training patterns consist of the following three types of data:

1. *categorical input attributes*
2. *continuous input attributes*
3. *continuous output classes*

The first data type contains the encoding of any nominal input attributes. If binary encoding is used, this encoding consists of creating columns of zeros and ones for each class value associated with every nominal attribute. If only one attribute is used for input, then the number of columns is equal to the number of classes for that attribute. If more columns appear in the data, then each nominal attribute is associated with several columns, one for each of its classes.

Each column consists of zeros, if that classification is not associated with this case, otherwise, one if that classification is associated. Consider an example with one nominal variable and two classes: *male* and *female* (male, male, female, male, female). With binary encoding, the following matrix is sent to the training engine to represent this data:

$$categoricalAtt = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Continuous input and output data are passed to the training engine using two double precision arrays: `continuous` and `outputs`. The number of rows in each of these matrices is `n_observations`. The number of columns in `continuous` and `outputs`, corresponds to the number of input and output variables, respectively.

Network Configuration

The network configuration consists of the following:

- *the number of inputs and outputs*
- *the number of hidden layers*

- *a description of the number of perceptrons in each layer*
- *and a description of the linkages among the perceptrons*

This description is passed into `imsls_f_mlff_network_trainer` using the structure *Imsls_f_NN_Network*. See `imsls_f_mlff_network`.

Training Efficiency

The training efficiency determines the time it takes to train the network. This is controlled by several factors. One of the most important factors is the initial weights used by the optimization algorithm. These are taken from the initial values provided in the structure *Imsls_f_NN_Network*, `ff_net->links[i].weight`. Equally important are the scaling and filtering applied to the training data.

In most cases, all variables, particularly output variables, should be scaled to fall within a narrow range, such as [0, 1]. If variables are unscaled and have widely varied ranges, then numerical overflow conditions can terminate network training before an optimum solution is calculated.

Output

Output from `imsls_f_mlff_network_trainer` consists of scaled values for the network outputs, a corresponding forecast array for these outputs, a weights array for the trained network, and the training statistics. The *Imsls_f_NN_Network* structure is updated with the weights and bias values and can be used as input to `imsls_f_mlff_network_forecast`. For more details about the weights and bias values, see Table 3.

Examples

Example 1

This example trains a two-layer network using 100 training patterns from one nominal and one continuous input attribute. The nominal attribute has three classifications which are encoded using binary encoding. This results in three binary network input columns. The continuous input attribute is scaled to fall in the interval [0,1].

The network training targets were generated using the relationship:

$$Y = 10*X_1 + 20*X_2 + 30*X_3 + 2.0*X_4,$$

where X_1, X_2, X_3 are the three binary columns, corresponding to the categories 1-3 of the nominal attribute, and X_4 is the scaled continuous attribute.

The structure of the network consists of four input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:

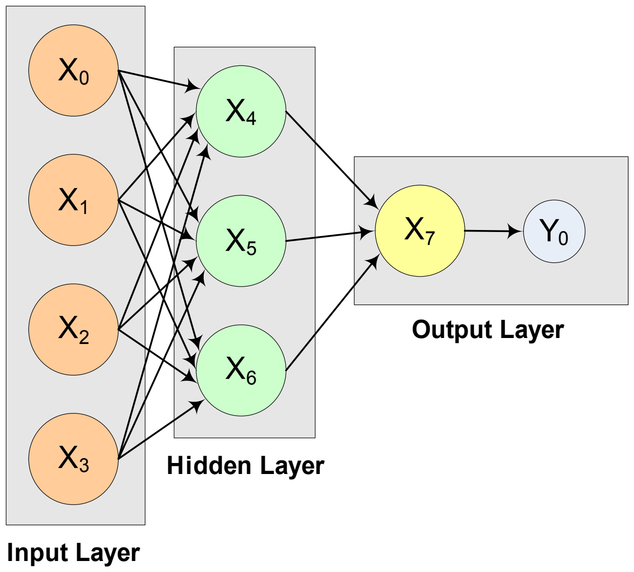


Figure 13- 11: A 2-layer, Feedforward Network with 4 Inputs and 1 Output

There are a total of 15 weights and 4 bias weights in this network. The activation functions are all linear.

Since the target output is a linear function of the input attributes, linear activation functions guarantee that the network forecasts will exactly match their targets. Of course, the same result could have been obtained using multiple regression. Printing is turned on to show progress during the training session.

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

void main()
{
    /* A 2D matrix of values for the categorical training
    attribute.  In this example,  the single categorical
    attribute has 3 categories that are encoded using binary
    encoding for input into the network.

    {1,0,0} = category 1
    {0,1,0} = category 2
    {0,0,1} = category 3
    */
    int categorical[300] =
    {
        1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,
        1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,
        1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,
        1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,

        0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,
        0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,
    0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,

    0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,
    0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,
    0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,
    0,0,1,0,0,1,0,0,1,0,0,1,0,0,1
};

/* A matrix of values for the continuous training attribute */
float continuous[100] = {
    4.007054658,7.10028447,4.740350984,5.714553211,6.205437459,
    2.598930065,8.65089967,5.705787357,2.513348184,2.723795955,
    4.1829356,1.93280416,0.332941608,6.745567628,5.593588463,
    7.273544478,3.162117939,4.205381208,0.16414745,2.883418275,
    0.629342241,1.082223406,8.180324708,8.004894314,7.856215418,
    7.797143157,8.350033996,3.778254431,6.964837082,6.13938006,
    0.48610387,5.686627923,8.146173848,5.879852653,4.587492779,
    0.714028533,7.56324211,8.406012623,4.225261454,6.369220241,
    4.432772218,9.52166984,7.935791508,4.557155333,7.976015058,
    4.913538616,1.473658514,2.592338905,1.386872932,7.046051685,
    1.432128376,1.153580985,5.6561491,3.31163251,4.648324851,
    5.042514515,0.657054195,7.958308093,7.557870384,7.901990083,
    5.2363088,6.95582150,8.362167045,4.875903563,1.729229471,
    4.380370223,8.527875685,2.489198107,3.711472959,4.17692681,
    5.844828801,4.825754155,5.642267843,5.339937786,4.440813223,
    1.615143829,7.542969339,8.100542684,0.98625265,4.744819569,
    8.926039258,8.813441887,7.749383991,6.551841576,8.637046998,
    4.560281415,1.386055087,0.778869034,3.883379045,2.364501589,
    9.648737525,1.21754765,3.908879368,4.253313879,9.31189696,
    3.811953836,5.78471629,3.414486452,9.345413015,1.024053777
};
/* A 2D matrix containing the training outputs for this network.
In this case there is an exact linear relationship between these
outputs and the inputs: output = 10*X1 +20*X2 + 30*X3 +2*X4,
where X1-X3 are the categorical variables and X4 is the continuous
attribute variable.   Output is unscaled.
*/
float output[100];
Imsls_f_NN_Network *ffnet;
float *stats;
int n_obs= 100, n_cat=3, n_cont=1;
int i;
float *residuals, *forecasts, *weights;
float bias, coef1, coef2, coef3, coef4;
int hidActFcn[3] = {IMSLS_LINEAR,IMSLS_LINEAR,IMSLS_LINEAR};

/* Scale continuous attribute into the interval [0, 1]
and generate outputs */
for(i=0; i < 100; i++)
{
    continuous[i] = continuous[i]/10.0;
    output[i] = (10 * categorical[i*3]) + (20 * categorical[i*3+1]) +
        (30 * categorical[i*3+2]) + (20 * continuous[i]);
}
```

```
    /* Create network */
    ffnet = imsls_f_mlff_network_init(4,1);
    imsls_f_mlff_network(ffnet, IMSLS_CREATE_HIDDEN_LAYER, 3,
        IMSLS_ACTIVATION_FCN, 1, &hidActFcn,
        IMSLS_LINK_ALL,  0);

    /*  Set initial weights */
    for (i=0; i<ffnet->n_links; i++)
    {
        /* hidden layer 1 */
        if (ffnet->nodes[ffnet->links[i].to_node].layer_id == 1)
            ffnet->links[i].weight = .25;
        /* output layer */
        if (ffnet->nodes[ffnet->links[i].to_node].layer_id == 2)
            ffnet->links[i].weight = .33;
    }

    /* Initialize seed for consisten results */
    imsls_random_seed_set(12345);
    stats = imsls_f_mlff_network_trainer(ffnet, n_obs, n_cat, n_cont,
        categorical,continuous, output,
        IMSLS_STAGE_I, 10, 100,
        IMSLS_MAX_FCN, 1000,
        IMSLS_REL_FCN_TOL, 1.0e-20,
        IMSLS_GRAD_TOL, 1.0e-20,
        IMSLS_MAX_STEP, 5.0,
        IMSLS_TOLERANCE, 1.0e-5,
        IMSLS_PRINT,
        IMSLS_RESIDUAL, &residuals,
        IMSLS_FORECASTS, &forecasts,
        0);

    printf("Predictions for Last Ten Observations: \n");

    for(i=90; i < 100; i++){
        printf("observation[%d] %f Prediction %f Residual %f \n", i,
output[i],
            forecasts[i], residuals[i]);
    }
    /* hidden layer nodes bias value * link weight */
    bias   = ffnet->nodes[ffnet->n_nodes-4].bias * ffnet->links[12].weight +
        ffnet->nodes[ffnet->n_nodes-3].bias * ffnet->links[13].weight +
        ffnet->nodes[ffnet->n_nodes-2].bias * ffnet->links[14].weight;
    bias  += ffnet->nodes[ffnet->n_nodes-1].bias;  /* the bias of the output
node */
    coef1  = ffnet->links[0].weight * ffnet->links[12].weight;
    coef1 += ffnet->links[4].weight * ffnet->links[13].weight;
    coef1 += ffnet->links[8].weight * ffnet->links[14].weight;
    coef2  = ffnet->links[1].weight * ffnet->links[12].weight;
    coef2 += ffnet->links[5].weight * ffnet->links[13].weight;
    coef2 += ffnet->links[9].weight * ffnet->links[14].weight;
    coef3  = ffnet->links[2].weight * ffnet->links[12].weight;
    coef3 += ffnet->links[6].weight * ffnet->links[13].weight;
    coef3 += ffnet->links[10].weight * ffnet->links[14].weight;
    coef4  = ffnet->links[3].weight * ffnet->links[12].weight;
    coef4 += ffnet->links[7].weight * ffnet->links[13].weight;
    coef4 += ffnet->links[11].weight * ffnet->links[14].weight;
    coef1 += bias;
    coef2 += bias;
    coef3 += bias;

    printf("Bias: %f \n", bias);
    printf("X1: %f \n", coef1);
    printf("X2: %f \n", coef2);
    printf("X3: %f \n", coef3);
    printf("X4: %f \n", coef4);

    imsls_f_mlff_network_free(ffnet);

}
```

Output

```
TRAINING PARAMETERS:
  Stage II Opt.   = 1
  n_epochs        = 10
  epoch_size      = 100
  max_itn         = 1000
  max_fcn         = 1000
  max_step        = 5.000000
  rfcn_tol        = 1e-20
  grad_tol        = 1e-20
  tolerance       = 0.000010

STAGE I TRAINING STARTING
Stage I: Epoch 1 - Epoch Error SS = 3.57886e-10 (Iterations=34)
Stage I Training Converged at Epoch = 1


STAGE I FINAL ERROR SS = 0.000000

OPTIMUM WEIGHTS AFTER STAGE I TRAINING:
weight[0] = 0.262463    weight[1] = 1.30687     weight[2] = 1.32345
weight[3] = 0.929833
weight[4] = -1.40295    weight[5] = 1.46973     weight[6] = 4.50657
weight[7] = 6.25732
weight[8] = 2.05971     weight[9] = 2.55983     weight[10] = 3.40746
weight[11] = 3.52705
weight[12] = 0.371129   weight[13] = 3.43777    weight[14] = -0.526312
weight[15] = 1.41332
weight[16] = 4.33401    weight[17] = 6.28003    weight[18] = 3.69105

STAGE I TRAINING CONVERGED
STAGE I ERROR SS = 0.000000


GRADIENT AT THE OPTIMUM WEIGHTS
g[0] =       0.000001        weight[0] =    0.262463
g[1] =       -0.000023       weight[1] =    1.306865
g[2] =       0.000027        weight[2] =    1.323447
```

```
g[3] =       0.000007         weight[3] =    0.929833
g[4] =       0.000010         weight[4] =    -1.402949
g[5] =       -0.000216        weight[5] =    1.469729
g[6] =       0.000249         weight[6] =    4.506571
g[7] =       0.000063         weight[7] =    6.257323
g[8] =       -0.000002        weight[8] =    2.059708
g[9] =       0.000033         weight[9] =    2.559830
g[10] =      -0.000038        weight[10] =   3.407457
g[11] =      -0.000010        weight[11] =   3.527051
g[12] =      0.000049         weight[12] =   0.371129
g[13] =      0.000399         weight[13] =   3.437771
g[14] =      0.000235         weight[14] =   -0.526312
g[15] =      0.000005         weight[15] =   1.413319
g[16] =      0.000043         weight[16] =   4.334013
g[17] =      -0.000007        weight[17] =   6.280032
g[18] =      0.000012         weight[18] =   3.691053

Training Completed

Predictions for Last Ten Observations:
observation[90] 49.297478 Prediction 49.297482 Residual 0.000004
observation[91] 32.435097 Prediction 32.435097 Residual 0.000000
observation[92] 37.817757 Prediction 37.817760 Residual 0.000004
observation[93] 38.506630 Prediction 38.506630 Residual 0.000000
observation[94] 48.623795 Prediction 48.623802 Residual 0.000008
observation[95] 37.623909 Prediction 37.623913 Residual 0.000004
observation[96] 41.569431 Prediction 41.569435 Residual 0.000004
observation[97] 36.828972 Prediction 36.828976 Residual 0.000004
observation[98] 48.690826 Prediction 48.690826 Residual 0.000000
observation[99] 32.048107 Prediction 32.048107 Residual 0.000000
Bias: 15.809660
X1: 9.999999
X2: 19.999996
X3: 30.000000
X4: 20.000002
```

mlff_network_forecast

Calculates forecasts for trained multilayered feedforward neural networks.

Synopsis

#include `<imsls.h>`

float `*imsls_f_mlff_network_forecast` (*Imsls_f_NN_Network* `*ff_net`,
int `n_categorical`, *int* `n_continuous`,
int `categorical[]`, *float* `continuous[]`, ..., 0)

The type *double* function is `imsls_d_mlff_network_forecast`.

Return Value

Pointer to an array of size `n_outputs` containing the forecasts, where `n_outputs` is the number of output perceptrons in the network.
`n_outputs` = `ff_net->layers[ff_net->n_layers-1].n_nodes`.

Required Arguments

Imsls_f_NN_Network `*ff_net` (Input)
Pointer to a structure of type *Imsls_f_NN_Network* containing the trained feedforward network. See `imsls_f_mlff_network`.

int `n_categorical` (Input)
Number of categorical attributes.

int `n_continuous` (Input)
Number of continous attributes.

int `categorical[]` (Input)
Array of size `n_categorical` containing the categorical input variables.

float `continuous[]` (Input)
Array of size `n_continuous` containing the continuous input variables.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_mlff_network_forecast` (*Imsls_f_NN_Network* `*ff_net`,
int `n_categorical`, *int* `n_continuous`, *int* `categorical[]`,
float `continuous[]`,
`IMSLS_RETURN_USER`, *float* `forecasts[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `forecasts[]` (Output)
If specified, the forecasts for the trained network is stored in array `forecasts` of size `n_outputs`, where `n_outputs` is the number of perceptrons in the network.
`n_outputs = ff_net->layers[ff_net->n_layers -1].n_nodes.`

Description

Function `imsls_f_mlff_network` calculates a forecast for a previously trained multilayered feedforward neural network using the same network structure and scaling applied during the training. The structure *Imsls_f_NN_Network* describes the network structure used to originally train the network. The weights, which are the key output from training, are used as input to this routine. The weights are stored in the *Imsls_f_NN_Network* structure.

In addition, two one-dimensional arrays are used to describe the values of the categorical and continuous attributes that are to be used as network inputs for calculating the forecast.

Function `imsls_f_mlff_network` returns a forecast, calculated using the network input attributes provided.

Training Data

Neural network training data consist of the following three types of data:
1. categorical input attribute data
2. continuous input attribute data

3. *continuous output data*

The first data type contains the encoding of any nominal input attributes. If binary encoding is used, this encoding consists of creating columns of zeros and ones for each class value associated with every nominal attribute. If only one attribute is used for input, then the number of columns is equal to the number of classes for that attribute. If more columns appear in the data, then each nominal attribute is associated with several columns, one for each of its classes.

Each column consists of zeros, if that classification is not associated with this case, otherwise, one if that classification is associated. Consider an example with one nominal variable and two classes: *male* and *female* (male, male, female, male, female). With binary encoding, the following matrix is sent to the training engine to represent this data:

$$categoricalAtt = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Continuous input and output data are passed to the training engine using two double precision arrays: `continuous` and `outputs`. The number of rows in each of these matrices is `n_observations`. The number of columns in `continuous` and `outputs`, corresponds to the number of input and output variables, respectively.

Network Configuration

The configuration of the network consists of a description of the number of perceptrons for each layer, the number of hidden layers, the number of inputs and outputs, and a description of the linkages among the perceptrons. This description is passed into this training routine through the structure *Imsls_f_NN_Network*. See `imsls_f_mlff_network`.

Forecast Calculation

The forecast is calculated from the input attributes, network structure and weights provided in the structure *Imsls_f_NN_Network*.

Examples

Example 1

This example trains a two-layer network using 90 training patterns from one nominal and one continuous input attribute. The nominal attribute has three classifications which are encoded using binary encoding. This results in three binary network input columns. The continuous input attribute is scaled to fall in the interval [0,1].

The network training targets were generated using the relationship:

$$Y = 10*X_1 + 20*X_2 + 30*X_3 + 2.0*X_4,$$

where X_1, X_2, X_3 are the three binary columns, corresponding to the categories 1-3 of the nominal attribute, and X_4 is the scaled continuous attribute.

The structure of the network consists of four input nodes ands two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:

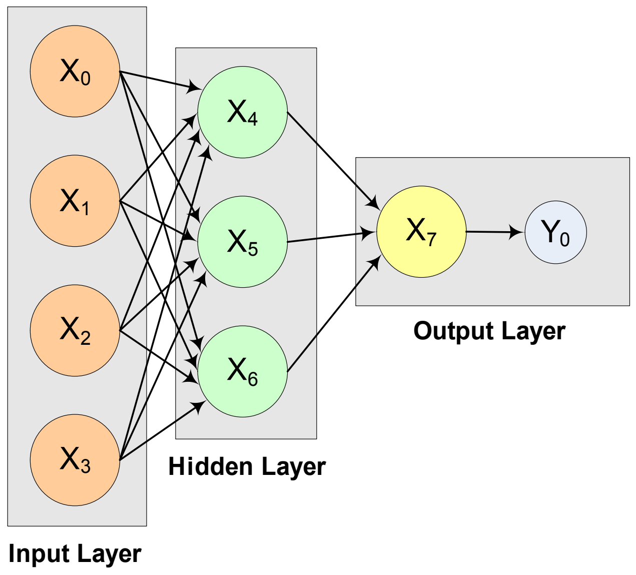


Figure 13- 12: A 2-layer, Feedforward Network with 4 Inputs and 1 Output

There are a total of 100 outputs. Training the first 90 and forecasting the 10 and compare the forecasted values with the actual outputs.

```
#include "imsls.h"
#include <stdio.h>
void
main ()
{
  static int categorical[300] = {
    1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
    0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
    0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
    0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
    0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1
  };
  static float continuous[100] = {
    4.007054658, 7.10028447, 4.740350984, 5.714553211, 6.205437459,
    2.598930065, 8.65089967, 5.705787357, 2.513348184, 2.723795955,
    4.1829356, 1.93280416, 0.332941608, 6.745567628, 5.593588463,
    7.273544478, 3.162117939, 4.205381208, 0.16414745, 2.883418275,
    0.629342241, 1.082223406, 8.180324708, 8.004894314, 7.856215418,
    7.797143157, 8.350033996, 3.778254431, 6.964837082, 6.13938006,
    0.48610387, 5.686627923, 8.146173848, 5.879852653, 4.587492779,
    0.714028533, 7.56324211, 8.406012623, 4.225261454, 6.369220241,
    4.432772218, 9.52166984, 7.935791508, 4.557155333, 7.976015058,
    4.913538616, 1.473658514, 2.592338905, 1.386872932, 7.046051685,
    1.432128376, 1.153580985, 5.6561491, 3.31163251, 4.648324851,
    5.042514515, 0.657054195, 7.958308093, 7.557870384, 7.901990083,
    5.2363088, 6.95582150, 8.362167045, 4.875903563, 1.729229471,
    4.380370223, 8.527875685, 2.489198107, 3.711472959, 4.17692681,
    5.844828801, 4.825754155, 5.642267843, 5.339937786, 4.440813223,
    1.615143829, 7.542969339, 8.100542684, 0.98625265, 4.744819569,
    8.926039258, 8.813441887, 7.749383991, 6.551841576, 8.637046998,
    4.560281415, 1.386055087, 0.778869034, 3.883379045, 2.364501589,
    9.648737525, 1.21754765, 3.908879368, 4.253313879, 9.31189696,
    3.811953836, 5.78471629, 3.414486452, 9.345413015, 1.024053777
  };
  static float output[100] = {
    18.01410932, 24.20056894, 19.48070197, 21.42910642, 22.41087492,
    15.19786013, 27.30179934, 21.41157471, 15.02669637, 15.44759191,
    18.3658712, 13.86560832, 10.66588322, 23.49113526, 21.18717693,
    24.54708896, 16.32423588, 18.41076242, 10.3282949, 15.76683655,
    11.25868448, 12.16444681, 26.36064942, 26.00978863, 25.71243084,
    25.59428631, 26.70006799, 17.55650886, 23.92967416, 22.27876012,
    10.97220774, 21.37325585, 26.2923477, 21.75970531, 19.17498556,
    21.42805707, 35.12648422, 36.81202525, 28.45052291, 32.73844048,
    28.86554444, 39.04333968, 35.87158302, 29.11431067, 35.95203012,
    29.82707723, 22.94731703, 25.18467781, 22.77374586, 34.09210337,
    22.86425675, 22.30716197, 31.3122982, 26.62326502, 29.2966497,
    30.08502903, 21.31410839, 35.91661619, 35.11574077, 35.80398017,
    30.4726176, 33.91164302, 36.72433409, 29.75180713, 23.45845894,
    38.76074045, 47.05575137, 34.97839621, 37.42294592, 38.35385362,
    41.6896576, 39.65150831, 41.28453569, 40.67987557, 38.88162645,
    33.23028766, 45.08593868, 46.20108537, 31.9725053, 39.48963914,
    47.85207852, 47.62688377, 45.49876798, 43.10368315, 47.274094,
    39.1205628, 32.77211017, 31.55773807, 37.76675809, 34.72900318,
    49.29747505, 32.4350953, 37.81775874, 38.50662776, 48.62379392,
    37.62390767, 41.56943258, 36.8289729, 48.69082603, 32.04810755
  };

  /* 2D Array Definitions */
#define CATEGORICAL(i,j) categorical[i*n_cat+j]
#define CATEGORICALOBS(i,j) categoricalObs[i*n_cat+j]

  Imsls_f_NN_Network *ffnet;

  float *stats;
  int n_obs = 100, n_cat = 3, n_cont = 1;
  int i, j;
  float *forecasts;
```

```
/*  for forecasting */
int categoricalObs[3] = { 0, 0, 0 };
float continuousObs[1] = { 0 };
float x, y;
float forecast[5];
float *cont;

/* Scale continuous attribute to the interval [0, 1] */
cont = imsls_f_scale_filter (n_obs, continuous, 1,
                         IMSLS_SCALE_LIMITS, 0.0, 10.0, 0.0, 1.0, 0);


ffnet = imsls_f_mlff_network_init (4, 1);

imsls_f_mlff_network (ffnet,
                  IMSLS_CREATE_HIDDEN_LAYER, 3, IMSLS_LINK_ALL, 0);

for (i = 0; i < ffnet->n_links; i++)
  {

    /* hidden layer 1 */
    if (ffnet->nodes[ffnet->links[i].to_node].layer_id == 1)
     {
       ffnet->links[i].weight = .25;
     }


    /* output layer */
    if (ffnet->nodes[ffnet->links[i].to_node].layer_id == 2)
     {
       ffnet->links[i].weight = .33;
     }

  }

imsls_random_seed_set (12345);
stats = imsls_f_mlff_network_trainer (ffnet, n_obs - 10, n_cat,
                               n_cont, categorical, continuous, output,
                               0);

printf ("Predictions for Observations 90 to 100: \n");

for (i = 90; i < 100; i++)
  {
    continuousObs[0] = continuous[i];
    for (j = 0; j < n_cat; j++)
     {
       CATEGORICALOBS (0, j) = CATEGORICAL (i, j);
     }

    forecasts = imsls_f_mlff_network_forecast (ffnet, n_cat, n_cont,
                                       categoricalObs,
                                       continuousObs, 0);

    x = output[i];
```

```
      y = forecasts[0];
      printf
       ("observation[%d] %8.4f    Prediction %8.4f    Residual %8.4f \n",
        i, x, y, x - y);

    }


  imsls_f_mlff_network_free (ffnet);
#undef CATEGORICAL
#undef CATEGORICALOBS
}
```

Output

```
NOTE: Because multiple optima are possible during training, the output of
this example can vary by platform.

Predictions for Observations 90 to 100:

observation[90]  49.2975    Prediction  43.8761    Residual   5.4213
observation[91]  32.4351    Prediction  23.6643    Residual   8.7708
observation[92]  37.8178    Prediction  30.4261    Residual   7.3916
observation[93]  38.5066    Prediction  31.2768    Residual   7.2298
observation[94]  48.6238    Prediction  43.1369    Residual   5.4869
observation[95]  37.6239    Prediction  30.1860    Residual   7.4379
observation[96]  41.5694    Prediction  35.0006    Residual   6.5688
observation[97]  36.8290    Prediction  29.1978    Residual   7.6311
observation[98]  48.6908    Prediction  43.2108    Residual   5.4800
observation[99]  32.0481    Prediction  23.1740    Residual   8.8742
```

scale_filter

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.

Synopsis

#include `<imsls.h>`

float `*` `imsls_f_scale_filter` (*int* `n_obs`, *float* `x[]`, *int* `method`, ...,0)

The type *double* function is `imsls_d_scale_filter`.

Required Arguments

int `n_obs` (Input)
Number of observations.

float `x[]` (Input)
An array of length `n_obs`. The values in `x` are either the scaled or unscaled values of a continuous variable. Missing values are allowed, and are indicated by placing a NaN (not a number) in `x`. See `imsls_f_machine`(6).

int `method` (Input)

The scaling method to apply to each variable. The association of the value in `method` and the scaling algorithm is summarized in the table below. The sign of `method` determines whether the values in `x` are scaled or unscaled. If `method` is positive then values in `x` are scaled. If `method` is negative then values in `x` are unscaled.

Method	Algorithm
0	No scaling.
±1	Bounded scaling and unscaling.
±2	Unbounded z-score scaling using the mean and standard deviation.
±3	Unbounded z-score scaling using the median and mean absolute difference.
±4	Bounded z-score scaling using the mean and standard deviation.
±5	Bounded z-score scaling using the median mean absolute difference.

Return Value

A pointer to an internally allocated array of length `n_obs` containing either the scaled or unscaled value of `x`, depending upon whether `method` is positive or negative, respectively. If errors are encountered, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*` `imsls_f_scale_filter` (*int* `n_obs`, *float* `x[]`, *int* `method`,
`IMSLS_RETURN_USER`, *float* `z[]`,
`IMSLS_SCALE_LIMITS`, *float* `real_min`, *float* `real_max`,
float `target_min`, *float* `target_max`,
`IMSLS_SUPPLY_CENTER_SPREAD`, *float* `center`, *float* `spread`,
`IMSLS_RETURN_CENTER_SPREAD`, *float* `*center`,
float `*spread`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `z[]` (Output)

A user-supplied array of length `n_obs` containing either the scaled or unscaled values of `x`, depending upon whether `method` is positive or negative, respectively.

`IMSLS_SCALE_LIMITS`, *float* `real_min`, *float* `real_max`, *float* `target_min`, *float* `target_max` (Input)

The real and target limits for `x`. This optional argument is required when bounded scaling is performed, i.e., `method`=±1, ±4, or ±5. `real_min` is the lowest value expected for each input variable in `x`. `real_max` is the largest value expected. `target_min` is lowest value allowed for the output variable, `z`. `target_max` is the largest value allowed for the output variable.

`IMSLS_SUPPLY_CENTER_SPREAD`, *float* `center`, *float* `spread` (Input)
The values `center` and `spread` are only used for z-score scaling or unscaling of `x`, that is, when `method` is one of ±2, ±3, ±4, and ±5. The value of `center` is either the mean or median, and the value of `spread` is either the standard deviation or mean absolute difference. When `method` is positive, this optional argument can be used to supply a user-defined center and spread rather than allowing `imsls_f_scale_filter` to compute the center and spread from the data in `x`. When `method` is one of -2, -3, -4, or -5, this optional argument must be used to supply the center and spread used during scaling.

`IMSLS_RETURN_CENTER_SPREAD`, *float* `*center`, *float* `*spread` (Output)
Pointers to scalars containing the computed center and spread of `x`. The values `center` and `spread` are only used for z-score scaling or unscaling of `x`. These methods, ±2, ±3, ±4, and ±5, require two numbers, either the mean or median, and either the standard deviation, or mean absolute difference. The value of `center` is either the mean or median for `x`. The value of `spread` is either the standard deviation or mean absolute difference.

Description

The function `imsls_f_scale_filter` is designed to either scale or unscale a continuous variable using one of four methods prior to their use as neural network input or output.

The specific encoding computations employed are specified by argument `method`. Scaling limits are supplied with the optional argument `IMSLS_SCALE_LIMITS`, and are required for the bounded scaling methods, i.e., `method`=±1, ±4, or ±5. Bounded scaling ensures that the scaled values in the returned array fall between a lower and upper bound.

If `method`=1 then the bounded method of scaling and unscaling is applied to `x` using the scaling limits in `scale_limit`.

If `method`=±2, ±3, ±4, or ±5, then the z-score method of scaling is used. These calculations are based upon the following scaling calculation:

$$z[i] = \frac{(x[i]-a)}{b},$$

where a is a measure of center for `x`, and b is a measure of the spread of `x`.

If `method`=±2 or ±4, then by default a and b are the arithmetic average and sample standard deviation of the training data. These values can be overridden using the optional argument `IMSLS_SUPPLY_CENTER_SPREAD`.

If `method`=±3 or ±5, then by default a and b are the median and $\tilde{s}$, where $\tilde{s}$ is a robust estimate of the population standard deviation:

$$\tilde{s} = \frac{MAD}{0.6745}$$, where MAD is the Mean Absolute Deviation

$$MAD = median\{|x_j - median\{x\}|\}.$$

Again, the values of a and b can be overridden using the optional argument IMSLS_SUPPLY_CENTER_SPREAD.

Method ±1: Bounded Scaling and Unscaling

If method=1, then the optional argument IMSLS_SCALE_LIMITS is required and a scaling operation is conducted using the scale limits for x using the following calculation:

$$z[i] = r\left(x[i] - real\_min\right) + target\_min,$$

where

$$r = \frac{target\_max - target\_min}{real\_max - real\_min}.$$

If method=-1, then optional argument IMSLS_SCALE_LIMITS is required and an unscaling operation is conducted by inverting the following calculation:

$$x[i] = \frac{\left(z[i] - target\_min\right)}{r} + real\_min.$$

Method +2 or +3: Unbounded z-score Scaling

If method=2 or method=3, then a scaling operation is conducted using the scale limits of x using a z-score calculation:

$$z[i] = \frac{\left(x[i] - center\right)}{spread},$$

If either center or spread are missing, (a NaN), then appropriate values are calculated from the non-missing values of x. If method=2, then center is set equal to the arithmetic average $\bar{x}$, and spread is set equal to the sample standard deviation, s.

If method=3, then center is set equal to the median $\tilde{m}$, and center is set equal to the Mean Absolute Difference (MAD).

Method -2 or -3: Unbounded z-score Unscaling

If method=-2 or method=-3, then an unscaling operation is conducted using the inverse calculation for the equation shown in the above section, "*Method* +2 *or* +3*: Unbounded z-score Scaling*."

$$x[i] = spread \cdot z[i] + center.$$

For these values of `method`, missing values for `center` and `spread` are not allowed. If `method`=-2, then `center` and `spread` are assumed to be equal to the arithmetic average and standard deviation, respectively. These values would normally be the same used in scaling the variable with `method`=+2. If `method`= -3, then `center` and `spread` are assumed to be equal to the median and mean absolute difference, respectively. These values would normally be the same used in scaling the variable with `method`=+3.

Method +4 or +5: Bounded z-score Scaling

This method is essentially the same as the z-score calculation described for `method`=+2 and `method`=+3 with additional scaling or unscaling using the scale limits. If `method`=4, then the optional argument `IMSLS_SCALE_LIMITS` is required and a scaling operation is conducted using the scale limits for `x` using the widely known z-score calculation:

$$z[i] = \frac{r \cdot \left(x[i] - center\right)}{spread} - r \cdot real\_min + target\_min \; .$$

If either `center` or `spread` are missing, (a NaN), then appropriate values are calculated from the non-missing values in `x`. If `center` is missing and `method`=+4, then `center` is set equal to the arithmetic average $\bar{x}$, and `spread` is set equal to the Sample Standard Deviation, s . If `center` is missing and `method`=+5, then `x_stats`[i] is set equal to the median $\tilde{m}$, and `spread` is set equal to the MAD.

In bounded scaling, if `z`[*i*] exceeds its bounds, it is set to the boundary it exceeded.

Method -4 or -5: Bounded z-score unscaling

If `method`=-4 or `method`=-5, then the optional argument `IMSLS_SCALE_LIMITS` is required and an unscaling operation is conducted using the inverse calculation for the equation below.

$$x[i] = \frac{spread \cdot \left(z[i] - target\_min\right)}{r} + spread \cdot real\_min + center$$

For these values of `method`, missing values for `center` and `spread` are not allowed. If `method`=-4, then `center` and `spread` are assumed to be equal to the arithemetic average and standard deviation, respectively. These values would normally be the same used in scaling `x` with `method`=+4. If `method`=-5, then `center` and `spread` are assumed to be equal to the median and mean absolute difference, respectively. These values would normally be the same used in scaling the `x` with `method`=+5.

Examples

Example 1

In this example two data sets are filtered using bounded *z*-score scaling.

```
#include <imsls.h>
void main()
```

```
{
    int n_obs=5;
    float x1[] = {3.5, 2.4, 4.4, 5.6, 1.1};
    float x2[] = {3.1, 1.5, - 1.5, 2.4, 4.2};
    float *z1, *z2;
    float *y1, *y2;
    float center1, spread1;
    float center2, spread2;

    z1 = imsls_f_scale_filter(n_obs, x1, 4,
                              IMSLS_SCALE_LIMITS, -6.0, 6.0, -3.0, 3.0,
                              IMSLS_RETURN_CENTER_SPREAD, &center1, &spread1,
                              0);
    z2 = imsls_f_scale_filter(n_obs, x2, 5,
                              IMSLS_SCALE_LIMITS, -3.0, 3.0, -3.0, 3.0,
                              IMSLS_RETURN_CENTER_SPREAD, &center2, &spread2,
                              0);

    imsls_f_write_matrix("z1", n_obs, 1, z1, 0);
    printf("Center = %f\nSpread = %f\n", center1, spread1);
    imsls_f_write_matrix("z2", n_obs, 1, z2, 0);
    printf("Center = %f\nSpread = %f\n", center2, spread2);

    /* Un-scale z1 and z2. */
    y1 = imsls_f_scale_filter(n_obs, z1, -4,
                              IMSLS_SCALE_LIMITS, -6.0, 6.0, -3.0, 3.0,
                              IMSLS_SUPPLY_CENTER_SPREAD, center1, spread1,
                              0);
    y2 = imsls_f_scale_filter(n_obs, z2, -5,
                              IMSLS_SCALE_LIMITS, -3.0, 3.0, -3.0, 3.0,
                              IMSLS_SUPPLY_CENTER_SPREAD, center2, spread2,
                              0);
    imsls_f_write_matrix("y1", n_obs, 1, y1, 0);
    imsls_f_write_matrix("y2", n_obs, 1, y2, 0);
}
```

Output

3.

```
     z1
1      0.0287
2     -0.2870
3      0.2870
4      0.6314
5     -0.6601
Center = 3.400000
Spread = 1.742125

     z2
1       0.525
2      -0.674
3      -2.923
4       0.000
5       1.349
Center = 2.400000
Spread = 1.334342
```

```
     y1
1         3.5
2         2.4
3         4.4
4         5.6
5         1.1

     y2
1         3.1
2         1.5
3        -1.5
4         2.4
5         4.2
```

time_series_filter

Converts time series data to the format required for processing by a neural network.

Synopsis

#include `<imsls.h>`

*float** `imsls_f_time_series_filter` (*int* `n_obs`, *int* `n_var`, *int* `max_lag`,
float `x[]`, ...,0)

The type *double* function is `imsls_d_time_series_filter`.

Required Arguments

int `n_obs` (Input)
Number of observations. The number of observations must be greater than `n_lags`.

int `n_var` (Input)
Number of variables (columns) in `x`. The number of variables must be one or greater, `n_var`>0.

int `max_lag` (Input)
The number of lags. The number of lags must be one or greater, `max_lag`>0.

float `x[]` (Input)
An array of size `n_obs` by `n_var`. All data must be sorted in chronological order from most recent to oldest observations.

Return Value

A pointer to an internally allocated array of size (`n_obs-max_lag`) by `n_var`*(`max_lag`+1)) If errors are encountered, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

*float** `imsls_f_time_series_filter` (*int* `n_obs`, *int* `n_var`,
int `max_lag`, *float* `x[]`,

IMSLS_RETURN_USER, *float* z[],
0)

Optional Arguments

IMSLS_RETURN_USER, *float* z[] (Output)
User supplied array of size (n_obs-max_lag) by n_var*(max_lag+1) containing the filtered data.

Description

Function imsls_f_time_series_filter accepts a data matrix and lags every column to form a new data matrix. The input matrix, x, contains n_var columns. Each column is transformed into (max_lag+1) columns by lagging its values.

Since a lag of zero is always included in the output matrix z, the total number of lags is n_lags = max_lag+1.

The output data array, z, can be represented symbolically as:

z = |x(0) : x(1) : x(2) : … : x(max_lag)|,

where x(i) is the *i*th lag of the incoming data matrix, x. For example, if x={1, 2, 3, 4, 5} and n_var=1, then n_obs=5, and x(0)=x, x(1)={2, 3, 4, 5}, x(2)={3, 4, 5}, etc.

Consider, an example in which n_obs=5 and n_var=2 with all variables continuous input attributes. It is assumed that the most recent observations are in the first row and the oldest are in the last row.

$$x = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & 8 \\ 4 & 9 \\ 5 & 10 \end{bmatrix}.$$

If max_lag=1, then the number of columns will be n_var*(max_lag+1)=2*2=4, and the number of rows will be n_obs-max_lag=5-1=4:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 \\ 2 & 7 & 3 & 8 \\ 3 & 8 & 4 & 9 \\ 4 & 9 & 5 & 10 \end{bmatrix}.$$

If max_lag=2, then the number of columns will be n_var*(max_lag+1)=2*3=6. , and the number of rows will be n_obs-max_lag=5-2=3:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \\ 2 & 7 & 3 & 8 & 4 & 9 \\ 3 & 8 & 4 & 9 & 5 & 10 \end{bmatrix}.$$

Example 1

In this example, the matrix `x` with 5 rows and 2 columns is lagged twice, i.e. `max_lag`=2. This produces an output two-dimensional matrix with 5(`n_obs-max_lag`)=5-2=3 rows, but 2*3=6 columns. The first two columns correspond to lag=0, which simply places the original data into these columns. The 3rd and 4th columns contain the first lags of the original 2 columns and the 5th and 6th columns contain the second lags. Note that the number of rows for the output matrix `z` is less than the number for the input matrix `x`.

```
#include <imsls.h>
void main ()
{
#define N_OBS 5
#define N_VAR 2
#define MAX_LAG 2
  float x[N_OBS*N_VAR] = {1, 6,
                          2, 7,
                          3, 8,
                          4, 9,
                          5, 10};


  float *z;

  z = imsls_f_time_series_filter(N_OBS, N_VAR, MAX_LAG, (float*)x, 0);
  imsls_f_write_matrix("X", N_OBS, N_VAR, (float*)x, 0);
  imsls_f_write_matrix("Z", N_OBS-MAX_LAG, N_VAR*(MAX_LAG+1), z, 0);
}
```

Output

```
              X
              1            2
1             1            6
2             2            7
3             3            8
4             4            9
5             5           10

                                         Z
              1            2            3            4            5            6
1             1            6            2            7            3            8
2             2            7            3            8            4            9
3             3            8            4            9            5           10
```

time_series_class_filter

Converts time series data sorted within nominal classes in decreasing chronological order to a useful format for processing by a neural network.

Synopsis

#include `<imsls.h>`

*float** `imsls_f_time_series_class_filter` (*int* `n_obs`, *int* `n_lags`,
int `n_classes`, *int* `i_class[]`, *float* `x[]`, …,0)

The type *double* function is `imsls_d_time_series_class_filter`.

Required Arguments

int `n_obs` (Input)
Number of observations. The number of observations must be greater than `n_lags`.

int `n_lags` (Input)
The number of lags. The number of lags must be one or greater.

int `n_classes` (Input)
The number of classes associated with these data. The number of classes must be one or greater.

int `i_class[]` (Input)
An array of length `n_obs`. The *i*th element in `i_class` is equal to the class associated with the *i*th element of `x`. The classes must be numbered from 1 to `n_classes`.

float `x[]` (Input)
A sorted array of length `n_obs`. This array is assumed to be sorted first by class designations and then descending by chronological order, i.e., most recent observations appear first within a class.

Return Value

A pointer to an internally allocated array of size `n_obs` by `n_lags` columns. If errors are encountered, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

*float** `imsls_f_time_series_class_filter` (*int* `n_obs`, *int* `n_lags`,
int `n_classes`, *int* `i_class[]`, *float* `x[]`,
`IMSLS_RETURN_USER`, *float* `z[]`,
`IMSLS_LAGS`, *int* `lag[]`,
0)

The type *double* function is `imsls_d_time_series_class_filter`.

Optional Arguments

`IMSLS_RETURN_USER`, *float* `z[]` (Output)
: A user-supplied array of size `n_obs` by `n_lags`. The *i*th column contains the lagged values of `x` for a lag equal to the number of lags in `lag[i]`.

`IMSLS_LAGS`, *int* `lag[]` (Input)
: An array of length `n_lags`. The *i*th element in `lag` is equal to the lag requested for the *i*th column of `z`. Every lag must be non-negative.
: Default: `lag[i]=i`

Description

The function `imsls_f_time_series_class_filter` accepts a data array, `x[]`, and returns a new data array, `z[]`, containing `n_lags` columns, each containing a lagged version of `x`.

The output data array, `z`, can be represented symbolically as:

$$z = |x(0) : x(1) : x(2) : \ldots : x(\texttt{n\_lags-1})|,$$

where `x`(i) is the *i*th lagged column of the incoming data array, `x`. Notice that `n_lags` is the number of lags and not the maximum lag. The maximum number of lags is `max_lag= n_lags-1`, unless the optional input `log[]` is given, the highest lag is `max_lags`. If `n_lags` = 2 and the optional input `log[]` is not given, then the output array contains the lags 0, 1.

Consider, an example in which `n_obs`=10, `n_lags` =2 and

$$x^T = \{1,2,3,4,5,6,7,8,9,10\}.$$

If $lag^T = \{0,2\}$ and

$$i\_class^T = \{1,1,1,1,1,1,1,1,1,1\}.$$

then, `n_classes`=1 and `z` would contain 2 columns and 10 rows:

$$z = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & 6 \\ 5 & 7 \\ 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & NaN \\ 10 & NaN \end{bmatrix}.$$

Note that since $lag^T = [0,1]$, the first column of z is formed using a lag of zero and the second is formed using a lag of two. A zero lag corresponds to no lag, which is why the first column of `z` in this example is equal to the original data in `x`.

On the other hand, if the data were organized into two classes with

$$i\_class^T = \{1,1,1,1,1,2,2,2,2,2\},$$

then z is still a 2 by 10 matrix, but with the following values:

$$z = \left[\begin{array}{cc} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & NaN \\ 5 & NaN \\ \hline 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & NaN \\ 10 & NaN \end{array}\right]$$

The first 5 rows of `z` are the lagged columns for the first class, and the last five are the lagged columns for the second class.

Example 1

Suppose that the training data to the neural network consists of the following data matrix consisting of a single nominal variable coded into two binary columns and a single time series variable:

$$\begin{bmatrix} 0 & 1 & 2.1 \\ 0 & 1 & 2.3 \\ 0 & 1 & 2.4 \\ 0 & 1 & 2.5 \\ 1 & 0 & 1.1 \\ 1 & 0 & 1.2 \\ 1 & 0 & 1.3 \\ 1 & 0 & 1.4 \end{bmatrix}$$

In this case, `n_obs`=8 and `n_classes`=2. If we wanted to lag the 3rd column by 2 time lags, i.e., `n_lags`=2,

$$lag^T = \{0,1\},$$

$$i\_class^T = \{1,1,1,1,2,2,2,2\}$$, and

$$x^T = \{2.1, 2.3, 2.4, 2.5, 1.1, 1.2, 1.3, 1.4\}$$.

The resulting data matrix would have 4 rows and 2 columns:

$$z = \begin{bmatrix} x(0) & x(1) \end{bmatrix} = \left[\begin{array}{cc} 2.1 & 2.3 \\ 2.3 & 2.4 \\ 2.4 & 2.5 \\ 2.5 & NaN \\ \hline 1.1 & 1.2 \\ 1.2 & 1.3 \\ 1.3 & 1.4 \\ 1.4 & NaN \end{array}\right].$$

```
void main(){
#define N_OBS 8
#define N_LAGS 2
        float x[N_OBS] = {2.1, 2.3, 2.4, 2.5, 1.1, 1.2, 1.3, 1.4};
        float *z;
        int n_classes = 2;
        int i_class[] =  {1,1,1,1,2,2,2,2};
        z = imsls_f_time_series_class_filter(N_OBS, N_LAGS, n_classes,
                                             i_class, x,
                                             0);
        imsls_f_write_matrix("z", N_OBS, N_LAGS, (float*)z, 0);
}
```

Output

```
                z
              1            2
1           2.1          2.3
2           2.3          2.4
3           2.4          2.5
4           2.5  ...........
5           1.1          1.2
6           1.2          1.3
7           1.3          1.4
8           1.4  ...........
```

unsupervised_nominal_filter

Converts nominal data into a series of binary encoded columns for input to a neural network. Optionally, it can also reverse the binary encoding, accepting a series of binary encoded columns and returning a single column of nominal classes.

Synopsis

#include `<imsls.h>`

*int** `imsls_unsupervised_nominal_filter` (*int* `n_obs`,
int `n_classes`, *int* `x[]`, *...,* 0)

Required Arguments

int `n_obs` (Input)
Number of observations.

int `* n_classes` (Input/Output)
A pointer to the number of classes in `x[]`. `n_classes` is output for `IMSLS_ENCODE` and input for `IMSLS_DECODE`.

int `x[]` (Input)
A one or two-dimensional array depending upon whether encoding or decoding is requested. If encoding is requested, `x` is an array of length `n_obs` containing the categories for a nominal variable numbered from 1 to `n_classes`. If decoding is requested, then `x` is an array of size `n_obs` by `n_classes`. In this case, the columns contain only zeros and ones that are interpreted as binary encoded representations for a single nominal variable.

Return Value

A pointer to an internally allocated array, `z[]`. The values in `z` are either the encoded or decoded values for `x`, depending upon whether `IMSLS_ENCODE` or `IMSLS_DECODE` is requested. If errors are encountered, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

*int** `imsls_f_unsupervised_nominal_filter` (*int* `n_obs`, *int* `x[]`,
`IMSLS_RETURN_USER`, *int* `z[]`,
`IMSLS_ENCODE` or
`IMSLS_DECODE`,
0)

Optional Arguments

`IMSLS_ENCODE` *or* `IMSLS_DECODE` (Input)
If `IMSLS_ENCODE` is specified, binary encoding is requested. Classes must be numbered sequentially from 1 to `n_classes`. `IMSLS_DECODE` is used to request that `x` be decoded. The values in each column should be zeros and ones. The values in the *i*th column of `x` are associated with the *i*th class of the

nominal variable.
Default: `IMSLS_ENCODE`.

`IMSLS_RETURN_USER`, *int* `z[]` (Output)
A user-supplied array of size `n_obs` by `n_classes`. If `IMSLS_DECODE` is specified, then `z` should be length `n_obs`. The value in `z[i]` is either the encoded or decoded value for `x[i]`, depending upon whether `IMSLS_ENCODE` or `IMSLS_DECODE` is specified.

Description

The function `imsls_unsupervised_nominal_filter` is designed to either encode or decode nominal variables using a simple binary mapping.

Binary Encoding: IMSLS_ENCODE

In this case, `x[]` is an input array to which a binary filter is applied. Binary encoding takes each category in `x[]`, and creates a column in `z[]`, the output matrix, containing all zeros and ones. A value of zero indicates that this category is not present and a value of one indicates that it is present.

For example, if `x[]={2, 1, 3, 4, 2, 4}` then `n_classes=4`, and

$$z = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that the number of columns in `z` is equal to the number of distinct classes in `x`. The number of rows in `z` is equal to the length of `x`.

Binary Decoding: IMSLS_DECODE

Binary decoding takes each column in `x[]`, and returns the appropriate class in `z[]`.

For example, if `x[]` is the same as described above:

$$x = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

then `z[]` would be returned as `z[]`={2, 1, 3, 4, 2, 4}. Notice this is the same as the original array because classes are numbered sequentially from 1 to `n_classes`. This ensures that the *i*th column of `x[]` is associated with the *i*th class in the output array.

```
#include <imsls.h>

void main ()
{
#define N_OBS 7
    int x[N_OBS] = {3, 3, 1, 2, 2, 1, 2};
    int *x2;
    int *z, n_classes;
    /* Binary Filtering. */
    z = imsls_unsupervised_nominal_filter(N_OBS, &n_classes, x, 0);
       printf("n_classes = %d\n",n_classes);
    imsls_i_write_matrix("X", N_OBS, 1, (int*)x, 0);
    imsls_i_write_matrix("Z", N_OBS, n_classes, z, 0);
    /* Binary Unfiltering. */
    x2 = imsls_unsupervised_nominal_filter(N_OBS, &n_classes, z,
                                  IMSLS_DECODE, 0);
    imsls_i_write_matrix("Unfiltering result", N_OBS, 1, x2, 0);
 }
```

Output

```
n_classes = 3

  X
1   3
2   3
3   1
4   2
5   2
6   1
7   2

      Z
    1   2   3
1   0   0   1
2   0   0   1
3   1   0   0
4   0   1   0
5   0   1   0
6   1   0   0
7   0   1   0

Unfiltering result
       1   3
       2   3
       3   1
       4   2
       5   2
       6   1
       7   2
```

unsupervised_ordinal_filter

Converts ordinal data into proportions. Optionally, it can also reverse encoding, accepting proportions and converting them into ordinal values.

Synopsis

#include `<imsls.h>`

void `imsls_f_unsupervised_ordinal_filter` (*int* `n_obs`,
int `x[]`, *float* `z[]` ..., 0)

The type *double* function is `imsls_d_unsupervised_ordinal_filter`.

Required Arguments

int `n_obs` (Input)
Number of observations.

int `x[]` (Input/Output)
An array of length `n_obs` containing the classes for the ordinal data. Classes must be numbered 1 to `IMSLS_N_CLASSES`. This is an output argument if `IMSLS_DECODE` is specified, otherwise it is input.

float `z[]` (Input/Output)
An array of length `n_obs` containing the encoded values for `x` represented as cumulative proportions associated with each ordinal class (values between 0.0 and 1.0 inclusive). This is an input argument if `IMSLS_DECODE` is specified, otherwise it is output.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_unsupervised_ordinal_filter` (*int* `n_obs`, *int* `x[]`,
float `z[]`,
`IMSLS_ENCODE` *or*
`IMSLS_DECODE`,
`IMSLS_NO_TRANSFORM`, *or*
`IMSLS_SQUARE_ROOT`, *or*
`IMSLS_ARC_SIN`,
`IMSLS_N_CLASSES`, *int* * `n_classes`,
0)

The type *double* function is `imsls_d_unsupervised_ordinal_filter`.

Optional Arguments

`IMSLS_ENCODE` or `IMSLS_DECODE` (Input)
If `IMSLS_ENCODE` is specified, `z` is an output array and `x` is an input array that is filtered by converting each ordinal class value into a cumulative proportion (a value between 0.0 and 1.0 inclusive). If `IMSLS_DECODE` is specified, `x` is an output array and `z` is an input array that contains transformed cumulative proportions. In this case, the transformed cumulative proportions are

converted into ordinal class values using the coding class=1, 2, … etc.
Default: `IMSLS_ENCODE`.

`IMSLS_SQUARE_ROOT` or `IMSLS_ARC_SIN` or `IMSLS_NO_TRANSFORM` (Input)
`IMSLS_NO_TRANSFORM` indicates that the cumulative proportions used to encode the ordinal variable are not transformed. If `IMSLS_SQUARE_ROOT` is specified, cumulative proportions are transformed using the square root transformation. If `IMSLS_ARC_SIN` is specified, the cumulative proportions are transformed using the arcsin of the square root of the cumulative proportions.
Default: `IMSLS_NO_TRANSFORM` .

`IMSLS_N_CLASSES`, *int* `* n_classes` (Output)
The number of ordinal classes in `x` and the number of unique proportions in `z`.

Description

The function `imsls_f_unsupervised_ordinal_filter` is designed to either encode or decode ordinal variables. Filtering consists of transforming the ordinal classes into proportions, with each proportion being equal to the proportion of the data at or below this class.

Ordinal Filtering: IMSLS_ENCODE

In this case, `x` is an input array that is filtered by converting each ordinal class value into a cumulative proportion.

For example, if `x[]={2, 1, 3, 4, 2, 4, 1, 1, 3, 3}` then `n_obs`=10 and `IMSLS_N_CLASSES`=4. This function then fills `z` with cumulative proportions represented as proportions displayed in the table below. Cumulative proportions are equal to the proportion of the data in this class or a lower class.

Ordinal Class	Frequency	Cumulative Proportion
1	3	30%
2	2	50%
3	3	80%
4	2	100%

If `IMSLS_NO_TRANSFORM` is specified, then the equivalent proportions in `z` are

`z[]`={0.50, 0.30, 0.80, 1.00, 0.50, 1.00, 0.30, 0.30, 0.80, 0.80}.

If `IMSLS_SQUARE_ROOT` is specified, then the square root of these values is returned, i.e.,

$$z[i] = \sqrt{\frac{z[i]}{100}}$$

`z[]`={0.71, 0.55 , 0.89, 1.0, 0.71, 1.0, 0.55, 0.55, 0.89, 0.89};

If `IMSLS_ARC_SIN` is specified, then the arcsin square root of these values is returned using the following calculation:

$$z[i] = \arcsin\left(\sqrt{\frac{z[i]}{100}}\right)$$

Ordinal UnFiltering: IMSLS_DECODE

Ordinal Unfiltering takes the transformed cumulative proportions in `z` and converts them into ordinal class values using the coding class=1, 2, … etc.

For example, if `IMSLS_NO_TRANSFORM` is specified and `z[]`={0.20, 1.00, 0.20, 0.40, 1.00, 1.00, 0.40, 0.10, 1.00, 1.00} then upon return, the output array would consist of the ordinal classes `x[]`={2, 4, 2, 3, 4, 4, 3, 1, 4, 4}.

If one of the transforms is specified, the same operation is performed since the transformations of the proportions are monotonically increasing. For example, if the original observations consisted of {2.8, 5.6, 5.6, 1.2, 4.5, 7.1}, then input x for encoding would be `x[]`={2, 4, 4, 1, 3, 5} and output `IMSLS_N_CLASSES`=5. The output array `x` after decoding would consist of the ordinal classes `x[]`={2, 4, 4, 1, 3, 5}.

Example 1

A taste test was conducted yielding the following data:

Individual	Rating
1	Poor
2	Good
3	Very Good
4	Very Poor
5	Very Good

The data in the table above would have the coded values shown below. This assumes that the rating scale is: very poor, poor, good, and very good.

`x`={2, 3, 4, 1, 4}

The returned values are:

`z`={0.40, 0.60, 1.00, 0.20, 1.00}.

```
#include <imsls.h>

void main () {
#define N_OBS 5
       int x[N_OBS] = {2,3,4,1,4};
       int x2[N_OBS], n_classes;
       float z[N_OBS];

       /* Filtering. */
       imsls_f_unsupervised_ordinal_filter(N_OBS, x, z,
              IMSLS_N_CLASSES, &n_classes,
              0);
       printf("n_classes = %d\n", n_classes);
       imsls_i_write_matrix("x", N_OBS, 1, x, 0);
       imsls_f_write_matrix("z", N_OBS, 1, z, 0);

       /* Unfiltering. */
       imsls_f_unsupervised_ordinal_filter(N_OBS, x2, z,
              IMSLS_DECODE,
              IMSLS_N_CLASSES, &n_classes,
              0);
       printf("\nn_classes = %d\n", n_classes);
       imsls_i_write_matrix("x-unfiltered", N_OBS, 1, x2, 0);
}
```

Output

```
n_classes = 4

  x
1   2
2   3
3   4
4   1
5   4

       z
1          0.4
2          0.6
3          1.0
4          0.2
5          1.0

n_classes = 4

x-unfiltered
    1   2
    2   3
    3   4
    4   1
    5   4
```

Chapter 14: Printing Functions

Routines

Print a matrix or vector	write_matrix	981
Set the page width and length	page	986
Set the printing options	write_options	987

write_matrix

Prints a rectangular matrix (or vector) stored in contiguous memory locations.

Synopsis

#include `<imsls.h>`

void `imsls_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`, …, 0)

For *int* `a[]`, use `imsls_i_write_matrix`.
For *double* `a[]`, use `imsls_d_write_matrix`.

Required Arguments

char `*title` (Input)
Matrix title. Use `\n` within a title to create a new line. Long titles are automatically wrapped.

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

int `nca` (Input)
Number of columns in the matrix.

float `a[]` (Input)
Array of size `nra` × `nca` containing the matrix to be printed.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`,
`IMSLS_TRANSPOSE`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,

IMSLS_PRINT_ALL, *or*
IMSLS_PRINT_LOWER, *or*
IMSLS_PRINT_UPPER, *or*
IMSLS_PRINT_LOWER_NO_DIAG, *or*
IMSLS_PRINT_UPPER_NO_DIAG,
IMSLS_WRITE_FORMAT, *char* *fmt,
IMSLS_NO_ROW_LABELS, *or*
IMSLS_ROW_NUMBER, *or*
IMSLS_ROW_NUMBER_ZERO, *or*
IMSLS_ROW_LABELS, *char* *rlabel[],
IMSLS_NO_COL_LABELS, *or*
IMSLS_COL_NUMBER, *or*
IMSLS_COL_NUMBER_ZERO, *or*
IMSLS_COL_LABELS, *char* *clabel[],
0)

Optional Arguments

IMSLS_TRANSPOSE
Print a^T.

IMSLS_A_COL_DIM, *int* a_col_dim (Input)
Column dimension of *a*.
Default: a_col_dim = nca

IMSLS_PRINT_ALL, *or*
IMSLS_PRINT_LOWER, *or*
IMSLS_PRINT_UPPER, *or*
IMSLS_PRINT_LOWER_NO_DIAG, *or*
IMSLS_PRINT_UPPER_NO_DIAG
Exactly one of these optional arguments can be specified to indicate that either a triangular part of the matrix or the entire matrix is to be printed. If omitted, the entire matrix is printed.

Keyword	Action
IMSLS_PRINT_ALL	Entire matrix is printed (the default).
IMSLS_PRINT_LOWER	Lower triangle of the matrix is printed, including the diagonal.
IMSLS_PRINT_UPPER	Upper triangle of the matrix is printed, including the diagonal.
IMSLS_PRINT_LOWER_NO_DIAG	Lower triangle of the matrix is printed, without the diagonal.
IMSLS_PRINT_UPPER_NO_DIAG	Upper triangle of the matrix is printed, without the diagonal.

IMSLS_WRITE_FORMAT, *char* *fmt (Input)
Character string containing a list of C conversion specifications (formats) to be used when printing the matrix. Any list of C conversion specifications suitable

for the data type can be given. For example, `fmt` = `"%10.3f"` specifies the conversion character `f` for the entire matrix. For the conversion character `f`, the matrix must be of type *float* or *double*.
Alternatively,`fmt` = `"%10.3e%10.3e%10.3f%10.3f%10.3f"` specifies the conversion character `e` for columns 1 and 2 and the conversion character `f` for columns 3, 4, and 5. If the end of `fmt` is encountered and if some columns of the matrix remain, format control continues with the first conversion specification in `fmt`.

Aside from restarting the format from the beginning, other exceptions to the usual C formatting rules are as follows:

Characters not associated with a conversion specification are not allowed. For example, in the format `fmt` = `"1%d2%d"`, the characters 1 and 2 are not allowed and result in an error.

A conversion character `d` can be used for floating-point values (matrices of type *float* or *double*). The integer part of the floating-point value is printed.

For printing numbers whose magnitudes are unknown, the conversion character `g` is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The `w` (or `W`) conversion character is a special conversion character used by this function to select a conversion specification so that the decimal points will be aligned. The conversion specification ending with `w` is specified as `"%n.dw"`. Here, `n` is the field width and `d` is the number of significant digits generally printed. Valid values for `n` are 3, 4, …, 40. Valid values for `d` are 1, 2, …, `n` – 2. If `fmt` specifies one conversion specification ending with `w`, all elements of `a` are examined to determine one conversion specification for printing. If `fmt` specifies more than one conversion specification, separate conversion specifications are generated for each conversion specification ending with `w`. Set `fmt` = `"10.4w"` for a single conversion specification selected automatically with field width 10 and with four significant digits.

`IMSLS_NO_ROW_LABELS`, *or*
`IMSLS_ROW_NUMBER`, *or*
`IMSLS_ROW_NUMBER_ZERO`, *or*
`IMSLS_ROW_LABELS`, *char* `*rlabel[]` (Input)

If `IMSLS_ROW_LABELS` is specified, `rlabel` is a vector of length `nra` containing pointers to the character strings comprising the row labels. Here, `nra` is the number of rows in the printed matrix. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no row labels are desired, use the `IMSLS_NO_ROW_LABELS` optional argument. If the numbers 1, 2, …, `nra` are desired, use the `IMSLS_ROW_NUMBER` optional argument. If the numbers 0, 1, 2, …, `nra` – 1 are desired, use the `IMSLS_ROW_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nra` are used for the row labels by default whenever `nra` > 1.
If `nra` = 1, the default is no row labels.

`IMSLS_NO_COL_LABELS`, *or*
`IMSLS_COL_NUMBER`, *or*
`IMSLS_COL_NUMBER_ZERO`, *or*
`IMSLS_COL_LABELS`, *char* `*clabel[]` (Input)

If `IMSLS_COL_LABELS` is specified, `clabel` is a vector of length `nca` + 1 containing pointers to the character strings comprising the column headings. The heading for the row labels is `clabel` [0]; `clabel` [*i*], $i = 1, \ldots,$ `nca`, is the heading for the *i*-th column. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no column labels are desired, use the `IMSLS_NO_COL_LABELS` optional argument. If the numbers 1, 2, …, `nca`, are desired, use the `IMSLS_COL_NUMBER` optional argument. If the numbers 0, 1, …, `nca` – 1 are desired, use the `IMSLS_COL_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nca` are used for the column labels by default whenever `nca` > 1. If `nca` = 1, the default is no column labels.

Description

Function `imsls_write_matrix` prints a real rectangular matrix (stored in *a*) with optional row and column labels (specified by `rlabel` and `clabel`, respectively, regardless of whether a or a^T is printed). An optional format, `fmt`, can be used to specify a conversion specification for each column of the matrix.

In addition, the write matrix functions can restrict printing to the elements of the upper or lower triangles of a matrix by using the `IMSLS_PRINT_UPPER`, `IMSLS_PRINT_LOWER`, `IMSLS_PRINT_UPPER_NO_DIAG`, and `IMSLS_PRINT_LOWER_NO_DIAG` options. Generally, these options are used with symmetric matrices, but this is not required. Vectors can be printed by specifying a row or column dimension of 1.

Output is written to the file specified by the function `imsls_output_file` (Chapter 15, "Utilities"). The default output file is standard output (corresponding to the file pointer `stdout`). A page width of 78 characters is used. Page width and page length can be reset by invoking function `imsls_page`.

Horizontal centering, the method for printing large matrices, paging, the method for printing NaN (Not a Number), and whether or not a title is printed on each page can be selected by invoking function `imsls_write_options`.

Examples

Example 1

This example is representative of the most common situation in which no optional arguments are given.

```
#include <imsls.h>

#define NRA 3
#define NCA 4

main()
{
    int      i, j;
    float   a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }

    }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, (float*) a, 0);
}
```

Output

```
                   matrix
                      a
            1           2           3           4
1         1.1         1.2         1.3         1.4
2         2.1         2.2         2.3         2.4
3         3.1         3.2         3.3         3.4
```

Example 2

In this example, some of the optional arguments available in the `imsls_write_matrix` functions are demonstrated.

```
#include <imsls.h>

#define NRA     3
#define NCA     4

main()
{
    int         i, j;
    float       a[NRA][NCA];
    char        *fmt = "%10.6W";
    char        *rlabel[] = {"row 1", "row 2", "row 3"};
    char        *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }
    }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, (float *)a,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_ROW_LABELS, rlabel,
        IMSLS_COL_LABELS, clabel,
        IMSLS_PRINT_UPPER_NO_DIAG,
        0);
}
```

Output

```
                     matrix
                        a
             col 2       col 3       col 4
row 1          1.2         1.3         1.4
row 2                      2.3         2.4
row 3                                  3.4
```

Example 3

In this example, a row vector of length four is printed.

```
#include <imsls.h>

#define NRA 1
#define NCA 4

main()
{
    int         i;
    float       a[NCA];
    char        *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0; i < NCA; i++) {
    a[i] = i + 1;
   }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, a,
        IMSLS_COL_LABELS, clabel,
        0);
}
```

Output

```
                  matrix
                     a
     col 1       col 2       col 3       col 4
         1           2           3           4
```

page

Sets or retrieves the page width or length.

Synopsis

#include `<imsls.h>`

void `imsls_page` (*Imsls_page_options* `option`, *int* `*page_attribute`)

Required Arguments

Imsls_page_options `option` (Input)
Option giving which page attribute is to be set or retrieved. The possible values are shown in the table below.

Keyword	Description
`IMSLS_SET_PAGE_WIDTH`	Sets the page width.
`IMSLS_GET_PAGE_WIDTH`	Retrieves the page width.
`IMSLS_SET_PAGE_LENGTH`	Sets the page length.
`IMSLS_GET_PAGE_LENGTH`	Retrieves the page length.

int `*page_attribute` (Input, if the attribute is set; Output, otherwise.)
The value of the page attribute to be set or retrieved. The page width is the number of characters per line of output (default 78), and the page length is the number of lines of output per page (default 60). Ten or more characters per line and 10 or more lines per page are required.

Example

The following example illustrates the use of `imsls_page` to set the page width to 40 characters. Function `imsls_f_write_matrix` is then used to print a 3×4 matrix A, where $a_{ij} = i + j/10$.

```
#include <imsls.h>

#define NRA 3
#define NCA 4
main()
{
    int         i, j, page_attribute;
    float       a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
    page_attribute = 40;
    imsls_page(IMSLS_SET_PAGE_WIDTH, &page_attribute);
    imsls_f_write_matrix("a", NRA, NCA, (float *)a, 0);
}
```

Output

```
                  a
            1           2           3
1         1.1         1.2         1.3
2         2.1         2.2         2.3
3         3.1         3.2         3.3

            4
1         1.4
2         2.4
3         3.4
```

write_options

Sets or retrieves an option for printing a matrix.

Synopsis

#include <imsls.h>

void imsls_write_options (*Imsls_write_options* option, *int* *option_value)

Required Arguments

Imsls_write_options option (Input)
Option giving the type of the printing attribute to set or retrieve.

Keyword for Setting	Keyword for Retrieving	Attribute Description
IMSLS_SET_DEFAULTS		uses the default settings for all parameters
IMSLS_SET_CENTERING	IMSLS_GET_CENTERING	horizontal centering
IMSLS_SET_ROW_WRAP	IMSLS_GET_ROW_WRAP	row wrapping
IMSLS_SET_PAGING	IMSLS_GET_PAGING	paging
IMSLS_SET_NAN_CHAR	IMSLS_GET_NAN_CHAR	method for printing NaN
IMSLS_SET_TITLE_PAGE	IMSLS_GET_TITLE_PAGE	whether or not titles appear on each page
IMSLS_SET_FORMAT	IMSLS_GET_FORMAT	default format for real and complex numbers

int *option_value (Input, if option is to be set; Output, otherwise)
Value of the option attribute selected by option. The values to be used when setting attributes are described in a table in the description section.

Description

Function imsls_write_options allows the user to set or retrieve an option for printing a matrix. Options controlled by imsls_write_options are horizontal centering, method for printing large matrices, paging, method for printing NaN, method for printing titles, and the default format for real and complex numbers. (NaN can be retrieved by functions imsls_f_machine and imsls_d_machine (Chapter 15, “Utilities”).

The following values can be used for the attributes:

Keyword	Value	Meaning
`CENTERING`	0	Matrix is left justified.
	1	Matrix is centered.
`ROW_WRAP`	0	Complete row is printed before the next row is printed. Wrapping is used if necessary.
	m	Here, *m* is a positive integer. Let n_1 be the maximum number of columns that fit across the page, as determined by the widths in the conversion specifications starting with column 1. First, columns 1 through n_1 are printed for rows 1 through *m*. Let n_2 be the maximum number of columns that fit across the page, starting with column n_1+1. Second, columns n_1+1 through n_1+n_2 are printed for rows 1 through *m*. This continues until the last columns are printed for rows 1 through *m*. Printing continues in this fashion for the next *m* rows, etc.
`PAGING`	–2	No paging occurs.
	–1	Paging is on. Every invocation of an function `imsls_write_matrix` begins on a new page, and paging occurs within each invocation as is needed.
	0	Paging is on. The first invocation of an `imsls_f_write_f_matrix` function begins on a new page, and subsequent paging occurs as is needed. Paging occurs in the second and all subsequent calls to an `imsls_f_write_matrix` function only as needed.
	k	Turn paging on and set the number of lines printed on the current page to *k* lines. If *k* is greater than or equal to the page length, then the first invocation of an `imsls_write_matrix` function begins on a new page. In any case, subsequent paging occurs as is needed.
`NAN_CHAR`	0	. . . . is printed for NaN.
	1	A blank field is printed for NaN.
`TITLE_PAGE`	0	Title appears only on first page.
	1	Title appears on the first page and all continuation pages.
`FORMAT`	0	Format is "`%10.4x`".
	1	Format is "`%12.6w`".
	2	Format is "`%22.5e`".

The w conversion character used by the FORMAT option is a special conversion character that can be used to automatically select a pretty C conversion specification ending in either e, f, or d. The conversion specification ending with w is specified as "%n.dw". Here, n is the field width, and d is the number of significant digits generally printed.

Function imsls_write_options can be invoked repeatedly before using a function imsls_f_write_matrix to print a matrix. The matrix printing functions retrieve the values set by imsls_write_options to determine the printing options. It is not necessary to call imsls_write_options if a default value of a printing option is desired. The defaults are as follows:

Keyword	Default Value	Meaning
CENTERING	0	left justified
ROW_WRAP	1000	lines before wrapping
PAGING	–2	no paging
NAN_CHAR	0	. . . .
TITLE_PAGE	0	title appears only on the first page
FORMAT	0	%10.4w

Example

The following example illustrates the effect of imsls_write_options when printing a 3 × 4 real matrix A with function imsls_f_write_matrix, where $a_{ij} = i + j/10$. The first call to imsls_write_options sets horizontal centering so that the matrix is printed centered horizontally on the page. In the next invocation of imsls_f_write_matrix, the left-justification option has been set by function imsls_write_options so the matrix is left justified when printed.

```
#include <imsls.h>

#define NRA 4
#define NCA 3

main()
{
    int         i, j, option_value;
    float       a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
                                /* Activate centering option */
    option_value = 1;
    imsls_write_options (IMSLS_SET_CENTERING, &option_value);
                                /* Write a matrix */
    imsls_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
                                /* Activate left justification */
    option_value = 0;
    imsls_write_options (IMSLS_SET_CENTERING, &option_value);
    imsls_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
}
```

Output

```
                                          a
                                1            2            3
                   1          1.1          1.2          1.3
                   2          2.1          2.2          2.3
                   3          3.1          3.2          3.3
                   4          4.1          4.2          4.3

                      a
            1            2            3
1         1.1          1.2          1.3
2         2.1          2.2          2.3
3         3.1          3.2          3.3
4         4.1          4.2          4.3
```

Chapter 15: Utilities

Routines

Set Output Files

Set output files	output_file	993
Get library version and license number	version	997

Error Handling

Error message options	error_options	998
Get error code	error_code	1004

Constants

Integer machine constants	machine (integer)	1005
Float machine constants	machine (float)	1007
Common data sets	data_sets	1009

Mathematical Support

Matrix-vector, matrix-matrix, vector-vector products	mat_mul_rect	1012
Rearrange elements of vector	permute_vector	1015
Interchange rows and columns of matrices	permute_matrix	1017
Evaluate the binomial coeficient	binomial_coefficient	1018
Evaluate the complete beta function	beta	1020
Evaluate the real incomplete beta function	beta_incomplete	1021
Evaluate the log of the real beta function	log_beta	1022
Evaluate the real gamma function	gamma	1023
Evaluate the incomplete gamma function	gamma_incomplete	1025
Evaluate the logarithm of the absolute value of the gamma function	log_gamma	1027
Return the number of CPU seconds used	ctime	1029

output_file

Sets the output file or the error message output file.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_output_file` (
    `IMSLS_SET_OUTPUT_FILE`, *FILE* `*ofile`,
    `IMSLS_GET_OUTPUT_FILE`, *FILE* `**pofile`,
    `IMSLS_SET_ERROR_FILE`, *FILE* `*efile`,
    `IMSLS_GET_ERROR_FILE`, *FILE* `**pefile`,
    0)

Optional Arguments

`IMSLS_SET_OUTPUT_FILE`, *FILE* `*ofile` (Input)
    Sets the output file to `ofile`.
    Default: `ofile = stdout`

`IMSLS_GET_OUTPUT_FILE`, *FILE* `**pofile` (Output)
    Sets the *FILE* pointed to by `pofile` to the current output file.

`IMSLS_SET_ERROR_FILE`, *FILE* `*efile` (Input)
    Sets the error message output file to `efile`.
    Default: `efile = stderr`

`IMSLS_GET_ERROR_FILE`, *FILE* `**pefile` (Output)
    Sets the *FILE* pointed to by `pefile` to the error message output file.

Description

This function allows the file used for printing by IMSL functions to be changed.

If multiple threads are used then default settings are valid for each thread. When using threads it is possible to set different output files for each thread by calling `imsls_output_file` from within each thread. See Example 2 for more details.

Example 1

This example opens the file *myfile* and sets the output file to this new file. Function `imsls_f_write_matrix` then writes to this file.

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

main()
{
    FILE          *ofile;
    float        x[] = {3.0, 2.0, 1.0};

    imsls_f_write_matrix ("x (default file)", 1, 3, x, 0);

    ofile = fopen("myfile", "w");
    imsls_output_file(IMSLS_SET_OUTPUT_FILE, ofile,
                     0);
    imsls_f_write_matrix ("x (myfile)", 1, 3, x, 0);
}
```

Output

```
x (default file)
1           2           3
3           2           1
```

File myfile

```
x (myfile)
1           2           3
3           2           1
```

Example 2

The following example illustrates how to direct output from IMSL routines that run in separate threads to different files. First, two threads are created, each calling a different IMSL function, then the results are printed by calling `imsls_f_write_matrix` from within each thread. Note that `imsls_output_file` is called from within each thread to change the default output file.

```
#include <pthread.h>
#include <stdio.h>
#include "imsls.h"
void *ex1(void* arg);
void *ex2(void* arg);
void main()
{
  pthread_t       thread1;
  pthread_t       thread2;

  /* Disable IMSL signal trapping. */
  imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);

  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}
void *ex1(void* arg)
{
  float *rand_nums = NULL;
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex1.out", "w");
  /* Set the output file for this thread. */
  imsls_output_file(IMSLS_SET_OUTPUT_FILE, file_ptr, 0);
  /* Compute 5 random numbers. */
  imsls_random_seed_set(12345);
  rand_nums = imsls_f_random_uniform(5, 0);
  /* Output random numbers. */
  imsls_f_write_matrix("Random Numbers", 5, 1, rand_nums, 0);
  if (rand_nums) free(rand_nums);
  fclose(file_ptr);
}
void *ex2(void* arg)
{
  int n_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};
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex2.out", "w");
  /* Set the output file for this thread. */
  imsls_output_file(IMSLS_SET_OUTPUT_FILE, file_ptr, 0);
  table = imsls_f_table_oneway (n_observations, x, n_intervals, 0);
  imsls_f_write_matrix("counts", 1, n_intervals, table, 0);

  if (table) free(table);
  fclose(file_ptr);
}
```

ex1.out

```
Random Numbers
 1      0.4919
 2      0.3909
 3      0.2645
 4      0.1814
 5      0.7546
```

ex2.out

```
                               counts
         1            2            3            4            5            6
         4            8            5            5            3            1

         7            8            9           10
         3            0            0            1
```

version

Returns information describing the version of the library, serial number, operating system, and compiler.

Synopsis

#include `<imsls.h>`

char `*imsls_version` (*Imsls_keyword* `code`)

Required Arguments

Imsls_keyword `code` (Input)
Index indicating which value is to be returned. It must be `IMSLS_LIBRARY_VERSION`, `IMSLS_OS_VERSION`, `IMSLS_COMPILER_VERSION`, or `IMSLS_LICENSE_NUMBER`.

Return Value

The requested value is returned. If `code` is out of range, then `NULL` is returned. Use `free` to release the returned string.

Description

Function `imsls_version` returns information describing the version of the library, the version of the operating system under which it was compiled, the compiler used, and the IMSL serial number.

Example

This example prints all the values returned by `imsls_version` on a particular machine. The output is omitted because the results are system dependent.

```
#include <imsls.h>

main()
{
    char    *library_version, *os_version;
    char    *compiler_version, *license_number;

    library_version  = imsls_version(IMSLS_LIBRARY_VERSION);
    os_version       = imsls_version(IMSLS_OS_VERSION);
    compiler_version = imsls_version(IMSLS_COMPILER_VERSION);
    license_number   = imsls_version(IMSLS_LICENSE_NUMBER);

    printf("Library version = %s\n", library_version);
    printf("OS version = %s\n", os_version);
    printf("Compiler version = %s\n", compiler_version);
    printf("Serial number = %s\n", license_number);
}
```

error_options

Sets various error handling options.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_error_options` (
`IMSLS_SET_PRINT`, *Imsls_error* `type`, *int* `setting`,
`IMSLS_SET_STOP`, *Imsls_error* `type`, *int* `setting`,
`IMSLS_SET_TRACEBACK`, *Imsls_error* `type`, *int* `setting`,
`IMSLS_FULL_TRACEBACK`, *int* `setting`,
`IMSLS_GET_PRINT`, *Imsls_error* type, *int* `*psetting`,
`IMSLS_GET_STOP`, *Imsls_error* `type`, *int* `*psetting`,
`IMSLS_GET_TRACEBACK`, *Imsls_error* `type`, *int* `*psetting`,
`IMSLS_SET_ERROR_FILE`, *FILE* `*file`,
`IMSLS_GET_ERROR_FILE`, *FILE* `**pfile`,
`IMSLS_ERROR_MSG_PATH`, *char* `*path`,
`IMSLS_ERROR_MSG_NAME`, *char* `*name`,
`IMSLS_ERROR_PRINT_PROC`, *Imsls_error_print_proc* `print_proc`,
`IMSLS_SET_SIGNAL_TRAPPING`, *int* `setting`,
0)

Optional Arguments

`IMSLS_SET_PRINT`, *Imsls_error* type, *int* `setting` (Input)
Printing of type `type` error messages is turned off if `setting` is 0; otherwise, printing is turned on.
Default: Printing turned on for `IMSLS_WARNING`, `IMSLS_FATAL`, `IMSLS_TERMINAL`, `IMSLS_FATAL_IMMEDIATE`, `and` `IMSLS_WARNING_IMMEDIATE` messages

`IMSLS_SET_STOP`, *Imsls_error* `type`, *int* `setting` (Input)
Stopping on type `type` error messages is turned off if `setting` is 0; otherwise, stopping is turned on.
Default: Stopping turned on for `IMSLS_FATAL` and `IMSLS_TERMINAL` and `IMSLS_FATAL_IMMEDIATE` messages

`IMSLS_SET_TRACEBACK`, *Imsls_error* `type`, *int* `setting` (Input)
Printing of a traceback on type `type` error messages is turned off if `setting` is 0; otherwise, printing of the traceback turned on.
Default: Traceback turned off for all message types

`IMSLS_FULL_TRACEBACK`, *int* `setting` (Input)
Only documented functions are listed in the traceback if `setting` is 0; otherwise, internal function names also are listed.
Default: Full traceback turned off

`IMSLS_GET_PRINT`, *Imsls_error* `type`, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for printing of type type error messages.

`IMSLS_GET_STOP`, *Imsls_error* `type`, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for stopping on type type error messages.

`IMSLS_GET_TRACEBACK`, *Imsls_error* type, *int* `*psetting` (Output)
Sets the integer pointed to by `psetting` to the current setting for printing of a traceback for type `type` error messages.

`IMSLS_SET_ERROR_FILE`, *FILE* `*file` (Input)
Sets the error output file.
Default: `file` = `stderr`

`IMSLS_GET_ERROR_FILE`, *FILE* `**pfile` (Output)
Sets the *FILE* `*` pointed to by `pfile` to the error output file.

`IMSLS_ERROR_MSG_PATH`, *char* `*path` (Input)
Sets the error message file path. On UNIX systems, this is a colon-separated list of directories to be searched for the file containing the error messages.
Default: system dependent

`IMSLS_ERROR_MSG_NAME`, *char* `*name` (Input)
Sets the name of the file containing the error messages.
Default: `file` = `"imsls_e.bin"`

`IMSLS_ERROR_PRINT_PROC`, *Imsls_error_print_proc* `print_proc` (Input)
Sets the error printing function. The procedure `print_proc` has the form
void `print_proc` (*Imsls_error* `type`, *long* `code`,
char `*function_name`, *char* `*message`).

In this case, `type` is the error message type number (`IMSLS_FATAL`, etc.), code is the error message code number (`IMSLS_MAJOR_VIOLATION`, etc.), `function_name` is the name of the function setting the error, and message is the error message to be printed. If `print_proc` is `NULL`, then the default error printing function is used.

`IMSLS_SET_SIGNAL_TRAPPING`, *int* `setting` (Input)
C/Stat/Library will use its own signal handler if setting is 1; otherwise the C/Stat/Library signal handler is not used. If C/Stat/Library is called from a multi-threaded application, signal handling by C/Stat/Library must be turned off. See Example 3 for details.
Default: setting = 1

Return Value

The return value is void.

Description

This function allows the error handling system to be customized.

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options (excluding `IMSLS_SET_SIGNAL_TRAPPING`) for each thread by calling `imsls_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);
```

See Example 3 and Example 4 for multithreaded examples.

NOTE: Signal handlers are installed when a C/Stat/Library function is called, then uninstalled prior to returning from the C/Stat/Library function. The library function `imsls_error_options` can be used to perform many different tasks with regard to error handling and it will install signal handlers when first called, even if the call is being made to disable signal handling through the use of the optional argument `IMSLS_SET_SIGNAL_TRAPPING`. However, there may be cases when it is desirable to completely avoid any installation of signal handlers by C/Stat/Library functions. In these cases, the following function can be called.

#include <`imsls.h`>

void `imsls_skip_signal_handler( );`

Examples

Example 1

In this example, the `IMSLS_TERMINAL` print setting is retrieved. Next, stopping on `IMSLS_TERMINAL` errors is turned off, output to standard output is redirected, and an error is deliberately caused by calling `imsls_error_options` with an illegal value.

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

main()
{
    int         setting;
                            /* Turn off stopping on IMSLS_TERMINAL */
                            /* error messages and write error */
                            /* messages to standard output */
    imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                       IMSLS_SET_ERROR_FILE, stdout,
                       0);
                            /* Call imsls_error_options() with */
                            /* an illegal value */
    imsls_error_options(-1);
                          /* Get setting for IMSLS_TERMINAL */
    imsls_error_options(IMSLS_GET_PRINT, IMSLS_TERMINAL, &setting,
                       0);
    printf("IMSLS_TERMINAL error print setting = %d\n", setting);
}
```

Output

```
*** TERMINAL Error from imsls_error_options.  There is an error with
*** argument number 1.  This may be caused by an incorrect number of
*** values following a previous optional argument name.

IMSLS_TERMINAL error print setting = 1
```

Example 2

In this example, IMSL's error printing function has been substituted for the standard function. Only the first four lines are printed below.

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

void        print_proc(Imsls_error, long, char*, char*);

main()
{
                           /* Turn off tracebacks on IMSLS_TERMINAL */
                           /* error messages and use a custom */
                           /* print function */
    imsls_error_options(IMSLS_ERROR_PRINT_PROC, print_proc,
                       0);
                           /* Call imsls_error_options() with an */
                           /* illegal value */
    imsls_error_options(-1);
}

void print_proc(Imsls_error type, long code, char *function_name,
                char *message)
{
    printf("Error message type %d\n", type);
    printf("Error code %d\n", code);
    printf("From function %s\n", function_name);
    printf("%s\n", message);
}
```

Output

```
Error message type 5
Error code 103
From function imsls_error_options
There is an error with argument number 1.  This may be caused by an
incorrect number of values following a previous optional argument name.
```

Example 3

In this example, two threads are created and error options is called within each thread to set the error handling options slightly different for each thread. Since we expect to

generate terminal errors in each thread, we must turn off stopping on terminal errors for each thread. Also notice that `imsls_error_options` is called from `main` to disable the IMSL signal-trapping mechanism.
See Example 4 for a similar example, using WIN32 threads. Note since multiple threads are executing, the order of the errors output may differ on some systems.

```
#include <pthread.h>
#include <stdio.h>
#include "imsls.h"

void *ex1(void* arg);
void *ex2(void* arg);
void main()
{
  pthread_t       thread1;
  pthread_t       thread2;

  /* Disable IMSL signal trapping. */
  imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);

  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}

void *ex1(void* arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0, 0);
  res = imsls_f_beta(-1.0, .5);
}
void *ex2(void* arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.  Notice that tracebacks are
   * turned on for IMSLS_TERMINAL errors.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                      IMSLS_SET_TRACEBACK, IMSLS_TERMINAL, 1, 0);
  res = imsls_f_gamma(-1.0);
}
```

Output

```
*** TERMINAL Error from imsls_f_beta.  Both "x" = -1.000000e+00 and "y" =
***          5.000000e-01 must be greater than zero.


*** TERMINAL Error from imsls_f_gamma.  The argument for the function can
***          not be a negative integer. Argument "x" = -1.000000e+00.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code                Routine
  ----------        ----------                -------
 IMSLS_TERMINAL    IMSLS_NEGATIVE_INTEGER    imsls_f_gamma
```

Example 4

In this example the WIN32 API is used to demonstrate the same functionality as shown in Example 3 above. Note since multiple threads are executing, the order of the errors output may differ on some systems.

```
#include <windows.h>
#include <stdio.h>
#include "imsls.h"

DWORD WINAPI ex1(void *arg);
DWORD WINAPI ex2(void *arg);

int main(int argc, char* argv[])
{
       HANDLE thread[2];

       imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);

       thread[0] = CreateThread(NULL, 0, ex1, NULL, 0, NULL);
       thread[1] = CreateThread(NULL, 0, ex2, NULL, 0, NULL);

       WaitForMultipleObjects(2, thread, TRUE, INFINITE);

}
DWORD WINAPI ex1(void *arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.
   */
imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                   0);
  res = imsls_f_beta(-1.0, .5);
       return(0);
}
```

```
DWORD WINAPI ex2(void *arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.  Notice that tracebacks are
   * turned on for IMSLS_TERMINAL errors.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                      IMSLS_SET_TRACEBACK, IMSLS_TERMINAL, 1,
                      0);
  res = imsls_f_gamma(-1.0);
  return(0);
}
```

Output

```
*** TERMINAL Error from imsls_f_beta.  Both "x" = -1.000000e+000 and "y" =
***          5.000000e-001 must be greater than zero.


*** TERMINAL Error from imsls_f_gamma.  The argument for the function can
***          not be a negative integer. Argument "x" = -1.000000e+000.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code              Routine
  ----------        ----------              -------
 IMSLS_TERMINAL    IMSLS_NEGATIVE_INTEGER    imsls_f_gamma USER
```

error_code

Gets the code corresponding to the error message from the last function called.

Synopsis

#include `<imsls.h>`

long `imsls_error_code` ( )

Return Value

This function returns the error message code from the last function called. The include file *imsls.h* defines a name for each error code.

Example

In this example, stopping on `IMSLS_TERMINAL` error messages is turned off and an error is then generated by calling function `imsls_error_options` with an illegal value for `IMSLS_SET_PRINT`. The error message code number is then retrieved and printed. In *imsls.h*, `IMSLS_INTEGER_OUT_OF_RANGE` is defined to be 132.

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

```
main()
{
    long        code;
                                /* Turn off stopping IMSLS_TERMINAL */
                                /* messages and print error messages */
                                /* on standard output */
    imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                        IMSLS_SET_ERROR_FILE, stdout,
                        0);
                                /* Call imsls_error_options() with */
                                /* an illegal value */
    imsls_error_options(IMSLS_SET_PRINT, 100, 0,
                        0);
                                /* Get the error message code */
    code = imsls_error_code();
    printf("error code = %d\n", code);
}
```

Output

```
*** TERMINAL error from imsls_error_options.  "type" must be between 1 and
***          5, but "type" = 100.

error code = 132
```

machine (integer)

Returns integer information describing the computer's arithmetic.

Synopsis

#include `<imsls.h>`

int `imsls_i_machine` (*int* `n`)

Required Arguments

int `n` (Input)
Index indicating which value is to be returned. It must be between 0 and 12.

Return Value

The requested value is returned. If `n` is out of range, NaN is returned.

Description

Function `imsls_i_machine` returns information describing the computer's arithmetic. This can be used to make programs machine independent.

`imsls_i_machine`(0) = Number of bits per byte

Assume that integers are represented in *M*-digit, base-*A* form as

$$\sigma \sum_{k=0}^{M} x_k \; A^k$$

where σ is the sign and $0 \le x_k < A$ for $k = 0, \ldots, M$. Then,

N	Definition
0	C, bits per character
1	A, the base
2	M_s, the number of base-A digits in a *short int*
3	$A^{M_s} - 1$, the largest *short int*
4	M_l, the number of base-A digits in a *long int*
5	$A^{M_l} - 1$, the largest *long int*

Assume that floating-point numbers are represented in N-digit, base B form as

$$\sigma B^E \sum_{k-1}^{N} x_k B^{-k}$$

where σ is the sign and $0 \le x_k < B$ for $k = 1, \ldots, N$ and $E_{\min} \le E \le E_{\max}$. Then

N	Definition
6	B, the base
7	N_f, the number of base-B digits in *float*
8	$E_{\min_f}$, the smallest *float* exponent
9	$E_{\max_f}$, the largest *float* exponent
10	N_d, the number of base-B digits in *double*
11	$E_{\max_f}$, the largest long *int*
12	$E_{\max_d}$, the number of base-B digits in *double*

Example

In this example, all the values returned by `imsls_i_machine` on a machine with IEEE (Institute for Electrical and Electronics Engineer) arithmetic are printed.

```
#include <imsls.h>

main()
{
    int         n, ans;

    for (n = 0;  n <= 12;  n++) {
        ans = imsls_i_machine(n);
        printf("imsls_i_machine(%d) = %d\n", n, ans);
    }
}
```

Output

```
imsls_i_machine(0) = 8
imsls_i_machine(1) = 2
imsls_i_machine(2) = 15
imsls_i_machine(3) = 32767
imsls_i_machine(4) = 31
imsls_i_machine(5) = 2147483647
imsls_i_machine(6) = 2
imsls_i_machine(7) = 24
imsls_i_machine(8) = -125
imsls_i_machine(9) = 128
imsls_i_machine(10) = 53
imsls_i_machine(11) = -1021
imsls_i_machine(12) = 1024
```

machine (float)

Returns information describing the computer's floating-point arithmetic.

Synopsis

#include `<imsls.h>`

float `imsls_f_machine` (*int* `n`)

The type *double* function is `imsls_d_machine`.

Required Arguments

int `n` (Input)
Index indicating which value is to be returned. The index must be between 1 and 8.

Return Value

The requested value is returned. If `n` is out of range, NaN is returned.

Description

Function `imsls_f_machine` returns information describing the computer's floating-point arithmetic. This can be used to make programs machine independent. In addition, some of the functions are also important in setting missing values.

Assume that *float* numbers are represented in N_f-digit, base B form as

$$\sigma B^E \sum_{k=1}^{N_f} x_k B^{-k}$$

where σ is the sign; $0 \le x_k < B$ for $k = 1, 2, \ldots, N_f$; and

$$E_{\min_f} \le E \le E_{\max_f}$$

Note that B = `imsls_i_machine`(6); N_f = `imsls_i_machine`(7);

$$E_{\min_f} = \texttt{imsls\_i\_machine}(8)$$

and

$$E_{\max_f} = \texttt{imsls\_i\_machine}(9)$$

The ANSI/IEEE 754-1985 standard for binary arithmetic uses NaN as the result of various otherwise illegal operations, such as computing 0/0. On computers that do not support NaN, a value larger than `imsls_d_machine`(2) is returned for `imsls_f_machine`(6). On computers that do not have a special representation for infinity, `imsls_f_machine`(2) returns the same value as `imsls_f_machine`(7).

Function `imsls_f_machine` is defined by the following table:

N	Definition
1	$B^{E_{\min_f}-1}$, the smallest positive number
2	$B^{E_{\max_f}}(1-B^{-N_f})$, the largest number
3	B^{-N_f}, the smallest relative spacing
4	B^{1-N_f}, the largest relative spacing
5	$\log_{10}(B)$
6	NaN
7	positive machine infinity
8	negative machine infinity

Function `imsls_d_machine` retrieves machine constants that define the computer's double arithmetic. Note that for *double B* = `imsls_i_machine`(6), N_d = `imsls_i_machine`(10),

$$E_{\min_d} = \texttt{imsls\_i\_machine}(11)$$

and

$$E_{\max_d} = \texttt{imsls\_i\_machine}(12)$$

Missing values in functions are always indicated by NaN. This is `imsls_f_machine`(6) in single precision and `imsls_d_machine`(6) in double precision. There is no missing-value indicator for integers. Users will almost always have to convert from their missing value indicators to NaN.

Example

In this example, all eight values returned by `imsls_f_machine` and by `imsls_d_machine` on a machine with IEEE arithmetic are printed.

```
#include <imsls.h>

main()
{
    int             n;
    float           fans;
    double          dans;

    for (n = 1;  n <= 8;  n++) {
        fans = imsls_f_machine(n);
        printf("imsls_f_machine(%d) = %g\n", n, fans);
    }

    for (n = 1;  n <= 8;  n++) {
        dans = imsls_d_machine(n);
        printf("imsls_d_machine(%d) = %g\n", n, dans);
    }
}
```

Output

```
imsls_f_machine(1) = 1.17549e-38
imsls_f_machine(2) = 3.40282e+38
imsls_f_machine(3) = 5.96046e-08
imsls_f_machine(4) = 1.19209e-07
imsls_f_machine(5) = 0.30103
imsls_f_machine(6) = NaN
imsls_f_machine(7) = Inf
imsls_f_machine(8) = -Inf
imsls_d_machine(1) = 2.22507e-308
imsls_d_machine(2) = 1.79769e+308
imsls_d_machine(3) = 1.11022e-16
imsls_d_machine(4) = 2.22045e-16
imsls_d_machine(5) = 0.30103
imsls_d_machine(6) = NaN
imsls_d_machine(7) = Inf
imsls_d_machine(8) = -Inf
```

data_sets

Retrieves a commonly analyzed data set.

Synopsis

#include `<imsls.h>`

float `*imsls_f_data_sets` (*int* `data_set_choice`, ..., 0)

The type *double* function is `imsls_d_data_sets`.

Required Arguments

int `data_set_choice` (Input)
Data set indicator. Set `data_set_choice` = 0 to print a description of all nine data sets. In this case, any optional arguments are ignored.

data_set_choice	N_observations	n_variables	Description of Data Set
1	16	7	Longley
2	176	2	Wolfer sunspot
3	150	5	Fisher iris
4	144	1	Box and Jenkins Series G
5	13	5	Draper and Smith Appendix B
6	197	1	Box and Jenkins Series A
7	296	2	Box and Jenkins Series J
8	100	4	Robinson Multichannel Time Series
9	113	34	Afifi and Azen Data Set A

Return Value

If `data_set_choice` ≠ 0, the requested data set is returned. If `data_set_choice` = 0 or an error occurs, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_data_sets` (*int* `data_set_choice`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_N_OBSERVATIONS`, *int* `*n_observations`,
`IMSLS_N_VARIABLES`, *int* `*n_variables`,
`IMSLS_PRINT_NONE`,
`IMSLS_PRINT_BRIEF`,
`IMSLS_PRINT_ALL`,
`IMSLS_RETURN_USER`, *float* `x[]`,
0)

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of user allocated space.

`IMSLS_N_OBSERVATIONS`, *int* `*n_observations` (Output)
Number of observations or rows in the output matrix.

`IMSLS_N_VARIABLES`, *int* `*n_variables` (Output)
Number of variables or columns in the output matrix.

`IMSLS_PRINT_NONE`
No printing is performed. This option is the default.

`IMSLS_PRINT_BRIEF`
Rows 1 through 10 of the data set are printed.

`IMSLS_PRINT_ALL`
All rows of the data set are printed.

`IMSLS_RETURN_USER`, *float* `x[]` (Output)
User-supplied array containing the data set.

Description

Function `imsls_f_data_sets` retrieves a standard data set frequently cited in statistics text books or in this manual. The following tables gives the references for each data set:

Data_set_choice	Reference
1	Longley (1967)
2	Anderson (1971, p.660)
3	Fisher (1936); Mardia et al. (1979, Table 1.2.2)
4	Box and Jenkins (1976, p. 531)
5	Draper and Smith (1981, pp. 629-630)
6	Box and Jenkins (1976, p. 525)
7	Box and Jenkins (1976, pp. 532-533)
8	Robinson (1976, p. 204)
9	Afifi and Azen (1979, pp. 16-22)

Example

In this example, `imsls_f_data_sets` is used to copy the Draper and Smith (1981, Appendix B) data set into `x`.

```
#include <imsls.h>

main()
{
    float *x;

    x = imsls_f_data_sets (5, 0);

    imsls_f_write_matrix("Draper and Smith, Appendix B", 13, 5, x, 0);
}
```

Output

```
                Draper and Smith, Appendix B
            1           2           3           4           5
 1        7.0        26.0         6.0        60.0        78.5
 2        1.0        29.0        15.0        52.0        74.3
 3       11.0        56.0         8.0        20.0       104.3
 4       11.0        31.0         8.0        47.0        87.6
 5        7.0        52.0         6.0        33.0        95.9
 6       11.0        55.0         9.0        22.0       109.2
 7        3.0        71.0        17.0         6.0       102.7
 8         1.0        31.0        22.0        44.0        72.5
 9         2.0        54.0        18.0        22.0        93.1
10        21.0        47.0         4.0        26.0       115.9
11         1.0        40.0        23.0        34.0        83.8
12        11.0        66.0         9.0        12.0       113.3
13        10.0        68.0         8.0        12.0       109.4
```

mat_mul_rect

Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any triple product.

Synopsis

#include `<imsls.h>`

float `*imsls_f_mat_mul_rect` (*char* `*string`, ..., 0)

The type *double* function is `imsls_d_mat_mul_rect`.

Required Arguments

char `*string` (Input)
String indicating operation to be performed. See the "Description" section below for more details."

Return Value

The result of the operation. This is always a pointer to a *float*, even if the result is a single number. If no answer was computed, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_mat_mul_rect` (*char* `*string`,
`IMSLS_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *float* `a[]`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,
`IMSLS_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *float* `b[]`,
`IMSLS_B_COL_DIM`, *int* `b_col_dim`,
`IMSLS_X_VECTOR`, *int* `nx`, *float* `*x`,
`IMSLS_Y_VECTOR`, *int* `ny`, *float* `*y`,
`IMSLS_RETURN_USER`, *float* `ans[]`,
`IMSLS_RETURN_COL_DIM`, *int* `return_col_dim`,
0)

Optional Arguments

`IMSLS_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *float* `a[]` (Input)
The `nrowa` × `ncola` matrix *A*.

`IMSLS_A_COL_DIM`, *int* `a_col_dim` (Input)
Column dimension of *A*.
Default: `a_col_dim` = `ncola`

`IMSLS_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *float* `b[]` (Input)
The `nrowb` × `ncolb` matrix *A*.

`IMSLS_B_COL_DIM`, *int* `b_col_dim` (Input)
Column dimension of *B*.
Default: `b_col_dim = ncolb`

`IMSLS_X_VECTOR`, *int* `nx`, *float* `*x` (Input)
Vector *x* of size `nx`.

`IMSLS_Y_VECTOR`, *int* `ny`, *float* `*y` (Input)
Vector *y* of size `ny`.

`IMSLS_RETURN_USER`, *float* `ans[]` (Output)
User-allocated array containing the result.

`IMSLS_RETURN_COL_DIM`, *int* `return_col_dim` (Input)
Column dimension of the answer.
Default: `return_col_dim` = the number of columns in the answer

Description

This function computes a matrix-vector product, a matrix-matrix product, a bilinear form of a matrix, or a triple product according to the specification given by `string`. For example, if "`A*x`" is given, *Ax* is computed. In `string`, the matrices *A* and *B* and the vectors *x* and *y* can be used. Any of these four names can be used with `trans`, indicating transpose. The vectors *x* and *y* are treated as $n \times 1$ matrices.

If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose, is returned. If `string` contains one multiplication, such as "`A*x`" or "`B*A`", then the indicated product is returned. Some other legal values for `string` are "`trans(y)*A`", "`A*trans(B)`", "`x*trans(y)`", or "`trans(x)*y`".

The matrices and/or vectors referred to in `string` must be given as optional arguments. If `string` is "`B*x`", then `IMSLS_B_MATRIX` and `IMSLS_X_VECTOR` must be given.

Example

Let *A*, *B*, *x*, and *y* equal the following matrices:

$$A = \begin{bmatrix} 1 & 2 & 9 \\ 5 & 4 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 7 & 4 \\ 9 & 1 \end{bmatrix} \quad x = \begin{bmatrix} 7 \\ 2 \\ 1 \end{bmatrix} \quad y = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

The arrays A^T, Ax, x^TA^T, AB, B^TA^T, x^Ty, xy^T and x^TAy are computed and printed.

```
#include <imsls.h>

main()
{
    float       A[] = {1, 2, 9,
                       5, 4, 7};
    float       B[] = {3, 2,
                       7, 4,
                       9, 1};
    float       x[] = {7, 2, 1};
    float       y[] = {3, 4, 2};
    float       *ans;

    ans = imsls_f_mat_mul_rect("trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        0);
    imsls_f_write_matrix("trans(A)", 3, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("A*x",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_X_VECTOR, 3, x,
        0);
    imsls_f_write_matrix("A*x", 1, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_X_VECTOR, 3, x,
        0);
    imsls_f_write_matrix("trans(x)*trans(A)", 1, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("A*B",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_B_MATRIX, 3, 2, B,
        0);
    imsls_f_write_matrix("A*B", 2, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(B)*trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_B_MATRIX, 3, 2, B,
        0);
    imsls_f_write_matrix("trans(B)*trans(A)", 2, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*y",
        IMSLS_X_VECTOR, 3, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("trans(x)*y", 1, 1, ans, 0);

    ans = imsls_f_mat_mul_rect("x*trans(y)",
        IMSLS_X_VECTOR, 3, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("x*trans(y)", 3, 3, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*A*y",
        IMSLS_A_MATRIX, 2, 3, A,
                                /* use only the first 2 components of x */
        IMSLS_X_VECTOR, 2, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("trans(x)*A*y", 1, 1, ans, 0);
}
```

Output

```
        trans(A)
           1           2
1          1           5
2          2           4
3          9           7

          A*x
        1           2
       20          50

   trans(x)*trans(A)
        1           2
       20          50

           A*B
           1           2
1         98          19
2        106          33

    trans(B)*trans(A)
           1           2
1         98         106
2         19          33

trans(x)*y
        31

             x*trans(y)
           1           2           3
1         21          28          14
2          6           8           4
3          3           4           2

trans(x)*A*y
        293
```

permute_vector

Rearranges the elements of a vector as specified by a permutation.

Synopsis

#include `<imsls.h>`

float `*imsls_f_permute_vector` (*int* `n_elements`, *float* `x[]`,
int `permutation[]`, *Imsls_permute* `permute`, ..., 0)

The type *double* function is `imsls_d_permute_vector`.

Required Arguments

int `n_elements` (Input)
Number of elements in the input vector `x`.

float `x[]` (Input)
Array of length `n_elements` to be permuted.

int `permutation[]` (Input)
Array of length `n_elements` containing the permutation.

Imsls_permute `permute` (Input)
Keyword of type *Imsls_permute*. Argument `permute` must be either `IMSLS_FORWARD_PERMUTATION` or `IMSLS_BACKWARD_PERMUTATION`. If `IMSLS_FORWARD_PERMUTATION` is specified, then a forward permutation is performed, i.e., `x(permutation[i])` is moved to location *i* in the return vector. If `IMSLS_BACKWARD_PERMUTATION` is specified, then a backward permutation is performed, i.e., `x[i]` is moved to location `permutation[i]` in the return vector.

Return Value

An array of length `n_elements` containing the input vector `x` permuted.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_permute_vector` (*int* `n_elements`, *float* `x[]`,
int `permutation[]`, *Imsls_permute* `permute`,
`IMSLS_RETURN_USER`, *float* `permuted_result[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `permuted_result[]` (Output)
User-allocated array containing the result of the permutation.

Description

Function `imsls_f_permute_vector` rearranges the elements of a vector according to a permutation vector. The function can perform both forward and backward permutation.

Example

This example rearranges the vector `x` using `permutation`. A forward permutation is performed.

```
#include <imsls.h>

void main()
{
    float x[] = {5.0, 6.0, 1.0, 4.0};
    int permutation[] = {2, 0, 3, 1};
    float     *output;
    int        n_elements = 4;

    output = imsls_f_permute_vector (n_elements, x, permutation,
        IMSLS_FORWARD_PERMUTATION, 0);

    imsls_f_write_matrix ("permuted result", 1, n_elements, output,
                           IMSLS_COL_NUMBER_ZERO, 0);
}
```

Output

```
          permuted result
     0          1          2          3
     1          5          4          6
```

permute_matrix

Permutes the rows or columns of a matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_permute_matrix` (*int* `n_rows`, *int* `n_columns`, *float* `a[]`,
    int `permutation[]`, *Imsls_permute* `permute`, ..., 0)

The type *double* function is `imsls_d_permute_matrix`.

Required Arguments

int `n_rows` (Input)
    Number of rows in the input matrix `a`.

int `n_columns` (Input)
    Number of columns in the input matrix `a`.

float `a[]` (Input)
    Matrix of size `n_rows` × `n_columns` to be permuted.

int `permutation[]` (Input)
    Array of length `n_elements` containing the permutation.

Imsls_permute `permute` (Input)
    Keyword of type *Imsls_permute*. Argument `permute` must be either `IMSLS_PERMUTE_ROWS`, if the rows of `a` are to be interchanged, or `IMSLS_PERMUTE_COLUMNS`, if the columns of `a` are to be interchanged.

Return Value

Array of size `n_rows` × `n_columns` containing the permuted input matrix `a`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_permute_matrix` (*int* `n_rows`, *int* `n_columns`, *float* `a[]`,
    int `permutation[]`, *Imsls_permute* `permute`,
    `IMSLS_RETURN_USER`, *float* `permuted_result[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `permuted_result[]` (Output)
User-allocated array of size `n_rows` × `n_columns` containing the result of the permutation.

Description

Function `imsls_f_permute_matrix` interchanges the rows or columns of a matrix using a permutation vector. The function permutes a column (row) at a time using function `imsls_f_permute_vector`. This process is continued until all the columns (rows) are permuted. On completion, let B = result and p_i = permutation $[i]$, then $B_{ij} = A_{p_i j}$ for all i, j.

Example

This example permutes the columns of a matrix `a`.

```
#include <imsls.h>

void main()
{
    float a[] = {3.0, 5.0, 1.0, 2.0, 4.0,
                 3.0, 5.0, 1.0, 2.0, 4.0,
                 3.0, 5.0, 1.0, 2.0, 4.0};
    int permutation[] = {2, 3, 0, 4, 1};
    float     *output;
    int        n_rows = 3;
    int        n_columns = 5;

    output = imsls_f_permute_matrix (n_rows, n_columns, a, permutation,
        IMSLS_PERMUTE_COLUMNS,
        0);

    imsls_f_write_matrix ("permuted matrix", n_rows, n_columns, output,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_COL_NUMBER_ZERO,
        0);
}
```

Output

```
                      permuted matrix
             0           1           2           3           4
0            1           2           3           4           5
1            1           2           3           4           5
2            1           2           3           4           5
```

binomial_coefficient

Evaluates the binomial coefficient.

Synopsis

#include `<imsls.h>`

int `imsls_f_binomial_coefficient` (*int* `n`, *int* `m`)

The type *double* procedure is `imsls_d_binomial_coefficient`.

Required Arguments

int `n` (Input)
First parameter of the binomial coefficient. Argument `n` must be nonnegative.

int `m` (Input)
Second parameter of the binomial coefficient. Argument `m` must be nonnegative.

Return Value

The binomial coefficient

$$\binom{n}{m}$$

is returned.

Description

The binomial function is defined to be

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

with $n \geq m \geq 0$. Also, n must not be so large that the function overflows.

Example

In this example, $\binom{9}{5}$ is computed and printed.

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

main()
{
    int       n = 9;
    int       m = 5;
    int       ans;

    ans = imsls_f_binomial_coefficient(n, m);
    printf("binomial coefficient = %d\n", ans);
}
```

Output

```
binomial coefficient = 126
```

beta

Evaluates the complete beta function.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta` (*float* `a`, *float* `b`)

The type *double* procedure is `imsls_d_beta`.

Required Arguments

float `a` (Input)
First beta parameter. It must be positive.

float `b` (Input)
Second beta parameter. It must be positive.

Return Value

The value of the beta function β(`a`, `b`). If no result can be computed, then NaN is returned.

Description

The beta function, β(`a`, `b`), is defined to be

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

Example

Evaluate the beta function β(0.5, 0.2).

```
#include <imsls.h>

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

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

Output

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

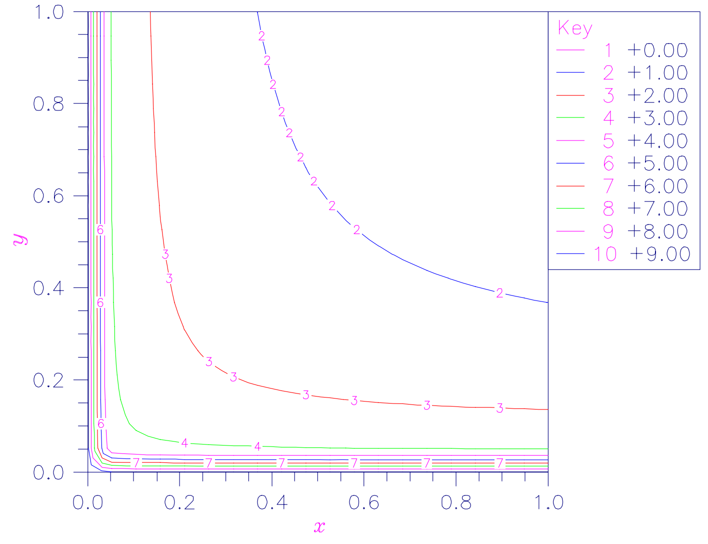


Figure 15-1 Plot of β (x, b*)*

The beta function requires that $a > 0$ and $b > 0$. It underflows for large arguments.

Alert Errors

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

Fatal Errors

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

beta_incomplete

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

Synopsis

#include `<imsls.h>`

float `imsls_f_beta_incomplete` (*float* `x`, *float* `a`, *float* `b`)

The type *double* procedure is `imsls_d_beta_incomplete`.

Required Arguments

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

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

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

Return Value

The value of the incomplete beta function.

Description

The incomplete beta function is defined to be

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

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

Example

Evaluate the log of the incomplete beta function $I_{0.61} = \beta_{0.61}$ (2.2,3.7)/β(2.2,3.7).

```
#include <imsls.h>

main()
{
    float       x = 0.61;
    float       a = 2.2;
    float       b = 3.7;
    float       ans;

    ans = imsls_f_beta_incomplete(x, a, b);
    printf("beta incomplete = %f\n", ans);
}
beta incomplete = 0.8822;
```

log_beta

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

Synopsis

#include `<imsls.h>`

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

The type *double* procedure is `imsls_d_log_beta`.

Required Arguments

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

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

Return Value

The value of the logarithm of the beta function β(*x*, *y*).

Description

The beta function, β(*x*, *y*), is defined to be

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

and `imsls_f_log_beta` returns ln β(*x*, *y*).

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

Warning Errors

`IMSLS_X_IS_TOO_CLOSE_TO_NEG_1`	The result is accurate to less than one precision because the expression $-x/(x+y)$ is too close to -1.

Example

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

```
#include <imsls.h>

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

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

Output

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

gamma

Evaluates the real gamma function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma` (*float* `x`)

The type *double* procedure is `imsls_d_gamma`.

Required Arguments

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

Return Value

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

Description

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

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

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

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

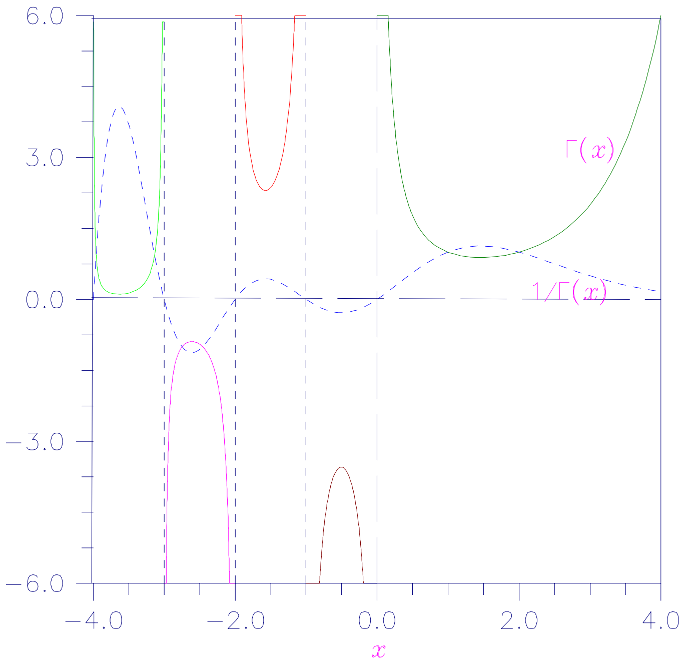


Figure 15-2 Plot of Γ(x) and 1/Γ(x)

Alert Errors

`IMSLS_SMALL_ARG_UNDERFLOW`	The argument x must be large enough that $\Gamma(x)$ does not underflow. The underflow limit occurs first for arguments close to large negative half integers. Even though other arguments away from these half

	integers may yield machine-representable values of $\Gamma(x)$, such arguments are considered illegal.

Warning Errors

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

Example

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

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

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

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

Output

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

Fatal Errors

`IMSLS_ZERO_ARG_OVERFLOW`	The argument for the gamma function is too close to zero.
`IMSLS_NEAR_NEG_INT_FATAL`	The argument for the function is too close to a negative integer.
`IMSLS_LARGE_ARG_OVERFLOW`	The function overflows because x is too large.
`IMSLS_CANNOT_FIND_XMIN`	The algorithm used to find x_{min} failed. This error should never occur.
`IMSLS_CANNOT_FIND_XMAX`	The algorithm used to find x_{max} failed. This error should never occur.

gamma_incomplete

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

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_incomplete` (*float* `a`, *float* `x`)

The type *double* procedure is `imsls_d_gamma_incomplete`.

Required Arguments

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

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

Return Value

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

Description

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

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt$$

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

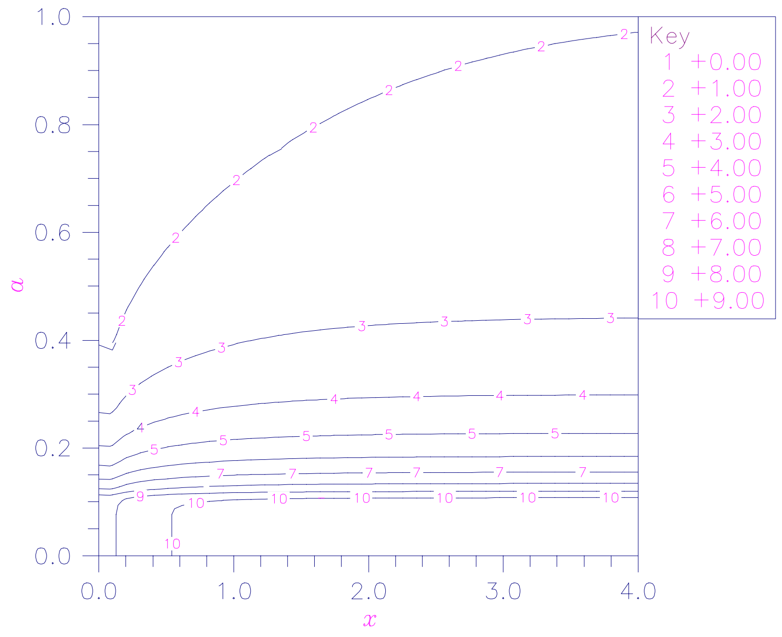


Figure 15-3 Contour Plot of γ(a, x)

Example

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

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

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

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

Output

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

Fatal Errors

`IMSLS_NO_CONV_200_TS_TERMS`	The function did not converge in 200 terms of Taylor series.
`IMSLS_NO_CONV_200_CF_TERMS`	The function did not converge in 200 terms of the continued fraction.

log_gamma

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

Synopsis

#include `<imsls.h>`

float `imsls_f_log_gamma` (*float* `x`)

The type *double* procedure is `imsls_d_log_gamma`.

Required Arguments

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

Return Value

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

Description

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

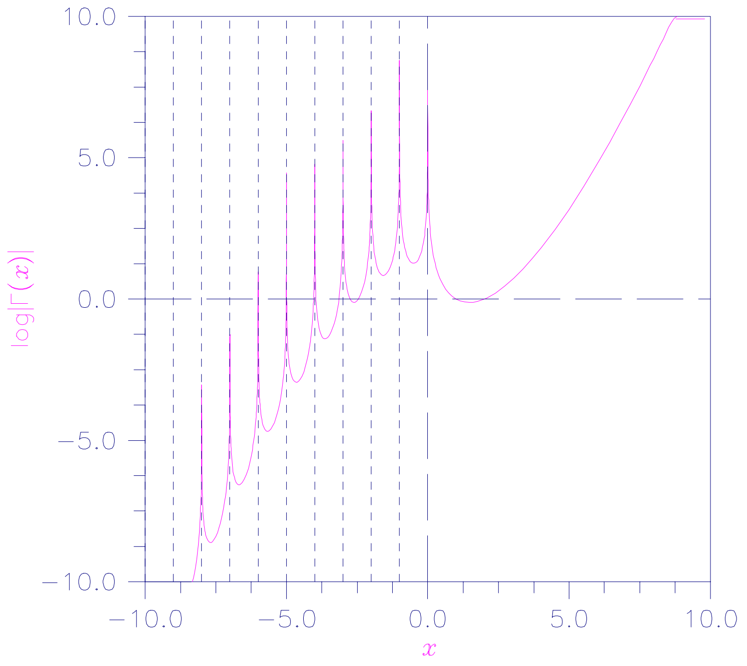


Figure 15-4 Plot of log|Γ(x)|

Example

In this example, log |Γ(3.5)| is computed and printed.

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

main()
{
    float       x = 3.5;
    float       ans;
    ans = imsls_f_log_gamma(x);
    printf("log gamma(%f) = %f\n", x, ans);
}
```

Output

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

Warning Errors

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

Fatal Errors

`IMSLS_NEGATIVE_INTEGER`	The argument for the function cannot be a negative integer.
`IMSLS_NEAR_NEG_INT_FATAL`	The argument for the function is too close to a negative integer.
`IMSLS_LARGE_ABS_ARG_OVERFLOW`	$\|x\|$ must not be so large that the result overflows.

ctime

Returns the number of CPU seconds used.

Synopsis

#include `<imsls.h>`

double `imsls_ctime` ()

Return Value

The number of CPU seconds used by the program.

Example

The CPU time needed to compute

$$\sum_{k=0}^{1,000,000} k$$

is obtained and printed. The time needed is machine dependent. The CPU time needed will varies slightly from run to run on the same machine.

```
#include <imsls.h>

main()
{
    int     k;
    double  sum, time;
                                /* Sum 1 million values */
    for (sum=0, k=1;  k<=1000000; k++)
         sum += k;
                                /* Get amount of CPU time used */
    time = imsls_ctime();
    printf("sum = %f\n", sum);

    printf("time = %f\n", time);
}
```

Output

```
sum = 500000500000.000000
time = 0.820000
```

Reference Material

User Errors

IMSL functions attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, various levels of severity of errors are recognized, and the extent of the error in the context of the purpose of the function also is considered; a trivial error in one situation can be serious in another. IMSL attempts to report as many errors as can reasonably be detected. Multiple errors present a difficult problem in error detection because input is interpreted in an uncertain context after the first error is detected.

What Determines Error Severity

In some cases, the user's input may be mathematically correct, but because of limitations of the computer arithmetic and of the algorithm used, it is not possible to compute an answer accurately. In this case, the assessed degree of accuracy determines the severity of the error. In cases where the function computes several output quantities, some are not computable but most are, an error condition exists. The severity of the error depends on an assessment of the overall impact of the error.

Kinds of Errors and Default Actions

Five levels of severity of errors are defined in IMSL C/Stat/Library. Each level has an associated PRINT attribute and a STOP attribute. These attributes have default settings (YES or NO), but they may also be set by the user. The purpose of having multiple error types is to provide independent control of actions to be taken for errors of different levels of severity. Upon return from an IMSL function, exactly one error state exists. (A code 0 "error" is no error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except `IMSLS_TERMINAL` may be informational errors. The include file, *imsls.h*, defines each of `IMSLS_NOTE`, `IMSLS_ALERT`, `IMSLS_WARNING`, `IMSLS_FATAL`, `IMSLS_TERMINAL`, `IMSLS_WARNING_IMMEDIATE`, and `IMSLS_FATAL_IMMEDIATE` as enumerated data type *Imsls_error*.

`IMSLS_NOTE`. A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations.
Default attributes: PRINT=NO, STOP=NO

`IMSLS_ALERT`. An *alert* indicates that a function value has been set to 0 due to underflow.
Default attributes: PRINT=NO, STOP=NO

`IMSLS_WARNING`. A *warning* indicates the existence of a condition that may require corrective action by the user or calling function. A warning error may be issued because the results are accurate to only a few decimal places; because some of the output may be erroneous, but most of the output is correct; or because some assumptions underlying the analysis technique are violated. Usually no corrective action is necessary, and the condition can be ignored.
Default attributes: PRINT=YES, STOP=NO

`IMSLS_FATAL`. A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling function must take corrective action to recover.
Default attributes: PRINT=YES, STOP=YES

`IMSLS_TERMINAL`. A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors can also be caused by various programming errors impossible to diagnose correctly in C. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the function with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error, because corrective action within the program is generally not reasonable. In normal use, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur.
Default attributes: PRINT=YES, STOP=YES

`IMSLS_WARNING_IMMEDIATE`. An *immediate warning* error is identical to a warning error, except it is printed immediately.
Default attributes: PRINT=YES, STOP=NO

`IMSLS_FATAL_IMMEDIATE`. An *immediate fatal* error is identical to a fatal error, except it is printed immediately.
Default attributes: PRINT=YES, STOP=YES

The user can set PRINT and STOP attributes by calling function `imsls_error_options` as described in Chapter 14, "Utilities."

Errors in Lower-level Functions

It is possible that a user's program may call an IMSL function that in turn calls a nested sequence of lower-level IMSL functions. If an error occurs at a lower level in such a nest of functions and if the lower-level function cannot pass the information up to the original user-called function, then a traceback of the functions is produced. The only common situation in which this can occur is when an IMSL function calls a user-supplied routine that in turn calls another IMSL function.

Functions for Error Handling

The user may interact in two ways with the IMSL error-handling system: (1) to change the default actions and (2) to determine the code of an informational error so as to take corrective action. The IMSL functions to use are `imsls_error_options` and `imsls_error_code`. Function `imsls_error_options` sets the actions to be taken when errors occur. Function `imsls_error_code` retrieves the integer code for an informational error. These functions are documented in Chapter 15, "Utilities."

Threads and Error Handling

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options using `imsls_error_options` (excluding `IMSLS_SET_SIGNAL_TRAPPING`) for each thread by calling `imsls_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);
```

See Chapter 15, "Utilities", examples 3 and 4 of `imsls_error_options` for multithreaded examples.

Use of Informational Error to Determine Program Action

In the program segment below, a factor analysis is to be performed on the matrix covariances. If it is determined that the matrix is singular (and often this is not immediately obvious), the program is to take a different branch.

```
x = imsls_f_factor_analysis (nobs, covariances,
        n_factors, 0);
if (imsls_error_code() == IMSLS_COV_IS_SINGULAR) {
       /*  Handle a singular matrix  */
}
```

Additional Examples

See functions `imsls_error_options` and `imsls_error_code` in Chapter 15, "Utilities" for additional examples.

Product Support

Contacting Visual Numerics Support

Users within support warranty may contact Visual Numerics regarding the use of the IMSL C Numerical Libraries. Visual Numerics can consult on the following topics:

- Clarity of documentation
- Possible Visual Numerics-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem

Not included in these topics are mathematical/statistical consulting and debugging of your program.

Contact Visual Numerics Product Support emailing:

- `http://www.vni.com/tech/imsl/phone.html`

Electronic addresses are not handled uniformly across the major networks, and some local conventions for specifying electronic addresses might cause further variations to occur; contact your E-mail postmaster for further details.

The following describes the procedure for consultation with Visual Numerics:

1. Include your VNI license number
2. Include the product name and version number: IMSL C Numerical Library Version 6.0
3. Include compiler and operating system version numbers
4. Include the name of the routine for which assistance is needed and a description of the problem

Appendix A: References

Abe

Abe, S. (2001) Pattern Classification: Neuro-Fuzzy Methods and their Comparison, Springer-Verlag.

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.

Akaike

Akaike, H., (1978), *Covariance Matrix Computation of the State Variable of a Stationary Gaussian Process*, Ann. Inst. Statist. Math. 30 , Part B, 499-504.

Akaike et al

Akaike, H. , Kitagawa, G., Arahata, E., Tada, F., (1979), Computer Science Monographs No. 13, The Institute of Statistical Mathematics, Tokyo.

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, T. W. (1994) *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.

Berry

Berry, M. J. A. and Linoff, G. (1997) *Data Mining Techniques*, John Wiley & Sons, Inc.

Best and Fisher

Best, D.J., and N.I. Fisher (1979), Efficient simulation of the von Mises distribution, *Applied Statistics*, **28**, 152–157.

Bishop

Bishop, C. M. (1995) *Neural Networks for Pattern Recognition*, Oxford University Press.

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 (1970) *Time Series Analysis: Forecasting and Control*, Holden-Day, Inc.

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.

Box et al.

Box, George E.P., Jenkins,Gwilym M. and Reinsel G.C., (1994) *Time Series Analysis*, Third edition, Prentice Hall, Englewood Cliffs, New Jersey.

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.

Breiman

Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984) *Classification and Regression Trees*, Chapman & Hall. For the latest information on CART visit http://www.salford-systems.com/cart.php.

Breslow

Breslow, N.E. (1974), Covariance analysis of censored survival data, *Biometrics*, **30**, 89–99.

Bridel

Bridle, J. S. (1990) Probabilistic Interpretation of Feedforward Classification Network Outputs, with relationships to statistical pattern recognition, in F. Fogelman Soulie and J. Herault (Eds.), *Neuralcomputing: Algorithms, Architectures and Applications*, Springer-Verlag, 227-236.

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.

Calvo

Calvo, R. A. (2001) Classifying Financial News with Neural Networks, *Proceedings of the 6th Australasian Document Computing Symposium*.

Chen and Liu

Chen, C. and Liu, L., *Joint Estimation of Model Parameters and Outlier Effects in Time Series*, Journal of the American Statistical Association, Vol. 88, No.421, March 1993.

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.

Clarkson and Jenrich

Clarkson, Douglas B. and Robert B Jenrich (1991), Computing extended maximum likelihood estimates for linear parameter models, submitted to *Journal of the Royal Statistical Society*, Series B, **53**, 417-426.

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.

Elliot

Elliot, D.L. (1993) A better activation function for artificial neural networks. Tech. Report. TR 93-8. Institute for Systems Research, Univ. of Maryland.

Elman

Elman, J. L. (1990) Finding Structure in Time, *Cognitive Science,* 14, 179-211.

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.

Giudici

Giudici, P. (2003) *Applied Data Mining: Statistical Methods for Business and Industry*, John Wiley & Sons, Inc.

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.

Hamilton

Hamilton, James D., *Time Series Analysis*, Princeton University Press, Princeton (NewJersey), 1994.

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.

Hebb

Hebb, D. O. (1949) *The Organization of Behaviour: A Neuropsychological Theory*, John Wiley.

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.

Hopfield

Hopfield, J. J. (1987) Learning Algorithms and Probability Distributions in Feed-Forward and Feed-Back Networks, *Proceedings of the National Academy of Sciences*, 84, 8429-8433.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hutchinson

Hutchinson, J. M. (1994) *A Radial Basis Function Approach to Financial Timer Series Analysis*, Ph.D. dissertation, Massachusetts Institute of Technology.

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.

Hwang

Hwang, J. T. G. and Ding, A. A. (1997) Prediction Intervals for Artificial Neural Networks, *Journal of the American Statistical Society*, 92(438) 748-757.

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.

Jacobs

Jacobs, R. A., Jorday, M. I., Nowlan, S. J., and Hinton, G. E. (1991) Adaptive Mixtures of Local Experts, *Neural Computation*, 3(1), 79-87.

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.

Kohonen

Kohonen, T. (1995), Self-Organizing Maps, Third Edition. Springer Series in Information Sciences., New York.

Kuehl, R. O.

Kuehl, R. O. (2000) *Design of Experiments: Statistical Principles of Research Design and Analysis*, 2^{nd} 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.

Kitagawa and Akaike

Kitagawa, G. and Akaike, H., A Procedure for the modeling of non-stationary time series, Ann. Inst. Statist. Math. 30 (1978), Part B, 351-363.

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.

Lawrence et al

Lawrence, S., Giles, C. L, Tsoi, A. C., Back, A. D. (1997) Face Recognition: A Convolutional Neural Network Approach, *IEEE Transactions on Neural Networks, Special Issue on Neural Networks and Pattern Recognition*, 8(1), 98-113.

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.

Li

Li, L. K. (1992) Approximation Theory and Recurrent Networks, Proc. Int. Joint Conf. On Neural Networks, vol. II, 266-271.

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.

Lippmann

Lippmann, R. P. (1989) Review of Neural Networks for Speech Recognition, Neural Computation, I, 1-38.

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.

Loh

Loh, W.-Y. and Shih, Y.-S. (1997) Split Selection Methods for Classification Trees, *Statistica Sinica*, 7, 815-840. For information on the latest version of QUEST see http://www.stat.wisc.edu/~loh/quest.html.

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.

Matsumoto and Nishimure

Makoto Matsumoto and Takuji Nishimura, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, Pages 3–30.

Mandic

Mandic, D. P. and Chambers, J. A. (2001) *Recurrent Neural Networks for Prediction*, John Wiley & Sons, LTD.

Manning

Manning, C. D. and Schütze, H. (1999) *Foundations of Statistical Natural Language Processing,* MIT Press.

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.

McCulloch

McCulloch, W. S. and Pitts, W. (1943) A Logical Calculus for Ideas Imminent in Nervous Activity, *Bulletin of Mathematical Biophysics*, 5, 115-133.

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.

Ozaki and Oda

Ozaki, T and Oda H (1978) Nonlinear time series model identification by Akaike's information criterion. *Information and Systems, Dubuisson eds*, Pergamon Press. 83-91.

Pao

Pao, Y. (1989) *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley Publishing.

Palm

Palm, F. C. (1996), GARCH models of volatility. In *Handbook of Statistics*, Vol. 14, 209-240. Eds: Maddala and Rao. Elsevier,New York.

Parker

Parker, D. B., (1985), *Learning Logic*. Technical Report TR-47, Cambridge, MA: MIT Center for Research in computational Economics and Management Science.

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.

Poli

Poli, I. and Jones, R. D. (1994) A Neural Net Model for Prediction, *Journal of the American Statistical Society*, 89(425) 117-121.

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.

Quinlan

Quinlan, J. R. (1993). C4.5 Programs for Machine Learning, Morgan Kaufmann. For the latest information on Quinlan's algorithms see http://www.rulequest.com/.

Rao

Rao, C. Radhakrishna (1973), *Linear Statistical Inference and Its Applications*, 2d ed., John Wiley & Sons, New York.

Reed

Reed, R. D. and Marks, R. J. II (1999) *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, The MIT Press, Cambridge, MA.

Ripley

Ripley, B. D. (1994) Neural Networks and Related Methods for Classification, *Journal of the Royal Statistical Society B*, 56(3), 409-456.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*, Cambridge University Press.

Robinson

Robinson, Enders A. (1967), *Multichannel Time Series Analysis with Digital Computer Programs*, Holden-Day, San Francisco.

Rosenblatt

Rosenblatt, F. (1958) The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, *Psychol. Rev.*, 65, 386-408.

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.

Rumelhart et al.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) Learning Representations by Back-Propagating Errors, *Nature*, 323, 533-536.

Rumelhart, D. E. and McClelland, J. L. eds. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1, 318-362, Cambridge, MA, MIT Press.

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.

Smith

Smith, M. (1993) *Neural Networks for Statistical Modeling*, New York: Van Nostrand Reinhold.

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.

Studenmund

Studenmund, A. H. (1992) Using Economics: A Practical Guide, New York: Harper Collins.

Swingler

Swingler, K. (1996) Applying Neural Networks: A Practical Guide, Academic Press.

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.

Tesauro

Tesauro, G. (1990) Neurogammon Wins Computer Olympiad, *Neural Computation*, 1, 321-323.

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.

Warner

Warner, B. and Misra, M. (1996) Understanding Neural Networks as Statistical Tools, *The American Statistician*, 50(4) 284-293.

Weisberg

Weisberg, S. (1985), *Applied Linear Regression*, 2d ed., John Wiley & Sons, New York.

Werbos

Werbos, P. (1974) *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science*, PhD thesis, Harvard University, Cambridge, MA.

Werbos, P. (1990) Backpropagation Through Time: What It Does and How to do It, *Proc. IEEE*, 78, 1550-1560.

Williams

Williams, R. J. and Zipser, D. (1989) A Learning Algorithm for Continuously Running Fully Recurrent Neural Networks, *Neural Computation,* 1, 270-280.

Witten

Witten, I. H. and Frank, E. (2000) *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann Publishers.

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.

Wu

Wu, S-I (1995) Mirroring Our Thought Processes, *IEEE Potentials*, 14, 36-41.

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

Routines

Function/Page	Purpose Statement
A	
anova_balanced on page 254	Analyzes a balanced complete experimental design for a fixed, random, or mixed model.
anova_factorial on page 237	Analyzes a balanced factorial design with fixed effects.
anova_nested on page 245	Analyzes a completely nested random model with possibly unequal numbers in the subgroups.
anova_oneway on page 228	Analyzes a one-way classification model.
arma on page 511	Computes least-square estimates of parameters for an ARMA model.
arma_forecast on page 527	Computes forecasts and their associated probability limits for an ARMA model.
autocorrelation on page 588	Computes the sample autocorrelation function of a stationary time series.
auto_arima on page 555	Automatically identifies time series outliers, determines parameters of a multiplicative seasonal ARIMA $(p,0,q)\times(0,d,0)_s$ model and produces forecasts that incorporate the effects of outliers whose effects persist beyond the end of the series
auto_uni_ar on page 532	Automatic selection and fitting of a univariate autoregressive time series model.
B	
beta on page 1020	Evaluates the complete beta function.
beta_cdf on page 783	Evaluates the beta probability distribution function.
beta_incomplete on	Evaluates the real incomplete beta function.

page 1021	
beta_inverse_cdf on page 785	Evaluates the inverse of the beta distribution function.
binomial_cdf on page 774	Evaluates the binomial distribution function.
binomial_coefficient on page 1018	Evaluates the binomial coefficient.
binomial_pdf on page 775	Evaluates the binomial probability function.
bivariate_normal_cdf on page 786	Evaluates the bivariate normal distribution function.
box_cox_transform on page 584	Performs a Box-Cox transformation.

C

categorical_glm on page 422	Analyzes categorical data using logistic, Probit, Poisson, and other generalized linear models.
chi_squared_cdf on page 788	Evaluates the chi-squared distribution function.
chi_squared_inverse_cdf on page 789	Evaluates the inverse of the chi-squared distribution function.
chi_squared_test on page 475	Performs a chi-squared goodness-of-fit test.
cluster_hierarchical on page 645	Performs a hierarchical cluster analysis given a distance matrix.
cluster_k_means on page 653	Performs a *K*-means (centroid) cluster analysis.
cluster_number on page 649	Computes cluster membership for a hierarchical cluster tree.
cochran_q_test on page 466	Performs a Cochran *Q* test for related observations.
contingency_table on page 402	Performs a chi-squared analysis of a two-way contingency table.
continuous_table_setup on page 862	Sets up table to generate pseudorandom numbers from a general continuous distribution.
covariances on page 185	Computes the sample variance-covariance or correlation matrix.
cox_stuart_trends_test on page 448	Performs the Cox and Stuart' sign test for trends in location and dispersion.
crd_factorial on page 266	Analyzes data from balanced and unbalanced completely randomized experiments.
crosscorrelation on page 593	Computes the sample cross-correlation function of two stationary time series

D

data_sets on page 1009	Retrieves a commonly analyzed data set.

difference on page 572	Differences a seasonal or nonseasonal time series.
discrete_table_setup on page 832	Sets up a table to generate pseudorandom numbers from a general discrete distribution.
discriminant_analysis on page 682	Performs discriminant function analysis.

E

error_code on page 1004	Returns the code corresponding to the error message from the last function called.
error_options on page 998	Sets various error handling options.
estimate_missing on page 614	Estimates missing values in a time series.
exact_enumeration on page 414	Computes exact probabilities in a two-way contingency table, using the total enumeration method.
exact_network on page 416	Computes exact probabilities in a two-way contingency table using the network algorithm.

F

factor_analysis on page 640	Extracts initial factor-loading estimates in factor analysis.
faure_next_point on page 911	Computes a shuffled Faure sequence
friedmans_test on page 462	Performs Friedman's test for a randomized complete block design.

G

gamma on page 1023	Evaluates the real gamma functions.
gamma_cdf on page 798	Evaluates the gamma distribution function.
gamma_incomplete on page 1025	Evaluates the incomplete gamma function.
gamma_inverse_cdf on page 799	Evaluates the inverse of the gamma distribution function.
garch on page 621	Computes estimates of the parameters of a GARCH (p, q) model

H

homogeneity on page 376	Conducts Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.
hypergeometric_cdf on page 777	Evaluates the hypergeometric distribution function.
hypergeometric_pdf on page 778	Evaluates the hypergeometric probability function.
hypothesis_partial on page 95	Constructs a completely testable hypothesis.

hypothesis_scph on page 100	Sums of cross products for a multivariate hypothesis.
hypothesis_test on page 105	Tests for the multivariate linear hypothesis.

I

J

K

kalman on page 626	Performs Kalman filtering and evaluates the likelihood function for the state-space model.
kaplan_meier_estimates on page 708	Computes Kaplan-Meier estimates of survival probabilities in stratified samples.
kolmogorov_one on page 487	Performs a Kolmogorov-Smirnov's one-sample test for continuos distributions.
kolmogorov_two on page 490	Performs a Kolmogorov-Smirnov's two-sample test
kruskal_wallis_test on page 459	Performs a Kruskal-Wallis's test for identical population medians.
k_trends_test on page 469	Performs k-sample trends test against ordered alternatives.

L

lack_of_fit on page 611	Performs lack-of-fit test for an univariate time series or transfer function given the appropriate correlation function.
latin_square on page 287	Analyzes data from latin-square experiments.
lattice on page 296	Analyzes balanced and partially-balanced lattice experiments.
life_tables on page 764	Produces population and cohort life tables.
Lnorm_regression on page 166	Fits a multiple linear regression model using criteria other than least squares.
log_beta on page 1022	Evaluates the log of the real beta function.
log_gamma on page 1027	Evaluates the logarithm of the absolute value of the gamma function.

M

machine (float) on page 1007	Returns information describing the computer's floating-point arithmetic.
machine (integer) on page 1005	Returns integer information describing the computer's arithmetic.
mat_mul_rect on page 1012	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any

	triple product.
max_arma on page 521	Exact maximum likelihood estimation of the parameters in a univariate ARMA (autoregressive, moving average) time series model.
mlff_network on page 934	Creates a multilayered feedforward neural network.
mlff_network_forecast on page 954	Calculates forecasts for trained multilayered feedforward neural networks.
mlff_network_trainer on page 944	Trains a multilayered feedforward neural network.
multi_crosscorrelation on page 599	Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.
multiple_comparisons on page 383	Performs Student-Newman-Keuls multiple comparisons test.
multivar_normality_test on page 493	Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.

N

noether_cyclical_trend on page 444	Performs the Noether's test for cyclical trend.
non_central_chi_sq on page 791	Evaluates the noncentral chi-squared distribution function.
non_central_chi_sq_inv on page 793	Evaluates the inverse of the noncentral chi-squared function.
non_central_t_cdf on page 807	Evaluates the noncentral Student's *t* distribution function.
non_central_t_inv_cdf on page 809	Evaluates the inverse of the noncentral Student's *t* distribution function.
nonlinear_optimization on page 157	Fits a nonlinear regression model using Powell's algorithm.
nonlinear_regression on page 147	Fits a nonlinear regression model.
nonparam_hazard_rate on page 756	Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.
normal_cdf on page 801	Evaluates the standard normal (Gaussian) distribution function.
normal_inverse_cdf on page 802	Evaluates the inverse of the standard normal (Gaussian) distribution function.
normal_one_sample on page 7	Computes statistics for mean and variance inferences using a sample from a normal population.
normal_two_sample on page 11	Computes statistics for mean and variance inferences using samples from two normal population.
normality_test on page 483	Performs a test for normality.

O

`output_file` on page 993	Sets the output file or the error message output file.

P

`page` on page 986	Sets or retrieves the page width or length.
`partial_autocorrelation` on page 608	Computes the sample partial autocorrelation function of a stationary time series.
`partial_covariances` on page 192	Computes partial covariances or partial correlations from the covariance or correlation matrix.
`permute_matrix` on page 1017	Permutes the rows or columns of a matrix.
`permute_vector` on page 1015	Rearranges the elements of a vector as specified by a permutation.
`poisson_cdf` on page 779	Evaluates the Poisson distribution function.
`poisson_pdf` on page 781	Evaluates the Poisson probability function.
`poly_prediction` on page 137	Computes predicted values, confidence intervals, and diagnostics after fitting a polynomial regression model.
`poly_regression` on page 130	Performs a polynomial least-squares regression.
`pooled_covariances` on page 197	Computes a pooled variance-covariance from the observations.
`principal_components` on page 640	Computes principal components.
`prop_hazards_gen_lin` on page 713	Analyzes time event data via the proportional hazards model.

Q

R

`random_arma` on page 880	Generates pseudorandom ARMA process numbers.
`random_beta` on page 837	Generates pseudorandom numbers from a beta distribution.
`random_binomial` on page 816	Generates pseudorandom binomial numbers.
`random_cauchy` on page 838	Generates pseudorandom numbers from a Cauchy distribution.
`random_chi_squared` on page 840	Generates pseudorandom numbers from a chi-squared distribution.
`random_exponential` on page 841	Generates pseudorandom numbers from a standard exponential distribution.
`random_exponential_mix` on page 843	Generates pseudorandom mixed numbers from a standard exponential distribution.
`random_gamma` on page 845	Generates pseudorandom numbers from a standard gamma

	distribution.
`random_general_continuous` on page 859	Generates pseudorandom numbers from a general continuous distribution.
`random_general_discrete` on page 828	Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.
`random_geometric` on page 818	Generates pseudorandom numbers from a geometric distribution.
`random_GFSR_table_get` on page 902	Retrieves the current table used in the GFSR generator.
`random_GFSR_table_set` on page 901	Sets the current table used in the GFSR generator.
`random_hypergeometric` on page 819	Generates pseudorandom numbers from a hypergeometric distribution.
`random_logarithmic` on page 822	Generates pseudorandom numbers from a logarithmic distribution.
`random_lognormal` on page 846	Generates pseudorandom numbers from a lognormal distribution.
`random_MT32_init` on page 905	Initializes the 32-bit Mersenne Twister generator using an array.
`random_MT32_table_get` on page 905	Retrieves the current table used in the 32-bit Mersenne Twister generator.
`random_MT32_table_set` on page 907	Sets the current table used in the 32-bit Mersenne Twister generator.
`random_MT64_init` on page 908	Initializes the 64-bit Mersenne Twister generator using an array.
`random_MT64_table_get` on page 908	Retrieves the current table used in the 64-bit Mersenne Twister generator.
`random_MT64_table_set` on page 910	Sets the current table used in the 64-bit Mersenne Twister generator.
`random_multinomial` on page 871	Generates pseudorandom numbers from a multinomial distribution.
`random_mvar_from_data` on page 868	Generates pseudorandom numbers from a multivariate distribution determined from a given sample.
`random_neg_binomial` on page 823	Generates pseudorandom numbers from a negative binomial distribution.
`random_normal` on page 848	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.
`random_normal_multivariate` on page 864	Generates pseudorandom numbers from a multivariate normal distribution.
`random_npp` on page 884	Generates pseudorandom numbers from a nonhomogeneous Poisson process.
`random_option` on page 894	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.
`random_option_get` on page 895	Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.

`random_order_normal` on page 876	Generates pseudorandom order statistics from a standard normal distribution.
`random_order_uniform` on page 878	Generates pseudorandom order statistics from a uniform (0, 1) distribution
`random_orthogonal_matrix` on page 866	Generates a pseudorandom orthogonal matrix or a correlation matrix.
`random_permutation` on page 887	Generates a pseudorandom permutation.
`random_poisson` on page 825	Generates pseudorandom numbers from a Poisson distribution.
`random_sample` on page 890	Generates a simple pseudorandom sample from a finite population.
`random_sample_indices` on page 889	Generates a simple pseudorandom sample of indices.
`random_seed_get` on page 896	Retrieves the current value of the seed used in the IMSL random number generators.
`random_seed_set` on page 899	Initializes a random seed for use in the IMSL random number generators.
`random_sphere` on page 873	Generates pseudorandom points on a unit circle or `K`-dimensional sphere.
`random_stable` on page 850	Sets up a table to generate pseudorandom numbers from a general discrete distribution.
`random_student_t` on page 852	Generates pseudorandom Student's *t*.
`random_substream_seed_get` on page 897	Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.
`random_table_get` on page 900	Retrieves the current table used in the shuffled generator.
`random_table_set` on page 900	Sets the current table used in the shuffled generator.
`random_table_twoway` on page 875	Generates a pseudorandom two-way table.
`random_triangular` on page 853	Generates pseudorandom numbers from a triangular distribution.
`random_uniform` on page 854	Generates pseudorandom numbers from a uniform (0, 1) distribution.
`random_uniform_discrete` on page 826	Generates pseudorandom numbers from a discrete uniform distribution.
`random_von_mises` on page 856	Generates pseudorandom numbers from a von Mises distribution.
`random_weibull` on page 857	Generates pseudorandom numbers from a Weibull distribution.
`randomness_test` on page 497	Performs a test for randomness.
`ranks` on page 34	Computes the ranks, normal scores, or exponential scores

	for a vector of observations.
`rcbd_factorial` on page 277	Analyzes data from balanced and unbalanced randomized complete-block experiments.
`regression` on page 64	Fits a multiple linear regression model using least squares.
`regression_prediction` on page 84	Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.
`regression_selection` on page 112	Selects the best multiple linear regression models.
`regression_stepwise` on page 122	Builds multiple linear regression models using forward selection, backward selection or stepwise selection.
`regression_summary` on page 76	Produces summary statistics for a regression model given the information from the fit.
`regressors_for_glm` on page 55	Generates regressors for a general linear model.
`robust_covariances` on page 203	Computes a robust estimate of a covariance matrix and mean vector.
`random_arma` on page 880	Generates pseudorandom ARMA process numbers.
`random_beta` on page 837	Generates pseudorandom numbers from a beta distribution.
`random_binomial` on page 816	Generates pseudorandom binomial numbers.
`random_cauchy` on page 838	Generates pseudorandom numbers from a Cauchy distribution.
`random_chi_squared` on page 840	Generates pseudorandom numbers from a chi-squared distribution.
`random_exponential` on page 841	Generates pseudorandom numbers from a standard exponential distribution.
`random_exponential_mix` on page 843	Generates pseudorandom mixed numbers from a standard exponential distribution.
`random_gamma` on page 845	Generates pseudorandom numbers from a standard gamma distribution.
`random_general_continuous` on page 859	Generates pseudorandom numbers from a general continuous distribution.
`random_general_discrete` on page 828	Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.
`random_geometric` on page 818	Generates pseudorandom numbers from a geometric distribution.
`random_GFSR_table_get` on page 902	Retrieves the current table used in the GFSR generator.
`random_GFSR_table_set` on page 901	Sets the current table used in the GFSR generator.
`random_hypergeometric` on page 819	Generates pseudorandom numbers from a hypergeometric distribution.
`random_logarithmic` on page 822	Generates pseudorandom numbers from a logarithmic distribution.
`random_lognormal` on page	Generates pseudorandom numbers from a lognormal

846	distribution.
random_multinomial on page 871	Generates pseudorandom numbers from a multinomial distribution.
random_mvar_from_data on page 868	Generates pseudorandom numbers from a multivariate distribution determined from a given sample.
random_neg_binomial on page 823	Generates pseudorandom numbers from a negative binomial distribution.
random_normal on page 848	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.
random_normal_multivariate on page 864	Generates pseudorandom numbers from a multivariate normal distribution.
random_npp on page 884	Generates pseudorandom numbers from a nonhomogeneous Poisson process.
random_option on page 894	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.
random_option_get on page 895	Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.
random_order_normal on page 876	Generates pseudorandom order statistics from a standard normal distribution.
random_order_uniform on page 878	Generates pseudorandom order statistics from a uniform (0, 1) distribution
random_orthogonal_matrix on page 866	Generates a pseudorandom orthogonal matrix or a correlation matrix.
random_permutation on page 887	Generates a pseudorandom permutation.
random_poisson on page 825	Generates pseudorandom numbers from a Poisson distribution.
random_sample on page 890	Generates a simple pseudorandom sample from a finite population.
random_sample_indices on page 889	Generates a simple pseudorandom sample of indices.
random_seed_get on page 896	Retrieves the current value of the seed used in the IMSL random number generators.
random_seed_set on page 899	Initializes a random seed for use in the IMSL random number generators.
random_sphere on page 873	Generates pseudorandom points on a unit circle or K-dimensional sphere.
random_stable on page 850	Sets up a table to generate pseudorandom numbers from a general discrete distribution.
random_student_t on page 852	Generates pseudorandom Student's *t*.
random_substream_seed_get on page 897	Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.
random_table_get on page 900	Retrieves the current table used in the shuffled generator.

random_table_set on page 900	Sets the current table used in the shuffled generator.
random_table_twoway on page 875	Generates a pseudorandom two-way table.
random_triangular on page 853	Generates pseudorandom numbers from a triangular distribution.
random_uniform on page 854	Generates pseudorandom numbers from a uniform (0, 1) distribution.
random_uniform_discrete on page 826	Generates pseudorandom numbers from a discrete uniform distribution.
random_von_mises on page 856	Generates pseudorandom numbers from a von Mises distribution.
random_weibull on page 857	Generates pseudorandom numbers from a Weibull distribution.
randomness_test on page 497	Performs a test for randomness.
ranks on page 34	Computes the ranks, normal scores, or exponential scores for a vector of observations.
rcbd_factorial on page 277	Analyzes data from balanced and unbalanced randomized complete-block experiments.
regression on page 64	Fits a multiple linear regression model using least squares.
regression_prediction on page 84	Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.
regression_selection on page 112	Selects the best multiple linear regression models.
regression_stepwise on page 122	Builds multiple linear regression models using forward selection, backward selection or stepwise selection.
regression_summary on page 76	Produces summary statistics for a regression model given the information from the fit.
regressors_for_glm on page 55	Generates regressors for a general linear model.
robust_covariances on page 203	Computes a robust estimate of a covariance matrix and mean vector.

S

scale_filter on page 960	Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.
seasonal_fit on page 576	Estimates the optimum seasonality parameters for a time series using an autoregressive model, AR(*p*), to represent the time series.
sign_test on page 438	Performs a sign test.
simple_statistics on page 1	Computes basic univariate statistics.
sort_data on page 26	Sorts observations by specified keys, with option to tally cases into a multi-way frequency table.

split_plot on page 314	Analyzes a wide variety of split-plot experiments with fixed, mixed or random factors.
split_split_plot on page 326	Analyzes data from split-split-plot experiments.
strip_plot on page 342	Analyzes data from strip-plot experiments.
strip_split_plot on page 353	Analyzes data from strip-split-plot experiments.
survival_estimates on page 750	Estimates using various parametric models.
survival_glm on page 727	Analyzes survival data using a generalized linear model.

T

t_cdf on page 804	Evaluates the Student's t distribution function.
t_inverse_cdf on page 805	Evaluates the inverse of the Student's t distribution function.
table_oneway on page 17	Tallies observations into one-way frequency table.
table_twoway on page 22	Tallies observations into a two-way frequency table.
tie_statistics on page 453	Computes tie statistics for a sample of observations.
time_series_class_filter on page 969	Converts time series data sorted with nominal classes in decreasing chronological order to useful format for processing by a neural network.
time_series_filter on page 966	Converts time series data to the format required for processing by a neural network.
ts_outlier_forecast on page 547	Computes forecasts, their associated probability limits and ψ -weights for an outlier contaminated time series whose underlying outlier free series follows a general seasonal or nonseasonal ARMA model
ts_outlier_identification on page 537	Detects and determines outliers and simultaneously estimates the model parameters in a time series whose underlying outlier free series follows a general seasonal or nonseasonal ARMA model.

U

unsupervised_nominal_filter on page 973	Converts nominal data into a series of binary encoded columns for input to a neural network.
unsupervised_ordinal_filter on page 976	Converts ordinal data into percentages.

V

version on page 997	Returns integer information describing the version of the library, license number, operating system, and compiler.

W

`wilcoxon_rank_sum` on page 455	Performs a Wilcoxon rank sum test.
`wilcoxon_sign_rank` on page 441	Performs a Wilcoxon sign rank test.
`write_matrix` on page 981	Prints a rectangular matrix (or vector) stored in contiguous memory locations.
`write_options` on page 987	Sets or retieves an option for printing a matrix.

X

Y

Z

Index

A

additive (AO) 539
AIC 535, 579, 580
Airline Data 579
Akaike's information criterion 532
alpha factor analysis 673
ANOVA
  balanced 254
  factorial 237
  multiple comparisons 383
  nested 245
  oneway 228
ANSI C ix
ARIMA models 555
  forecasts 527
  least squares estimates 537
  least-square estimates 511
  maximum likelihood estimates 537
  method of moments 513
  method of moments estimates 524
  method of moments estimation 515
  multiplicative seasonal 560
ARIMA models XE "ARIMA models:least squares estimates" \r "ARIMA" XE 537
ARMA model 521
association, measures of 407
Autoregressive model (AR) 617
Autoregressive Moving Average Model 510

B

backward selection 122
balanced 254
balanced experimental design 254
beta distribution function 783
  inverse 785
beta distribution, simulation 837
beta functions 1020, 1021, 1022
binary encoded 973
binomial coefficient 1018
binomial distribution 774
binomial distributions 816, 823, 832, 862
binomial probability 775
bivariate normal distribution function 786
Bonferroni method 232
bounded scaling 961
Box-Cox transformation 584

C

Cartesian coordinates 873
cauchy distributions 838
chi-squared analysis 402
chi-squared distribution function 788, 789
chi-squared distributions 840
chi-squared goodness-of-fit test 475
chi-squared statistics 401, 405
chi-squared test 475
classification model
  one-way 228
cluster analysis 639, 653
cluster membership 649
cluster_hierarchical 645
cluster_number 649
Cochran Q test 466
coefficient
  excess (kurtosis) 2
  skewness 2
  variation 5
compiler 997
computer constants 1005, 1007
confidence intervals 137
  mean 2
constants 1005, 1007
contingency coefficient 406
contingency tables 414, 416
  two-way 402
correlation matrix 185, 866
correlations 192
counts 2, 26
covariances 203
Cox and Stuart sign test 448
CPU 1029
Cramer's V 406
Crd factorial 266
  factorial experiments 271
  pooled location interaction 271
  unbalanced 266

unbalanced completely
randomized experiments 266
crosscorrelation 593
cross-correlation function 593, 599, 708, 714, 756, 764
cubic spline interpolation 617

D

data sets 1009
detection 538
deviation, standard 2
diagnostic checking 510
diagnostics 137
discrete uniform distributions 826
discriminant function analysis 682
dissimilarities 641
distribution functions
beta 783
inverse 785
bivariate normal 786
chi-squared 788
inverse 789
chi-squared, noncentral 791, 793
inverse 793
F_cdf
inverse 794
F_inverse_cdf 796
gamma 798
Gaussian 801
hypergeometric 777
inverse 802
normal 801
Poisson 779
Student's t 804
inverse 805
Student's t, noncentral 807
inverse 809
Dunn-Sidák method 232

E

eigensystem analysis 640
empirical tests 816
error handling xiv, 998, 1004, 1031
error messages 993
estimate of scale
simple robust 6
excess 5
exponential distribution, simulation 841
exponential scores 34

F

F statistic 15
factor analysis 640, 663
factorial 237
factorial design
analysis 237
Faure 913
Faure sequence 911, 912
faure_next_point 912
finite difference gradient 157
finite population 890
Fisher's LSD 233
forecasting 510
forecasts 547
ARMA models 527, 547
GARCH 547, 621
forward selection 122
frequency tables 17, 22
multi-way 26
Friedman's test 462

G

gamma distribution function 798
gamma distribution, simulation 845
gamma functions 1023, 1025, 1027
gamma_inverse_cdf 799
GARCH
(Generalized Autoregressive Conditional Heteroskedastic) 621
Gaussian distribution functions 801
inverse 802
general continuous distribution 859
general discrete distribution 828, 829, 832, 862
general distributions 475
general linear models 55
Generalized Feedback Shift Register 815
generalized feedback shift register method 814
generalized linear models 401
geometric distributions 818
GFSR 894
GFSR generator 815, 901, 902
goodness-of-fit tests 475
Gray code 914

H

Haar measure 867
hierarchical cluster analysis 645
hierarchical cluster tree 649

homogeneity 376
hypergeometric distribution function 777
hypergeometric distributions 819
hypergeometric_pdf 778
hyper-rectangle 911
hypothesis 95, 100, 105

I

image analysis 673
innovational (AO) 563
innovational (IO) 539, 563
integrated rate function 886
invertible/invertiblility 525

K

Kalman filtering 626
Kaplan_meier estimates 709
Kaplan_meier_estimates 708
Kaplan-Meier estimates
  computes 708
Kappa analysis 401
K-dimensional sphere 873
kernel functions 708, 756
K-means analysis 653
Kolmogorov one-sample test 487
Kolmogorov two-sample test 490
Kruskal-Wallis test 459
k-sample trends test 469
kurtosis 2, 5

L

lack-of-fit test 611
lack-of-fit tests 52
Latin square 287
Lattice 296
  3x3 balanced-lattice 301
  balanced lattice experiments 300
  intra-Block Error 301
  partially-balanced lattice experiments 296, 300
Least Absolute Value 54, 166, 170, 178
Least Maximum Value 54, 166, 183
Least Squares
  Alternatives
    Least Absolute Value 54
    Least Maximum Value 54
    Lp Norm 54
least-squares fit 64, 166, 245, 254, 441, 444, 448, 453, 462, 487, 490, 608
Lebesque measure 913
level shift (LS) 563
level Shift (LS) 539
library version 997
linear dependence 48
linear discriminant function analysis 682
linear regression
  multiple 44
  simple 44
logarithmic distributions 822
low-discrepancy 913
L*p* Norm 54, 171

M

MAD (Median Absolute Deviation) 6
Mardia’s multivariate measures 495
Mardia’s multivariate tests 493
matrices 641, 1012
matrix of dissimilarities 641
matrix storage modes xi
maximum 2, 5
maximum likelihood estimates 632
mean 2, 5, 7, 9
  for two normal populations 11
  normal population 7
Mean Absolute Deviation 962
measures of association 401, 406
measures of prediction 407
measures of uncertainty 407
median 6, 617
  absolute deviation 6
memory allocation xii
Mersenne Twister 905, 907, 908, 910, 1071
minimum 2, 5
missing values 55, 617
models 147
  general linear 55
  multiple linear regression 112
  nonlinear regression 49
  polynomial 45
  polynomial regression 137
Monte Carlo applications 816
multinomial distribution 871
Multiple comparisons 383
Multiple comparisons test
  Bonferroni, Tukey’s, or Duncan’s MRT 383

Student-Newman-Keuls 383
multiple linear regression models 64, 112, 122, 166, 245, 254, 441, 444, 448, 453, 462, 487, 490, 608
multiple_crosscorrelation 599
multiplicative congruential generator 814
multiplicative generator 814
multiplying matrices 1012
multivariate distribution 868
multivariate general linear hypothesis 100, 105
multivariate normal distribution, simulation 864

N

nested 245
nested random model 245, 249
network 934
Noether test 444
nominal data 973
non-ANSI C ix
noncentral chi-squared distribution function 791
inverse 793
noncentral Student's t distribution function 807, 809
nonhomogeneous Poisson process 884
nonlinear model 157
nonlinear regression 147
nonlinear regression models 49, 147
nonparam_hazard_rate 756
nonparametric hazard rate estimation 756
nonuniform generators 816
normal distribution function 802
normal distribution, simulation 848
normal populations
mean 7
variance 7
normal scores 34
normality test 483

O

observations
number of 2
one-step-ahead forecasts 615
oneway 228
one-way classification model 228
one-way frequency table 17
operating system 997
order statistics 876, 878
ordinal data 976
orthogonal matrix 866
outlier
description 563
outlier contaminated series 547
output files 993
overflow xiv

P

parameter estimation 510
partial correlations 192
partial covariances 192
partially tested hypothesis 95
permutations 1015, 1017
phi 406
Poisson distribution function 779
Poisson distribution, simulation 825
poisson_pdf 781
polynomial models 45
polynomial regression 130
polynomial regression models 137
pooled variance-covariance 197
population 764
predicted values 137
prediction coefficient 407
principal components 657
printing
matrices 981
options 987
retrieving page size 986
setting paper size 986
vectors 981
probability limits
ARMA models 527, 547
outlier contaminated series 549
prop_hazards_gen_lin 713
pseudorandom number generators 475
pseudorandom numbers 829, 832, 846, 852, 856, 857, 862
pseudorandom permutation 887
pseudorandom sample 889
p-values 406

Q

quadratic discriminant function analysis 682

R

random number generator 905, 906, 908, 909, 911
random numbers
  beta distribution 837
  exponential distribution 841
  gamma distribution 845
  Poisson distribution 825
  seed
    current value 896
    initializing 899
  selecting generator 894, 895
random numbers generators 848
random_MT32_init 905
random_MT32_table_get 905
random_MT32_table_set 907
random_MT64_init 908
random_MT64_table_get 908
random_MT64_table_set 910
randomness test 497
range 2, 5
ranks 34
Rcbd factorial 277
regression models 44, 76, 84
regressors 55
robust covariances 203

S

sample autocorrelation function 588
sample correlation function 510
sample partial autocorrelation function 608
scale filter 960
scales 960
Scheffé method 232
scores
  exponential 34
  normal 34
seasonal adjustment 577
seasonality parameters 576
seed 897
Seed 815
serial number 997
shuffled generator 900
sign test 438
simulation of random variables 813
skewness 2, 5
Split plot 314
  blocking factor 321
  completely randomized 314
  completely randomized design 321
  experiments 314
  fixed effects 321
  IMSLS_RCBD default setting 322
  random effects 323
  randomized complete block design 314, 321
  randomizing whole-plots 322
  split plot factor 322
  split plot factors 321
  whole plot 321
  whole plot factor 322
  whole plot factors 321
Split Plots
  whole-plots 314
Split-split plot 326
  split-plot factors 327
  split-split-plot experiments 326
  sub-plot factors 327
  whole plot factors 327
stable distribution 850
standard deviation 2, 9
standard errors 406
state vector 626
statespace model 626
stationary/stationarity 525
stepwise selection 122
Strip plot 342
Strip-split plot 353
Student's t distribution function 804
  inverse 805
summary statistics 50
survival probabilities 708, 709

T

t statistic 15
temporary change (TC) 539, 563
tests for randomness 475
Thread Safe x
  multithreaded application x
  single-threaded application xi
  threads and error handling 1033
tie statistics 453
time domain methodology 510
time event data 707, 713
time series 510, 880
  difference 572
time series class filter 969
time series data 966, 969
time series filter 966
transformation 510
transformations 54
transposing matrices 1012
triangular distributions 853
Tukey method 231
Tukey-Kramer method 231

two-way contingency table 875
two-way frequency tables 22
two-way table 875

U

unable to identify (UI) 539, 563
uncertainty, measures of 407
underflow xiv
uniform distribution, simulation 854
unit sphere 873
univariate statistics 1, 422, 727, 750, 843
unscales 960
unsupervised nominal filter 973
unsupervised ordinal filter 976
update equations 627
user-supplied gradient 157

V

variable selection 45
variance 2, 5, 7
  for two normal populations 11
  normal population 7
variance-covariance matrix 185
variation, coefficient of 5

W

weighted least squares 50
white noise
  Gaussian 541, 544, 551
  process 540
white noise process 524
Wilcoxon rank sum test 455
Wilcoxon signed rank test 441
Wilcoxon two-sample test 460
Wolfer Sunspot series 524, 535

Y

yates 388

Z

z-score scaling 961